FernUniversität in Hagen
Fakultät für Mathematik und Informatik
Lehrgebiet Programmiersysteme
Abschlussarbeit im Studiengang Master of Computer Science

Ein Refaktorisierungswerkzeug für die Injektion minimierter Abhängigkeiten

Autor:	Dipl.-Inform. (FH) Michael Mondl
Matrikel-Nr.:	7146124
Adresse:	Sturzgasse 1c/9 A-1140 Wien Österreich
Telefon:	+43 1 990 990 6
E-Mail:	michael@michaelmondl.de
Betreuer:	Prof. Dr. Friedrich Steimann

Wien, den 15. September 2008

Abstract

Während dem Lebenszyklus eines Softwaresystems verschlechtern sich dessen Qualitätseigenschaften, wenn keine aktiven Gegenmaßnahmen ergriffen werden. Eine mögliche Verbesserung der internen Struktur ist die Entkopplung von Modulen durch den Einsatz des Dependency Injection-Patterns. Bei bestehenden komplexen Softwaresystemen ist es jedoch keine triviale Tätigkeit, um dieses Entwurfsmuster einzuführen. Durch die Erzeugung und Anpassungen diverser Typen ist dieser Vorgang sehr fehleranfällig und zeitaufwendig.

Die Aufgabe dieser Abschlussarbeit ist die Planung und Realisierung eines Refaktorisierungswerkzeugs für die Programmiersprache Java, durch das dieses Refactoring automatisiert wird. Die technologische Basis des Werkzeugs bildet die IDE eclipse und das im Lehrgebiet Programmiersysteme der FernUniversität in Hagen entwickelte Infer Type-Refactoring. Neben der Entkopplung der Klassen durch die Einführung des Dependency Injection-Patterns ist das Werkzeug in der Lage eine entsprechende Konfiguration zu erzeugen. Durch die integrierte Generator-Schnittstelle können verfügbare Dependency Injection-Frameworks direkt unterstützt werden.

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich diese Abschlussarbeit selbständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Michael Mondl

Wien, den 15. September 2008

Inhaltsverzeichnis

1 Einleitung

1.1 Ausgangslage

Software zu schreiben, die sowohl funktionale als auch nicht-funktionale Anforderungen erfüllt, ist nicht die einzige Aufgabe eines Softwareentwicklers. Die Software sollte auch nach den Prinzipien des Software Engineering geplant und realisiert werden, damit ein erforderliches Maß an Qualität[1] erreicht wird: Ein Softwaresystem muss nach der Inbetriebnahme gepflegt, gewartet und bei zusätzlichen Anforderungen angepasst werden können. Während der Entwicklung ist deshalb die Berücksichtigung von Qualitätseigenschaften wie Verständlichkeit, Erweiterbarkeit und Wiederverwendbarkeit unabdingbar, um später der Softwarefäulnis entgegenwirken zu können.

Gerade bei größeren objektorientierten Softwaresystemen muss aufgrund der hohen Komplexität deren Struktur mit Bedacht gewählt werden, damit den Forderungen von Softwarequalität nachgekommen werden kann. Eine bewährte Möglichkeit der Strukturierung ist der systematische Einsatz von Interfaces im Sinne der interfacebasierten Programmierung[2]. Dieser Ansatz stützt sich vor allem auf die Software Engineering-Prinzipien

- Lose Kopplung,
- Hohe Kohäsion,
- Trennung der Zuständigkeiten
- und das Geheimnisprinzip.

Hierdurch wird ein Softwaresystem durch Dekomposition in mehrere Module zerlegt, die ausschließlich über Schnittstellen miteinander kommunizieren. In der objektorientierten Programmiersprache Java wird das Konzept der Module auf Packages abgebildet.

Um das Ideal einer vollständigen Entkopplung zu erreichen, dürfen die Typen eines Package lediglich von den Schnittstellentypen eines anderen Package Kenntnis besitzen. Da die Klassentypen eines anderen Package nicht bekannt sein dürfen, können von diesen

[1] Für eine Vertiefung in das Thema Softwarequalität siehe [Bal96] bzw. [Bal98].
[2] Siehe [SM05].

Typen keine Objekte mehr erzeugt werden. Dies führt zu einem Problem, wenn ein Objekt zur Erfüllung seiner Aufgabe auf ein anderes Objekt angewiesen ist, dessen Klassentyp jedoch aus einem anderen Package stammt. Das benötigte Objekt muss deshalb von einer weiteren Komponente erzeugt und an das abhängige Objekt übergeben werden. Da der konkrete Klassentyp des übergebenen Objekts nicht vom abhängigen Objekt genutzt werden kann, muss die Referenz einen Schnittstellentyp haben. Dieser Typ muss sich in der Typhierarchie des benötigten Objekts befinden.

Diese Entkopplung der Klassen und die Injektion von Objekten ist als Dependency Injection-Pattern bekannt und wird von Fowler in [Fow08a] ausführlich beschrieben. Dieses Entwurfsmuster bezieht sich auf zwei Klassen, wobei die abhängige Klasse die Rolle eines Client und die benötigte Klasse die eines Service einnimmt. Die Relation, die zwischen diesen beiden herrscht, wird von diesem Entwurfsmuster als Abhängigkeit der Client- zur Service-Klasse verstanden und beschreibt, auf welche Weise das notwendige Service-Objekt erzeugt und injiziert werden kann.

In der Realität stellt es sich meistens jedoch als schwierig heraus, Programmcode immer nach den Vorgaben und Richtlinien des Software Engineerings zu erstellen, indem u.a. Entwurfsmuster von Beginn an in das Strukturierungsschema integriert werden. Allzu oft sind die Rahmenbedingungen eines Softwareprojekts so eng gesteckt, dass den Entwicklern kein Spielraum bleibt während der Realisierung eines Softwaresystems Qualitätsmaßnahmen zu berücksichtigen. Allgegenwärtig ist dabei das durch Sneed bekannte Teufelsquadrat[3], welches die Tatsache unterstreicht, dass bei einem gegebenen Funktionsumfang und engen Budget- und Zeitvorgaben zwangsläufig an Qualität eingespart werden muss. Die Mängel der unsauberen Umsetzung des interfaceorientierten Ansatzes drücken sich vor allem in der engen Kopplung zwischen Modulen aus.

Um der weiteren Degeneration von mangelhaftem Programmcode entgegen zu wirken, hat sich in der objektorientierten Programmierung eine Methodik etabliert, die sich gezielt mit dessen Umstrukturierung befasst. Eingeprägt hat sich hierfür der englische Begriff Refactoring[4], der von Fowler folgendermaßen definiert wird:

> "Refactoring is the process of changing a software system in such a way that it does not alter the external behaviour of the code yet improves its internal structure."
>
> M. Fowler

Das Ziel des Refactorings ist die Verbesserung der Struktur von Programmcode, ohne das von außen beobachtete Verhalten zu verändern. Bei den agilen Vorgehensmodellen

[3] Siehe [Sne87].
[4] Siehe [Fow00].

hat das Refactoring bereits einen festen Platz als eigenständige Aktivität eingenommen. Extreme Programming[5], als prominentester Vertreter der modernen Vorgehensmodellen, sieht beispielsweise Zyklen vor, die aus Testen, Implementierung, Refactoring und Integration bestehen.

1.2 Problemstellung

Die Auswirkung einer Einführung des Dependency Injection-Pattern läßt sich bereits an einem einfachen Beispiel veranschaulichen.

```
1  public class A {
2    private B b;
3
4    public A() {
5      b = new B();
6    }
7  }
8
9  public class B {
10 }
```

Die Klasse `A` besitzt hierbei eine starke Kopplung zur Klasse `B`, da dessen Typ in der Deklaration des Objektattributs und zur Erzeugung des Objekts genutzt wird.

Die Abhängigkeit der Klasse `A` zur Klasse `B` wird gelöst, indem das Objekt der Klasse `B` als Konstruktorparameter übergeben wird und das entsprechende Objektattribut mit einem Schnittstellentyp deklariert wird.

```
1  public class A {
2    private IB b;
3
4    public A(IB b) {
5      this.b = b;
6    }
7  }
8
9  public interface IB {
10 }
11
```

[5] Siehe [Bec03].

```
12 public class B implements IB {
13 }
```

Der Konstruktoraufruf `new B()` wurde vollständig entfernt und findet im gesamten Programmcode keine Entsprechung mehr. Erst durch eine Konfiguration, die die Erzeugung und Injektion von Service-Objekten beschreibt, ist die Einführung des Dependency Injection-Patterns vollständig. Diese Konfiguration ist jedoch nicht generisch, sondern ist durch die unterschiedlichen Ansätze der Dependency Injection-Frameworks immer frameworkspezifisch.

1.3 Aufgabenstellung und Zielsetzung

Die nachträgliche Einführung des Dependency Injection-Patterns in ein bestehendes Softwaresystem ist zwar möglich, stellt jedoch eine fehleranfällige und zeitaufwendige Tätigkeit dar. Da hierbei neue Typen erstellt und vorhandene modifiziert werden müssen, sind die Änderungen gerade bei gewachsenen Strukturen nicht immer überschaubar. Es besteht die Gefahr, dass das Softwaresystem durch die Änderungen in einen fehlerhaften Zustand überführt wird und nicht mehr von einem Compiler übersetzbar ist. Die größere Gefahr geht jedoch von der Einführung logischer Fehler aus. Diese Fehler sind für einen Compiler nicht erkennbar und werden zumeist erst zur Laufzeit aufgedeckt, wenn diese zu einem unerwarteten Verhalten des Softwaresystems führen. Ein typisches Beispiel ist das Auftreten von `NullPointerExceptions`.

Die Aufgabe dieser Abschlussarbeit ist es ein Refactoring zu beschreiben, durch das das Dependency Injection-Pattern in vorhandene Programmstrukturen eingeführt werden kann. Hierfür muss einerseits die notwendige Änderung des Programmcodes durchgeführt werden, andererseits die notwendige Konfiguration erzeugt werden. Auf Basis des konzeptionellen Entwurfs soll deshalb ebenfalls ein Refaktorisierungswerkzeug für die Entwicklungsumgebung eclipse[6] implementiert werden. Das Ziel ist es den Benutzer mit dem Inject Dependency Plug-In in die Lage zu versetzen, die notwendigen Transformationen automatisiert und einfach durchzuführen.

Das Dependency Injection-Pattern lässt die Frage nach der Beschaffenheit des Assemblers offen und beschreibt lediglich dessen rudimentäre Funktion. Durch eine integrierte Generator-Schnittstelle sollen andere Plug-Ins in der Lage sein, frameworkspezifische Konfigurationen für die verfügbaren Dependency Injection-Frameworks erzeugen zu können.

Das Inject Dependency Plug-In selbst bietet zur Erzeugung der Konfiguration die Möglichkeit eine einfache Assemblerklasse zu erzeugen. Das ebenfalls im Rahmen dieser Ab-

[6] Siehe http://www.eclipse.org/.

schlussarbeit realisierte Inject Dependency Guice Plug-In nutzt die erwähnte Generator-Schnittstelle, um für das Dependency Injection-Framework Guice[7] Konfigurationscode erzeugen zu können.

1.4 Aufbau der Arbeit

Dieses erste Kapitel dient als Einführung und beschreibt den Kontext, die Problemstellung und das Ziel dieser Abschlussarbeit.

Im zweiten Kapitel werden die notwendigen Grundlagen für das zu realisierende Plug-In gelegt. Hierfür wird das das Prinzip das Dependency Injection-Pattern, das Refactoring von Programmcode und das Infer Type-Refactoring vorgestellt.

In Kapitel 3 wird die Funktionsweise des Inject Dependency-Refactoring als Transformationsprozess beschrieben. In diesem Zuge wird ebenfalls aufgezeigt, welche Vorbedingungen für dieses Refactoring gelten.

Im anschließenden Kapitel 4 werden die Grenzen der Verwendbarkeit des Refactoring aufgezeigt.

Kapitel 5 beschreibt die Anforderungen an das Plug-In, dessen Entwurf und relevante Auszüge der Implementierung.

Die Beschreibung des ebenfalls realisierten Konfigurationsgenerator für Guice wird in Kapitel 6 beschrieben.

In Kapitel 7 wird das Inject Dependency-Refactoring an einem Fallbeispiel angewendet. Hierbei werden die Funktionsweise und die Bedienung des Plug-Ins demonstriert.

Das achte Kapitel wertet die Ergebnisse des Fallbeispiels aus und formuliert die Schlussfolgerung dieser Abschlussarbeit.

Da das Inject Dependency-Refactoring auf einem anderen Refactoring aufsetzt und zukünftig selbst von anderen Plug-Ins integriert werden soll, wird in Anhang A die Integration von Refactorings diskutiert. Zu diesem Thema wurde im Rahmen dieser Abschlussarbeit eine Schnittstelle ausgearbeitet, die solch eine Integration ermöglicht.

Vor allem durch die Komplexität der Analyse und Transformation von Programmcode ist das Testen relevanter Teile eines Refactoring Plug-Ins notwendig. eclipse bietet hierfür standardmäßig bereits die Möglichkeit Plug-Ins per JUnit[8] im PDE[9] zu testen. In Anhang B wird diese Thematik aufgegriffen, indem gezeigt wird, dass auch für PDE-JUnit-Tests

[7] Siehe http://code.google.com/p/google-guice/.
[8] Siehe http://www.junit.org/.
[9] Plug-in Development Environment.

Mock-Objekte durch JMock[10] eingesetzt werden können.

[10]Siehe http://www.jmock.org/.

2 Grundlagen

Die Beschreibung des Refactoring und die Realisierung des Refaktorisierungswerkzeugs stützen sich auf existierende Konzepte und Technologien, die in diesem Kapitel vorgestellt werden.

2.1 Das Dependency Injection-Pattern

Das Dependency Injection-Pattern gehört zu den Erzeugungsmustern[11] und wird detailliert in [Fow08a] beschrieben. Es ist ein Entwurfsmuster, bei dem die Zuständigkeit für die Objekterzeugung und -verknüpfung aus den Klassen herausgelöst und in eine Art Factory[12] transferiert wird. Durch diese Maßnahme wird der Kontrollfluß der Objekterzeugung und -verknüpfung invertiert und kann aus diesem Grund als eine Variante des Inversion of Control verstanden werden.

2.1.1 Inversion of Control

Das Prinzip des Inversion of Control beruht auf der Änderung des Kontrollflußes innerhalb eines Softwaresystems. Dieses Prinzip beschreibt im wesentlichen die Arbeitsweise eines Frameworks und grenzt dieses gleichzeitig aus einer konzeptionellen Sicht von Funktionsbibliotheken ab.

Beim Einsatz einer Bibliothek obliegt es der Applikation Bibliotheksobjekte zu erzeugen und an diese Nachrichten zu senden. Durch diese klassische Verfahrensweise wird die Applikation um die Funktionen der Bibliothek erweitert. Der Kontrollfluß wird dabei immer von der Applikation gesteuert.

Umgekehrt verhält es sich, wenn für die Realisierung einer Applikation ein Framework eingesetzt wird. Der Kontrollfluß wird hierbei vom Framework gesteuert und die Applikation wird von diesem integriert. Diese Integration wird in der Regel durch die Implementierung vordefinierter Interfaces oder durch den Einsatz von Reflection realisiert, ist jedoch immer frameworkspezifisch.

[11] Siehe [GHJV95].
[12] Ein Beispiel einer Factory wird in [GHJV95] als Abstract Factory beschrieben.

2.1.2 Die Charakteristik dieses Entwurfsmusters

Die Idee hinter Dependency Injection ist, dass eine Klasse seine Abhängigkeiten bekannt gibt, indem die Injektion von Service-Objekten durch Konstruktoren oder Methoden ermöglicht wird. Ein Assembler ist zusammen mit einer entsprechenden Konfiguration in der Lage die Abhängigkeiten zwischen den Klassen zu erkennen, die relevanten Service-Objekte zu erzeugen und den Client-Objekten zu injizieren. Hierbei wird jedoch keine Vorgabe gemacht, wie der Assembler realisiert sein muss bzw. auf welche Weise die Konfiguration formuliert werden soll.

Um die Entkopplung von Klassen zu ermöglichen, folgt Dependency Injection dem Prinzip der „Trennung von Schnittstelle und Implementierung“. Hierfür ist es notwendig, dass die Service-Klasse ein idealerweise clientspezifisches Abstraction-Interface implementiert. Durch diese Abstraktion vom Service-Typ kann die Injektion mit dem Typ des Interface erfolgen. Der Client ist hierdurch in der Lage Nachrichten an das Service-Objekt durch das Service-Interface zu senden, ohne den konkreten Typ der Service-Klasse kennen zu müssen.

2.1.3 Varianten der Injektion

Das Dependency Injection-Pattern sieht drei Varianten für die Injektion von Service-Objekten vor. Zur Veranschaulichung wird von der folgenden Klassenkonstellation ausgegangen:

```
1 public class A {
2   private B b = new B();
3 }
4
5 public class B {
6 }
```

Ein Objekt der Klasse `A` erzeugt im Konstruktor ein Objekt der Klasse `B` und initialisiert das entsprechende Objektattribut `b`. Die Klasse `A` hat somit eine Abhängigkeit zur Klasse `B`.

Interface Injection

Diese Variante der Injektion wird auch „Typ 1 IoC“ genannt. Hierbei implementiert eine Client-Klasse ein dediziertes Interface, das ausschließlich dessen Abhängigkeiten kommuniziert. Jede Injektionsmöglichkeit eines Service-Objekts wird im Injection-Interface als eine Setter-Methode deklariert, die ebenfalls von der Client-Klasse implementiert wird.

```
1  public interface IA {
2    public void injectB(IB b);
3  }
4
5  public class A implements IA {
6    private IB b;
7
8    public void injectB(IB b) {
9      this.b = b;
10   }
11 }
12
13 public interface IB {
14 }
15
16 public class B implements IB {
17 }
```

Die Klasse `A` deklariert die Abhängigkeit zu einem Service-Objekt vom Typ `IB` im neu eingeführten Injection-Interface `IA`. Die Klasse `B` implementiert das neu eingeführte Abstraction-Interface `IB` und ist somit ein geeigneter Kandidat[13], dass von dieser durch den Assembler ein Objekt erzeugt und injiziert werden kann.

Neben dem Injection-Interface sieht die Interface Injection-Variante eine weitere Schnittstelle vor, durch die die eigentliche Injektion erfolgt. Dieses Injector-Interface[14] trägt beispielsweise den generischen Namen `Injector` und deklariert eine einzige Methode mit der Signatur `public void inject(object target)`. Diese Schnittstelle kann für die Injektion eines Service-Objekts durch eine dedizierte Klasse oder durch die Service-Klasse selbst implementiert werden.

Soll das Injector-Interface durch eine dedizierte Klasse implementiert werden, so könnte beim Einsatz einer Assemblerklasse eine interne Klasse genutzt werden:

```
1  public interface Injector {
2    public void inject(Object target);
3  }
4
```

[13] Erst durch eine entsprechende Konfiguration wird definiert, dass tatsächlich ein Objekt der Klasse `B` erzeugt und injiziert werden muss.

[14] Fowler beschreibt in [Fow00a] den Einsatz dieser Schnittstelle. Die Notwendigkeit für diese Schnittstelle liegt in der Art der Implementierung eines Dependency Injection-Frameworks, das die Interface Injection-Variante unterstützt. Eine Erklärung folgt im anschließenden Absatz 2.1.4.

```
5  public class Assembler {
6    //...
7
8    private static class InjectorImpl implements Injector {
9      public void inject(Object target) {
10       IA a = (IA) target;
11       a.injectB(...);
12     }
13   }
14
15 }
```

Die Implementierung der `inject`-Methode muss das Objekt zuerst auf den Typ des Injection-Interface casten, bevor dann im zweiten Schritt das Service-Objekt injiziert werden kann.

Wird das Injector-Interface hingegen von der Service-Klasse direkt implementiert, so muss von dieser zum Injector-Typ eine `implements`-Beziehung hergestellt und die `inject`-Methode implementiert werden:

```
1 public class B implements IB, Injector {
2   public void inject(object target) {
3     IA a = (IA) target;
4     a.injectB(this);
5   }
6 }
```

Diese Variante hat jedoch einen entscheidenden Nachteil: Die Service-Klasse muss eine Schnittstelle implementieren, die nichts mit deren eigentlichen Funktion zu tun hat. Aus Sicht des Single Responsibility Principle[15] sollte die Service-Klasse nicht gleichzeitig ihre funktionale Aufgabe und die technische Realisierung der Injektion eines Objekts implementieren. Für ein Service-Klasse sollte die Einführung des Dependency Injection-Patterns möglichst nicht invasiv sein. Zudem wird der Service-Klasse eine Abhängigkeit zum Injection-Interface der Client-Klasse eingeführt. Diese neue Abhängigkeit würde somit zu einer Verschlechterung der Gesamtstruktur führen.

Unabhängig davon, welche der beiden Möglichkeiten zur Implementierung des Injector-Interface genutzt wird, ist deren Abhängigkeit zu einem Dependency Injection-Framework. Während das Injection-Interface frameworkunabhängig ist, bilden das

[15] Siehe [Mar02].

Injector-Interface und die dedizierte Klasse bereits einen Teil der Konfiguration. Somit ist es die Aufgabe eines Generator Plug-Ins diese beiden Typen zu erstellen, falls durch das zugehörige Dependency Injection-Framework die Interface Injection-Variante unterstützt wird.

Setter Injection

Diese Variante der Injektion wird auch „Typ 2 IoC“ genannt. Wie bei der Interface Injection-Variante implementiert die Client-Klasse für jede Abhängigkeit eine Setter-Methode, kommuniziert diese jedoch nicht durch ein zusätzliches Injection-Interface.

```
1  public class A {
2    private IB b;
3
4    public void setB(IB b) {
5      this.b = b;
6    }
7  }
8
9  public interface IB {
10 }
11
12 public class B implements IB {
13 }
```

Die Klasse `A` deklariert eine Abhängigkeit zu einem Service-Objekt vom Typ `IB` durch die Setter-Methode `setB`.

Constructor Injection

Diese Variante der Injektion wird auch „Typ 3 IoC“ genannt. Im Gegensatz zu den ersten beiden Varianten werden Service-Objekte bei der Constructor Injection-Variante nicht durch Methoden injiziert, sondern werden dem Client-Objekt bereits während der Erzeugung durch den Konstruktor übergeben.

```
1  public class A {
2    private IB b;
3
4    public A(IB b) {
5      this.b = b;
6    }
```

```
7  }
8
9  public interface IB {
10 }
11
12 public class B implements IB {
13 }
```

Die Klasse `A` deklariert einen Parameter im Konstruktor, durch den das notwendige Service-Objekt injiziert werden kann.

2.1.4 Vergleich der Varianten

Die beiden Varianten Interface Injection und Setter Injection sind prinzipiell identisch, wenn man das Injection-Interface für einen Moment ignoriert: Bei beiden Varianten werden die Abhängigkeiten durch eine Setter-Methode injiziert. Der tatsächliche Unterschied liegt in der Funktionsweise des Assemblers, genauer gesagt in der Art, wie der Assembler Service-Objekte in ein Client-Objekt injiziert. Bei der Interface Injection wird dies durch das Injection-Interface ermöglicht. Mit Hilfe des `instanceof`-Operators und dem Casting des Client-Objekts auf den Typ des Injection-Interface werden die Service-Objekte durch dieses Interface injiziert. Für die Setter Injection-Variante muss ein Assembler auf Reflection zurückgreifen um die Setter-Methoden zu erreichen. Auch die Injektion der Service-Objekte wird anschließend mit Hilfe von Reflection realisiert. Neben der Setter Injection-Variante wird auch für die Constructor Injection-Variante vom Assembler auf Reflection zurückgegriffen. Durch die technische Nähe der Setter Injection und Constructor Injection-Varianten ist es naheliegend, dass verfügbare Dependency Injection-Frameworks diese beiden Varianten immer zusammen zur Verfügung stellen. Nennenswert für die Implementierung der Interface Injection-Variante ist bisher nur das Apache Avalon-Projekt[16], das mittlerweile jedoch eingestellt ist.

Die „richtige“ Injektion der Abhängigkeiten ist eine philosophische Streitfrage zwischen den Nutzern von Dependency Injection-Frameworks und führt auch in diversen Foren immer wieder zu heftigen Diskussionen. Eine sachlichere Betrachtungsweise ist die Kategorisierung der Abhängigkeiten nach „notwendig“ und „optional“. Notwendige Abhängigkeiten werden dem Konstruktor des Client-Objekts übergeben um sicherzustellen, dass das Objekt von Beginn an vollständig initialisiert ist. Auch wenn ein Assembler sicherstellen kann, dass weitere Abhängigkeiten durch Setter-Methoden injiziert werden, bevor das Client-Objekt zur Verfügung gestellt wird, haben Setter-Methoden einen optionalen Charakter.

[16] Siehe http://avalon.apache.org/closed.html.

Wird neben einem Dependency Injection-Framework ein weiteres Framework eingesetzt, ist es unter Umständen nicht mehr möglich, dass der Erzeugungsvorgang von Client-Objekten durch den Assembler übernommen wird. Werden beispielsweise persistente Objekte, die zur Realisierung eines Rich Domain Models[17] Abhängigkeiten besitzen, durch ein Persistenz-Framework wie Hibernate[18] erzeugt, so können die Abhängigkeiten erst nach deren Erzeugung injiziert werden. Unabhängig davon, ob die Injektion durch dynamische Proxies[19], einen Aspekt[20] oder andere Konzepte angestoßen wird, können die Abhängigkeiten nicht durch die Constructor Injection-Variante injiziert werden. Um diese nachträgliche Injektion von Abhängigkeiten zu unterstützen, stellen die Assembler der Dependency Injection-Frameworks in der Regel eine `configure`-Methode zur Verfügung, durch die ein extern erzeugtes Client-Objekt nachträglich initialisiert werden kann.

Weitere Gedanken zur Unterscheidung zwischen Setter und Constructor Injection diskutiert Folwer in [Fow08a] ausführlich.

2.1.5 Konfiguration des Geflechts der Service-Objekte

Das Objektgeflecht der Service-Klassen ist inhärent unveränderlich und wird durch eine Konfiguration abgebildet, deren Art und Ausprägung abhängig vom eingesetzten Dependency Injection-Framework ist. In der Regel bieten diese Frameworks die Möglickeit die Konfiguration in Form von XML-Konfigurationsdateien oder Konfigurationscode zu formulieren. Fowler diskutiert die Vor- und Nachteile beider Varianten ebenfalls in [Fow08a] ausführlich.

Die Unveränderlichkeit des Objektgeflechts rührt aus dem Kontrollverlust der Client-Klasse für die Initialisierung ihrer Objektattribute, der in Absatz 2.1.1 bereits diskutiert wurde. Zur Veranschaulichung ist der Vergleich des Dependency Injection-Patterns mit dem Service Locator-Pattern[21] hilfreich.

Beim Einsatz des Service Locator-Pattern wird ebenfalls eine vollständige Enkopplung der Client-Klasse von der Service-Klasse erreicht.

```
1 public class A {
2   private IB b = ServiceLocator.getB();
3 }
4
5 public interface IB {
```

[17] Siehe [Fow08b].
[18] Siehe http://www.hibernate.org/.
[19] Siehe http://docs.codehaus.org/display/YAN/Transparent+Dependency+Injection+for+Rich+Domain+Objects.
[20] Siehe http://www.aspectj.org.
[21] Siehe [Fow08a].

```
6  }
7
8  public class B implements IB {
9  }
10
11 public class ServiceLocator {
12   private static ServiceLocator soleInstance;
13   private B b;
14
15   //...
16
17   public static IB getB() {
18     return soleInstance.b;
19   }
20 }
```

Benötigt ein Client-Objekt ein Service-Objekt, so bezieht dieses die notwendige Referenz durch den Service Locator. Durch diese aktive Anforderung der Referenz kann ein vorhandenes Objektattribut in der Client-Klasse prinzipiell mehrfach zugewiesen werden. Da es für das Abstraction-Interface potentiell mehrere Service-Klassen gibt, kann sich die Referenz des Objektattributs während dem Lebenszyklus eines Client-Objekts mehrfach ändern.

```
1  public class A {
2    private IB b;
3
4    public void work1() {
5      b = ServiceLocator.getB1();
6      internalWork();
7    }
8
9    public void work2() {
10     b = ServiceLocator.getB2();
11     internalWork();
12   }
13
14   public void internalWork() {
15     b.serve();
16   }
```

```
17 }
18
19 public interface IB {
20   public void serve();
21 }
22
23 public class B1 implements IB {
24   public void serve() {
25   }
26 }
27
28 public class B2 implements IB {
29   public void serve() {
30   }
31 }
32
33 public class ServiceLocator {
34   private static ServiceLocator soleInstance;
35   private B1 b1;
36   private B2 b2;
37
38   //...
39
40   public static IB getB1() {
41     return soleInstance.b1;
42   }
43
44   public static IB getB2() {
45     return soleInstance.b1;
46   }
47 }
```

Hierbei kann sich die Referenz des Objektattributs in Abhängigkeit zur aufgerufenen Methode des Client-Objekts mehrfach ändern.

Dieses dynamische Verhalten kann durch die Konfiguration des Dependency Injection-Patterns nicht abgebebildet werden. Das Dependency Injection-Pattern zielt auf die einmalige und endgültige Initialisierung des Objektattributs eines Client-Objekts ab. Bei der Constructor Injection-Variante kann das entsprechende Objektattribut als `final` deklarariert werden, da eine weitere Zuweisung nicht vorgesehen bzw. möglich ist.

2.1.6 Bootstrapping

Wird für die Erzeugung und Injektion von Service-Objekten ein Dependency Injection-Framework eingesetzt, so ist ein sogenanntes Bootstrapping notwendig. Hierbei wird der frameworkspezifische Dependency Injection-Assembler zusammen mit der Angabe der Konfiguration erzeugt. Für das Guice Framework könnte das Bootstrapping folgendermaßen aussehen:

```
1 public class Program {
2
3   public static void main(String args[]) {
4     Injector assembler =
5         Guice.createInjector(new Config1(), new Config2());
6   }
7
8 }
```

Der Assembler wird durch die Methode `createInjector` erzeugt. Durch die Angabe zweier Objekte, die die Konfiguration durch frameworkspezifischen Konfigurationscode formulieren, erstellt der Assembler das Objektgeflecht der Service-Objekte. Von diesem Assembler können registrierte Service-Objekte bezogen und vollständig initialisierte Client-Objekte erzeugt werden.

Das Bootstrapping muss sich nicht zwingend in der `main`-Methode befinden. So gibt es für Guice beispielsweise die Möglichkeit in das Struts2-Framework[22] eingebunden zu werden. In diesem Fall wird das Bootstrapping und das damit verbundene Erzeugen des Assemblers direkt von Stuts2 übernommen. Durch die Integration von Guice in Struts2 können hierbei Service-Objekte in Actions injiziert werden.

Desweiteren gibt es Umstände, bei denen weder ein Assembler, noch eine Konfiguration für die Injektion von Service-Objekten notwendig ist. In [Bau08] wird ein Refactoring beschrieben, mit dem im Rahmen von Unit-Tests[23] die für eine Client-Klasse notwendigen Service-Objekte durch Mock-Objekte automatisiert ersetzt werden können. In diesem Fall ist die Erzeugung der Mock-/Service-Objekte und die Injizierung in das zu testende Client-Objekt die Aufgabe des Testcodes.

```
1 public class BankTest {
2   .....
3   static final boolean collaboratorsAreMocked =
```

[22] Siehe http://struts.apache.org/2.x/.
[23] Siehe http://www.junit.org/

```
4       BankTest.class.isAnnotationPresent(DoMockCollaborators.class);
5     private IDatenbank mock_kontenDb_testKontoAusgabe;
6     private Ausgabe mock_outputStream_testKontoAusgabe;
7     .....
8
9     @Test
10    public void testKontoAusgabe() throws Throwable {
11      // Mock-objects for collaborator simulation
12      mock_kontenDb_testKontoAusgabe = context.mock(IDatenbank.class);
13      mock_outputStream_testKontoAusgabe = context.mock(Ausgabe.class);
14      // TODO Specify the behavior of the mock-objects
15      .....
16      // Inject mock-objects or real collaborators into CUT-instance
17      if (collaboratorsAreMocked)
18        dieBank = new Bank(mock_kontenDb_testKontoAusgabe,
19          mock_outputStream_testKontoAusgabe);
20      else
21        dieBank = new Bank(kontenDb, outputStream);
22      .....
23    }
24    .....
25  }
```

In diesem aus [Bau08] entnommenen Codebeispiel ist es die Aufgabe der Testmethode die Mock-Objekte zu erzeugen und dem zu testenden `dieBank`-Objekt per Constructor Injection zu injizieren.

2.2 Refactoring

Unter Refactoring versteht man die Umstrukturierung von Code mit dem Ziel eine qualitative Verbesserung herbeizuführen. Fowler fordert in [Fow00] die Einhaltung der Verhaltenskonservation, die in der Realität meist nur schwierig einzuhalten ist. Es ist nicht ausreichend, dass nach der Umstrukturierung der Code vom Compiler fehlerfrei übersetzbar ist. Die Einführung logischer Fehler treten erst während Ausführung auf und machen sich beispielsweise durch `NullPointerExceptions` bemerkbar.

2.2.1 Bedingte Transformation

Um generell Fehler zu vermeiden führt Opdyke in [Opd92] das Konzept der bedingten Transformation ein. Die bedingte Transformation basiert auf sieben Programminvarianten, die durch eine Transformation nicht verletzt werden dürfen:

- Unique Superclass
- Distinct Class Names
- Distinct Member Names
- Inherited Member Variables Not Redefined
- Compatible Signatures In Member Function Redefinition
- Type-Safe Assignements
- Semantically Equivalent References And Operations

Opdyke ergänzt Transformationen um Vorbedingungen und definiert diese als eine untrennbare Einheit. Vor der Ausführung einer Transformation sind deren Vorbedingungen zu überprüfen. Erst wenn alle Vorbedingungen erfüllt sind darf die Transformation ausgeführt und Programmcode manipuliert werden. Zu jeder Transformation ist somit Analysecode notwendig, der auf Verletzung der Invarianten überprüft.

2.2.2 Kategorisierung von Refactorings

Vor allem bei großen Refactorings ist die Formulierung und Überprüfung von Vorbedingungen notwendig. Bei kleinen Refactorings sind die Änderungen des Codes vergleichsweise minimal und beziehen sich des öfteren nur auf eine einzige Quellcode-Datei. Fowler beschreibt in [Fow00] einen Katalog von kleinen Refactorings und beschreibt ebenfalls die Durchführung von einigen großen Refactorings.

Während die Forderung nach Verhaltenskonservation in der Regel von den kleinen Refactorings erfüllt werden kann, ist dies bei den großen Refactorings meist nicht möglich. Große Refactorings führen zu Änderungen an vielen Dateien und bestehen aus mehr als nur der Abarbeitung von mehreren kleinen Refactorings. Um nach der Durchführung eines großen Refactorings wieder ein ausführbares Softwaresystem zu erhalten ist oft zusätzliche Implementierungsarbeit notwendig.

Die Einführung eines Entwurfsmusters[24] wird in der Literatur zu den großen Refactorings zugeordnet und wird beispielsweise in [Ker04] ausführlich beschrieben.

[24] Siehe [GHJV95].

2.2.3 Beschreibungssprache

In den Werken von Fowler und Kerievsky werden die einzelnen Transformationsschritte ausschließlich informell beschrieben. Ó Cinnéides Dissertation[25] baut auf der Arbeit von Opdyke auf führt die Idee der bedingten Transformation weiter. Er identifiziert für die Einführung der klassischen Entwurfsmuster von Gamma et al. generische und atomare „primitve Refactorings“ und definiert desweiteren diverse Analyse- und Hilfsfunktionen.

Um ein Refactoring detailierter beschreiben zu können, führt er eine Beschreibungssprache ein, die in der Lage ist Transformationen formal zu beschreiben. Diese Beschreibungssprache wird innerhalb dieser Abschlussarbeit für die Beschreibung der Transformationen der drei Dependency Injection-Varianten eingesetzt.

2.3 Das Infer Type-Refactoring

Infer Type ist in der Lage zu einem gegebenen Deklarationstyp einen minimalen Typ abzuleiten. Dieser minimale Typ besitzt ausschließlich diejenigen Methoden, die für die gegebene Deklaration notwendig sind. Die konzeptionelle Basis für dieses Refactoring stellt die Typinferenz dar, die in [SMM05] ausführlich beschrieben wird.

Das Infer Type-Refactoring wurde am Lehrgebiet Programmiersysteme der FernUniversität in Hagen im Rahmen einer Abschlußarbeit als Plug-In für eclipse realisiert und wird ausführlich in [Keg07] beschrieben. Mit diesem Refactoring kann beispielsweise für die Deklaration eines Objektattributs ein minimaler Typ abgeleitet und eingeführt werden. Falls der abgeleitete Typ bereits existiert, kann dieser direkt für die Deklaration eingesetzt werden. Im anderen Fall ist das Refactoring in der Lage den neuen Typ[26] zu erzeugen.

2.3.1 Abstraktion des Referenz-Typs

Soll von Infer Type für einen existierenden Typ ein minimaler Typ berechnet werden, ist die Auswahl eines Deklarationselements zwingend notwendig. Im folgenden Beispiel wird die Deklaration des Objektattributs `b` der Klasse `A` als Ausgangspunkt des Infer Type-Refactoring gewählt:

```
1 public class A {
2   private B b;
```

[25] Siehe [Cin00].

[26] In der Regel erzeugt das Infer Type-Refactoring einen Schnittstellentyp. Müsste die Schnittstelle in der Typhierarchie eine Klasse spezialisieren, so wird anstatt einer Schnittstelle eine abstrakte Klasse erzeugt.

```
3
4     public A() {
5       b = new B();
6     }
7
8     public void work() {
9       b.serve1();
10    }
11  }
12
13  public class B {
14    public void serve1() {
15    }
16
17    public void serve2() {
18    }
19  }
```

Durch die Typinferenz wird ein neuer Schnittstellentyp `IB` erzeugt, der ausschließlich die Methode `serve1` deklariert. Da an die Methode `serve2` während der Ausführung des Programms niemals eine Nachricht gesendet wird, ist diese nicht im neu erzeugten Typ enthalten. Neben der Erzeugung des Schnittstellentyps führt das Infer Type-Refactoring den neuen Typ in die Typhierarchie der Klasse `B` ein und passt die Deklaration des Objektattributs `b` an:

```
1   public class A {
2     private IB b;
3
4     public A() {
5       b = new B();
6     }
7
8     public void work() {
9       b.serve1();
10    }
11  }
12
13  public interface IB {
14    public void serve1();
15  }
```

```
16
17 public class B implements IB {
18   public void serve1() {
19   }
20
21   public void serve2() {
22   }
23 }
```

2.3.2 Entkopplung von Klassen

Durch das Infer Type-Refactoring wird das ausgewählte Deklarationselement angepasst, indem der konkrete Klassentyp durch den Typ der Schnittstelle ersetzt wird. Dies führt bereits zu einer gewissen Entkopplung der Klasse `A` von der Klasse `B`. Mit Ausnahme des Konstruktoraufrufs durch den `new`-Operator wird der Typ der Klasse `B` an keiner Stelle mehr innerhalb der Klasse `A` benötigt.

Erst durch einen weiteren Refactoringschritt, in dem der Konstruktoraufruf eliminiert wird, kann die Klasse `A` vollständig von der Klasse `B` entkoppelt werden. Da hierdurch die Objekterzeugung nicht mehr von der Klasse `A` übernommen werden kann, ist eine weitere Komponente notwendig, die ein Objekt der Klasse `B` erzeugen und zur Verfügung stellen kann.

An dieser Stelle schließt das Inject Dependency-Refactoring an, das in der Lage ist die Objekterzeugung aus einem Client herauszulösen, die Konfiguration für einen Assembler zu generieren und die Abhängigkeit durch eine der drei vorgestellten Varianten zu injizieren.

3 Bechreibung des Inject Dependency-Refactoring

Das Dependency Injection-Pattern ermöglicht die vollständige Entkopplung eines Clients von einem Service. Diese Entkopplung wird durch drei Maßnahmen herbeigeführt, die charakteristisch für dieses Entwurfsmuster sind:

- Abstraktion der Service-Klasse durch die Einführung eines Schnittstellentyps.
- Verlagerung der Erzeugung des Service-Objekts in einen Assembler.
- Injektion des Service-Objekts in das Client-Objekt durch einen Assembler.

Eine enge Kopplung von Klassen ist zwar über Packages hinweg unerwünscht, stellt aber beispielsweise in packageinternen Strukturen kein Problem dar. Die möglichst enge Kopplung von Klassen innerhalb von Packages wird dabei hohe Kohäsion genannt und ist eines der angestrebten Prinzipen des Software Engineering. Die Entscheidung, an welchen Stellen im Programmcode die Anwendung des Dependency Injection-Refactoring angebracht ist, muss daher von einem Softwareentwickler getroffen werden.

Alternativ zum Dependency Injection-Pattern kann ein Client ein anderes Erzeugungsmuster (z.B. das Abstract Factory-Pattern[27]) als Vermittler benutzen, um ein Service-Objekt zu beziehen. Ein Vermittler stellt wie ein Assembler dem Client eine Abhängigkeit zur Verfügung, jedoch mit dem Unterschied, dass der Client eine Abhängigkeit von diesem aktiv anfordert. Für das Dependency Injection-Pattern sind ausschließlich Abhängigkeiten relevant, für die der Client die Service-Objekte selbst erzeugt. Nur für diese Abhängigkeiten kann die Objekterzeugung in einem Assembler verlagert werden.

3.1 Initialisierung des Objektattributs im Client

Damit ein Client auf einen Service zugreifen kann, muss das entsprechende Objektattribut vor dem ersten Zugriff initialisiert werden. Ohne den Einsatz des Dependency

[27] Siehe [GHJV95].

Injection-Patterns muss sich jedes Objekt selbst um die Beschaffung seiner Abhängigkeiten kümmern: Mit Hilfe des `new`-Operators erzeugt der Client ein Objekt des Service und speichert die Referenz in einem Objektattribut.

Ist das Objektattribut mit dem Modifier `final` deklariert, kann die Initialisierung ausschließlich an zwei Stellen im Code des Clients geschehen: Direkt bei der Deklaration des Objektattributs und innerhalb eines Konstruktors[28]. Fehlt der `final`-Modifier in der Deklaration des Objektattributs ist es ebenfalls möglich, dass dieses innerhalb einer Methode initialisiert wird.

Wird die Objekterzeugung des Service-Objekts durch die Einführung des Dependency Injection-Patterns in einen Assembler verlagert, wird die Referenz dieses Objekts durch einen Konstruktor oder eine Methode[29] an den Client übergeben. Erst nach der Injektion aller Abhängigkeiten durch den Assembler ist das Client-Objekt vollständig initialisiert.

Die Verlagerung der Objekterzeugung des Service ist einhergehend mit der eventuellen Verschiebung des Initialisierungzeitpunkts des zugehörigen Objektattributs. Neben der Initialisierung bei der Deklaration eines Objektattributs und der verzögerten Initialisierung führt das Dependency Injection-Pattern einen weiteren Zeitpunkt ein: Die Initialisierung nach der Erzeugung des Client-Objekts. Dieser Zeitpunkt wird genauer gesagt durch die Dependency Injection-Methoden Setter Injection und Interface Injection eingeführt, da diese erst nach der Erzeugung des Client-Objekts die Abhängigkeiten injizieren können.

Zusammenfassend spielen im Kontext des Dependency Injection-Patterns drei Zeitpunkte eine Rolle:

- Die Deklarationen der Objektattribute.
- Der Ausführung eines Konstruktors.
- Nach der Objekterzeugung.

Die eventuelle Verschiebung des Initialisierungszeitpunkts eines Objektattributs ist direkt abhängig von der Wahl der Dependency Injection-Methode. Durch Constructor Injection wird der Zeitpunkt der Initialisierung in die Ausführung eines Konstruktors verschoben. Interface Injection und Setter Injection sind zeitlich weiter nach hinten versetzt, da hierbei die Initialisierung erst nach der Objekterzeugung des Clients erfolgt. Die Initialisierung während der Deklaration eines Objektattributs kann im Kontext des Dependency Injection-Patterns hingegen nicht realisiert werden.

[28] Diese Art der Initialisierung wird auch *verzögerte Initialisierung* genannt.

[29] Wurde das zugehörige Objektattribut mit dem Modifier `final` deklariert, muss dieser in diesem Fall im Zuge der Codetransformation natürlich entfernt werden.

Auch die Zugriffe auf Objektattribute können in die drei unterschiedlichen Zeitpunkte eingeteilt werden. Für eine genauere Betrachtung sei das folgende Beispiel gegeben, bei dem eine Abhängigkeit bereits zum frühstmöglichen Zeitpunkt initialisiert wird:

```
1  public class Service {
2    public int calculate() {
3      //...
4    }
5  }
6
7  public class Client {
8    private Service service = new Service();
9    private int accessAtFieldDeclaration = service.calculate();
10
11   public Client() {
12     int accessAtConstructorInvocation = service.calculate();
13   }
14
15   public void work() {
16     int accessAfterConstruction = service.calculate();
17   }
18 }
```

Mit der Programmiersprache Java ist es möglich, dass das Objektattribut `accessAtFieldDeclaration` bereits während der Deklaration auf das Objekattribut `service` zugreift. Zeitlich nach hinten versetzt wird der Wert der Variablen `accessAtConstructorInvocation` durch den Zugriff auf den Service während der Ausführung des Konstruktors zugewiesen. Zuletzt wird für die Wertzuweisung der Variablen `accessAfterConstruction` auf den Service zugegriffen. Dies kann jedoch erst nach der Erzeugung des Client-Objekts geschehen.

Während der Initialisierungszeitpunkt eines Objektattributs durch die Einführung des Dependency Injection-Patterns zeitlich nach hinten verschoben werden kann, bleiben die Zeitpunkte der Zugriffe auf diese hingegen unverändert. Wie die folgende Matrix zeigt, können sich Zugriffszeitpunkte und die Einführung einer Dependency Injection-Methode gegenseitig ausschließen:

	Constructor Injection	**Setter Injection**	**Interface Injection**
Zugriff während Deklaration der Objektattribute	Nicht möglich	Nicht möglich	Nicht möglich
Zugriff während der Ausführung eines Konstruktors	Möglich	Nicht möglich	Nicht möglich
Zugriff nach der Objekterzeugung	Möglich	Möglich	Möglich

Tabelle 3.1: Gegenseitiger Ausschluss der Zeitpunkte „Initialisierung eines Objektattributs“ und „Zugriff auf ein Objektattribut“

Keine der Dependency Injection-Methoden kann eingesetzt werden, wenn auf eine Abhängigkeit bereits während der Deklarationen anderer Objektattribute zugegriffen wird. Wird auf eine Abhängigkeit innerhalb eines Konstruktors zugegriffen, kommt nur die Constructor Injection-Methode in Frage. Für den letzten Fall hingegen ist der Einsatz aller drei Dependency Injection-Methoden möglich.

3.2 Abstraktion des Servicetyps

Um einen Client vom Service zu entkoppeln ist es notwendig, dass dem Client die Kenntnis über den Klassentyp des Service vollständig entzogen wird. Im Code des Clients ist für die starke Kopplung zum Service mitunter das für die Abhängigkeit notwendige Objektattribut verantwortlich, da dieses mit dessen Klassentyp des Service deklariert wird. Um die Kopplung an dieser Stelle lösen zu können, muss der Typ der Deklaration angepasst werden, indem dieser durch einen Schnittstellentyp ersetzt wird.

Dieser Schnittstellentyp sollte im Sinne der interfacebasierten Programmierung ein maximal verallgemeinerter Typ sein, d.h. dieser Typ sollte nur die tatsächlich benötigten Methoden enthalten. Da dieser Typ im Regelfall nicht existiert, muss dieser aus dem Benutzungsverhalten abgeleitet werden. Mit der Typinferenz steht ein Mechanismus zur Verfügung, mit dem solch ein Schnittstellentyp abgeleitet und in die Typhierarchie des Service eingeführt werden kann. Im Rahmen der Diplomarbeit von Kegel wurde auf Basis von Infer Type bereits ein praxistaugliches Refactoring realisiert und wird ausführlich in [Keg07] beschrieben.

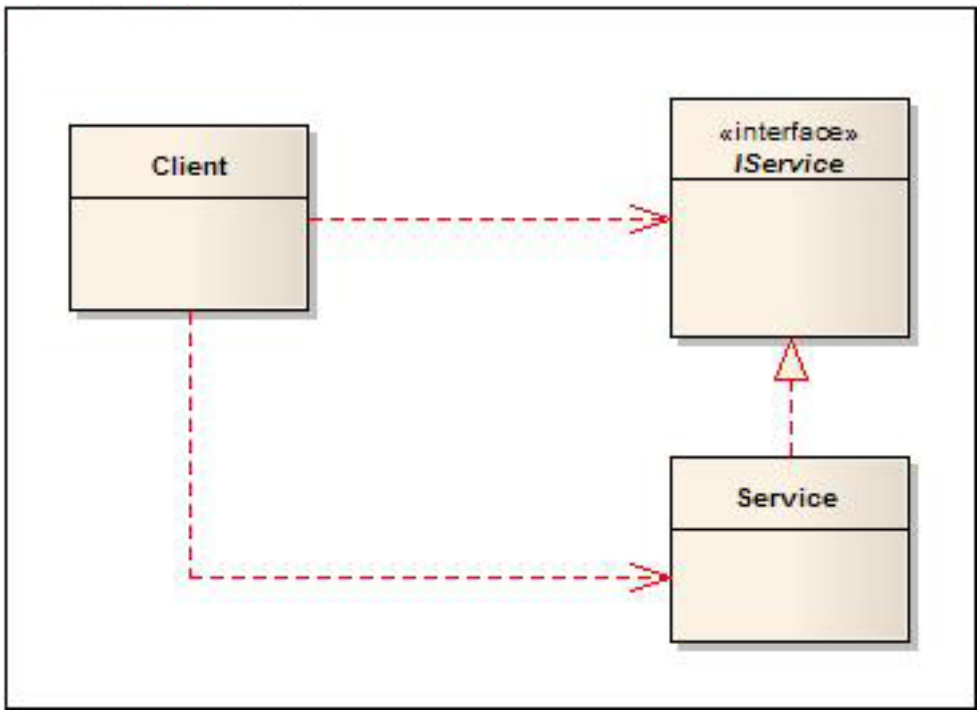

Abbildung 3.1: Abstraktion des Servicetyps

Technisch gesehen wird ein neuer Schnittstellentyp erzeugt, zu dem der Service eine `implements`-Beziehung herstellt. Die Modifikation der Service-Klasse ist somit minimal.

Der erste Schritt der Enkopplung besteht aus der Änderung der Deklaration des Objektattributs. Der Klassentyp des Service wird durch den neuen Schnittstellentyp ersetzt. Unter gewissen Umständen, die ebenfalls in [Keg07] diskutiert werden, kann es sein, dass Infer Type anstatt eines Schnittstellentyps den Typ einer abstrakten Klasse erzeugt. Im Rahmen des Dependency Injection-Patterns ist diese Unterscheidung nicht von Bedeutung, weshalb im weiteren ausschließlich von der Erzeugung eines Schnittstellentyps ausgegangen wird.

Trotz der Erzeugung eines neuen Typs und der Änderung der Deklaration des Objektattributs ist eine vollständige Entkopplung durch diese Maßnahmen noch nicht erreicht. Da der Client weiterhin für die Objekterzeugung des Service zuständig ist, muss dieser dessen Klassentyp weiterhin kennen.

3.3 Verlagerung der Objekterzeugung

Erst durch die Verlagerung der Objekterzeugung des Service kann der Client vollständig vom Service entkoppelt werden. Hierzu sieht das Dependency Injection-Pattern eine Komponente vor, deren Aufgabe die Objekterzeugung und Injektion von Abhängigkeiten ist.

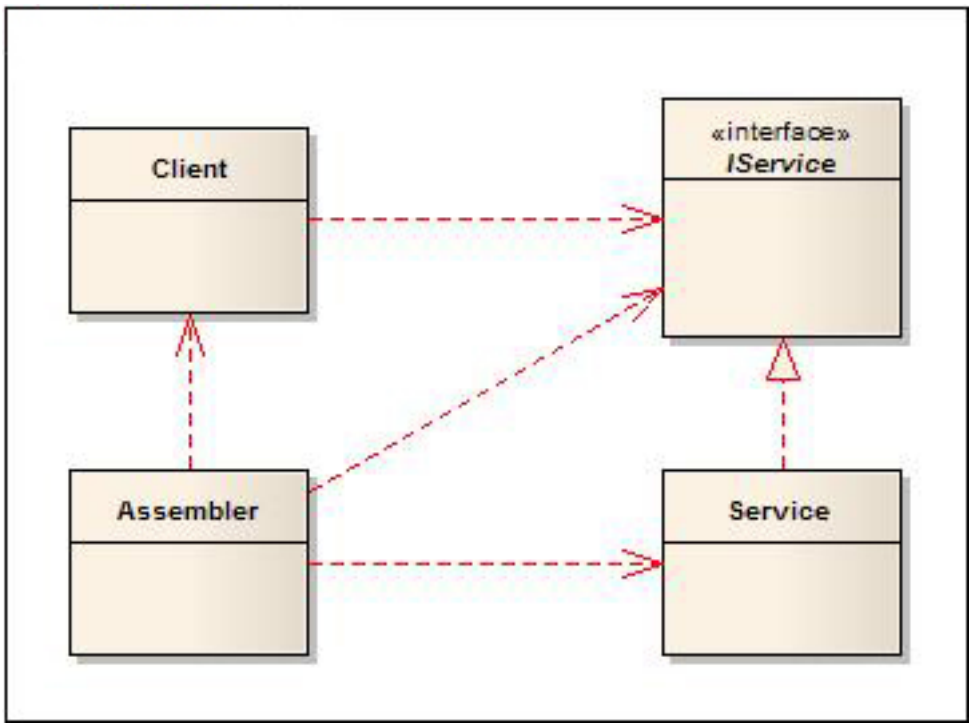

Abbildung 3.2: Verlagerung der Objekterzeugung

Der Assembler übernimmt die Objekterzeugung des Service und injiziert das Objekt dem Client durch eine der drei Dependency Injection-Methoden. Damit der Assembler jedoch ein Objekt der Service-Klasse erzeugen kann, müssen von diesem die folgenden Bedingungen erfüllt sein:

- Da sich der Assembler nicht zwangsläufig im selben Package wie die Service-Klasse befindet, muss diese als `public` deklarariert sein.
- Die Service-Klasse darf nicht abstrakt sein.

Neben diesen Bedingungen, die an die Deklaration der Service-Klasse gerichtet sind, existiert eine weitere Anforderung, die direkt im Bezug zu der Objekterzeugung steht. Wird die Erzeugung des Service-Objekts vom Client in den Assembler verschoben, so liegt es anschließend in dessen Verantwortung den entsprechenden `new`-Operator aufzurufen.

Besitzt die Service-Klasse keinen expliziten Konstruktor, so wird der implizite Standard-Konstruktor für die Objekterzeugung genutzt, der per Definition keine Parameter besitzt. Unabhängig vom Kontext, in dem der `new`-Operator aufgerufen wird, kann somit ein Objekt der Service-Klasse erzeugt werden. Deklariert die Service-Klasse hingegen einen oder mehrere Parameter, müssen die Werte des Konstruktoraufrufs im Client überprüft werden.

Erzeugt der Client ein Objekt eines Service, so spielt der Client in diesem Moment zusätzlich die Rolle eines Assemblers, und nimmt dabei neben der Objekterzeugung eben-

falls die Konfigurationsaufgabe wahr. Konzeptionell werden durch eine Konfiguration ausschließlich Literale - also Werte, die bereits vor der Ausführung feststehen - an den Konstruktor des Service übergeben. Erst bei den Aufrufen der öffentlichen Objektmethoden des Service können clientabhängige Parameter übergeben werden.

Wird durch den Client ein Objekt eines Service erzeugt und dieser übergibt hierbei dem Konstruktor Variablen als Parameter, so kann das Dependency Injection-Pattern nicht eingeführt werden. Die Objekterzeugung kann nicht in einen Assembler verlagert werden, da die Parameterwerte des Konstruktors abhängig vom Client sind.

Damit der Assembler die Objekterzeugung und Konfiguration des Service übernehmen kann, müssen sämtliche Parameterwerte des `new`-Operators durch Literale geben sein.

Neben der Service-Klasse muss auch der für die Objekterzeugung relevante Konstruktor als `public` deklariert sein, damit es dem Assembler ermöglicht wird ein Objekt zu erzeugen.

3.4 Injektion der Abhängigkeit

Die Verschiebung der Objekterzeugung aus dem Client geht einher mit der Einführung eines Mechanismus, durch den die notwendige Referenz durch den Assembler injiziert werden kann.

Für die Constructor Injection-Methode wird einem Konstruktor ein neuer Parameter eingeführt, durch den die Referenz zum Service-Objekt übergeben wird. Gesetz dem Fall, dass der Client noch keinen expliziten Konstruktor besitzt, wird diesem im Rahmen der Einführung der Constructor Injection-Methode ein neuer Konstruktor eingefügt. Für die Deklaration des neuen Parameters wird hierbei der Deklarationstyp des Objektattributs verwendet. Durch die Erweiterung des Konstruktors durch einen neuen Parameter wird dessen Signatur geändert. Existiert ein weiterer Konstruktor mit einer identischen Signatur, so wird eine Kollision verursacht und der Programmcode wäre vom Compiler nicht übersetzbar. Dieser Fall muss durch eine entsprechene Analyse des Programmcodes verhindert werden.

Bei der Setter Injection-Methode wird dem Client eine neue Methode hinzugefügt, durch die der Assembler die Referenz zu einem Service-Objekt übergeben kann. Sowohl der Name der Methode, als auch der Name und Typ des Parameters werden durch die Berücksichtigung der Java Beans-Spezifikation[30] aus dem jeweiligen Objektattribut bestimmt. Für ein Objektattribut `IService service` wird somit eine Setter-Methode mit der Signatur `public void setService(IService service)` eingeführt. Existiert jedoch bereits eine Methode mit einer identischen Signatur wird eine Kollision verursacht

[30]Siehe http://java.sun.com/javase/technologies/desktop/javabeans/index.jsp.

und muss ebenfalls durch Codeanalyse verhindert werden.

Die Interface Injection-Methode entspricht für den Client im wesentlichen der Setter Injection-Methode. Aus einem Objektattribut `IService service` wird ebenfalls eine Setter-Methode eingeführt, die lediglich in der Namensgebung abweicht: `public void injectService(IService service)`. Zusätzlich wird ein Schnittstellentyp erzeugt, durch den der Client explizit seine Abhängigkeiten vermittelt. Diese Schnittstelle enthält die Deklaration der Setter-Methode, die vom Client implementiert wird. Durch die Herstellung einer `implements`-Beziehung vom Client wird diese neue Schnittstelle in dessen Typhierarchie eingeführt.

3.5 Der Client im Kontext des Assemblers

Damit der Assembler die Abhängigkeiten eines Clients injizieren kann, muss dieser ebenfalls die Kontrolle über dessen Erzeugung haben. Zu diesem Zweck sieht ein auf dem Dependency Injection-Framework ein sogenanntes Bootstrapping vor. Nach der Erzeugung des Assemblers können von diesem, neben den registrierten Service-Objekten, vollständig initialisierte Client-Objekte bezogen werden. Um dies zu ermöglichen, müssen vom Client einige Bedingungen erfüllt werden. Die Bedingungen an die Deklaration der Klasse entsprechen zunächst denen der Service-Klasse:

- Da sich der Assembler nicht zwangsläufig im selben Package wie die Client-Klasse befindet, muss diese als `public` deklarariert sein.
- Die Client-Klasse darf nicht abstrakt sein.

Auch die Bedingung, dass bereits deklarierte Konstruktoren als `public` deklariert sein müssen, kann aus der Betrachtung der Service-Konstruktoren übernommen werden. Die weitere Betrachtung der Objekterzeugung des Client geht jedoch einen Schritt weiter als die eines Service.

Besitzt eine Service-Klasse mehrere Konstruktoren, so kann durch die Initialisierung des entsprechenden Objektattributs im Client eindeutig festgestellt werden, welcher für die Erzeugung der Abhängigkeit genutzt wird. Besitzt hingegen ein Client mehrere Konstruktoren, so kann für den Assembler nicht eindeutig bestimmt werden, welcher für die Objekterzeugung genutzt werden soll. Ein Objekt des Clients kann in einer vorhandenen Applikation an diversen Stellen im Code durch unterschiedliche Konstruktoren erzeugt werden. Aus diesem Grund kann die Objekterzeugung eines Clients nur vom Assembler berücksichtigt werden, wenn dessen Klasse maximal einen expliziten Konstruktor deklariert. Damit dieser Konstruktor unabhängig vom Erzeugungskontext ist, muss dieser parameterlos sein.

3.6 Beschreibung der Transformation

Da die Umstrukturierung von Programmcode ein aufwendiger und fehleranfälliger Vorgang ist, stellt die Implementierung eines entsprechenden Refaktorisierungswerkzeugs eine sinnvolle Maßnahme dar. Bisher wurden die Änderungsschritte, die für die Einführung des Dependency Injection-Patterns notwendig sind, ausschließlich in natürlichsprachlichen Anweisungen beschrieben. Bevor das Refactoring jedoch realisiert werden kann ist es im Vorfeld erforderlich, dass die zugrunde liegende Transformation durch eine formale Beschreibung wohl definiert ist.

3.6.1 Steckbrief nach Fowler

Um das allgemeine Grundverständnis für das Thema Refactoring zu fördern, wird zunächst das von Fowler in [Fow00a] verwendete Beschreibungsformat auch für die Beschreibung des Inject Dependency-Refactoring genutzt.

Inject Dependency (Abhängigkeit injizieren)

Eine nutzende Klasse (Client) ist mit einer benutzten Klasse (Service) durch Methoden- und Konstruktoraufrufe eng gekoppelt.

Abstrahiere vom Service durch die Einführung eines Schnittstellentyps und überlasse die Objekterzeugung einem Assembler.

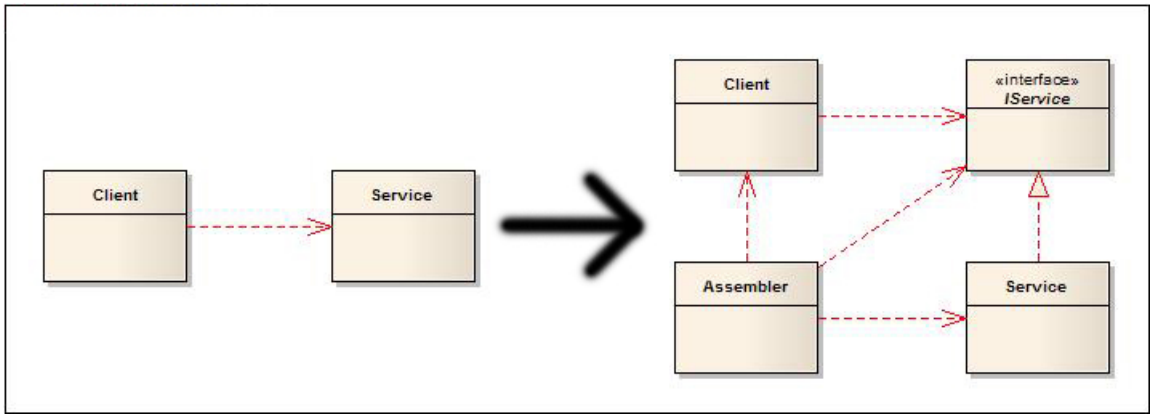

Abbildung 3.3: Einführung des Dependency Injection-Patterns

Motivation

Abhängigkeiten zwischen Klassen in objektorientierten Softwaresystemen können aus vielerlei Gründen bestehen, sind jedoch nicht immer erwünscht. Speziell Methoden- und

Konstruktoraufrufe über Packages hinweg führen zu einer engen Kopplung zwischen Klassen, die es zu lösen gilt.

Um eine vollständige Entkopplung des Clients zu erreichen wird ein Schnittstellentyp erzeugt, der in die Typhierarchie des Service eingeführt wird. Anstatt des konkreten Klassentyps des Service wird innerhalb des Clients der neu eingeführte Typ verwendet.

Die Objekterzeugung des Service wird in einen Assembler verlagert. Die entsprechende Referenz wird dem Client durch diesen Assembler anschließend zur Verfügung gestellt. Zu diesem Zweck gibt es die Varianten Constructor Injection, Setter Injection und Interface Injection, durch die der Assembler dem Client diese Referenz injizieren kann.

Mechanik

Für jede der drei Varianten wird im Folgenden die Mechanik informal beschrieben. Da sich diese Varianten jedoch nur in der Art der Injizierung unterscheiden, kann man die Mechaniken in einen gemeinsamen und einen variantenspezifischen Teil zerlegen.

Gemeinsam

- (1) Es wird ein leeres Interface erzeugt.
- (2) Alle genutzen öffentlichen Objektmethoden des Service werden im neuen Interface deklariert. Durch diese minimale Schnittstelle wird eine maximale Generalisierung erreicht.
- (3) Die Deklaration des Service wird geändert, damit dieser das neue Interface implementiert.
- (4) Die Deklaration des Objektattributs im Client wird geändert, indem der Typ der Service-Klasse durch den neuen Schnittstellentyp ersetzt wird.

Mit diesen Schritten erzielt man bereits eine gewisse Entkopplung, da nun sämtliche Methodenaufrufe an den neuen Schnittstellentyp gerichtet werden. Die Schritte (1) bis (4) führen diesen neuen Schnittstellentyp ein und ändern den Typ der Deklaration des Objektattributs im Client. Dieser Teil der Mechanik wird bereits vollständig vom Infer Type-Refactoring realisiert.

Constructor Injection

- (5) Ist im Client kein expliziter Konstruktor deklariert, wird diesem ein leerer Konstruktor ohne Parameter eingeführt.
- (6) Wird das Objektattribut des Clients direkt bei der Deklaration initialisiert, wird der Aufruf des `new`-Operators in den einzigen Konstruktor des Clients verschoben.

- (7) Der Aufruf des `new`-Operators wird wie beim Introduce Parameter-Refactoring[31] aus dem Konstruktor herausgeschoben. Hierdurch wird dem Konstruktor ein neuer Parameter hinzugefügt, der mit dem Typ der neuen Schnittstelle deklariert ist.
- (8) Ist das Objektattribut nicht mit dem Modifier `final` deklariert, so wird dieser der Deklaration hinzugefügt.

Durch die Schritte (5) bis (7) der variantenspezifischen Mechanik wird der neue respektive vorhandene Konstruktor angepasst, damit über diesen das Objektattribut initialisiert werden kann. Schritt (8) stellt sicher, dass das Objektattribut mit dem Modifier `final` deklariert ist. Die konzeptionelle Unveränderlichkeit des Objektattributs wird durch den final-`Modifier` deklarativ gestützt.

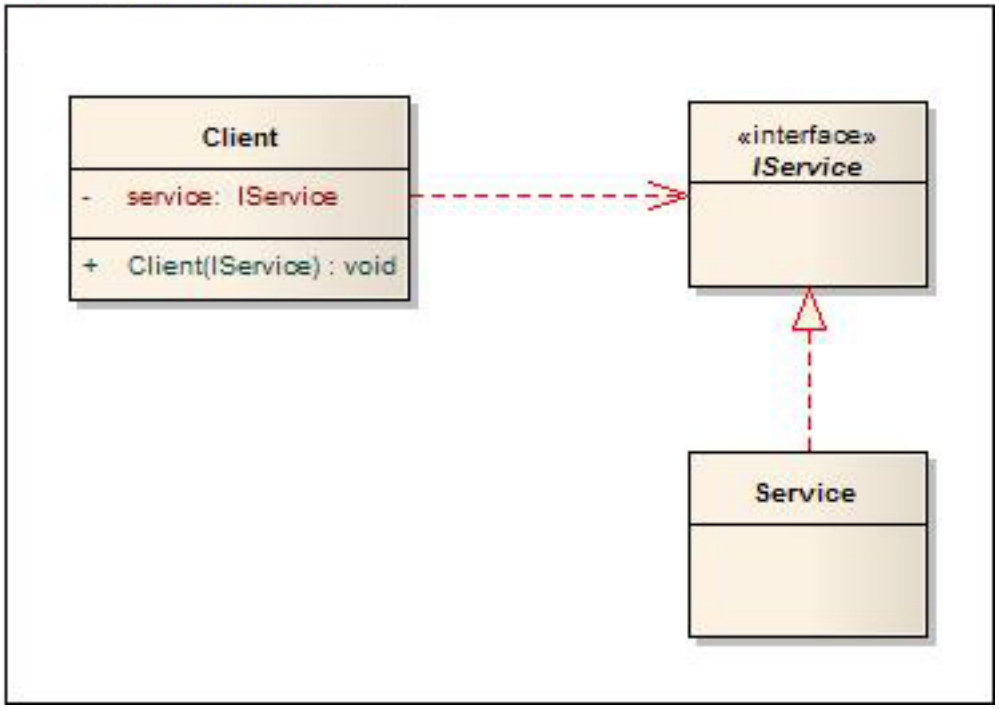

Abbildung 3.4: Constructor Injection-Variante

[31] Siehe [Fow00].

Setter Injection

- (5) Wird das Objektattribut verzögert initialisiert, so wird die Initialisierung direkt zu dessen Deklaration verschoben.
- (6) Für das Objektattribut wird dem Client eine Setter-Methode eingeführt.
- (7) Die Initialisierung des Objektattributs wird vollständig entfernt.
- (8) Ist das Objektattribut mit dem Modifier `final` deklariert, so wird dieser aus der Deklaration entfernt.

In Schritt (5) wird eine eventuell verzögerte Initialisierung zur Deklaration des Objektattributs verschoben, damit weitere auf Basis einer direkten Initialisierung aufbauen können. Die Schritte (6) und (7) führen die notwendige Setter-Methode ein und entfernen die Initialisierung aus der Deklaration des Objektattributs. Abschließend stellt Schritt (8) sicher, dass der Modifier `final` nicht in der Deklaration des Objektattributs vorhanden ist. Verbleibt der `final`-Modifier in der Deklaration kann das Objektattribut nicht innerhalb einer Methode initialisiert werden.

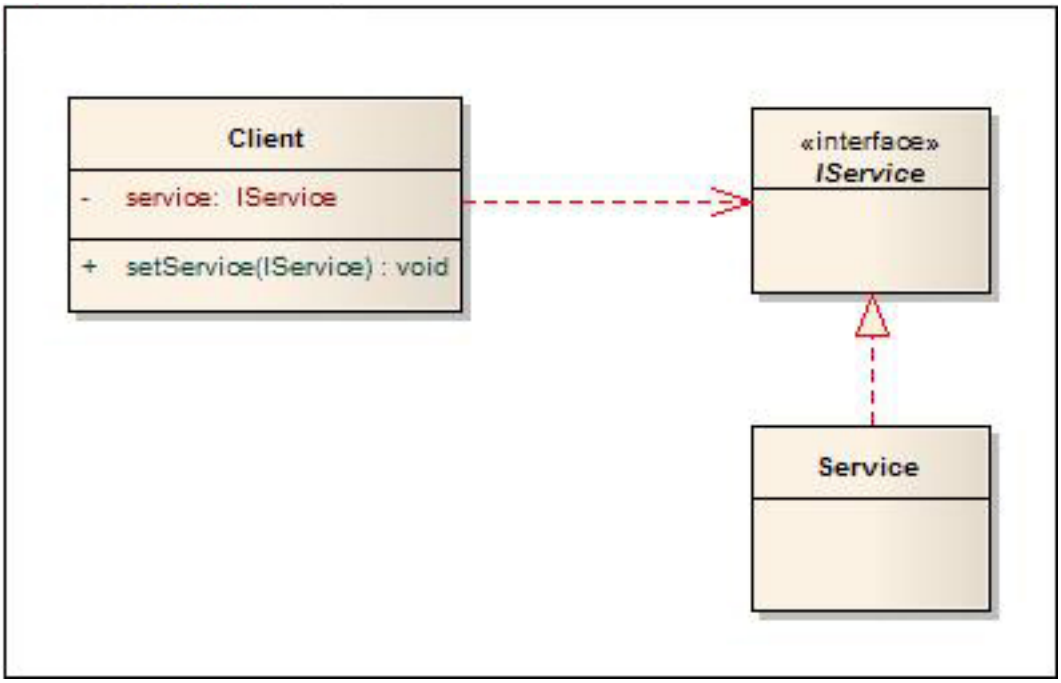

Abbildung 3.5: Setter Injection-Variante

Interface Injection

- (5) Wird das Objektattribut verzögert initialisiert, so wird die Initialisierung direkt zu dessen Deklaration verschoben.

- (6) Für das Objektattribut wird dem Client eine Setter-Methode eingeführt.
- (7) Die Initialisierung des Objektattributs wird vollständig entfernt.
- (8) Ist das Objektattribut mit dem Modifier `final` deklariert, so wird dieser aus der Deklaration entfernt.
- (9) Die Injection-Schnittstelle wird erzeugt, in der sich zudem die Deklaration der Setter-Methode befindet.
- (10) Die Deklaration des Clients wird geändert, damit dieser das neue Interface implementiert.

Die Schritte (5) bis (8) sind identisch zu denen der Setter Injection-Methode. Ergänzend wird in den Schritten (9) und (10) eine neue Schnittstelle erzeugt, die anschließend in die Typhierarchie des Clients eingeführt wird.

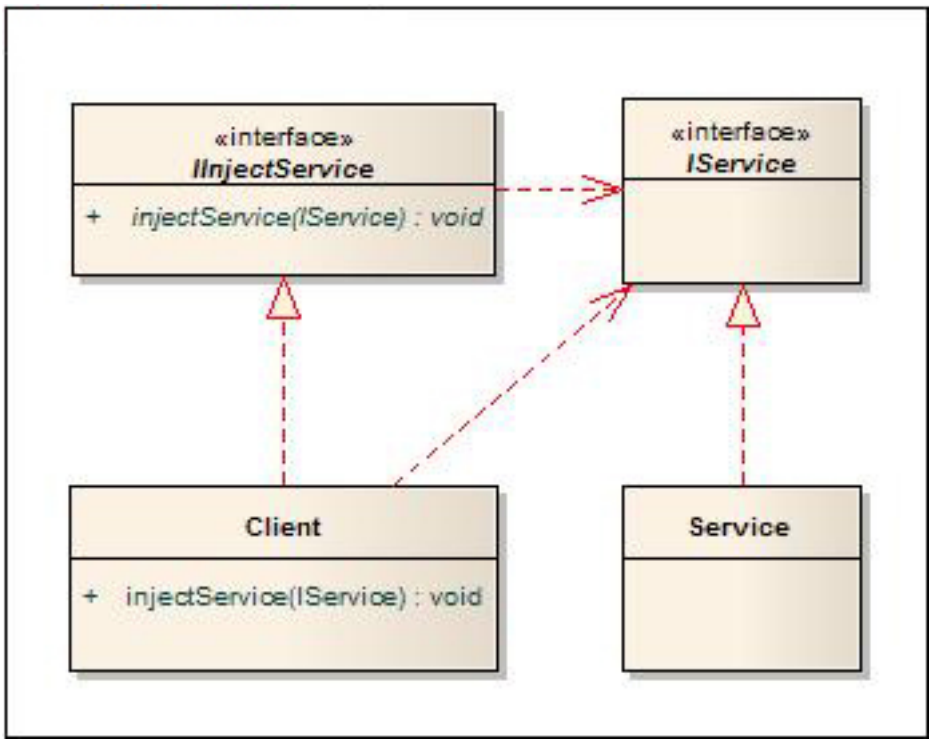

Abbildung 3.6: Interface Injection-Variante

3.6.2 Formale Beschreibung nach Ó Cinnéide

Um die Mechanik der Transformation exakt formulieren zu können, genügt die von Fowler eingesetzte Beschreibung in natürlicher Sprache jedoch nicht. Damit die Transformation

formal beschrieben werden kann, wird im Folgenden die von Ó Cinnéide[32] eingeführte Beschreibungssprache eingesetzt.

Einsatz der Beschreibungssprache

Mit Ó Cinnéides Beschreibungssprache ist es möglich die Transformationsschritte eines Refactoring als Pseudocode zu beschreiben. In dieser Sprache sind auf Basis der Prädikatenlogik elementare Typen und Methoden definiert, durch die die Einführung der Entwurfsmuster[33] von Gamma et al. formuliert werden können.

Um durch die Beschreibungssprache sowohl das Inject Dependency-Refactoring, als auch das eingebundene Refactoring Infer Type beschreiben zu können ist es jedoch erforderlich, dass deren Umfang etwas erweitert wird.

Zunächst werden die bereits vordefinierten Sprachmittel erläutert, die im Rahmen des Inject Dependency-Refactoring eingesetzt werden. Die Beschreibungen der Analysefunktionen und primitiven Refactorings wurden hierfür in gekürzter Form aus [Cin00] entnommen.

Von Ó Cinnéide definierte Analysefunktionen und primitive Refactorings:

addMethod	Primitives Refactoring
addImplementsLink	Primitives Refactoring
addInterface	Primitives Refactoring
classOf	Analysefunktion
nameOf	Analysefunktion
parameterizeField	Primitives Refactoring
sigOf	Analysefunktion

Tabelle 3.2: Von Ó Cinnéide definierte Analysefunktionen und primitive Refactorings

addMethod (Primitives Refactoring)

Definition

```
void addMethod(Class c, Method m)
```

Fügt der Klasse `c` die Methode `m` hinzu. Eine Methode mit derselben Signatur darf innerhalb der Klasse noch nicht existieren. Dieses Refactoring erweitert die externe Schnittstelle der Klasse.

addImplementsLink (Primitives Refactoring)

[32] Siehe [Cin00].
[33] Siehe [GHJV95].

Definition

`void addImplementsLink(Class concrete, Interface inf)`
Erzeugt eine `implements`-Beziehung von der Klasse `concrete` zum Interface `inf`. Die Klasse `concrete` darf nicht abstrakt sein, d.h. diese muß alle abstrakten Methoden des Interface `inf` implementieren.

addInterface (Primitives Refactoring)

Definition

`void addInterface(Interface i)`
Fügt dem Programm das Interface `i` hinzu. Eine Klasse oder ein Interface mit diesem Namen darf noch nicht existieren.

classOf (Analysefunktion)

Definition

`Class classOf(Constructor/Method/Field a)`
Liefert die Klasse zu der der gegebene Konstruktor, die gegebene Methode oder das gegebene Feld `a` gehört.

nameOf (Analysefunktion)

Definition

`String nameOf(Class/Interface/Method/Constructor x)`
Liefert den Namen der gegebenen Klasse, des gegebenen Interface der gegebenen Methode oder des gegebenen Konstruktors `x`.

parameterizeField (Primitives Refactoring)

Definition

`void parameterizeField(Field f, Constructor con)`[34]
Schiebt die Initialisierung des Objektattributs `f` aus dem Konstruktor `con` heraus, so dass der Initialwert für dieses Objektattribut dem Konstruktor als Argument übergeben werden muss.

[34] Die Signatur und die Definition dieses primitiven Refactorings wurde modifiziert, damit ein bestimmtes Objektattribut gezielt refaktorisiert werden kann. Ebenfalls der Konstruktor, durch den der Initialwert als Argument übergeben werden muss, kann hierdurch gezielt angegeben werden.

sigOf (Analysefunktion)

Definition

`Signature sigOf(Method/Constructor x)`
Liefert die Signatur der gegebenen Methode oder des gegebenen Konstruktors `x`.

Zusätzlich zu diesen bereits definierten Sprachmitteln werden im Rahmen dieser Abschlussarbeit weitere definiert, die für die formale Beschreibung der drei Varianten des Dependency Injection notwendig sind.

Neue Analysefunktionen, Hilfsfunktionen und primitive Refactorings:

addConstructor	Primitives Refactoring
addFinalModifier	Primitives Refactoring
addMethodSignature	Primitives Refactoring
createEmptyConstructor	Hilfsfunktion
createEmptyInterface	Hilfsfunktion
createSetterMethod	Hilfsfunktion
constructorsOf	Analysefunktion
declarationTypeOf	Analysefunktion
directInitializeField	Primitives Refactoring
inferType	Hilfsfunktion
isLazyInitialized	Analysefunktion
lazyInitializeField	Primitives Refactoring
noOfConstructors	Analysefunktion
removeFinalModifier	Primitives Refactoring
removeInitialization	Primitives Refactoring
updateFieldType	Primitives Refactoring

Tabelle 3.3: Neue Analysefunktionen, Hilfsfunktionen und primitive Refactorings

addConstructor (Primitives Refactoring)

Definition

`void addConstructor(Class c, Constructor con)`
Fügt der Klasse `c` den Konstruktor `con` hinzu. Ein Konstruktor mit derselben Signatur darf innerhalb der Klasse noch nicht existieren.

addFinalModifier (Primitives Refactoring)

Definition

`void addFinalModifier(Field f)`
Fügt dem Objektattribut `f` den Modifier `final` hinzu, falls dieser in dessen Deklaration noch nicht vorhanden ist.

addMethodSignature (Primitives Refactoring)

Definition

`void addMethodSignature(Interface i, Signature s)`
Fügt dem Interface `i` die MethodenSignatur `s` hinzu. Eine identische Methodensignatur darf innerhalb des Interface noch nicht existieren.

createEmptyConstructor (Hilfsfunktion)

Definition

`Constructor createEmptyConstructor(String s)`
Erzeugt und liefert einen neuen, leeren und parameterlosen Konstruktor mit dem Namen `s`.

createEmptyInterface (Hilfsfunktion)

Definition

`Interface createEmptyInterface(String s)`
Erzeugt und liefert ein neues und leeres Interface mit dem Namen `s`.

createSetterMethod (Hilfsfunktion)

Definition

`Method createSetterMethod(String prefix, Field f)`
Erzeugt und liefert für das Objektattribut `f` eine Setter-Methode. Der Name wird dabei aus dem gegebenen Präfix `prefix` und dem Namen des Objektattributs bestimmt. Für das Präfix „set“ und dem Namen „service“ eines Objektattributs wird beispielsweise der Methodenname „setService“ erzeugt. Der Typ des einzigen Arguments wird dabei durch den Typ des Objektattributs bestimmt.

constructorsOf (Hilfsfunktion)

Definition

`Constructor[] constructorsOf(Class c)`

Liefert die explizit deklarierten Konstruktoren der Klasse `c` als Array.

declarationTypeOf (Hilfsfunktion)

Definition

`Class/Interface declarationTypeOf(Field f)`
Liefert den Typ der Deklaration des Objektattributs `f`.

directInitializeField (Primitives Refactoring)

Definition

`void directInitializeField(Field f, Constructor con)`
Verschiebt die verzögerte Initialisierung des Objektattributs `f` aus dem Konstruktor `con` zu dessen Deklaration.

Vor der Anwendung von `directInitializeField`:

```
1 public class Client {
2   private Service service;
3
4   public Client() {
5     service = new Service();
6   }
7 }
```

Nach der Anwendung von `directInitializeField`:

```
1 public class Client {
2   private Service service = new Service();
3
4   public Client() {
5   }
6 }
```

inferType (Hilfsfunktion)

Definition

`Interface inferType(Class service, String serviceInf, Field f)`

Verwendet Infer Type zum Ableiten eines Interface mit dem Namen `serviceInf`, durch das die Klasse `service` abstrahiert wird. Die in diesem neu erzeugten Interface enthaltenen Methoden werden dabei durch das Nutzungsverhalten des Objektattributs `f` bestimmt.

isLazyInitialized (Analysefunktion)

Definition

`bool isLazyInitialized(Field f)`
Überprüft ob das Objektattribut `f` verzögert initialisiert wird.

lazyInitializeField (Primitives Refactoring)

Definition

`void lazyInitializeField(Field f, Constructor con)`
Verschiebt die Initialisierung des Objektattributs `f` aus der Deklaration in den Konstruktor `con`.

Vor der Anwendung von `lazyInitializeField`:

```
1 public class Client {
2   private Service service = new Service();
3
4   public Client() {
5   }
6 }
```

Nach der Anwendung von `lazyInitializeField`:

```
1 public class Client {
2   private Service service;
3
4   public Client() {
5     service = new Service();
6   }
7 }
```

noOfConstructors (Analysefunktion)

Definition

`int noOfConstructors(Class c)`
Liefert die Anzahl der explizit deklarierten Konstruktoren in der Klasse `c`.

removeFinalModifier (Primitives Refactoring)

Definition

`void removeFinalModifier(Field f)`
Entfernt dem Objektattribut `f` den Modifier `final`, falls dieser in dessen Deklaration vorhanden ist.

removeInitialization (Primitives Refactoring)

Definition

`void removeInitialization(Field f)`
Entfernt dem Objektattribut `f` die Initialisierung.

Vor der Anwendung von `removeInitialization`:

```
1 public class Client {
2   private Service service = new Service();
3 }
```

Nach der Anwendung von `removeInitialization`:

```
1 public class Client {
2   private Service service;
3 }
```

updateFieldType (Primitives Refactoring)

Definition

`void updateFieldType(Field f, Class/Interface x)`
Ändert den Deklarationstyp des Objektattributs `f` in den der gegebenen Klasse oder des gegebenen Interface `x`.

Für die Abstraktion des Service wird das bereits ausgearbeitete Refactoring Infer Type

genutzt. Die Ableitung des neuen Schnittstellentyps und die Typänderung der Objektattributdeklaration wird als Minitransformation formal beschrieben.

Neue Minitransformation:

```
1 InferredAbstraction(Field field, String serviceInf) {
2   Class service = declarationTypeOf(field);
3   Interface abstractionInterface =
4     InferType(service, serviceInf, field);
5   addInterface(abstractionInterface);
6   addImplementsLink(service, abstractionInterface);
7   updateFieldType(field, abstractionInterface);
8 }
```

Die `InferredAbstraction`-Minitransformation wird verwendet, wenn eine Client eine andere Klasse im Sinne eines Service benutzt und von diesem Service durch eine abgeleitete Schnittstelle mit dem Namen `serviceInf` abstrahiert werden soll. Die Methodendeklarationen der Schnittstelle werden dabei durch das Nutzungsverhalten bestimmt. Die neue Schnittstelle wird in die Typhierarchie des Service eingeführt und der Deklarationstyp des Objektattributs `field` wird dementsprechend geändert.

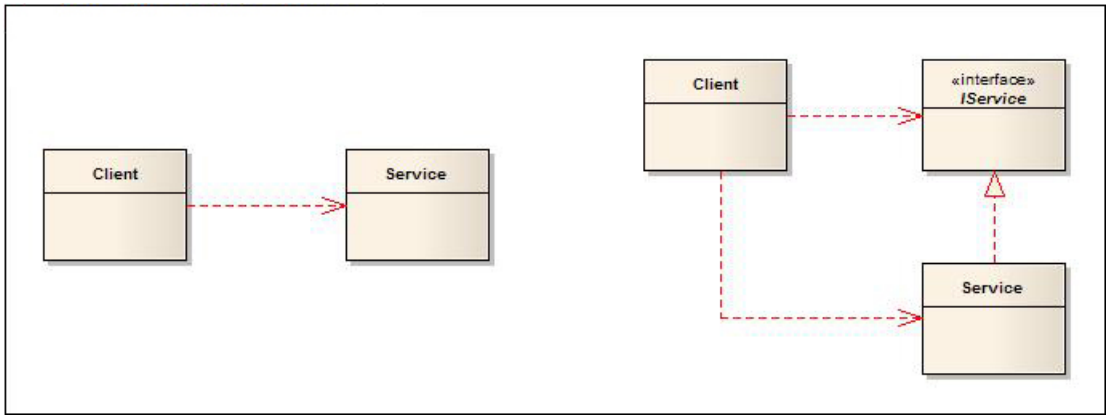

Abbildung 3.7: Anwendung der Inferred Abstraction-Minitransformation

Vorläufer

Als Vorläufer bezeichnet Ó Cinnéide eine schnelle und oberflächliche Überprüfung von Programmcode, die ermitteln soll, ob die grundsätzlichen Bedingungen für die Ausführung eines Refactoring gegeben sind.

Die Client-Klasse wird dahingehen analysiert, ob sich in ihr Objektattribute befinden, die geeignete Kandidaten für das Halten der Referenz zu einem Service-Objekt sind.

Hierzu wird für jedes Objektattribut überprüft, ob der Deklarationstyp einem Klassentyp entspricht und unveränderlich ist. Nur für unveränderliche Objektattribute ist eine gründlichere Analyse im Rahmen der späteren Überprüfung der Vorbedingungen sinnvoll. Für veränderliche Objektattribute ist es nicht möglich deren Initialisierung in einer Konfiguration abzubilden.

Der Vorläufer kann die Unveränderlichkeit durch zwei Maßnahmen ermitteln:

- Das Objektattribut ist mit dem Modifier `final` deklariert.
- Ohne den `final`-Modifier wird durch eine Analyse die Art der Initialisierung des Objektattributs ermittelt. Könnte das Objektattribut durch die Implementierung des Programmcodes prinzipell als `final` deklariert werden, wird dieses als konzeptionell unveränderlich angesehen.

Transformationen

Die Transformationen der drei Dependency Injection-Varianten bauen gemeinsam auf dem initialen Einsatz des Infer Type-Refactoring auf. Erst nachdem die Abstraktion vom Service eingeführt und die Typdeklaration des Objektattributs angepasst wurde, werden variantenspezifische Anweisungen durchgeführt.

Algorithmus der Constructor Injection-Transformation

```
1  ApplyConstructorInjection(Field field, String serviceInf) {
2    InferredAbstraction(field, serviceInf);
3
4    Class client = classOf(field);
5    Constructor con;
6    if (noOfConstructors(client) = 0) {
7      con = createEmptyConstructor(nameOf(client));
8      addConstructor(client, con);
9      lazyInitializeField(field, con);
10   }
11   else {
12     con = constructorsOf(client)[0];
13     if (!isLazyInitialized(field)) {
14       lazyInitializeField(field, con);
15     }
16   }
17
18   parameterizeField(field, con);
```

```
19    addFinalModifier(field);
20  }
```

Existiert in der Klasse `client` kein expliziter Konstruktor, so wird dieser ein neuer hinzugefügt und die Initialisierung des Objektattributs `field` wird in diesen neuen Konstruktor verschoben (Zeilen 6-10). Gesetz den Fall, dass bereits ein expliziter Konstruktor existiert, wird geprüft, ob das Objektattribut `field` direkt initialisiert wird und dessen Initialisierung wird gegebenenfalls verschoben (Zeile 11-16). Abschließend wird die Initialisierung aus dem Konstruktor entfernt, dem Konstruktor ein entsprechender Übergabeparameter hinzugefügt und der eventuell fehlende `final`-Modifier der Objektattributdeklaration hinzugefügt (Zeilen 18 und 19).

Algorithmus der Setter Injection-Transformation

```
1  ApplySetterInjection(Field field, String serviceInf) {
2    InferredAbstraction(field, serviceInf);
3
4    Class client = classOf(field);
5    if (isLazyInitialized(field)) {
6      Constructor con = constructorsOf(client)[0];
7      directInitializeField(field, con);
8    }
9
10   Method setter = createSetterMethod("set", field);
11   addMethod(client, setter);
12
13   removeInitialization(field);
14   removeFinalModifier(field);
15 }
```

Zunächst wird geprüft, ob das Objektattribut `field` verzögert initialisiert wird. Ist dem so wird die Initialisierung zur Deklaration verschoben (Zeilen 5-8). Danach wird der Klasse `client` eine Setter-Methode hinzugefügt (Zeilen 10 und 11). Abschließend werden die Initialisierung und der eventuell vorhandene `final`-Modifier aus der Objektattributdeklaration entfernt (Zeilen 13 und 14).

Algorithmus der Interface Injection-Transformation

```
1  ApplyInterfaceInjection(Field field, String serviceInf,
2      String injectInf) {
3    InferredAbstraction(field, serviceInf);
4
5    Class client = classOf(field);
6    if (isLazyInitialized(field)) {
7      Constructor con = constructorsOf(client)[0];
8      directInitializeField(field, con);
9    }
10
11   Method setter = createSetterMethod("inject", field);
12   addMethod(client, setter);
13
14   removeInitialization(field);
15   removeFinalModifier(field);
16
17   Interface injectionInterface = createEmptyInterface(injectInf);
18   addInterface(injectionInterface);
19   addMethodSignature(client, sigOf(setter));
20   addImplementsLink(client, injectionInterface);
21 }
```

Die Transformationsschritte dieser Variante bauen auf denen der Setter Injection auf, wobei sich lediglich der Präfix der eingeführten Setter-Methode unterscheidet (Zeilen 5-14). Zusätzlich wird eine Injection-Schnittstelle injectInf erzeugt und der Typhierarchie des Client hinzugefügt. Dieser neuen Schnittstelle wird die Deklaration der neuen Setter-Methode hinzugefügt (Zeilen 16-19).

Vorbedingungen

Die Eingabeparameter für den Transformationsprozess sind für das Inject Dependency-Refactoring die Argumente `Field field` und `String serviceInf`. Bei der Interface Injection-Variante kommt zusätzlich das Argument `String injectInf` hinzu.

Unabhängig von Benutzereingaben

- Der Initialisierungszeitpunkt des Objektattributs durch die Dependency Injection-Methode darf nicht dessen Zugriffszeitpunkt kollidieren.

- Die Klassen des Client und des Service müssen `public` sein, ebenso die darin deklarierten Konstruktoren. Beide Klassen dürfen nicht abstrakt sein.
- Die Parameter der Service-Konstruktoren müssen ausnahmslos durch Literale gegeben sein.
- Der Client darf nur maximal einen explizit deklarierten und parameterlosen Konstruktor besitzen.
- Infer Type muss in der Lage sein anhand des Benutzungsverhaltens eine Abstraktion des Service ableiten zu können.
- Das Objektattribut der Abhängigkeitsbeziehung muss entweder explizit oder konzeptionell unveränderlich sein.

Abhängig von Benutzereingaben

- Der Name `serviceInf` darf noch von keinem Typ verwendet werden.
- Der Name `injectInf` darf noch von keinem Typ verwendet werden (nur Interface Injection).
- Im Client dürfen noch keine Methoden existieren, die dieselben Signaturen wie die zu erstellenden Setter-Methoden haben (nur Setter Injection und Interface Injection).

Rahmenbedingungen

Neben den Vorbedingungen, die speziell für das Inject Dependency-Refactoring gelten, gibt es ergänzend noch weitere Rahmenbedingungen, die es zu erfüllen gilt. Diese Randbedingungen haben einen allgemeinen Charakter und gelten prinzipiell auch für andere Refactorings.

- Da die Transformation eines Refactoring die relevanten Typen modifiziert, müssen sowohl der Client als auch der Service im Quellcode vorhanden sein.
- Für ein Refactoring ist es für die Analyse und die Modifizierung des relevanten Programmcodes notwendig, dass dieser fehlerfrei übersetzbar ist.
- Codefragmente, die Reflection einsetzen, können nicht berücksichtigt werden.

3.7 Die Aufgabe des Assemblers

Während der Beschreibung des Inject Dependency-Refactoring wurde bisher davon ausgegangen, dass der Assembler einem Client-Objekt ein Service-Objekt injizieren kann, ohne jedoch genauer zu spezifizieren, wie dieser diese Funktionalität realisiert. Zwar gibt es bereits seit längerem ausgereifte Dependency Injection-Frameworks, wie beispielsweise Guice[35], wird im Folgenden jedoch zunächst von der Erzeugung einer Assembler-Klasse ausgegangen, um möglichst lange nah am Konzept des Dependency Injection-Patterns bleiben zu können.

Das Verknüpfungsgeflecht zwischen Clients und Services wird durch eine Konfiguration gegeben, die bereits vor der Ausführung des Programmcodes feststeht, unabhängig davon, ob diese Konfiguration durch Code oder Konfigurationsdateien definiert wird.

Um die Implementierung eines Assemblers und dessen Konfiguration durch Programmcode beispielhaft darstellen zu können, gehen wir im weiteren von einem einfachen Beispiel aus:

```
1  public class Service {
2  }
3
4  public class Client {
5    private Service service = new Service();
6  }
```

Angenommen der Client wird vom Typ des Service durch die Einführung der Constructor Injection-Methode vollständig entkoppelt, dann ändert das Inject Dependency-Refactoring die vorhandenen Klassen wie folgt:

```
1  public interface IService {
2  }
3
4  public class Service implements IService {
5  }
6
7  public class Client {
8    private final IService service;
9
10   public Client(IService service) {
11     this.service = service;
12   }
```

[35] Siehe http://http://code.google.com/p/google-guice/.

```
13 }
```

Abschließend ist es die Aufgabe des Refactoring einen Assembler zu erzeugen, der diese Konfiguration realisieren muss:

- Erzeugung eines Objekts der Klasse `Service`.
- Erzeugung eines Objekts der Klasse `Client`.
- Injektion der `Service`-Referenz in das `Client`-Objekt durch dessen Konstruktor.

Die folgende Implementierung eines Assemblers würde das gewünschte Verhalten realisieren:

```
1  public class Assembler {
2    private static final Service service;
3    private static final Client client;
4
5    static {
6      service = new Service();
7      client = new Client(service);
8    }
9
10   public Client getClient() {
11     return client;
12   }
13 }
```

Dieser Assembler erzeugt jeweils ein Objekt des Client und Service, injiziert dem Client die Abhängigkeit und stellt das Client-Objekt nach außen zur Verfügung. Die Konfiguration wird hierbei ausschließlich innerhalb des Klassenkonstruktors realisiert.

4 Grenzen der Verwendbarkeit

Um die Grenzen der Verwendbarkeit des Inject Dependency-Refactorings ermitteln zu können, ist die isolierte Betrachtung zweier Klassen nicht ausreichend. In gewachsenen Softwaresystemen befinden sich Klassen in einer komplexen Struktur, die für eine genauere Betrachtung nicht ignoriert werden darf.

4.1 Abbildung des Objektgeflechts der Service-Objekte

Die Bezeichnungen Client und Service sind die Namen der Rollen zweier Klassen, wenn zwischen diesen eine Abhängigkeitsbeziehung besteht. Es ist durchaus üblich, dass es innerhalb von Softwaresystemen Objekte gibt, deren Klassen in Abhängigkeit zur jeweiligen Betrachtung einmal die Rolle eines Clients und ein anderes mal die eines Service spielen.

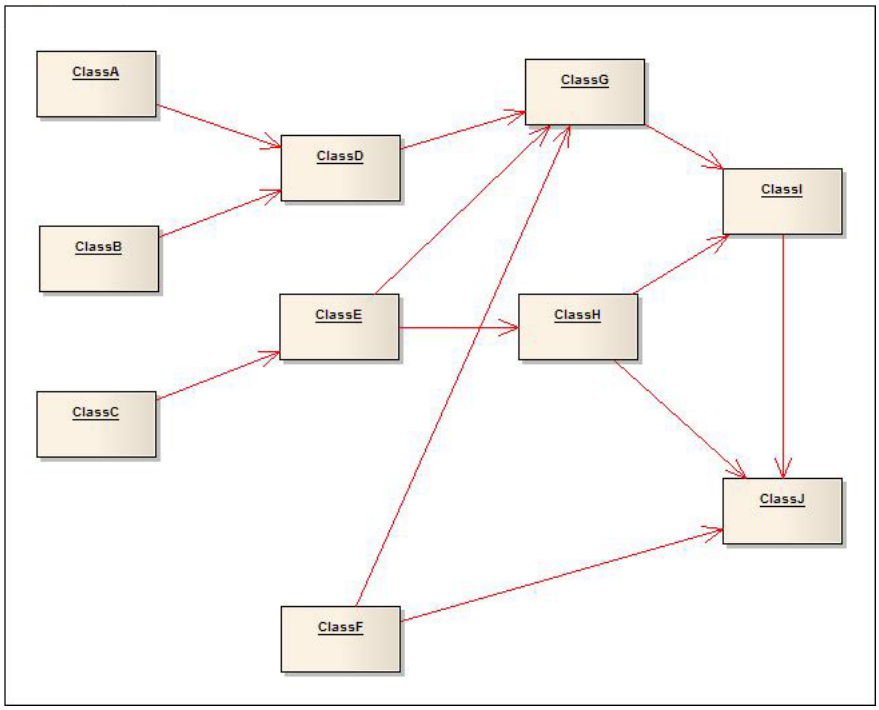

Abbildung 4.1: Klassengeflecht mit Kopplungen

In dieser Grafik hat beispielsweise `ClassD` eine Abhängigkeit zu `ClassG`, was `ClassD` zum Client und `ClassG` zum Service macht. Für `ClassG` stellt jedoch `ClassI` eine Abhängigkeit dar und macht diese somit zu einem Client.

Der Einsatz des Inject Dependency-Refactorings darf nicht als eine isolierte Anpassung gesehen werden, sonder ist vielmehr einer von mehreren Schritten um bestehendem Programmcode das Dependency Injection-Pattern einzuführen.

Angenommen `ClassG` sollte von `ClassI` entkoppelt werden, dann würden diese beiden Klassen dementsprechend modifiziert und die notwendige Konfiguration erzeugt werden.

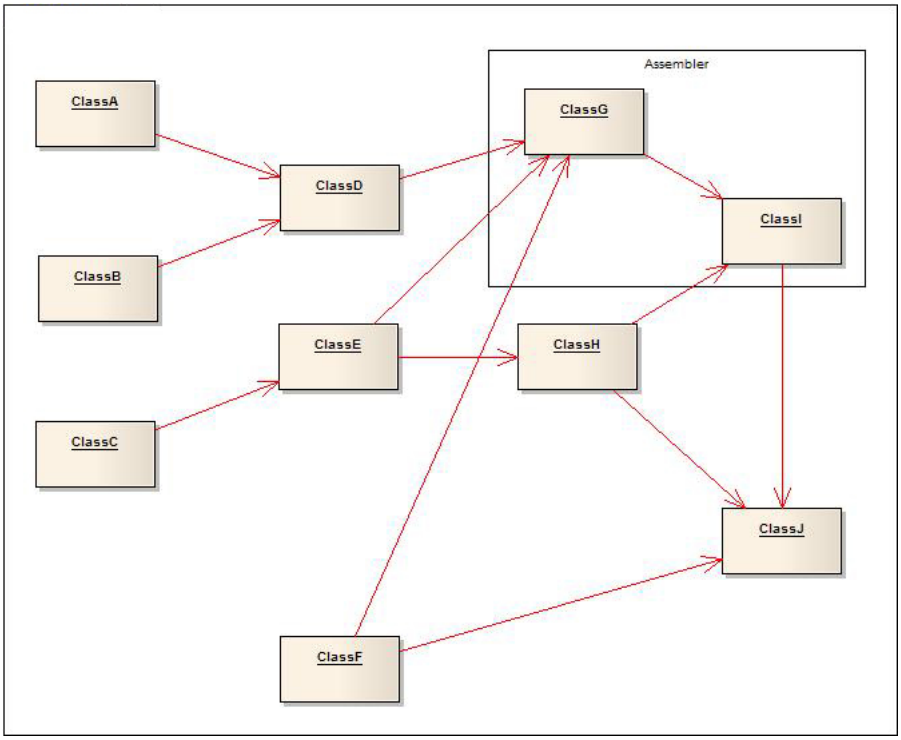

Abbildung 4.2: Klassengeflecht mit einer Entkopplung

Dieser Zwischenschritt führt den Programmcode in einen fehlerhaften Zustand. Wurde `ClassG` von `ClassI` durch den Einsatz der Constructor Injection-Methode entkoppelt, so wurde in dessen Konstruktor ein neuer Parameter eingeführt. Die Konstruktor-Aufrufe in `ClassD`, `ClassE` und `ClassF` können diesen neuen Parameter ohne Anpassung des Codes nicht bedienen. Der Programm wäre vom Compiler also nicht übersetzbar. Bei dem Einsatz der Setter bzw. Interface Injection-Methode wäre der Programmcode zwar übersetzbar und ausführbar, würde jedoch mit einer `NullPointerException` abbrechen, da die Abhängigkeit des `ClassG` nicht injiziert wurde.

Eine Lösung des Problems scheint auf den ersten Blick die Modifikation von `ClassD`, `ClassE` und `ClassF` zu sein, die bei dieser Betrachtung die Clients des Client wären. Hierfür müssten die Codefragmente, in denen ein Objekt von `ClassG` erzeugt wird, durch den Einsatz des Assemblers ersetzt werden. So könnte Beispielsweise der Konstuktoraufruf

`new ClassG()` durch einen Methodenaufruf `getClassG()` des erzeugten Assemblers ersetzt werden. Die relevanten Codestellen wären durch statische Codeanalyse problemlos identifizierbar. Der Programmcode wäre zweifellos übersetzbar, ausführbar und würde auch nicht durch eine `NullPointerException` abbrechen.

Dieser Lösungsansatz hätte jedoch die folgenden Konsequenzen:

- Mit Ausnahme des Bootstrapping sollte kein Typ eines Softwaresystems die Kenntnis vom Assembler besitzen. Im schlimmsten Fall kann zwar das Refactoring eine Abhängigkeit lösen, führt jedoch im Gegenzug in anderen Klassen eine Abhängigkeit zum Typ des Assemblers ein und ist somit kontraproduktiv.

- In den Clients des Client wird der Assembler als Service Locator eingesetzt. Dependency Injection soll eine vollwertige Alternative darstellen, weshalb dieser Kompromis nicht akzeptabel ist.

- Steht der ersetzte Konstruktoraufruf in den Clients des Client ursprünglich für die Erzeugung eines Service-Objekts, so wird das zugehörige Objektattribut für ein weiteres Refactoring verschleiert. Ohne diesen Konstruktoraufruf ist es für das Inject Dependency-Refactoring durch statische Codeanalyse nicht mehr möglich das Objektattribut als geeigneten Kandidaten zu erkennen.

Damit zwei Klassen innerhalb einer komplexen Struktur vollständig entkoppelt werden können, dürfen die Clients des Client innerhalb eines Refactoringschritts nicht angepasst werden. Beginnt man die Refactoringschritte bei denjenigen Klassen, die ausschließlich die Rolle des Client spielen, kann man ein Klassengeflecht vollständig entkoppeln.[36] Beginnend bei `ClassA`, `ClassB`, `ClassC` und `ClassF` wird die vollständige Entkopplung durch die im Folgenden informelle Konfiguration realisiert. Dabei wird davon ausgegangen, dass beispielsweise `ClassA` durch das Interface `ClassDA` vom Typ `ClassD` entkoppelt wird:

- Entkopplung der Abhängigkeiten von `ClassA`
  - Binden[37] eines Objekts von `ClassD` an den Typ `IClassDA`

- Entkopplung der Abhängigkeiten von `ClassB`

[36] Dies funktioniert natürlich nicht, wenn sich Zyklen durch die Abhängigkeitsbeziehungen bilden. Diese zyklische Abhängigkeiten werden von einigen Dependency Injection-Frameworks unterstützt und mit Hilfe eines Proxy-Mechnismus realisiert. Da solche Zyklen jedoch Symptome eines schlechten Software-Designs sind, werden diese nicht vom Inject Dependency-Refactoring unterstützt.

[37] Im Jargon des Dependency Injection-Patterns werden Klassentypen an zumeist Schnittstellentypen gebunden. Durch diese Bindung kann ein Assembler eine Abhängigkeit, die durch einen Schnittstellentyp gegeben ist, auflösen und von der zugehörigen Klasse ein Objekt erzeugen und injizieren.

  - Binden eines Objekts von `ClassD` an den Typ `IClassDB`

- Entkopplung der Abhängigkeiten von `ClassC`
  - Binden eines Objekts von `ClassE` an den Typ `IClassEC`
- Entkopplung der Abhängigkeiten von `ClassD`
  - Binden eines Objekts von `ClassG` an den Typ `IClassGD`
- Entkopplung der Abhängigkeiten von `ClassE`
  - Binden eines Objekts von `ClassG` an den Typ `IClassGE`
  - Binden eines Objekts von `ClassH` an den Typ `IClassHE`
- Entkopplung der Abhängigkeiten von `ClassF`
  - Binden eines Objekts von `ClassG` an den Typ `IClassGF`
  - Binden eines Objekts von `ClassJ` an den Typ `IClassJF`
- Entkopplung der Abhängigkeiten von `ClassG`
  - Binden eines Objekts von `ClassI` an den Typ `IClassIG`
- Entkopplung der Abhängigkeiten von `ClassH`
  - Binden eines Objekts von `ClassI` an den Typ `IClassIH`
  - Binden eines Objekts von `ClassJ` an den Typ `IClassJH`
- Entkopplung der Abhängigkeiten von `ClassI`
  - Binden eines Objekts von `ClassJ` an den Typ `IClassJI`

Zusammenfassend eine Übersicht über die Klassen, die Anzahl der durch den Assembler erzeugten Objekte und die Typen, durch die die Objekte auflösbar sind:

Name der Klasse	Anzahl Objekte	Auflösbar durch Typ
ClassD	2	IClassDA, IClassDB
ClassE	1	IClassEC
ClassG	3	ClassGD, IClassGE, IClassGF
ClassH	1	IClassHE
ClassI	2	IClassIG, IClassIH
ClassJ	3	IClassJF, IClassJH, IClassJI

Tabelle 4.1: Objektgeflecht der Service-Objekte infolge der erzeugten Konfiguration

Mit dieser Konfiguration werden alle Klassen vollständig entkoppelt, was dazu führt, dass der Assembler für die Objekterzeugungen der eliminierten `new`-Operatoren verantwortlich ist. Sobald der Assembler im Bootstrapping erzeugt wird, nutzt dieser die Konfiguration um sämtliche Service-Objekte zu erzeugen und die Abhängigkeiten zu injizieren. In dem eingeführten Beispiel sähe der erzeugte Objektgraph wie folgt aus:

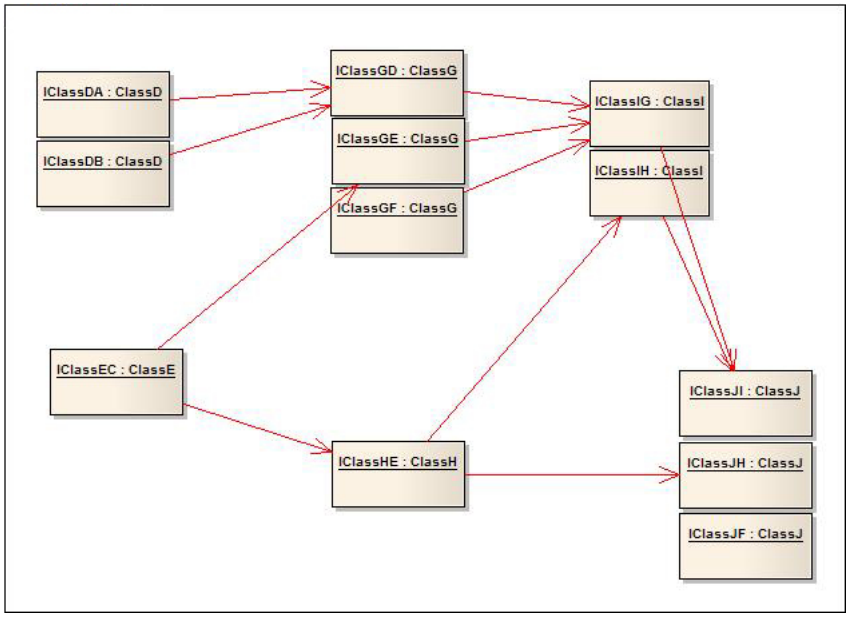

Abbildung 4.3: Objektgeflecht der Service-Objekte infolge der erzeugten Konfiguration

Auch wenn eine Klasse einmal die Rolle des Clients und ein anderes mal die Rolle des Service spielt, werden durch die Refactoringschritte keine überlappenden Konfigurationen erzeugt.

4.2 Erzeugung der Konfiguration

Es ist jedoch nicht praktikabel, dass für jede Konfiguration, die eine Entkopplung ergänzt, eine Assemblerklasse erzeugt wird. Würde jede Assemblerklasse einen Teil der gesamten Konfiguration realisieren, müsste ein Mechanismus eingeführt werden, durch den sich die Assemblerklassen austauschen könnten. Auch das Bootstrapping wäre sehr aufwendig, da von jeder Assemblerklasse ein Objekt erzeugt werden müsste um die gesamte Konfiguration zu laden.

Die hierfür notwendigen Anpassungen und Erweiterungen der Assemblerklasse würden dazu führen, dass eine Art Dependency Injection-Framework realisiert wird. Es wäre nur ein geringer zusätzlicher Aufwand, um die Assemblerklassen dann dahingehend zu erweitern, dass die Konfiguration beispielweise durch eine Konfigurationsklasse geladen

wird. Bereits an diesem Punkt wäre kein grundsätzlicher Unterschied mehr zu bereits existierenden Frameworks.

Es ist nicht die Aufgabe des Inject Dependency Plug-Ins ein eigenes Dependency Injection-Framework einzuführen, sondern soll einen generischen Charakter besitzen. Während die Einführung der Dependency Injection-Methoden frameworkunabhängig sein muss, ist die Erzeugung der Konfiguration hingegen direkt abhängig von einem Dependency Injection-Framework.

Das Refactoring stellt aus diesem Grund eine Schnittstelle zur Verfügung, durch die andere Plug-Ins einen Generator für frameworkspezifische Konfigurationen realisieren können. Das Inject Dependency Plug-In selbst nutzt diese Schnittstelle, damit eine einfache Assemblerklasse erzeugt werden kann. Diese Assemblerklasse realisiert jedoch ausschließlich die Konfiguration einer einzigen Abhängigkeitsbeziehung.

4.3 Reichweite der Transformation

Da das Inject Dependency-Refactoring zu den großen Refactorings gehört ist es nicht überraschend, dass es nicht vollständig verhaltenskonservierend ist. Durch seine Eigenschaft, dass die Entkopplungen schrittweise durchgeführt werden müssen und das Bootstrapping nicht zum eigentlichen Dependency Injection-Pattern gehört, sind vom Benutzer weitere Anpassungen notwendig. Die Transformation an sich ist unvollständig und führt erst dann wieder zu einem fehlerfrei ausführbaren Programm, wenn der Benutzer die ergänzende Konfiguration durch einen Assembler lädt und die notwendigen Objekte im Boostrapping vom Assembler bezieht.

Die Erzeugung einer einfachen Assemblerklasse wird eine Alternative zur Erzeugung einer Konfiguration bieten, wenn für kleine Projekte der Einsatz eines Dependency Injection-Frameworks nicht angebracht ist.

Durch die Schnittstelle für die Integration von Konfigurationsgeneratoren wird es möglich sein verfügbare Frameworks wie beispielsweise Guice direkt zu unterstützen. Neben der Erzeugung einer einfachen Assemblerklasse wird das Inject Dependency-Refactoring somit auch frameworkspezifische Konfigurationen generieren können.

5 Realisierung des Plug-Ins

Technische Anforderungen, Teile der Implementierung und die Architektur des Inject Dependency-Refactoring werden durch die direkte Einbindung in eclipse zu großen Teilen bereits vorgegeben. Das Plug-In wurde mit Hilfe der PDE[38] realisiert, setzt Java ab der Version 1.5 und eclipse in der aktuellen Version 3.4 voraus.

Dieses Kapitel stellt ausgewählte Auszüge des Entwurfs und der Implementierung vor, um dem interessierten Leser die relevanten Teile der Realisierung aufzuzeigen.

5.1 Refactoring mit eclipse

eclipse stellt seit der Version 3.1 für die Realisierung von Refactorings eine API zur Verfügung: Das Language Toolkit (LTK)[39]. Neben dieser sprachneutralen API definiert das LTK ebenfalls den Lebenszyklus von Refactorings, der durch eine exakte Prozedur beschrieben wird:

1. Das Refactoring wird vom Benutzer gestartet.

2. Eine initiale und schnelle Überprüfung wird durchgeführt die bestimmen soll, ob das Refactoring im gegebenen Kontext angewandt werden kann. Hierfür ist die Methode `checkInitialConditions` der Klasse `org.eclipse.ltk.core.refactoring.Refactoring` bzw. deren Subklasse vorgesehen.

3. Falls notwendig, wird der Benutzer nach zusätzlichen Informationen gefragt.

4. Nachdem alle notwendigen Informationen für die Ausführung des Refactoring verfügbar sind, wird nochmals eine diesmal gründlichere Überprüfung durchgeführt. Hierfür ist die Methode `checkFinalConditions` der Klasse `org.eclipse.ltk.core.refactoring.Refactoring` bzw. deren Subklasse vorgesehen.

[38] Siehe [MG08].
[39] Siehe [Fre08].

5. Die Änderungen des Programmcodes werden berechnet. Hierfür ist die Methode `createChange` der Klasse `org.eclipse.ltk.core.refactoring.Refactoring` bzw. deren Subklasse vorgesehen.

6. Ein Vorschaudialog erscheint, der Benutzer bestätigt den Dialog und die Änderungen werden durchgeführt.

Durch die beiden Überprüfungsroutinen `checkInitialConditions` und `checkFinalConditions` wird verhindert, dass Programmcode durch die Anwendung eines Refactoring in einen korrupten Zustand überführt wird. Während die erste Überprüfung ohne bemerkbare Verzögerung durchgeführt werden sollte, kann die zweite Überprüfung etwas länger dauern, sollte jedoch im Bereich weniger Sekunden liegen.

Eine zusätzliche und benutzerfreundliche Möglichkeit um zu verhindern, dass der Benutzer in die Irre geführt wird, ist das deaktivieren des Inject Dependency-Refactoring, wenn dieses nicht angewandt werden kann. Beispielsweise kann der Menüpunkt *Inject Dependency* im Kontextmenü des Texteditors technisch durch die Implementierung des Interfaces `org.eclipse.ui.IEditorActionDelegate` realisiert werden. Jede Änderung der Textselektion im Texteditor wird dabei der Methode `void selectionChanged(IAction action, ISelection selection)` der implementierenden Klasse mitgeteilt. Dort kann dann die schnelle Überprüfung durchgeführt werden. Werden bei der Überprüfung keine Probleme festgestellt, so wird der Menüpunkt durch `action.setEnabled(true)` sichtbar geschaltet. Andernfalls wird der Menüpunkt durch `action.setEnabled(false)` vor dem Benutzer verborgen.

5.2 Abbildung der bedingten Transformation

Für die Realisierung des Plug-Ins müssen die konzeptionellen Überlegungen aus Kapitel 3 gezielt abgebildet werden. Abbildung 5.1 zeigt, wie die Beschreibung eines Refactoring auf den vom LTK definierten Lebenszyklus abgebildet werden kann. Der Vorläufer eines Refactoring ist ein struktureller Hinweis im Programmcode, der schnell festgestellt werden kann. Die Überprüfung des Vorläufers wird deshalb als Kriterium für das Ein- bzw. Ausblenden des Menüpunkts für das Refactoring eingesetzt.

Die Vorbedingungen werden in einen von Benutzereingaben unabhängigen und in einen von Benutzereingaben abhängigen Teil aufgeteilt. Die unabhängigen Vorbedingungen werden deshalb zusammen mit dem Vorläufer[40] in der Methode `checkInitialConditions` implementiert. Die abhängigen Vorbedingungen werden in der Methode `checkFinalConditions` implementiert. Die Transformation wird direkt in der

[40]Da der Vorläufer auf jeden Fall überprüft werden muss, wird er hier ebenfalls überprüft.

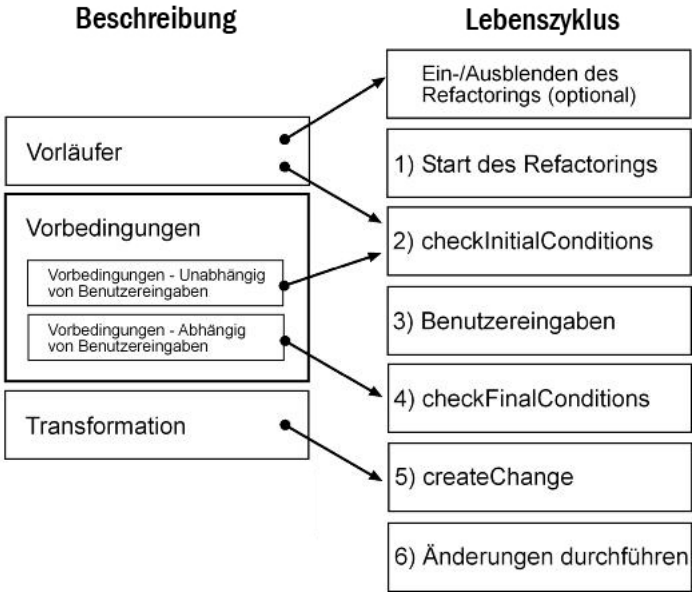

Abbildung 5.1: Lebenszyklus eines Refactoring

Methode `createChange` implementiert, da hier im Lebenszylus die Änderungen berechnet werden.

5.3 Anforderungen an das Plug-In

Während die Transformationen des Programmcodes, die aus der Einführung des Dependency Injection-Patterns folgen, direkt durch das Inject Dependency-Refactoring realisiert werden, muss für die Erzeugung des Abstraction-Interface Infer Type genutzt werden. Durch die Einbindung des Infer Type-Refactoring ist das Inject Dependency Plug-In direkt abhängig vom Infer Type Plug-In: Bevor das Inject Dependecy Plug-In in eclipse geladen werden kann, muss Infer Type bereits installiert sein.

Das Refactoring muss wie andere Standard-Refactorings von eclipse durch die GUI gestartet und gesteuert werden können. Der Startpunkt ist dabei eine geöffnete Java-Datei in der Java-Perspektive von eclipse. Durch das Kontextmenü soll der Menüpunkt *Inject Dependency* angezeigt werden, durch den das Refactoring gestartet werden kann. Eine schnelle Überprüfung des im Texteditor geöffneten Typs soll den Menüpunkt je nach Ergebnis der Analyse anklickbar oder ausgegraut darstellen.

Ein Code-Fragment muss hierbei nicht vom Benutzer markiert werden, da das Refactoring nicht auf ein bestimmtes Objektattribut abzielt, sondern für den aktuell dargestellten Typ gestartet wird. Dieser Typ gilt somit als Client, der von vorhandenen Services entkoppelt werden soll. Nachdem das Refactoring gestartet wurde, werden alle relevanten Objektattribute dargestellt, für die eine Refaktorisierung möglich ist. Für

jedes Objektattribut kann der Benutzer konfigurieren, ob der Client von dem jeweiligen Service entkoppelt wird und welche Dependency Injection-Methode hierfür eingesetzt werden soll.

Das Inject Dependency-Refactoring kann für die Entkopplung eines Client von einem Service eine Assemblerklasse generieren. Um jedoch auch verfügbare Dependency Injection-Frameworks unterstützen zu können, muss eine Generator-Schnittstelle definiert werden, durch die ein Generator für ein spezifisches Framework Konfigurationen erzeugen kann.

5.4 Entwurf

Für die Beschreibung der Architektur des Inject Dependency-Refactoring ist vor allem die *funktionale Sicht* von Interesse, da andere architektonischen Sichten[41] bereits durch eclipse vorgegeben und durch [CR06] bzw. [GB03] abgeleitet werden können.

5.4.1 Funktionale Sicht der Architektur

Die Implementierung untergliedert sich in zehn funktionale Module, die teilweise mit externen Modulen kooperieren.

Durch die Installation des Refactoring bindet `eclipse` zunächst die beiden Module `action` und `ui` ein. In diesen beiden Modulen befinden sich Klassen, die per Konfiguration für die Integration in die GUI und für die Teilnahme am Event-Handling[42] registriert werden. Im Kontextmenü der Java-Perspektive wird der Menüpunkt *Inject Dependency* eingefügt, wobei hier das `SelectionChanged`-Event[43] für eine Vorläufer-Überprüfung, wie in Absatz 3.6.2 beschrieben, genutzt wird: Eine oberflächliche und schnelle Überprüfung des Clients ermittelt, ob dieser Objektattribute besitzt, die potentiell für eine Abhängigkeit zu einem Service stehen. Abhängig vom Ergebnis dieser Analyse wird der Menüpunkt anklickbar oder ausgegraut dargestellt. Wird das Refactoring durch einen Klick auf den Menüpunkt gestartet, so nutzt das `action`-Modul das `ui`-Modul, um den Refactoring-Dialog darzustellen.

Nach den Anpassungen der Optionen im Dialog durch den Benutzer wird der Transformationsprozess durch das Modul `transformation` durchgeführt. Da sich dieses Modul für die Abstraktion des Service auf Infer Type stützt, wird durch das Modul `transformation.support` das `Infer Type-Refactoring` eingebunden.

[41] Für eine Vertiefung in das Thema architektonische Sichten und Perspektiven bei Software-Systemen siehe [WR05].
[42] Siehe [Pad08].
[43] Siehe [Hof08].

Neben den Code-Transformationen ist die Erzeugung von Konfigurationscode notwendig. Das Modul `generation` enthält eine Generator-Schnittstelle, die durch andere Plug-Ins implementiert werden kann. Diese Plug-Ins können dann ebenfalls von `eclipse` geladen werden und somit dem Inject Dependency-Refactoring durch das Modul `registry` zur Verfügung gestellt werden. Neben dem Generator für eine einfache Assemblerklasse im Modul `generation.builtin` wird in der Abbildung5.2 das Modul `Guice Configuration-Generator` dargestellt das für Guice geeignete Konfigurationen erzeugen kann.

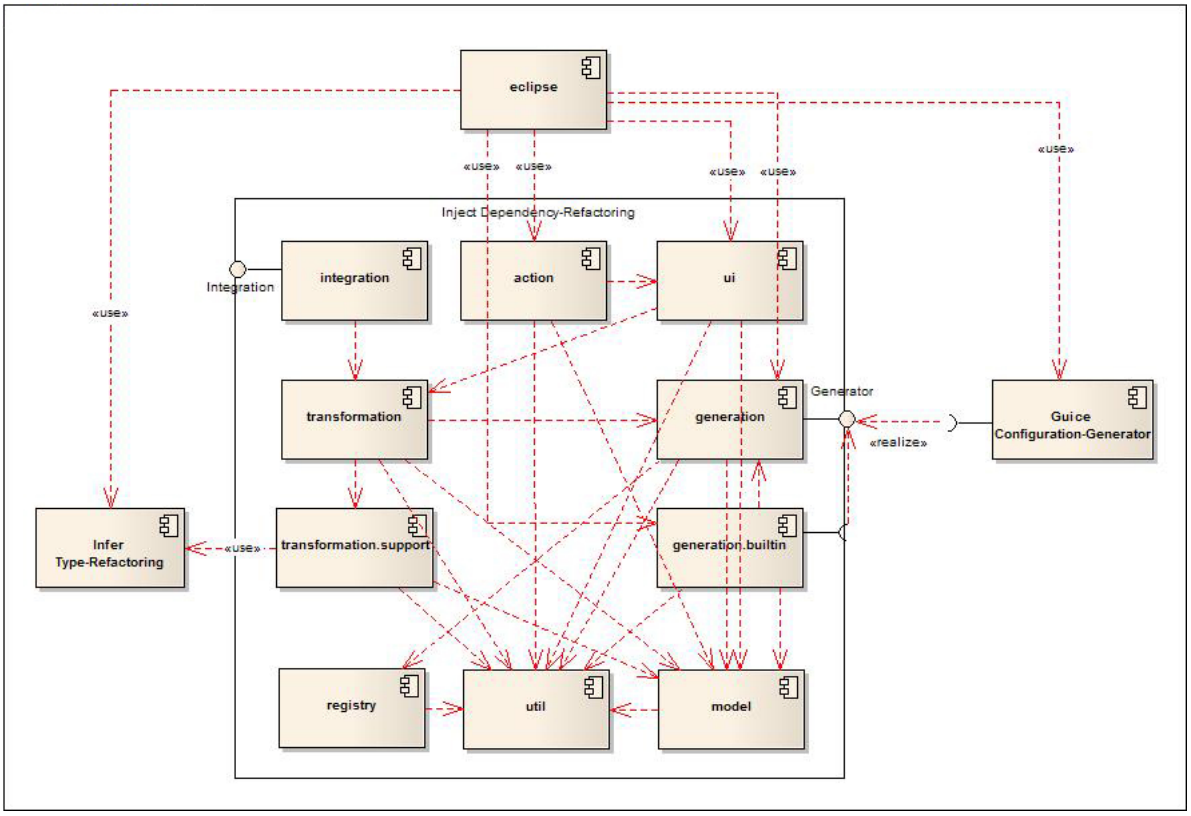

Abbildung 5.2: Architektur des Plug-In

Das Modul `model` enthält das Domänenmodell des Refactoring, durch welches ein Client und seine Services analysiert respektive repräsentiert werden. Da das Modell ebenfalls die Grundlage für die Transformation und die Generatoren darstellt, stützen sich die meisten anderen Module darauf.

Im verbleibenden Modul `util` befinden sich Hilfsklassen, wie beispielsweise das Logging, das von allen anderen Modulen genutzt wird.

5.4.2 Das Domänenmodell

Die konzeptionelle Basis des Refactoring bildet das domänenspezifische Modell, das sich aus dem Dependency Injection-Pattern ableitet. Dieses Modell ist in der Lage eine Client-Services-Konstellation auszudrücken, um darauf basierend Analysen zur Ermittlung möglicher Entkopplungen durchführen zu können.

Während es für das Dependency Injection-Pattern ausreichend ist, sich ausschließlich auf die Sprachkonstrukte Klasse, Schnittstelle, Objektattribut und Konstruktor zu stützen, ist diese Betrachtung für das Analysemodell zu grob. Für die Verfeinerung muss das Element Initialisierung hinzugefügt werden, desweiteren müssen die Elemente Klasse und Objektattribut durch Subtypen spezialisiert werden:

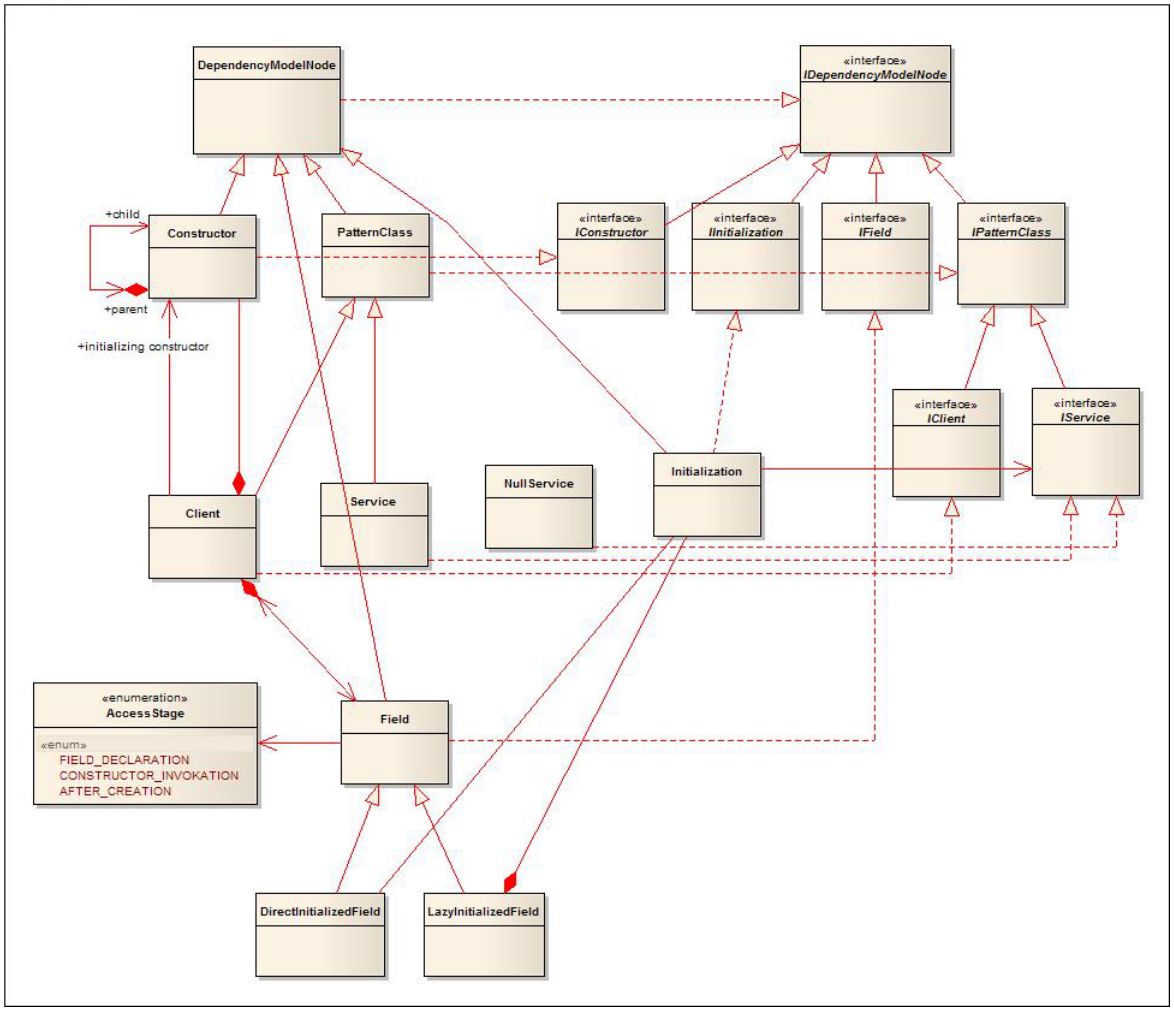

Abbildung 5.3: Das Domänenmodell des Refactoring

Die Typen des Models

Die Schnittstelle `IDependencyModelNode` ist die abstrakteste Ausprägung eines Elements des Analysemodells. Für die Elemente Konstruktor, Objektattribut, Initialisierung und „Klasse des Entwurfsmusters“ gibt es die Schnittstellen `IConstructor`, `IField`, `IInitialization` und `IPatternClass`, wobei letzteres nochmals zusätzlich durch die Schnittstellen `IClient` und `IService` verfeinert wird. Mit diesen Schnittstellen sind alle relevanten Elemente des Analysemodells abbildbar, wobei die Implementierung vollständig dahinter verborgen wird.

Die Klassen `DependencyModelNode`, `Constructor`, `Field`, `Initialization`, `PatternClass`, `Client` und `Service` implementieren die dementsprechenden Schnittstellen. Die Klasse `Field` wird zudem durch die beiden Klassen `DirectInitializedField` und `LazyInitializedField` spezialisiert, wobei erstere für die direkte Initialisierung und letztere für die verzögerte Initialisierung eines Objektattributs steht.

Die Klasse `NullServer` stellt einen Spezialfall dar. Diese Klasse realisiert das Null Object-Pattern[44] und wird dann eingesetzt, wenn die Initialisierung eines Objektattributs nicht durch den Einsatz des `new`-Operators erfolgt, sondern beispielsweise durch Reflection realisiert wird.

Durch die Enumeration `AccessStage` mit den deklarierten Instanzen `FIELD_DECLARATION`, `CONSTRUCTOR_INVOCATION` und `AFTER_CREATION` kann ein Objekt der Klasse `Field` die Information zur Verfügung stellen, ab welchem in Absatz 3.1 beschriebenen Zeitpunkt bereits auf das entsprechende Objektattribut zugegriffen wird.

Die Beziehungen unter den Modeltypen

Wird vom Inject Dependency-Refactoring aus einer Client-Services-Konstellation das Modell erzeugt, so bildet ein Objekt der Klasse `Client` die Wurzel.

Für jeden Konstruktor hält das `Client`-Objekt durch eine Collection eine Referenz zu einem `Constructor`-Objekt. Falls sich diese Konstruktoren untereinander mittels `this`-Operatoren aufrufen, so wird dies durch eine bidirektionale Beziehung zwischen den jeweiligen `Constructor`-Objekten ausgedrückt. Besitzt ein Client nur einen einzigen expliziten Konstruktor, dann hält das `Client`-Objekt eine direkte Referenz zu dem `Constructor`-Objekt, wobei letzteres in diesem Fall Initialisierungskonstruktor genannt wird.

Für jedes Objektattribut hält das `Client`-Objekt ebenfalls durch eine Collection eine Referenz zu einem `Field`-Objekt. Während ein Objekt der Klasse `DirectInitializedField` nur eine Referenz auf ein `Initialization`-Objekt hält, verweist ein Objekt der Klasse `LazyInitializedField` durch eine Collection auf mehrere

[44] Siehe [Mar02].

`Initialization`-Objekte: Ein Objektattribut kann an mehreren Stellen im Client initialisiert werden.

Ein Objekt der Klasse `Initialization` besitzt eine Referenz zu einem Service. Wird die Initialisierung durch den Aufruf eines `new`-Operators realisiert, dann zeigt die Referenz auf ein Objekt der Klasse `Service`. In allen anderen Fällen hält das `Initialization`-Objekt ein Objekt der Klasse `NullServer`.

5.4.3 Modellierung der Transformation

Durch die drei Dependency Injection-Methoden gibt es drei unterschiedliche Transformationsprozesse, die sich, wie bereits in Absatz 3.6.1 beschrieben, in einen gemeinsamen und einen individuellen Teil untergliedern lassen. Die gemeinsamen Transformationsschritte werden durch die Klasse `AbstractMethod` realisiert, die spezifischen Transformationsschritte in den spezialisierten Subklassen `SetterInjection`, `ConstructorInjection` und `InterfaceInjection`.

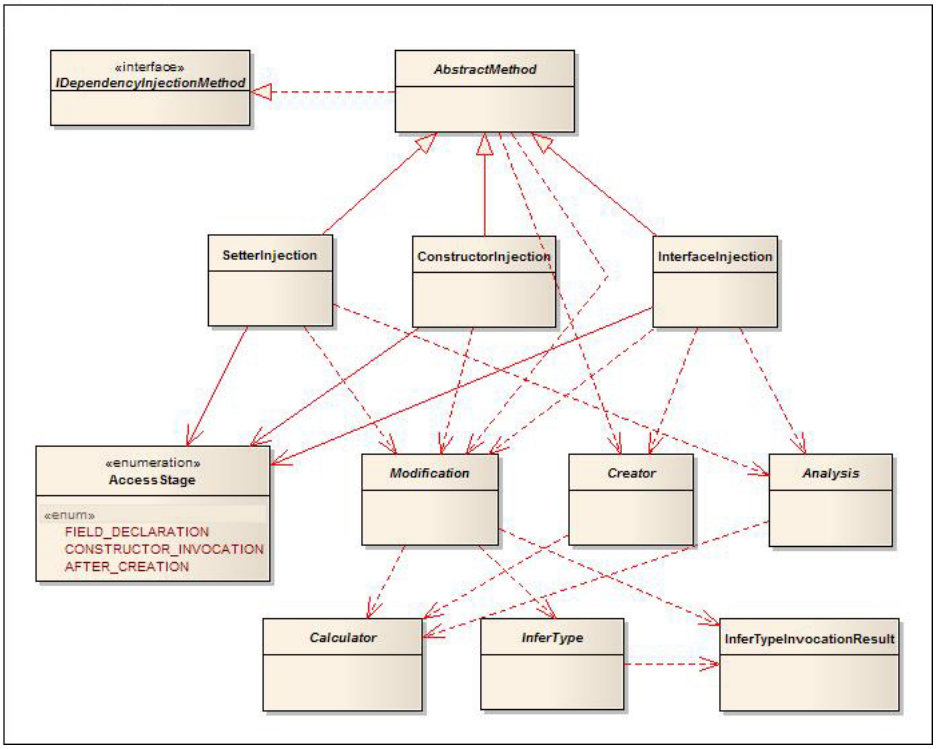

Abbildung 5.4: Modellierung der Transformation

Nach außen ist die Instanz einer Dependency Injection-Methode ausschließlich über die Schnittstelle `IDependencyInjectionMethod` verfügbar. Jede Methode stellt durch die Enumeration `AccessStage` die Information zur Verfügung, zu welchem Zeitpunkt durch sie ein zu einer Abhängigkeit gehörendes Objektattribut des Clients initialisiert wird.

Die Klassen `Analysis`, `Calculator`, `Creator`, `Infer Type` und `Modification` sind Funktionsbibliotheken mit den folgenden Aufgaben:

- **Analysis** - Die Funktionen dieser Klasse unterstützen bei der Ermittlung, ob eine Abhängigkeit durch eine gewählte Dependency Injection-Methode entkoppelt werden kann.
- **Calculator** - Diese Klasse hat die Aufgabe Namen von Parametern, Methoden und Typen zu berechnen, um Kollisionen3.4 durch bereits vorhandene Namen zu vermeiden.
- **Creator** - Die Klasse Creator ist in der Lage neue Typen, beispielsweise die Injection-Schnittstelle, zu erzeugen.
- **Infer Type** - Aufruf des Infer Type-Refactoring. Das Ergebnis der Ausführung wird als Objekt der Klasse `InferTypeInvocationResult` zurückgegeben. Dieses Objekt kapselt den Namen des neu generierten Typs und die zugehörige Modifikation als Instanz der Klasse `org.eclipse.ltk.core.refactoring.CompositeChange`.
- **Modification** - Sämtliche Transformationsschritte der drei Dependency Injection-Methoden werden in dieser Klasse realisiert und zur Verfügung gestellt.

Der Dependency Injection-Methode wird für die Durchführung der Transformation das Analysemodell, genauer gesagt die Objektattribute der zu entkoppelnden Abhängigkeiten, als Parameter übergeben. Zusammen mit den weiteren Angaben durch den Benutzer können die notwendigen Anpassungen des Codes durchgeführt werden.

5.4.4 Definition der Generator-Schnittstelle

Damit zu der Transformation die entsprechende Konfiguration erzeugt werden kann, gibt der Transformationsprozess alle relevanten Informationen an die Generator-Schnittstelle `IGenerator` weiter. Die Parameter der Methoden werden in Abbildung 5.5 aus Gründen der Übersichtlichkeit nicht dargestellt.

Durch die drei Methoden `supportsConstructorInjection`, `supportsSetterInjection` und
textttsupportsInterfaceInjection kann die implementierende Generatorklasse die Information zur Verfügung stellen, welche der drei Dependency Injection-Varianten durch sie unterstützt wird. Keines der verfügbaren Dependency Injection-Frameworks unterstützt derzeit alle drei Varianten.

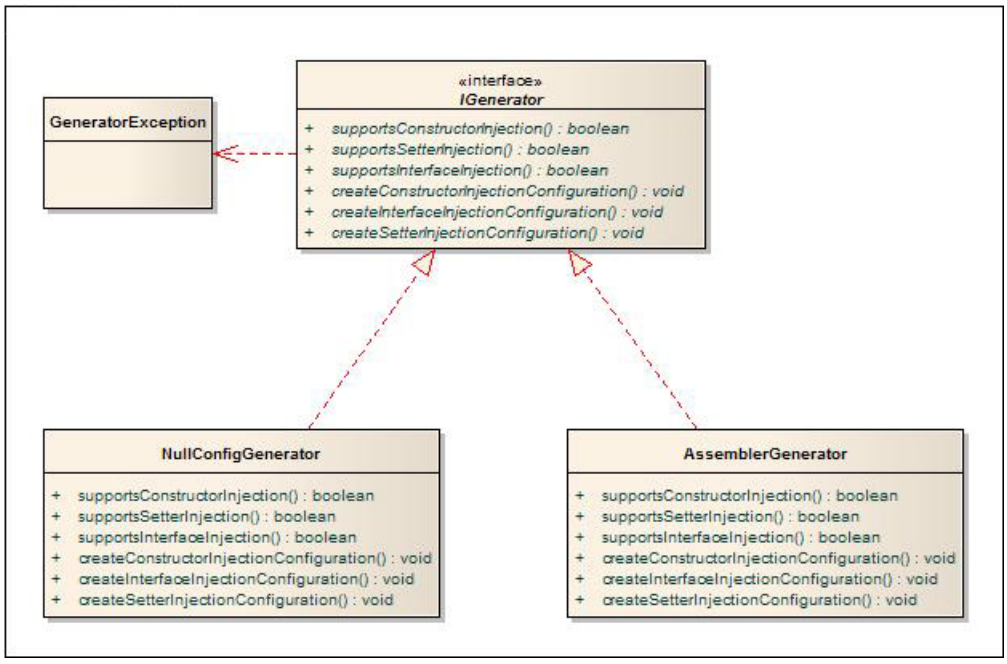

Abbildung 5.5: Modellierung der Generator-Schnittstelle

Durch die Methoden `createConstructorInjectionConfiguration`, `createSetterInjectionConfiguration` und `createInterfaceInjectionConfiguration` wird die Konfiguration der jeweils entsprechenden Dependency Injection-Variante generiert. Da manche Frameworks beispielsweise den relevanten Methoden und Konstruktoren Annotations anfügen müssen, wird den Generatoren der Zugriff auf das Analysemodell und das notwendige `CompilationUnitRewrite`-Objekt ermöglicht.

Das Inject Dependency-Plug-In stellt zudem bereits die zwei Implementierungen `AssemblerGenerator` und `NullConfigGenerator` zur Verfügung:

- **AssemblerGenerator** - Mit diesem Generator wird eine einfache Assemblerklasse generiert.

- **NullConfigGenerator** - Dieser Generator ist für den Fall, dass nur die Transformation ohne die Erzeugung von Konfiguration durchgeführt werden soll.

5.5 Implementierung

Für die Implementierung des Plug-In wurde in der PDE ein Plug-In Projekt mit dem Namen *InjectDependency Plug-In* erstellt. Durch die Integration des Infer Type-Refactoring bindet das Plug-In neben den Packages von eclipse und dem LTK zusätzlich das Package

`org.intoJ.inferType` ein. Das Inject Dependency-Plug-In selbst deklariert ein neues Package mit dem Namen `org.intoJ.injectDependency`.

Die Klasse `DependencyInjection` im Package `org.intoJ.injectDependency.model` ist die zentrale Stelle für die Überprüfung des Vorläufers und die Erzeugung des Analysemodells, auf welchem die Transformationen der Dependency Injection-Methoden ausgeführt werden können.

5.5.1 Überprüfung des Vorläufers

Da der Vorläufer sehr zügig durchgeführt werden muss, wird hierbei kein Analysemodell erzeugt. Die Methode `typeHasDependencyFields` führt deshalb eine schnelle oberflächliche Analyse durch:

```
1  public boolean typeHasDependencyFields(IType currentType) {
2    try {
3      IField[] fields = currentType.getFields();
4      for (IField field : fields) {
5        if (isNotDeclaredStatic(field)
6            && (isDeclaredFinal(field) || isConceptualFinal(field))
7            && isDeclaredWithClassType(field))
8          return true;
9      }
10   }
11   catch (JavaModelException ex) {
12     RefactoringLog.logError(ex);
13   }
14
15   return false;
16 }
```

Diese Überprüfung ist erfolgreich, wenn mindestens ein Objektattribut gefunden wurde, das prinzipiell eine Referenz zu einer Abhängigkeit speichern könnte. Bei einer erfolgreichen Überprüfung kann der Benutzer das Refactoring starten, da der Menüpunkt *Inject Dependency* anklickbar ist.

5.5.2 Erzeugung des Analysemodells

Das Analysemodell wird von der Klasse `DependencyInjection` durch die Methode `buildFrom` erzeugt und geliefert. Diese Methode erzeugt ein Objekt der Klasse

`DependencyFieldsResolver`, die eine Spezialisierung der Klasse `ASTVisitor`[45] ist. Der `DependencyFieldsResolver` und die Typen des Package `org.eclipse.jdt.core.dom` implementieren zusammen das Visitor-Pattern[46] und ermöglichen hierdurch die Ermittlung der für das Analysmodell relevanten Elemente.

Der Visitor beginnt die Traversierung des Objektgraphen bei dem Parameter `IType currentType`, der somit als der Client des Dependency Injection-Patterns behandelt wird. Für den Parameter erzeugt der Visitor ein Wrapper-Objekt der Klasse `Client`, und fügt diesem für jedes gefundene Objektattribut und jeden gefundenen Konstruktor ein Objekt der Klasse `Field` respektive `Constructor` hinzu. Die erzeugten Objekte kümmern sich anschließend selbst um die Erzeugung weiterer Objekte des Analysemodells. So ist es beispielsweise die Aufgabe der `Field`-Objekte die entsprechenden `Initialization`-Objekte zu erzeugen.

Jedes Element des Analysemodells hat die Aufgabe sich selbst zu überprüfen. Das Ergebnis einer Überprüfung wird durch die Enumerationen `MessageType` und `Message` ausgedrückt.

```
1  public enum MessageType {
2    INFO, ERROR;
3  }
4
5  public enum Message {
6    NOT_NEW_OPERATOR(MessageType.ERROR),
7    IS_NEW_OPERATOR(MessageType.INFO),
8    IS_NOT_CLASS(MessageType.ERROR),
9    JAVA_MODEL_EXCEPTION(MessageType.ERROR),
10   IS_NOT_PUBLIC(MessageType.ERROR),
11   IS_ABSTRACT(MessageType.ERROR),
12   IS_NOT_TOPLEVEL(MessageType.ERROR),
13   IS_NOT_FINAL(MessageType.ERROR),
14   IS_STATIC(MessageType.ERROR),
15   IS_NOT_SIMPLETYPE(MessageType.ERROR),
16   IS_NOT_DETERMINISTIC(MessageType.ERROR),
17   IS_NOT_SINGLE_DECLARATION(MessageType.ERROR),
18   CORE_EXCEPTION(MessageType.ERROR),
19   HAS_NON_CONSTANT_ARGUMENT(MessageType.ERROR),
20   IS_BINARY(MessageType.ERROR),
21   NO_INITIALIZATION_CONSTRUCTOR(MessageType.ERROR),
```

[45] Siehe [Kuh08].
[46] Siehe [GHJV95].

```
22    HAS_INITIALIZATION_CONSTRUCTOR(MessageType.INFO),
23    HAS_ARGUMENTS(MessageType.ERROR),
24    CANNOT_INFER_TYPE(MessageType.ERROR),
25    SOLVED_INFER_TYPE_FALSE_ERROR(MessageType.INFO);
26
27    private final MessageType type;
28
29    private Message(MessageType type) {
30      this.type = type;
31    }
32
33    public MessageType getType() {
34      return type;
35    }
36  }
```

Für das Analysemodell sind für die Ergebnisse der Überprüfungen dementsprechende Nachrichten modelliert. Jedes Element des Analysemodells ist eine Spezialisierung der abstrakten Klasse `DependencyModelNode`, die das Speichern und die Auswertung der Überprüfungsergebnisse implementiert und nach außen verfügbar macht:

```
1   public abstract class DependencyModelNode
2       implements IDependencyModelNode {
3     private final List<Message> msgs = new ArrayList<Message>();
4
5     public List<Message> getMessages() {
6       return Collections.unmodifiableList(msgs);
7     }
8
9     public boolean isValid() {
10      for (Message msg : msgs) {
11        if (msg.getType().compareTo(MessageType.ERROR) == 0)
12          return false;
13      }
14      return true;
15    }
16
17    protected void addMessage(Message msg) {
18      msgs.add(msg);
19    }
```

```
20
21    //...
22  }
```

Die Klasse `DependencyModelNode` ermöglicht es ihren Subtypen durch die Methode `addMessage` das Ergebnis einer Überprüfung zu speichern. Von außen kann jedes Element somit überprüft werden, ob es hinsichtlich der Analyse gültig ist (Methode `isValid`) und auf welchen Ergebnissen die Gültigkeit ermittelt wurde (Methode `getMessages`).

Durch diesen Ansatz ist es beispielsweise für den Refactoring-Dialog möglich, zwischen gültigen und ungültigen Objektattributen unterscheiden zu können. Gültige Objektattribute können für den Einsatz des Dependency Injection-Patterns ausgewählt werden, ungültige hingegen nicht, da diese ausgegraut dargestellt werden.

Die Überprüfung der benutzerunabhängigen Vorbedingungen stützt sich ebenfalls auf die Gültigkeit der Elemente des Analysemodells. Sollte sich bei der Überprüfung der benutzerunabhängigen Vorbedingungen herausstellen, dass es keine entkoppelbaren Abhängigkeiten gibt, so wird dies dem Benutzer durch einen entsprechenden Informations-Dialog angezeigt und das Refactoring wird nicht gestartet. Diese Überprüfung wird vom Analysemodell selbst durchgeführt, genauer gesagt durch eine Klasse `DependenciesResolver`, die das Modell traversieren und Überprüfungen durchführen kann. Der `DependenciesResolver` und die Typen des Analysmodells implementieren zusammen das Visitor-Pattern[47]:

```
1  public class Client extends PatternClass implements IClient,
2      IModelCreatedObserver {
3    //...
4    public boolean decouplingIsPossible() {
5      DependenciesResolver visitor = new DependenciesResolver();
6      this.accept(visitor);
7      return visitor.foundDependencies()
8        && !visitor.foundProblematicConstructors();
9    }
10   //...
11 }
```

Die Überprüfung ist erfolgreich, wenn der Visitor entkoppelbare Abhängigkeiten finden konnte und der Client keine Konstruktoren besitzt, die eine Objekterzeugung durch den Assembler verhindern würden.

[47] Siehe [GHJV95].

5.5.3 Auswahl der Dependency Injection-Methode

Nach der Auswahl des von dem Inject Dependency-Refactoring zu berücksichtigenden Objektattributs im Refactoring-Dialog, muss vom Benutzer eine Dependency Injection-Methode gewählt werden. Abhängig von der Wahl der Objektattribute sind nicht immer alle Methoden einsetzbar, da sich die in Absatz 3.1 eingeführten Initialisierungszeitpunkte und Zugriffszeitpunkte gegenseitig ausschließen können. Diese Überprüfungen werden von der Klasse `AbstractMethod` des Package `org.intoJ.injectDependency.transformation`, beziehungsweise deren Subklassen `SetterInjection`, `InterfaceInjection` und `ConstructorInjection`, realisiert:

```
1  public abstract class AbstractMethod
2      implements IDependencyInjectionMethod {
3    private final AccessStage maxAccessStageSupport;
4    //...
5
6    public AbstractMethod(AccessStage maxAccessStageSupport) {
7      this.maxAccessStageSupport = maxAccessStageSupport;
8    }
9
10   public abstract boolean canRefactor(IField field)
11     throws CoreException;
12
13   public final boolean canHandleAccessStageOf(IField field) {
14     return (!field.getAccessStage()
15       .isEarlierAs(maxAccessStageSupport));
16   }
17   //...
18 }
```

Durch die Subklassen wird dem Konstruktor der Klasse `AbstractMethod` der Parameter `AccessStage maxAccessStageSupport` übergeben. Diese Angabe wird innerhalb der Methode `canHandleAccessStageOf` verwendet um herauszufinden, ob die Dependency Injection-Methode für ein Objektattribut genutzt werden kann.

Nachdem alle Benutzereingaben im Refactoring-Dialog angegeben wurden ist es notwendig die benutzerabhängigen Vorbedingungen durchzuführen. Es ist möglich, dass die Namen neuer Deklarationselemente (Methoden respektive Schnittstellen) bereits in Verwendung sind und deshalb zu einer Namenskollision, wie in Absatz 3.4 beschrieben, führen würden. Die Ermittlung der Kollisionen wird direkt von den Subtypen der Klasse `AbstractMethod` durchgeführt. Hierzu wird die abstrakte Methode `canRefactor` genutzt,

die von jeder Subklasse implementiert wird.

5.5.4 Durchführung der Transformation

Bevor die eigentlichen Transformationsschritte der gewählten Dependency Injection-Methode ausgeführt werden können, wird Infer Type eingesetzt, um für die Abstraktion des Service eines jeden Objektattributs eine Schnittstelle zu erzeugen. Für die programmatische Durchführung des Infer Type-Refactoring wird eine Klasse `ExternalInferTypeRunnable` zur Verfügung gestellt, die für eine Integration durch ein anderes Refactoring nicht einsetzbar ist. Der Einsatz der Klasse `ExternalInferTypeRunnable` bringt die folgenden Nachteile mit sich:

- Infer Type kann auf unterschiedliche Deklarationselemente angewandt werden, beispielsweise auf Objektattribut-, Parameter- oder Klassenebene. Wird Infer Type auf Ebene von Objektattributen eingesetzt, so kann durch `ExternalInferTypeRunnable` nur ein einziges Objektattribut behandelt werden.

- Der Einsatz von `ExternalInferTypeRunnable` berechnet nicht nur die durchzuführenden Modifikationen des Codes, sondern führt diese sofort durch. Würde ein Benutzer während des Dialogs des Inject Dependency-Refactoring bei der Vorschau auf *Cancel* klicken, so würden zwar die berechneten Modifikationen durch Inject Dependency verworfen werden, diejenigen von Infer Type jedoch nicht.

- Sowohl das Inject Dependency-Refactoring als auch das Infer Type-Refactoring würden gleichzeitig Modifikationen am Client durchführen. Damit durch die Modifikationen kein korrupter Code entsteht, muss das Inject Dependency-Refactoring die Kontrolle über die Modifikationen des Infer Type-Refactoring haben. Dies ist beim Einsatz von `ExternalInferTypeRunnable` jedoch nicht möglich.

Aus diesem Grund verwendet das Inject Dependency-Refactoring nicht die Klasse `ExternalInferTypeRunnable`, sondern arbeitet direkt mit dessen internen API. Die Klasse `InferType` im Package `org.intoJ.injectDependency.transformation.support` realisiert hierbei für das Inject Dependency-Refactoring einen Adapter zu Infer Type:

```
1 public abstract class InferType {
2   //...
3
4   public static InferTypeInvocationResult createChange(IField field,
5       String newTypeName)
```

```
6         throws OperationCanceledException, CoreException {
7       InferTypeRefactoring refactoring =
8         getInferTypeRefactoring(field);
9       refactoring.setChangeTypeArguments(true);
10      refactoring.setNoAbstractClasses(false);
11      refactoring.setRedeclareDEs(true);
12      refactoring.checkInitialConditions(new NullProgressMonitor());
13      refactoring.setNewTypeName(newTypeName);
14      refactoring.setForceNew(true);
15      refactoring.checkFinalConditions(new NullProgressMonitor());
16      return new InferTypeInvocationResult(
17        (CompositeChange) refactoring.createChange(
18          new NullProgressMonitor()), refactoring.getNewTypeName());
19    }
20
21    private static InferTypeRefactoring getInferTypeRefactoring(
22        IField field) {
23      VariableDeclarationFragment fragment =
24        ((VariableDeclarationFragment) field.getFieldDeclaration()
25          .fragments().get(0));
26      SelectedElement selectedElement = new SelectedElement(
27        fragment.resolveBinding(),
28        field.getClient().getTypeDeclaration().resolveBinding(),
29        field.getFieldDeclaration().getType().getStartPosition(),
30          null);
31      return new InferTypeRefactoring(selectedElement, true);
32    }
33
34    //...
35  }
36
37  public class InferTypeInvocationResult {
38    private final CompositeChange change;
39    private final String newTypeName;
40
41    public InferTypeInvocationResult(CompositeChange change,
42        String newTypeName) {
43      this.change = change;
44      this.newTypeName = newTypeName;
45    }
```

```
46
47    public CompositeChange getChange() {
48      return change;
49    }
50
51    public String getNewTypeName() {
52      return newTypeName;
53    }
54  }
```

Der Methode `createChange` wird das Objektattribut übergeben, zu dessen Service eine Schnittstelle durch Infer Type berechnet und eingeführt werden soll. Im Parameter `newTypeName` befindet sich der Name der neuen Schnittstelle, die von Infer Type erzeugt wird. Das Ergebnis des Infer Type-Refactoring wird als Objekt der Klasse `InferTypeInvocationResult` zurückgegeben, das als Wrapper für den Namen der neuen Schnittstelle und der zugehörigen Modifikation dient.

Für die Integration des Infer Type-Refactoring sind zwei besondere Objekte notwendig:

```
1  //...
2  final Map<String, IType> fieldTypeReplacements =
3    new HashMap<String, IType>();
4  final Map<ICompilationUnit, String> serviceBuffers =
5    new HashMap<ICompilationUnit, String>();
6  //...
```

Die Variable `fieldTypeReplacements` speichert die neuen Deklarationstypen der Objektattribute, da die Ersetzung dieser Typen nicht mehr durch Infer Type realisiert wird. In der Variable `serviceBuffers` wird für jede `ICompilationUnit` einer Service-Klasse der initiale Zeicheninhalt des Buffer[48] gespeichert, weil die Service-Klasse während des Refactoring prinzipiell mehrfach modifiziert werden kann. Bei jedem Aufruf von Infer Type wird der aktuelle Buffer der Service-Klasse modifiziert, damit die `Change`-Objekte aufeinander aufbauen. Ohne diese Maßnahme würden alle Infer Type-Aufrufe vom Initialzustand des Service-Buffer ausgehen und deren Modifikationen würden den Code in einen korrupten Zustand überführen.

Zunächst wird für jedes gewählte Objektattribut Infer Type aufgerufen und das Objekt `CompositeChange change` des `InferTypeInvocationResult` an das Parameter-Objekt `unitChanges` übergeben.

[48] Siehe [Ecl08].

```
1  //...
2  for (DecouplingSpecification spec : specs) {
3    //...
4    IField field = client.resolveField(spec.getFieldName());
5    InferTypeInvocationResult result =
6      Modification.inferTypeFor(field, spec.getNewTypeNames()[0]);
7    unitChanges.add(result.getChange());
8    //...
9    }
10 //...
```

Das `CompositChange`-Objekt der Infer Type-Ausführung ist eine Sammlung von `Change`-Objekten, die anschließend einzeln überprüft werden:

- Ist das `Change`-Objekt die Modifikation der Client-Klasse, wird dieses durch `result.getChange().remove(change)` aus dem `CompositeChange`-Objekt entfernt, da nur das Inject Dependency-Refactoring den Client anpassen darf.
- Ist das `Change`-Objekt die Modifikation der Service-Klasse, wird die Modifikation sofort ausgeführt und durch `result.getChange().remove(change)` aus dem `CompositeChange`-Objekt entfernt. Ist in der Variable `serviceBuffers` der initiale Zeicheninhalt der Service-Klasse noch nicht gespeichert, wird dieser vor dem ersten Aufruf von Infer Type gesetzt.
- Ist das `Change`-Objekt die neu erzeugte Schnittstelle, wird deren Code der fehlende Name des eigenen Packages hinzugefügt. Anschließend wird in der Variable `fieldTypeReplacements` der neue Deklarationstyp des zugehörigen Objektattributs gesetzt.

Nach den Aufrufen des Infer Type-Refactoring wird für jedes Objektattribut die ausgewählte Dependency Injection-Methode eingesetzt:

```
1  //...
2  CompilationUnitRewrite compilationUnitRewrite
3    = new CompilationUnitRewrite(iClient.getICompilationUnit(),
4      iClient.getCompilationUnit());
5  for (DecouplingSpecification spec : decouplingSpecifications)
6    spec.getMethod().refactor(spec, iClient.getICompilationUnit(),
7      iClient.getSimpleName(), unitChanges, generator,
8      fieldTypeReplacements, serviceBuffer, iClient,
```

```
9       compilationUnitRewrite);
10  unitChanges.add(compilationUnitRewrite.createChange());
11  //...
```

Hierbei wird für den Client ein `CompilationUnitRewrite`-Objekt erzeugt, das zusammen mit allen relevanten Parametern an die Methode `refactor` übergeben werden. Die hierbei berechneten Transformationsschritte werden anschließend dem `unitChanges`-Objekt hinzugefügt.

Nach der Durchführung aller Transformationsschritte werden `Change`-Objekte erzeugt, die den Buffer der Service-Klasse vom Initialzustand in den Endzustand überführen können:

```
1   //...
2   for (ICompilationUnit serviceICU : serviceBuffer.keySet()) {
3     TextFileChange textFileChange =
4       new TextFileChange("Update ' " + serviceICU.getElementName()
5         + "' - " + serviceICU.getPath().toString(),
6         (IFile)serviceICU.getResource());
7     MultiTextEdit multiTextEdit = new MultiTextEdit();
8     multiTextEdit.addChild(new ReplaceEdit(0,
9       serviceBuffer.get(serviceICU).length(),
10      serviceICU.getBuffer().getContents()));
11    textFileChange.setEdit(multiTextEdit);
12    unitChanges.add(textFileChange);
13  }
14  //...
```

Diese `TextFileChange`-Objekte werden ebenfalls dem `unitChanges`-Objekt hinzugefügt.

Zur Verschmelzung der Transformationen des Infer Type- und des Inject Dependency-Refactoring wird ein `CompositeChange`-Objekt erstellt und befüllt:

```
1   //...
2   CompositeChange compositeChange = new CompositeChange(getName());
3   for (Change change : unitChanges) {
4     if (change instanceof CompositeChange)
5       compositeChange.merge((CompositeChange) change);
6     else
7       compositeChange.add(change);
8   }
```

```
9 //...
```

Die `CompositeChange`-Objekte des Infer Type-Refactoring müssen durch die `merge`-Methode hinzugefügt werden. Alle anderen Objekte können hingegen durch `add` hinzugefügt werden. Dieses `CompositeChange`-Objekte stellt die endültige Transformation des Inject Dependency-Refactoring dar.

5.5.5 Aktivierung des Generators

Damit es der Klasse `AbstractMethod` möglich ist, während der Transformation den Generator anzuweisen eine Konfiguration zu erzeugen, ist die Zuhilfenahme der `Registry` notwendig. Diese Klasse ist nicht nur in der Lage, die von eclipse geladenen Generatoren zu ermitteln, sondern auch die vom Benutzer gewählte Generator-Auswahl zur Verfügung zu stellen:

```
1  public class Registry implements IRegistry {
2    private static final String ATTRIBUTE_CLASS = "class";
3    private static final String ATRIBUTE_NAME = "name";
4    private static final String ATTRIBUTE_ID = "id";
5
6    private final static String NAMESPACE
7      = "org.intoJ.injectDependency";
8    private final static String EXTENSION_POINT_NAME
9      = "generators";
10   private final static String SELECTED_GENERATOR_STORE_KEY
11     = "injectDependency.selectedGenerator";
12
13   private Map<String, GeneratorInfo> infoMap
14     = new HashMap<String, GeneratorInfo>();
15
16   public Registry() {
17     init();
18   }
19
20   public Collection<GeneratorInfo> getGeneratorInfos() {
21     return infoMap.values();
22   }
23
24   public void storeSelectedGenerator(GeneratorInfo info) {
25     IPreferenceStore store = getPreferenceStore();
```

```
26      store.setValue(SELECTED_GENERATOR_STORE_KEY, info.getId());
27    }
28
29    public GeneratorInfo restoreSelectedGenerator() {
30      IPreferenceStore store = getPreferenceStore();
31      String id = store.getString(SELECTED_GENERATOR_STORE_KEY);
32      return (id == "" ? null : infoMap.get(id));
33    }
34
35    public void resetSelectedGenerator() {
36      getPreferenceStore().setToDefault(SELECTED_GENERATOR_STORE_KEY);
37    }
38
39    private void init() {
40      IExtension[] extensions =
41        Platform.getExtensionRegistry()
42          .getExtensionPoint(NAMESPACE, EXTENSION_POINT_NAME)
43          .getExtensions();
44
45      boolean first = true;
46      for (IExtension extension : extensions) {
47        IConfigurationElement configuration =
48          extension.getConfigurationElements()[0];
49        GeneratorInfo generatorInfo =
50          new GeneratorInfo(configuration.getAttribute(ATTRIBUTE_ID),
51            configuration.getAttribute(ATRIBUTE_NAME),
52            configuration.getAttribute(ATTRIBUTE_CLASS),
53              configuration);
54        infoMap.put(generatorInfo.getId(), generatorInfo);
55
56        if (first)
57          setDefaultGenerator(generatorInfo);
58
59        first = false;
60      }
61
62    }
63
64    private void setDefaultGenerator(GeneratorInfo info) {
65      IPreferenceStore store = getPreferenceStore();
```

```
66      store.setDefault(SELECTED_GENERATOR_STORE_KEY, info.getId());
67    }
68
69    private IPreferenceStore getPreferenceStore() {
70      return InjectDependencyPlugin.getDefault().getPreferenceStore();
71    }
72  }
73
74  public class GeneratorInfo {
75    private final String id;
76    private final String name;
77    private final String className;
78    private final IConfigurationElement configuration;
79
80    public GeneratorInfo(String id, String name, String className,
81        IConfigurationElement configuration) {
82      this.id = id;
83      this.name = name;
84      this.className = className;
85      this.configuration = configuration;
86    }
87
88    public String getId() {
89      return id;
90    }
91
92    public String getName() {
93      return name;
94    }
95
96    public String getClassName() {
97      return className;
98    }
99
100   public IConfigurationElement getConfiguration() {
101     return configuration;
102   }
103
104   //...
105 }
```

```
106
107 public class GeneratorFactory {
108   private static final String EXECUTABLE_EXTENSION_ATTR_NAME
109         = "class";
110   private final IRegistry registry = new Registry();
111
112   public IGenerator createGenerator() throws CoreException {
113     GeneratorInfo info = registry.restoreSelectedGenerator();
114     return (IGenerator) info.getConfiguration()
115       .createExecutableExtension(EXECUTABLE_EXTENSION_ATTR_NAME);
116   }
117
118 }
```

Jeder verfügbare Generator wird durch ein `GeneratorInfo`-Objekt repräsentiert. Neben der Ermittlung der verfügbaren Generator-Klassen hat die Klasse `Registry` die Aufgabe die Benutzerauswahl eines bestimmten Generators zu speichern. Der Benutzer hat die Möglichkeit durch den Preferences-Dialog einen Generator auszuwählen, der dann bei jeder Ausführung des Inject Dependency-Refactoring für die Erzeugung von Konfigurationscode eingesetzt wird. Wurde vom Benutzer kein Generator explizit ausgewählt, so wird standardmäßig die Klasse `NullConfigGenerator` eingesetzt.

Die Erzeugung des `Generator`-Objekts wird in der Methode `createGenerator` realisiert. Das `Configuration`-Objekt ist ein eclipse-spezifischer Typ, der in der Lage ist ein Objekt von einer durch den Plug-In Mechanismus eingebunden Klasse zu erzeugen.

6 Realisierung eines Generators

Die Integration des Generators für ein zu unterstützendes Dependency Injection-Framework wurde am Beispiel Guice realisiert. Das Inject Dependency Plug-In sieht hierfür die Generator-Schnittstelle `IGenerator` aus dem Package `org.intoJ.injectDependency.generation` vor, die aus diesem Grund vom Inject Dependency Guice Plug-In eingesetzt wird.

Damit ein anderes Plug-In diese Schnittstelle implementieren und die Generatorklasse in eclipse einbinden kann, stellt das Inject Dependency Plug-In den Extension Point *generators* zur Verfügung. Die als *Exported Packages* markierten Packages `org.intoJ.injectDependency.generation` und `org.intoJ.injectDependency.model` stellen sicher, dass der Generator auf alle notwendigen Typen zugreifen kann.

6.1 Einschränkung der Unterstützung von Dependency Injection-Varianten

Guice unterstützt ausschließlich die Dependency Injection-Varianten Setter Injection und Construktor Injection. Die Klasse `GuiceConfigurationGenerator` des Package `org.intoJ.injectDependency.guice.generation` implementiert die Generator-Schnittstelle deshalb wie folgt:

```
1  public class GuiceConfigurationGenerator implements IGenerator {
2    //...
3
4    public boolean supportsConstructorInjection() {
5      return true;
6    }
7
8    public boolean supportsInterfaceInjection() {
9      return false;
10   }
11
12   public boolean supportsSetterInjection() {
```

```
13      return true;
14    }
15
16    //...
17  }
```

Da die Dependency Injection-Variante Interface Injection nicht unterstützt wird, liefert die Methode `supportsInterfaceInjection` `false` zurück, während die übrigen zur Signalisierung der Unterstützung der anderen beiden Varianten jeweils `true` zurückgeben.

6.2 Erzeugung der Konfiguration für Guice

Guice sieht für die Konfiguration ausschließlich Konfigurationscode vor. Zur Modularisierung des Konfigurationscode stellt Guice die Schnittstelle `Module` aus dem Package `com.google.inject.Module` vor, die von der Generatorklasse genutzt wird. Der Programmcode der Konfigurationsklasse wird in der Methode `createConfiguration` erzeugt:

```
1  public class GuiceConfigurationGenerator implements IGenerator {
2    //...
3
4    private static final void createConfiguration(
5        List<Change> unitChanges, IJavaProject javaProject,
6        String encoding, String fqName, IClient client, IField field)
7         throws JavaModelException {
8      //...
9
10     StringBuilder builder = new StringBuilder();
11     if (!packageName.isEmpty())
12       builder.append("package " + packageName + ";");
13     if (!client.getPackageName().equals(packageName))
14       builder.append("import " + client.getFullyQualifiedName()
15         + ";");
16     if (!service.getPackageName().equals(packageName))
17       builder.append("import " + service.getFullyQualifiedName()
18         + ";");
19     if (!field.getIType().getPackageFragment().getElementName()
20         .equals(packageName))
21       builder.append("import " + field.getIType()
```

```
22            .getFullyQualifiedName() + ";");
23        builder.append("import␣com.google.inject.Binder;");
24        builder.append("import␣com.google.inject.Module;");
25        builder.append("import␣com.google.inject.Provider;");
26        //builder.append("import com.google.inject.Scopes;");
27        builder.append("public␣class␣" + typeName
28          + "␣implements␣Module,␣Provider<"
29          + field.getIType().getElementName() + ">␣{");
30        builder.append("public␣void␣configure(Binder␣binder)␣{");
31        builder.append("binder.bind(" + field.getIType().getElementName()
32          + ".class).toProvider(" + typeName + ".class);");
33        builder.append("}");
34        builder.append("public␣" + field.getIType().getElementName()
35          + "␣get()␣{");
36        builder.append("return␣new␣" + service.getSimpleName()
37          + "(");
38        for (Expression expression : expressions) {
39          if (expression instanceof CharacterLiteral)
40            builder.append("'"
41              + expression.resolveConstantExpressionValue() + "'");
42          else if (expression instanceof StringLiteral)
43            builder.append("\""
44              + expression.resolveConstantExpressionValue() + "\"");
45          else
46            builder.append(expression.resolveConstantExpressionValue());
47        }
48        builder.append(");");
49        builder.append("}");
50        builder.append("}");
51        String content = builder.toString();
52
53        //...
54      }
55
56      //...
57  }
```

Der Konfigurationscode wird durch ein `StringBuilder`-Objekt erzeugt. Die relevanten Typen und Packages der beteiltigten Client- und Server-Klasse werden an den entsprechenden Stellen im Zeicheninhalt eingefügt. Desweiteren wird der Konstruktoraufruf des

Service innerhalb einer `for`-Schleife mit den notwendigen Literalen als Parameter ergänzt. An dieser Stelle im Konfigurationscode wird der durch das Dependency Injection-Pattern eliminierte Konstruktoraufruf kompensiert.

6.3 Einführung von Annotations

Der erzeugte Konfigurationscode unterscheidet sich bei den beiden Varianten Setter Injection und Constructor Injection nicht, da Guice die Service-Objekte durch die Auflösung von Schnittstellen-Typen vornimmt. Jedoch benötigt Guice hierbei einen Hinweis, an welchen Stellen eines Client-Objekts notwendige Service-Objekte ermittelt und injiziert werden sollen:

```
1  public class GuiceConfigurationGenerator implements IGenerator {
2    private static final String INJECT_DEPENDENCY_GUICE_ID
3      = "org.intoJ.injectDependency.transformation.guice";
4    private final static GroupCategorySet INJECT_DEPENDENCY_GUICE_SET
5      = new GroupCategorySet(new GroupCategory(
6        INJECT_DEPENDENCY_GUICE_ID, "Inject␣Dependency␣Guice",
7        "Inject␣Dependency␣Guice"));
8    private static final String GUICE_INJECT_TYPE_NAME
9      = "com.google.inject.Inject";
10   private final List<MethodDeclaration> annotatedConstructors
11     = new ArrayList<MethodDeclaration>();
12
13   //...
14
15   public void createConstructorInjectionConfiguration(
16       CompilationUnitRewrite compilationUnitRewrite,
17       List<Change> unitChanges, IJavaProject javaProject,
18       String encoding, String fqName, IClient client, IField field,
19       MethodDeclaration methodDeclaration)
20       throws GeneratorException {
21     try {
22       if (!annotatedConstructors.contains(methodDeclaration)) {
23         addInjectAnnotation(compilationUnitRewrite,
24           methodDeclaration);
25         annotatedConstructors.add(methodDeclaration);
26       }
27       createConfiguration(unitChanges, javaProject, encoding, fqName,
```

```
28          client, field);
29        }
30        catch (Exception ex) {
31          throw new GeneratorException(ex);
32        }
33      }
34
35      public void createSetterInjectionConfiguration(
36          CompilationUnitRewrite compilationUnitRewrite,
37          List<Change> unitChanges, IJavaProject javaProject,
38          String encoding, String fqName, IClient client, IField field,
39          MethodDeclaration methodDeclaration)
40          throws GeneratorException {
41        try {
42          addInjectAnnotation(compilationUnitRewrite, methodDeclaration);
43          createConfiguration(unitChanges, javaProject, encoding, fqName,
44            client, field);
45        }
46        catch (Exception ex) {
47          throw new GeneratorException(ex);
48        }
49      }
50
51      private static void addInjectAnnotation(
52          CompilationUnitRewrite compilationUnitRewrite,
53          MethodDeclaration methodDeclaration)
54          throws JavaModelException {
55        boolean missing = true;
56        List<IExtendedModifier> modifiers =
57          methodDeclaration.modifiers();
58        for (IExtendedModifier extendedModifier : modifiers) {
59          if (extendedModifier.isAnnotation()) {
60            Annotation annotation = (Annotation) extendedModifier;
61            if (annotation.resolveTypeBinding().getQualifiedName()
62                .equals(GUICE_INJECT_TYPE_NAME)) {
63              missing = false;
64              break;
65            }
66          }
67        }
```

```
68
69      if (missing) {
70        ASTRewrite astRewrite = compilationUnitRewrite.getASTRewrite();
71        AST ast = compilationUnitRewrite.getAST();
72        NormalAnnotation inject = ast.newNormalAnnotation();
73        inject.setTypeName(ast.newSimpleName(
74          getTypeName(GUICE_INJECT_TYPE_NAME)));
75        ListRewrite listRewrite = astRewrite.getListRewrite(
76          methodDeclaration, MethodDeclaration.MODIFIERS2_PROPERTY);
77        listRewrite.insertFirst(inject, compilationUnitRewrite
78          .createCategorizedGroupDescription("Add final modifier",
79            INJECT_DEPENDENCY_GUICE_SET));
80        compilationUnitRewrite.getImportRewrite()
81          .addImport(GUICE_INJECT_TYPE_NAME);
82      }
83    }
84
85    //...
86  }
```

Dieser Hinweis wird bei Guice durch die Annotation `@Inject` realisiert. Die Methode `addInjectAnnotation` ist in der Lage einer Methode bzw. einem Konstruktor diese Annotation einzuführen.

Sowohl die Methode `createSetterInjectionConfiguration`, als auch die Methode `createConstructorInjectionConfiguration` setzen `addInjectionAnnotation` ein, wobei letztere das Objektattribut `annotatedConstructors` nutzt um Duplikate zu vermeiden.

Nach der Einführung der Annotation rufen beide Methode die `createConfiguration`-Methode auf um die Konfigurationsklasse zu erzeugen.

7 Anwendung der Plug-Ins

Das Inject Dependency-Refactoring wird in diesem Kapitel für das klassische Beispiel von Fowler durchgeführt, das seinen Ursprung in [Fow08a] hat. Zur Entkopplung wird hierbei die Constructor Injection-Variante gewählt, zur Konfiguration wird durch das Inject Dependency Guice Plug-In Konfgurationscode erzeugt. Zur Vervollständigung des Beispiels wird das Bootstrapping händisch erstellt.

7.1 Auswahl eines verfügbaren Generators

Das Inject Dependency-Refactoring kann durch weitere Plug-Ins erweitert werden, die die vorgegebene Generator-Schnittstelle implementieren. Die in eclipse installierten Generatoren werden automatisch erkannt und dem Benutzer zur Auswahl angeboten. Der Dialog für die Auswahl des gewünschten Generators wird durch *Window -> Preferences* in eclipse geöffnet.

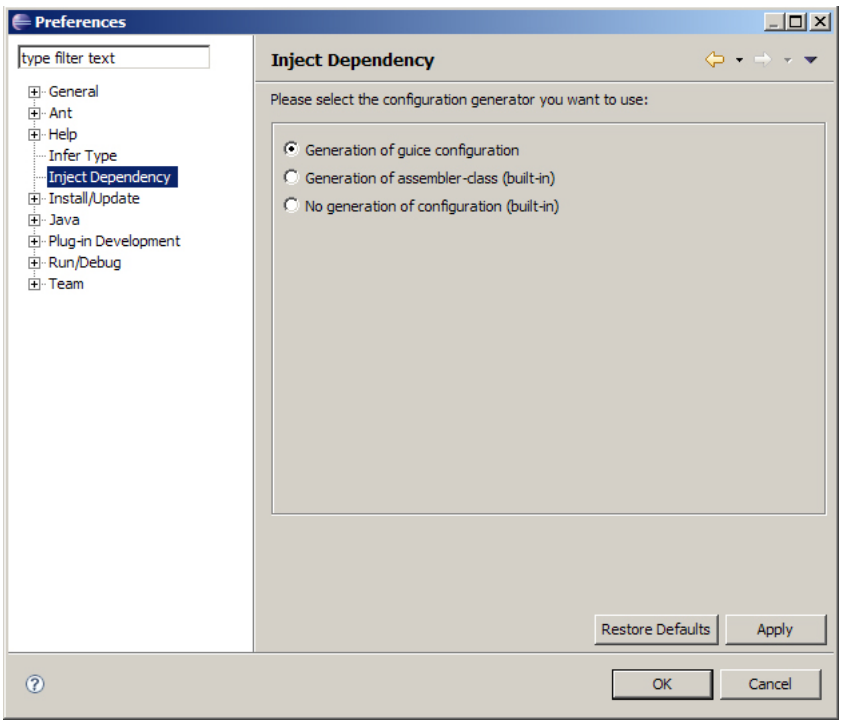

Abbildung 7.1: Auswahl der verfügbaren Generatoren

In diesem Fenster gibt es den Menüpunkt *Inject Dependency*, der den Einstellungsdialog für das Inject Dependency-Refactoring sichtbar macht. Durch die Auswahl des gewünschten Generators und die Bestätigung durch einen Klick auf *Apply* oder *OK* wird der gewählte Generator für alle zukünftigen Ausführungen des Refactoring eingesetzt.

7.2 Fowlers Beispiel

Fowlers Beispiel besteht aus den drei Klassen `Movie`, `MovieLister` und `ColonDelimetedMovieFinder`, die jedoch nicht vollständig beschrieben werden. Damit der Code übersetzbar und damit für die Durchführung des Inject Dependency-Refactorings geeignet ist, wurden die fehlenden Teile der Implementierung ergänzt:

```
1  public class Movie {
2    private String title;
3    private String director;
4
5    public Movie(String title, String director) {
6      this.title = title;
7      this.director = director;
8    }
9
10   public String getDirector() {
11     return director;
12   }
13
14   public String getTitle() {
15     return title;
16   }
17 }
18
19 public class ColonDelimitedMovieFinder {
20   private String filename;
21
22   public ColonDelimitedMovieFinder(String filename) {
23     this.filename = filename;
24   }
25
26   public List findAll() {
```

```
27      //List of movies is hardcoded!
28      List list = new ArrayList();
29      list.add(new Movie("Once␣Upon␣a␣Time␣in␣the␣West",
30          "Sergio␣Leone"));
31      return list;
32    }
33  }
34
35  public class MovieLister {
36    private ColonDelimitedMovieFinder finder;
37
38    public MovieLister() {
39      finder = new ColonDelimitedMovieFinder("movies1.txt");
40    }
41
42    public Movie[] moviesDirectedBy(String arg) {
43      List allMovies = finder.findAll();
44      for (Iterator it = allMovies.iterator(); it.hasNext();) {
45        Movie movie = (Movie) it.next();
46        if (!movie.getDirector().equals(arg)) it.remove();
47      }
48      return (Movie[]) allMovies.toArray(new Movie[allMovies.size()]);
49    }
50  }
```

Die Benutzung der Klasse MovieLister könnte beispielsweise wie folgt aussehen:

```
1  public class Program {
2    public static void main(String[] args) {
3      MovieLister lister = new MovieLister();
4      Movie[] movies = lister.moviesDirectedBy("Sergio␣Leone");
5      System.out.println(movies[0].getTitle());
6    }
7  }
```

Die Programmausführung erzeugt ein Objekt der Klasse `MovieLister` und ermittelt alle `Movie`-Objekte, deren Regisseur *Sergio Leone* ist. Das einzige `Movie`-Objekt ist innerhalb der Implementierung hart kodiert und trägt den Titel *Once Upon a Time in the West*, der deshalb auf der Konsole ausgegeben wird.

7.2.1 Durchführung des Inject Dependency-Refactoring

In der Java-Perspektive von eclipse wird die Klasse `MovieLister` geöffnet, da diese für das Dependency Injection-Pattern die Rolle des Client spielt. Im Kontextmenü erscheint der Menüpunkt *Inject Dependency*, durch welchen das Refactoring gestartet wird. Die Objektattribute werden nach deren Deklarationstyp gruppiert.

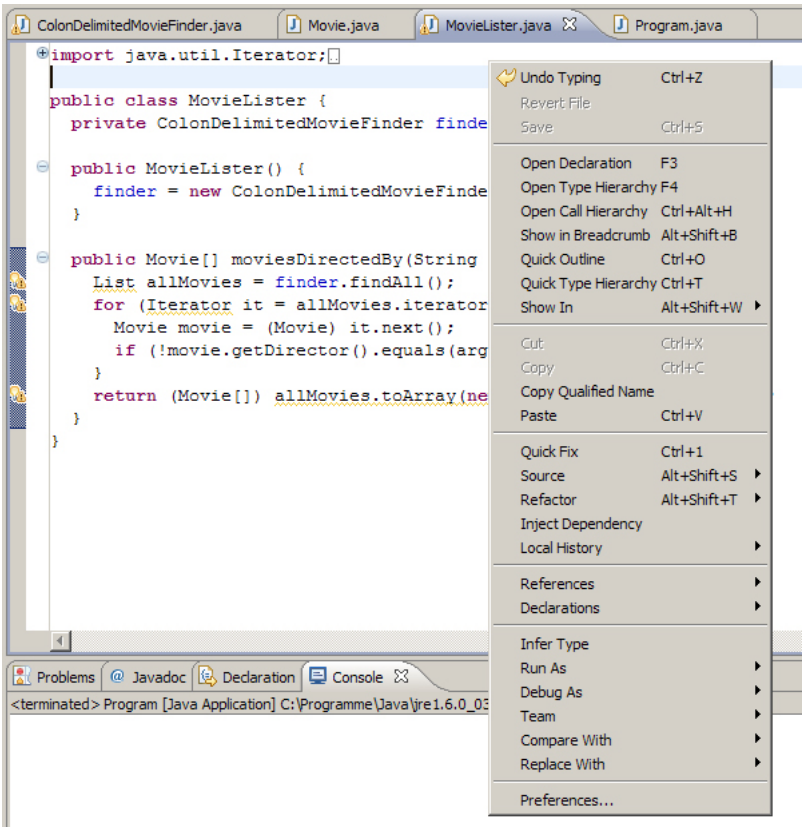

Abbildung 7.2: Start des Inject Dependency-Refactoring

Nach dem Klick auf *Inject Dependency* analysiert das Refactoring den Code und stellt im erscheinenden Refactoring-Dialog alle Objektattribute der Klasse `MovieLister` dar.

Ist ein Objektattribut nicht für die Durchführung des Refactoring geeignet, so wird dieses ausgegraut dargestellt. Zusätzlich werden Informationen angezeigt, weshalb dieses nicht für das Refactoring geeignet ist.

Im anderen Fall hat der Benutzer die Möglichkeit das Objektattribut auszuwählen, um den Client von der zugehörenden Abhängigkeit zu entkoppeln.[49] Hierzu muss die Checkbox des jeweiligen Objektattributs angehakt und der Name des neuen Schnittstellentyps, der durch Infer Type erzeugt werden soll, angegeben werden. Bei der Interface Injection-Methode ist zusätzlich die Angabe des Namens der Injection-Schnittstelle notwendig.

[49] Der Benutzer kann ebenfalls einen Deklarationstyp auswählen. Hierdurch werden automatisch alle untergeordneten Objektattribute ausgewählt.

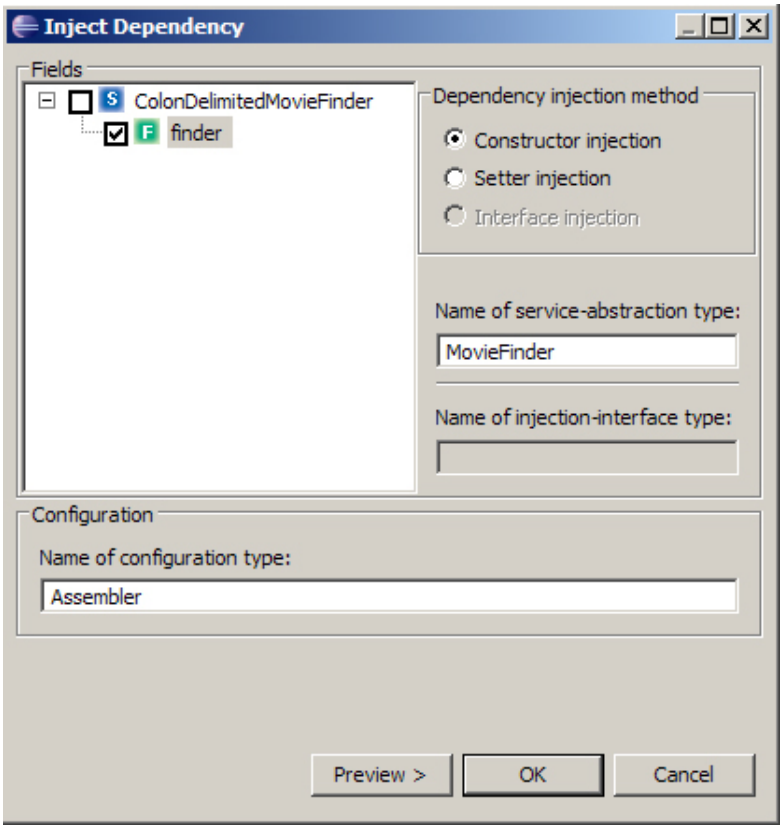

Abbildung 7.3: Einstellungsmöglichkeiten im Refactoring-Dialog

In diesem Beispiel fällt die Entscheidung für den Namen der neuen Schnittstelle auf *MovieFinder*. Für die Injektion der Abhängigkeit wird die Constructor Injection-Variante ausgewählt. Da das Guice Framework die Interface Injection-Variante nicht unterstützt, wird diese Option nicht angeboten und ausgegraut dargestellt. Die zu erzeugende Konfigurationsklasse soll *Assembler* genannt werden.

Erst wenn mindestens ein Objektattribut ausgewählt und alle notwendigen Angaben für dieses eingegeben wurden, ermöglicht der Dialog das Anklicken des *OK*- und *Preview*-Knopfes.

Im anschließenden Dialog erscheint die Vorschau der durchzuführenden Transformationen und der neu zu erzeugenden Typen.

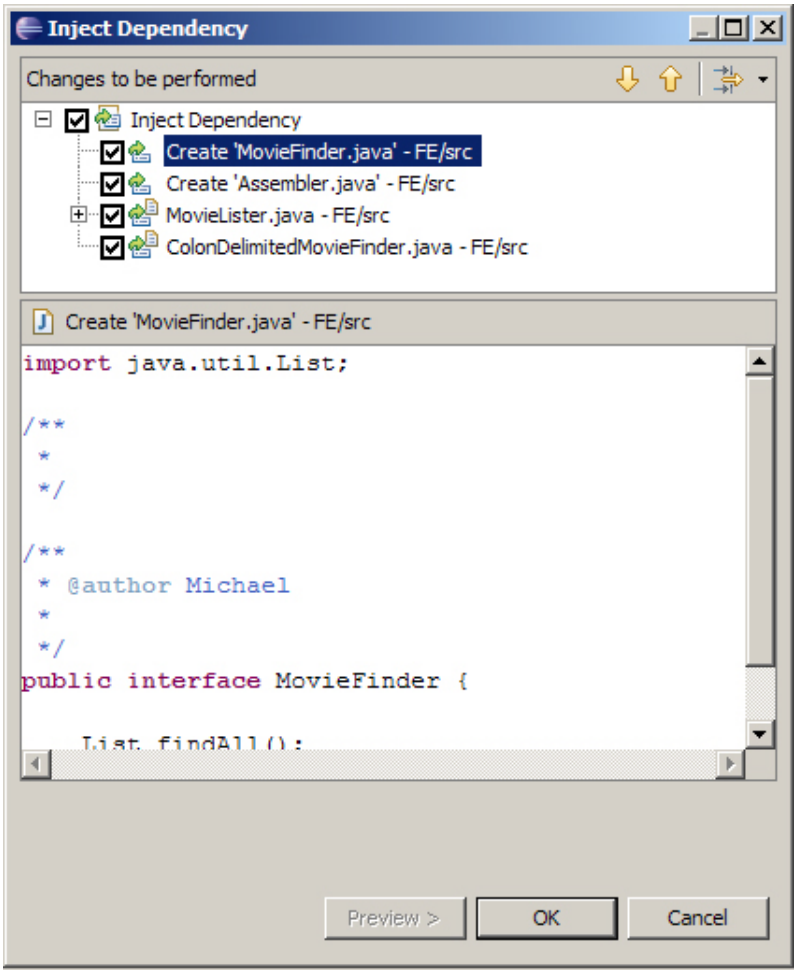

Abbildung 7.4: Vorschau der Modifikationen und neu erzeugten Typen

Durch den Klick auf *OK* wird das Refactoring ausgeführt.

7.2.2 Ergebnis der Ausführung

Das Refactoring erzeugt die neue Schnittstelle `MovieFinder` und modifiziert die Klassen `MovieLister` und `ColumnDelimitedMovieLister`:

```
1 public interface MovieFinder {
2   List findAll();
3 }
4
5 public class MovieLister {
6   private final MovieFinder finder;
```

```
7
8     @Inject()
9     public MovieLister(MovieFinder finder) {
10      this.finder = finder;
11    }
12
13    //...
14  }
15
16  public class ColonDelimitedMovieFinder implements MovieFinder {
17    //...
18  }
```

Die Schnittstelle `MovieFinder` enthält die Methodendeklaration `findAll`, da dies die einzige Methode der Klasse `ColonDelimitedMovieFinder` ist, die durch den `MovieLister` aufgerufen wird. Dem Konstruktor der Klasse `MovieLister` wurde ein neuer Parameter hinzugefügt, durch den die Abhängigkeit injiziert werden kann. Die Deklarationstypen des Objektattributs und des neuen Parameters wurden angepasst, indem der Deklarationstyp durch den neuen Schnittstellentyp ersetzt wurde. Die Klasse `ColonDelimitedMovieFinder` wurde angepasst, indem von dieser eine `implements`-Beziehung zu der neuen Schnittstelle `MovieFinder` hergestellt wurde.

Neben der Erzeugung der Konfigurationsklasse hat der Generator dem Konstruktor der Klasse `MovieLister` die Annotation `Inject` hinzugefügt. Diese Annotation wird vom Guice Framework zur Ermittlung der Abhängigkeiten des Clients benötigt. Die für das Beispiel erzeugte Konfigurationsklasse wurde wie folgt generiert:

```
1  public class Assembler implements Module, Provider<MovieFinder> {
2    public void configure(Binder binder) {
3      binder.bind(MovieFinder.class).toProvider(Assembler.class);
4    }
5
6    public MovieFinder get() {
7      return new ColonDelimitedMovieFinder("movies1.txt");
8    }
9  }
```

Die Konfiguration bindet sich selbst als Provider an die Schnittstelle `MovieFinder`. Da dem Konstruktor bei der Erzeugung des `ColonDelimitedMovieFinder`-Objekts ein Parameter übergeben werden muß, ist die Implementierung eines Providers notwendig. In der hierfür notwendigen Methode `get` findet sich somit der durch die Transformation

eliminierte Konstruktoraufruf wieder.

7.2.3 Erstellung des Bootstrapping

Zur Überprüfung des Refactoring ist die Erzeugung des Bootstrapping notwendig.

```
1 public class Bootstrapping {
2   public static void main(String[] args) {
3     Injector injector = Guice.createInjector(new Assembler());
4     MovieLister lister = injector.getInstance(MovieLister.class);
5     Movie[] movies = lister.moviesDirectedBy("Sergio Leone");
6     System.out.println(movies[0].getTitle());
7   }
8 }
```

Im diesem Bootstrapping wird der `Injector` des Guice Frameworks genutzt, dem die Konfigurationsklasse `Assembler` bei der Erzeugung übergeben wird. Von diesem kann nun ein vollständig initialisiertes Objekt der Klasse `MovieLister` bezogen werden. Danach kann wie bei der Ausführung der Klasse `Program` am Anfang des Kapitels 7.2 dieses Objekt benutzt werden, um alle `Movie`-Objekte zu ermitteln, deren Regisseur *Sergio Leone* ist. Auch bei der Ausführung des Bootstrapping wird der Titel *Once Upon a Time in the West* des einzigen `Movie`-Objekts auf der Konsole ausgegeben.

8 Schlussbetrachtung

Das Inject Dependency Plug-In ermöglicht die automatisierte Einführung des Dependency Injection-Patterns in bestehende Softwaresysteme. Es ist das Ergebnis des konzeptionellen Entwurfs des Inject Dependency-Refactoring und der anschließenden Implementierung eines Prototypen. Die vorangehende Analyse des Programmcodes ist in der Lage durch Vorbedingungen zu überprüfen, ob die Einführung des Dependency Injection-Patterns für eine Client-Services-Konstellation möglich ist und verhindert bei Bedarf die Durchführung der Transformation. Die Entkopplung der Client-Klasse von einer Service-Klasse wird zusätzlich durch den Einsatz des Infer Type-Refactoring optimiert, da zur Abstraktion des Typs der Service-Klasse ein maximal generalisierter Typ erzeugt wird.

8.1 Änderung komplexer Programmstrukturen

Wird das Dependency Injection-Pattern einer bestehenden Klassenstruktur eingeführt, so sind ebenfalls die Clients des Client von den Änderungen betroffen. Zwar wird in Absatz 4.1 beschrieben, dass eine sequentielle Entkopplung von Klassen prinzipiell möglich ist, beruht jedoch auf einer Struktur, die nicht immer zwangsläufig gegeben ist. Wird der Client-Klasse beispielsweise ein neues Konstruktorargument eingeführt, so führt das zu syntaktischen Fehlern innerhalb der Clients des Client, da das Refactoring die hierbei vorhandenen Konstruktoraufrufe nicht anpasst.

```
1  public class ClientOfClient {
2    private Client client;
3
4    public void work() {
5      client = new Client(); //Syntaxfehler
6    }
7  }
8
9  public class Client {
10   public final IService service;
11
```

```
12    public Client(IService service) {
13      this.service = service;
14    }
15  }
16
17  public interface IService {
18  }
19
20  public class Service implements IService {
21  }
```

Eine Lösung des Problems scheint die Zuhilfenahme des Assemblers zur Ermittlung des notwendigen Konstruktparameters zu sein.

```
1   public class ClientOfClient {
2     private Client client;
3
4     public void work() {
5       client =
6         new Client(Assembler.getService()); //Kein Syntaxfehler
7     }
8   }
```

Die hierdurch eingeführte Abhängigkeit zum Assembler bewirkt jedoch eine Verschlechterung der Programmstruktur und wurde deshalb bereits als Anti-Pattern mit dem Namen *Container Dependency*[50] formuliert.

Um die Abhängigkeit zum Assembler zu vermeiden, könnten die Clients des Client die Abhängigkeit des Clients zum Service als die eigene übernehmen, damit das Service-Objekt dann dem Konstrukoraufruf übergeben werden kann.

```
1   public class ClientOfClient {
2     private IService service;
3     private Client client;
4
5     public void work(IService service) {
6       this.service = service;
7       client =
8         new Client(service); //Kein Syntaxfehler
9     }
```

[50] Siehe http://www.picocontainer.org/container-dependency-antipattern.html.

```
10 }
```

Neben der Notwendigkeit, dass diejenigen Klassen, die `ClientOfClient`-Objekte erzeugen, ebenfalls angepasst werden müssten, besitzt die Klasse `ClientOfClient` eine Abhängigkeit, die nur zum Zweck der Übergabe an ein anderes Objekt existiert. Dieses Verfahren ist ebenfalls bereits als Anti-Pattern mit dem Namen *Propagating Dependency*[51] formuliert worden.

8.2 Fazit

Eine abschließende Antwort für dieses Problem kann im Rahmen dieser Abschlussarbeit nicht gegeben werden. Während der Einsatz des Inject Dependency-Refactoring für einzelne Client-Services-Konstellationen ohne weitere Anpassungen möglich ist, müssen bei verwobenen Programmstrukuren weitere Anpassungen durch den Benutzer gemacht werden. Erst durch den Aufbau eines vollständigen Abhängigkeitsgraphen aus der Programmstruktur eines Softwaresystems kann ermittelt werden, welche Entkopplungen aus ganzheitlicher Sicht vollständig möglich sind.

8.3 Weiterführende Arbeit

Eine Lösung des vorgestellten Problems wäre die vorangehende Entkopplung der Klasse `ClienOfClient` von der Klasse `Client`.

```
1  public class ClientOfClient {
2    private final IClient client;
3
4    public void work(IClient client) {
5      this.client = client;
6    }
7  }
8
9  public interface IClient {
10 }
11
12 public class Client implements IClient {
13   public final IService service;
14
```

[51] Siehe http://www.picocontainer.org/propagating-dependency-antipattern.html.

```
15    public Client(IService service) {
16      this.service = service;
17    }
18  }
```

Eine Erweiterung des Inject Dependency Plug-Ins müsste die Abhängigkeitsgraphen innerhalb eines Softwaresystems analysieren und visuell darstellen können. Das Refactoring müsste dann alle Abhängigkeitesbeziehungen, die zueinander in Relation stehen, gleichzeitig behandeln und würde somit einen korrupten Zustand des Softwaresystems verhindern.

Hierfür ist jedoch zuerst eine vorbereitende konzeptionelle Überlegung notwendig, ob die gleichzeitige Anpassung und Erzeugung durch einen Benutzer noch nachvollziehbar und konfigurierbar ist.

8.4 Ausblick

Im Moment wird ausschließlich das Guice Framework durch die Erzeugung von Konfigurationsklassen unterstützt. Um das Inject Dependency-Refactoring auch beim Einsatz anderer Dependency Injection-Frameworks sinnvoll einsetzen zu können, ist die Realisierung weiterer Generator Plug-Ins notwendig. Aus diesem Grund kann das Inject Dependency Plug-In der eclipse-Community zur Verfügung gestellt werden, damit von dieser weitere Generatoren realisiert werden können.

A Integration eines Refactoring

Der Einsatz des Inject Dependency-Refactorings erfolgt in der Regel über die Dialogoberfläche, durch die der Benutzer sämtliche Einstellungen vornehmen und das Refactoring starten kann. Zudem ist es möglich, dass die Ausführung des Refactorings durch ein anderes Plug-In gesteuert werden kann.

Für das Ausführen eines Plug-Ins aus einem anderen Plug-In heraus stellt eclipse die Schnittstelle `IRunnableWithProgress` zur Verfügung. Diese Schnittstelle steht für eine Aufgabe, die im Hintergund ausgeführt wird und sich in die Dialogoberfläche eines anderen Plug-Ins integriert. Soll ein Plug-In integrierbar sein, so muss eine Klasse des Plug-Ins diese Schnittstelle implementieren und kann in diesem Zuge Konfigurationseinstellungen durch Setter-Methoden ermöglichen. Die Ausführung eines integrierten Plug-Ins wird in der Methode `run` implementiert. Der Container, der in der Lage ist eine Hintergrundaufgabe entgegenzunehmen, wird von der Klasse `Wizard` durch die Methode `getContainer` zur Verfügung gestellt.

So kann beispielseise ein anderes Plug-In innerhalb der Methode `performFinish` einer Subklasse von `Wizard` ausgeführt werden:

```
1 public boolean performFinish() {
2   //...
3   IRunnableWithProgress op = ...
4   getContainer().run(false, false, op);
5   //...
6 }
```

Der erste Parameter der Methode `run` gibt an, ob die Hintergrundaufgabe in einem eigenen Thread ausgeführt werden soll. Der zweite Parameter definiert, ob die Hintergrundaufgabe abgebrochen werden darf. Hierfür würde in der Dialogoberfläche ein Abbrechen-Knopf dargestellt werden. Als dritter Parameter wird die Hintergrundaufgabe als Referenz übergeben.

A.1 Entwurf der Integrations-Schnittstelle

Der Versuch ein Refactoring durch diese Schnittstelle in ein anderes Plug-In zu integrieren führt zu den folgenden Problemen:

- Die Integration des Refactorings wird konzeptionell auf die Dialogoberfläche eingeschränkt.

- Sind Einstellungen für das Refactoring notwendig, so müssen diese durch Setter-Methoden realisiert werden. Vor der Ausführung der `run`-Methode kann somit nicht sichergestellt werden, dass alle Angaben vor der Ausführung des Refactorings gesetzt wurde. Notwendige und optionale Einstellungen können nur durch Code-Dokumentation vermittelt werden.

- Bei der Ausführung eines Refactoring ist es nicht zwingend erforderlich, dass die Transformationen innerhalb der `run`-Methode nach deren Berechnung auch ausgeführt werden. Das einbindende Plug-In muss in der Lage sein die Kontrolle über die Transformationen zu besitzen. Für diese Kontrolle müssen vom eingebundenen Refactoring das `Change`-Objekt und das `RefactoringStatus`-Objekt ausgewertet werden können. Diese beiden Objekte müssen umständlich durch Getter-Methoden bezogen werden.

Zur Verbesserung der Integration wurde eine Schnittstelle entworfen, die nicht von einer Dialogoberfläche abhängig ist. Anstelle einer `run`-Methode definiert die Schnittstelle `org.intoJ.injectDependency.integration.IRefactoringIntegration` zwei Methoden mit unterschiedlicher Bedeutung.

```
1 public interface IRefactoringIntegration {
2   IntegrationResult runCommitedRefactoring(IProgressMonitor monitor,
3     Object... arguments) throws IntegrationException;
4
5   IntegrationResult runUncommitedRefactoring(
6     IProgressMonitor monitor, Object... arguments)
7     throws IntegrationException;
8 }
```

Beide Methoden akzeptieren das Argument `monitor` um eine graphische Darstellung zu unterstützen. Das zweite Argument akzeptiert beliebige Parameter, die für die Durchführung des Refactoring notwendig sind. Als Rückgabewert wird ein Objekt der Klasse

`org.intoJ.injectDependency.integration.IntegrationResult` zurückgegeben. Dieses Objekt kapselt das `Change`-Objekt und das `RefactoringStatus`-Objekt der Transformationen. Auftretende refactoringspezifische Exceptions werden durch die Exception `org.intoJ.injectDependency.integration.IntegrationException` gekapselt.

Beide Methoden führen das Refactoring aus und erzeugen somit Transformationen von Programmcode. Während die Methode `runUncommitedRefactoring` diese Transformationen nur berechnet, werden diese während der Ausführung von `runCommitedRefactoring` abschließend auch durchgeführt.

A.2 Integration des Inject Dependency-Refactoring

Damit das Inject Dependency-Refactoring auch von anderen Plug-Ins integriert werden kann, implementiert die Klasse `org.intoJ.injectDependency.InjectDependencyRefactoring` die Schnittstelle `IRefactoringIntegration`.

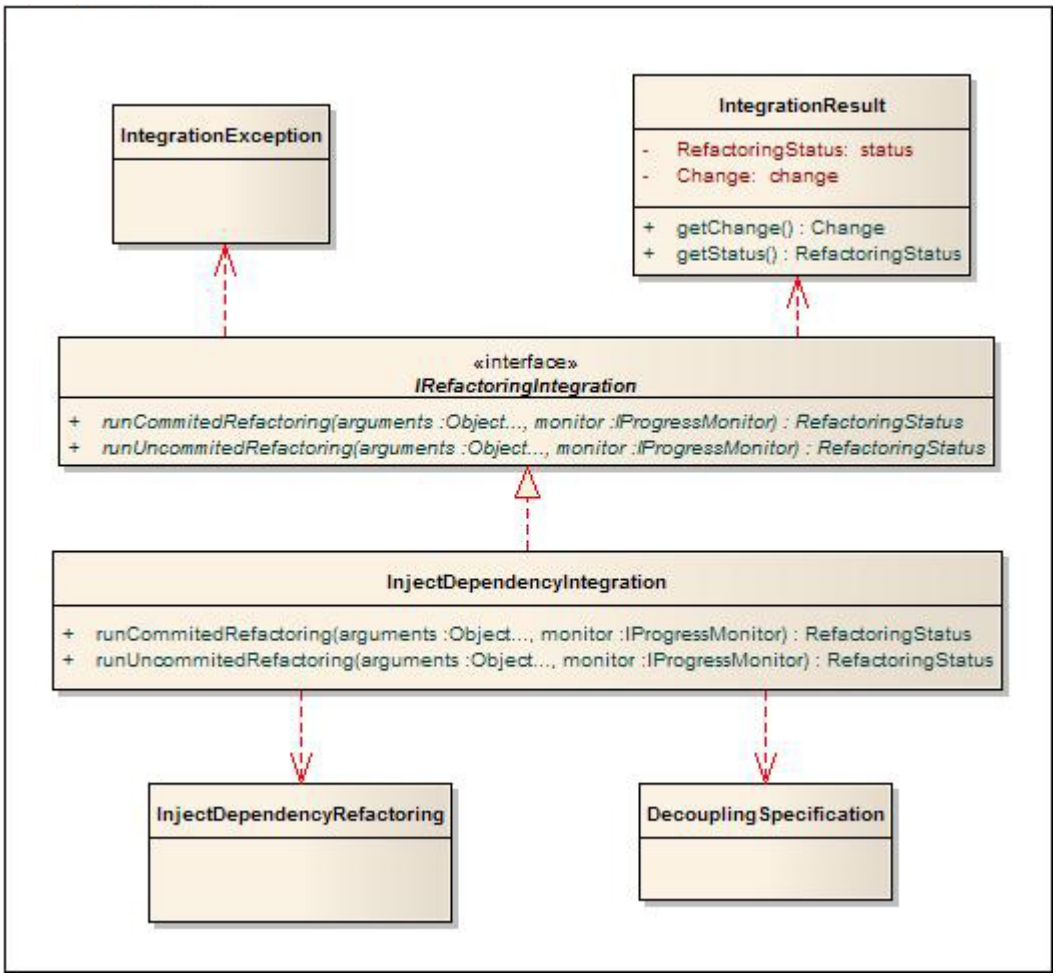

Abbildung A.1: Schnittstelle zur Integration eines Refactoring

Die Anforderungen an die Realisierung sind die Extraktion der Parameter aus dem Argument `Object.. arguments` und die Durchführung des Refactorings. Die beiden Methoden der Schnittstelle wurden wie folgt implementiert:

```
1  public class InjectDependencyIntegration
2      implements IRefactoringIntegration {
3    private static final String INTEGRATED_REFACTORING_NAME
4      = "Inject␣Dependency-Integration";
5    private ICompilationUnit icu;
6    private String typeName;
7    private List<DecouplingSpecification> decouplingSpecifications;
8
9    //...
10   public IntegrationResult runCommitedRefactoring(
11       IProgressMonitor monitor, Object... arguments)
12       throws IntegrationException {
13     return runRefactoring(true, monitor, arguments);
14   }
15
16   public IntegrationResult runUncommitedRefactoring(
17       IProgressMonitor monitor, Object... arguments)
18       throws IntegrationException {
19     return runRefactoring(false, monitor, arguments);
20   }
21   //...
22 }
```

Beide Methoden rufen die private Methode `runRefactoring` auf, wobei sich die Aufrufe nur durch den ersten Parameter unterscheiden. Dieser Parameter gibt die Information, ob die berechneten Transformationen durchgeführt werden sollen, an die aufgerufene private Methode weiter.

Die Extraktion der Parameter wird durch die private Methode `extractArguments` realisiert:

```
1  public class InjectDependencyIntegration
2      implements IRefactoringIntegration {
3    //...
4    private void extractArguments(Object... arguments)
5        throws IntegrationException {
6      if (arguments == null || arguments.length != 3)
```

```
7         throw new IntegrationException("Please␣provide␣arguments.");
8
9       if (arguments[0] instanceof ICompilationUnit)
10        icu = (ICompilationUnit) arguments[0];
11      else
12        throw new IntegrationException("Please␣provide␣"
13          + "compilation␣unit.");
14
15      if (arguments[1] instanceof String)
16        typeName = (String) arguments[1];
17      else
18        throw new IntegrationException("Please␣provide␣typename.");
19
20      if (arguments[2] instanceof List)
21        decouplingSpecifications
22          = (List<DecouplingSpecification>) arguments[2];
23      else
24        throw new IntegrationException("Please␣provide␣"
25          + "decoupling␣specifications.");
26    }
27    //...
28  }
```

Das Inject Dependency-Refactoring benötigt die Angabe einer Compilation Unit, des zugehörigen Typnamens und einer Liste von `DecouplingSpecification`-Objekten. Sind diese Angaben verfügbar werden die entsprechenden Objektattribute gesetzt. Sind die Angaben unvollständig oder fehlerhaft wird eine `IntegrationException` geworfen. Jedes `DecouplingSpecification`-Objekte kapselt hierbei die Informationen, welches Objektattribut durch welche Dependency Injection-Variante behandelt werden soll und wie die Namen der neuen Typen sind.

Die Durchführung des Refactoring geschieht in der Methode `runRefactoring`:

```
1   public class InjectDependencyIntegration
2       implements IRefactoringIntegration {
3     //...
4     private IntegrationResult runRefactoring(
5         final boolean commitChange, IProgressMonitor monitor,
6         Object... arguments) throws IntegrationException {
7       if (monitor == null)
8         monitor = new NullProgressMonitor();
9
```

```
10      try {
11        //Extract arguments
12        extractArguments(arguments);
13
14        //Determine the client type
15        IType type = icu.getType(typeName);
16        if (!type.exists())
17          throw new IntegrationException("Type " + typeName
18            + " does not exist in compilation unit.");
19
20        //Determine the fields
21        for (DecouplingSpecification spec : decouplingSpecifications) {
22          IField field =
23            type.getField(spec.getFieldName());
24          if (!field.exists())
25            throw new IntegrationException("Field "
26              + spec.getFieldName()
27              + " is not a member of type " + typeName + ".");
28        }
29
30        //Create refactoring
31        monitor.beginTask(INTEGRATED_REFACTORING_NAME, 5);
32        InjectDependencyRefactoring refactoring =
33          new InjectDependencyRefactoring(type);
34
35        //Check initial conditions
36        final RefactoringStatus status =
37          refactoring.checkInitialConditions(
38            new SubProgressMonitor(monitor, 1));
39        if (status.hasFatalError())
40          return new IntegrationResult(status, null);
41
42        //Check final conditions
43        refactoring
44          .setDecouplingSpecifications(decouplingSpecifications);
45        status.merge(refactoring.checkFinalConditions(
46          new SubProgressMonitor(monitor, 1)));
47        if (status.hasFatalError())
48          return new IntegrationResult(status, null);
49
```

```
50          //Create change
51          final Change change = refactoring.createChange(
52            new SubProgressMonitor(monitor, 1));
53          change.initializeValidationData(
54            new SubProgressMonitor(monitor, 1));
55
56          //Commit change if required
57          if (commitChange)
58            change.perform(monitor);
59
60          //Return status and change
61          return new IntegrationResult(status, change);
62        }
63        catch (OperationCanceledException ex) {
64          throw new IntegrationException(ex);
65        }
66        catch (CoreException ex) {
67          throw new IntegrationException(ex);
68        }
69        finally {
70          monitor.done();
71        }
72      }
73      //...
74  }
```

Nachdem die Parameter durch den Aufruf von `extractArguments` bestimmt wurden, wird zunächst überprüft, ob der Typname innerhalb der Compilation Unit existiert. Ebenfalls wird geprüft, ob die angegebenen Namen der Objektattribute innerhalb des Typs existieren.

Dem Konstruktor des `InjectDependencyRefactoring`-Objekts wird der Typ als Parameter übergeben. Während bei der Überprüfung der Vorbedingungen durch die `checkInitialConditions`-Methode keine weiteren Parameter notwendig sind, muss dem Refactoring-Objekt vor der Ausführung der `checkFinalConditions`-Methode die Liste der zu behandelnden Felder übergeben werden. Dies geschieht durch die `setDecouplingSpecifications`-Methode.

Nach der Überprüfung der Nachbedingungen werden die Transformationen durch die Methode `createChange`-Methode berechnet und als `Change`-Objekt geliefert. Die tatsächliche Durchführung der Transformationen hängt vom Methoden-Parameter `commitChange` ab.

A.3 Schlussbetrachtung

Durch diese Schnittstelle hat ein Plug-In die volle Kontrolle über die Transformationen eines eingebundenen Refactorings. Durch den Zugriff auf das `Change`-Objekt und das `RefactoringStatus`-Objekt kann ein integrierendes Refactoring die erhaltenen Transformationen und Status mit den eigenen verschmelzen. Die Klasse RefactoringStatus sieht hierfür die Methode `merge` vor. Für die Verschmelzung von Transformationen sieht das LTK von eclipse die Klasse `CompositeChange` vor, die ebenfalls eine Methode `merge` besitzt.

Da das integrierende Plug-In dafür zuständig ist, die Compilation Unit an das eingebundene Refactoring zu übergeben, kann dieses kontrollieren, ob das eingebundene Refactoring direkt auf dem Quellcode operiert oder sämtliche Transformationen auf Basis einer Arbeitskopie berechnet werden.

B Testen von Plug-Ins mit JUnit und Mock-Objekten

Auch während der Implementierung eines Refactorings für eclipse ist sehr hilfreich, einzelne Schritte und funktionale Einheiten durch Tests zu überprüfen. Das Ausführen von JUnit-Tests ist bei der Entwicklung von Plug-Ins jedoch etwas aufwändiger, da die Tests nicht nur auf Basis der JVM ausgeführt werden müssen, sondern auch die Ausführung innerhalb von eclipse erfordern.

Seit Milestone 3 von eclipse 3.0 ist PDE-JUnit ein fester Bestandteil von eclipse. Diese Erweiterung der PDE ermöglicht es JUnit-Tests auszuführen, indem das zugehörige Plug-In innerhalb einer neuen eclipse-Instanz ausgeführt wird. In [CR06] und [GB03] wird PDE-JUnit und die Ausführung der Tests ausführlich vorgestellt.

B.1 Einbindung einer Mock-Bibliothek

Als ergänzende Technik hat sich der Einsatz von Mock-Objekten[52] etabliert. Als konkretes Beispiel für eine Mock-Bibliothek wird im folgenden JMock[53] genutzt.

Wird JMock als Bibliothek in ein PDE-JUnit-Projekt eingebunden, können die Tests zwar fehlerfrei übersetzt werden, würden jedoch zu einem Abbruch während der Ausführung führen. Die für die Tests erzeugte eclipse-Instanz kennt die notwendige Bibliothek JMock nicht, weshalb die Ausführung eines Tests eine `ClassNotFoundException` verursachen würde.

Damit JMock innerhalb von PDE-Unit-Tests eingesetzt werden kann, muss JMock ebenfalls von eclipse als Plug-In eingebunden werden. Nachdem die aktuelle Version von JMock 2 heruntergeladen und entpackt wurde, wird ein neues Projekt in eclipse erstellt.

B.1.1 Plug-In erstellen

Mit *File -> New -> Project...* wird der Assistent in eclipse gestartet. Unter *Plug-In Development* gibt es die Option *Plug-In from existing JAR archives*, die ausgewählt

[52] Siehe [Bec02].
[53] Siehe http://www.jmock.org/.

werden muss. Im Dialog *JAR selection* werden alle JAR-Dateien der entpackten JMock-Version ausgewählt. Im Dialog *Plug-In Project Properties* wird als Projektname *org.jmock* eingegeben. Zuletzt muss sichergestellt werden, dass die Checkbox bei *Unzip the JAR archives into the project* nicht angekreuzt ist.

B.1.2 Konfiguration des Plug-In

Im Abschnitt *Overview* der Plug-In-Konfiguration sollten die folgenden Angaben eingetragen werden:

- Version - 2.5.0
- Name - JMock Plug-In

Im Abschitt *Dependencies* wird JUnit in den Versionen 3 und 4 ausgewählt, da JMock 2 beides unterstützt. Hierfür werden unter *Required Plug-Ins* durch *Add...* die beiden Optionen *org.junit* und *org.junit4* ausgewählt.

B.2 Anlegen eines PDE-JUnit Projekts

Für die Durchführung von PDE-Junit-Tests sieht das PDE einen eigenen Projekt-Typ vor. In diesem Projekt können die Tests dann implementiert und ausgeführt werden. Zur Veranschaulichung werden im folgenden die Klassengerüste eines Beispielprojekts dargestellt.

B.2.1 Projekt erstellen

Mit *File -> New - Project...* wird der Assistent in eclipse gestartet. Unter *Plug-In Development* wird die Option *Plug-In Project* ausgewählt. Im Dialog *Plug-In Project* muss ein Projektname angegeben werden, beispielsweise *org.intoJ.injectDependency.test.* Im Dialog *Plug-In Content* dürfen die Checkboxen bei *Generate an activator...* und *This Plug-In will make contributions...* nicht angekreuzt sein. Zuletzt muss bei *Rich Client Application* die Option *No* ausgewählt werden.

B.2.2 Konfiguration des Projekts

Im Abschnitt *Overview* der Projekt-Konfiguration können die folgenden Angaben eingetragen werden:

- Version - 0.0.1

- Name - InjectDependencyTest Plug-in

Im Abschnitt *Dependencies* müssen unter *Required Plug-Ins* durch *Add...* die folgenden Optionen hinzugefügt werden:

- *org.junit* oder *org.junit4*
- *org.jmock*
- Das Package des zu testenden Plug-In (beispielsweise *org.intoJ.injectDependency*)
- Zusätzliche Packages von eclipse bzw. PDE (beispielsweise *org.eclipse.ltk.core.refactoring*)

B.2.3 Einbindung der Hilfsklasse TestProject

Erich Gamma und Kent Beck stellen die Klasse `TestProject` zur Verfügung, mit der eine Fixture für PDE-JUnit-Tests erstellt werden kann. Diese Klasse kann wird ebenfalls in Absatz B.3 dieser Abschlussarbeit dargestellt. Mit dieser Klasse kann programmatisch sehr einfach ein Java-Projekt erzeugt werden.

Die Klasse `TestProject` muss in ein Verzeichnis des Projekts kopiert werden (beispielsweise in das Package `org.intoJ.injectDependency.test.support`).

B.2.4 Erstellung der Testsuite

Im Package `org.intoJ.injectDependency.test` wird die Klasse `InjectDependencyTestSuite` mit dem folgenden Code erstellt:

```
1 public class InjectDependencyTestSuite {
2   public static Test suite() {
3     TestSuite suite =
4       new TestSuite("org.intoJ.injectDependency tests");
5     suite.addTestSuite(ExampleTest.class);
6     return suite;
7   }
8 }
```

B.2.5 Erstellung eines Tests

Im Package `org.intoJ.injectDependency.test.example` wird die Klasse `ExampleTest` mit dem folgenden Code erstellt:

```
1  public class ExampleTest extends MockObjectTestCase {
2    private TestProject project;
3
4    @Override
5    protected void setUp() throws Exception {
6      project = new TestProject();
7    }
8
9    @Override
10   protected void tearDown() throws Exception {
11     project.dispose();
12   }
13
14   public void testExample() {
15     //Test-Code
16   }
17 }
```

Durch die Spezialisierung der Klasse `MockObjectTestCase` kann die Funktionalität von JMock 2 ohne Einschränkung genutzt werden.

B.2.6 Programmatische Erzeugung eines Java-Projekts

Beispielcode, bei dem eine Klasse `Client` das Objekt einer Klasse `Service` erzeugt und dessen Referenz in einem Objektattribut hält:

```
1  IPackageFragment servicePackage = project.createPackage("service");
2  IType service = project.createType(servicePackage, "Service.java",
3    ("public class Service {" +
4     "}"));
5
6  IPackageFragment clientPackage = project.createPackage("client");
7  IType client = project.createType(clientPackage, "Client.java",
8    ("import service.Service;" +
9     "" +
10    "public class Client {" +
11    "  private final Service service = new Service();" +
12    "}"));
```

B.2.7 Durchführung der Testsuite

Nach einem Rechtsklick im *Package Explorer* auf die Klasse `InjectDependencyTestSuite` kann im erscheinenden Kontextmenü auf *Run As -> JUnit Plug-in Test* geklickt werden. Dies startet die TestSuite und den von ihre eingebundenen Test.

B.3 Die Hilfsklasse Testproject

Die Klasse Testproject wurde aus [GB03] entnommen und geringfügig an die API der aktuellen Version von eclipse beziehungsweise des LTK angepasst.

```
1  import java.io.IOException;
2  import java.net.URL;
3
4  import org.eclipse.core.resources.IFolder;
5  import org.eclipse.core.resources.IProject;
6  import org.eclipse.core.resources.IProjectDescription;
7  import org.eclipse.core.resources.IWorkspaceRoot;
8  import org.eclipse.core.resources.ResourcesPlugin;
9  import org.eclipse.core.runtime.CoreException;
10 import org.eclipse.core.runtime.IPath;
11 import org.eclipse.core.runtime.IPluginDescriptor;
12 import org.eclipse.core.runtime.IPluginRegistry;
13 import org.eclipse.core.runtime.Path;
14 import org.eclipse.core.runtime.Platform;
15 import org.eclipse.jdt.core.IClasspathEntry;
16 import org.eclipse.jdt.core.ICompilationUnit;
17 import org.eclipse.jdt.core.IJavaElement;
18 import org.eclipse.jdt.core.IJavaProject;
19 import org.eclipse.jdt.core.IPackageFragment;
20 import org.eclipse.jdt.core.IPackageFragmentRoot;
21 import org.eclipse.jdt.core.IType;
22 import org.eclipse.jdt.core.JavaCore;
23 import org.eclipse.jdt.core.JavaModelException;
24 import org.eclipse.jdt.core.search.IJavaSearchConstants;
25 import org.eclipse.jdt.core.search.ITypeNameRequestor;
26 import org.eclipse.jdt.core.search.SearchEngine;
27 import org.eclipse.jdt.launching.JavaRuntime;
28
```

```
29 public class TestProject {
30   private IProject project;
31   private IJavaProject javaProject;
32   private IPackageFragmentRoot sourceFolder;
33
34   public TestProject() throws CoreException {
35     IWorkspaceRoot root = ResourcesPlugin.getWorkspace().getRoot();
36     project = root.getProject("Project-1");
37     project.create(null);
38     project.open(null);
39
40     javaProject = JavaCore.create(project);
41     IFolder binFolder = createBinFolder();
42     setJavaNature();
43     javaProject.setRawClasspath(new IClasspathEntry[0], null);
44     createOutputFolder(binFolder);
45     addSystemLibraries();
46   }
47
48   public IProject getProject() {
49     return project;
50   }
51
52   public IJavaProject getJavaProject() {
53     return javaProject;
54   }
55
56   public void addJar(String plugin, String jar) throws IOException,
57       JavaModelException {
58     Path result = findFileInPlugin(plugin, jar);
59     IClasspathEntry[] oldEntries = javaProject.getRawClasspath();
60     IClasspathEntry[] newEntries =
61       new IClasspathEntry[oldEntries.length + 1];
62     System.arraycopy(oldEntries, 0, newEntries, 0,
63       oldEntries.length);
64     newEntries[oldEntries.length] =
65       JavaCore.newLibraryEntry(result, null, null);
66     javaProject.setRawClasspath(newEntries, null);
67   }
68
```

```
69    public IPackageFragment createPackage(String name)
70        throws CoreException {
71      if (sourceFolder == null)
72        sourceFolder = createSourceFolder();
73      return sourceFolder.createPackageFragment(name, false, null);
74    }
75
76    public IType createType(IPackageFragment pack, String cuName,
77        String source) throws JavaModelException {
78      StringBuffer buf = new StringBuffer();
79      buf.append("package " + pack.getElementName() + ";\n");
80      buf.append("\n");
81      buf.append(source);
82      ICompilationUnit cu =
83        pack.createCompilationUnit(cuName, buf.toString(), false,
84          null);
85      return cu.getTypes()[0];
86    }
87
88    public void dispose() throws CoreException {
89      waitForIndexer();
90      project.delete(true, true, null);
91    }
92
93    private IFolder createBinFolder() throws CoreException {
94      IFolder binFolder = project.getFolder("bin");
95      binFolder.create(false, true, null);
96      return binFolder;
97    }
98
99    private void setJavaNature() throws CoreException {
100     IProjectDescription description = project.getDescription();
101     description.setNatureIds(new String[] {JavaCore.NATURE_ID});
102     project.setDescription(description, null);
103   }
104
105   private void createOutputFolder(IFolder binFolder)
106       throws JavaModelException {
107     IPath outputLocation = binFolder.getFullPath();
108     javaProject.setOutputLocation(outputLocation, null);
```

```
109   }
110
111   private IPackageFragmentRoot createSourceFolder()
112       throws CoreException {
113     IFolder folder = project.getFolder("src");
114     folder.create(false, true, null);
115     IPackageFragmentRoot root =
116       javaProject.getPackageFragmentRoot(folder);
117
118     IClasspathEntry[] oldEntries = javaProject.getRawClasspath();
119     IClasspathEntry[] newEntries =
120       new IClasspathEntry[oldEntries.length + 1];
121     System.arraycopy(oldEntries, 0, newEntries, 0,
122       oldEntries.length);
123     newEntries[oldEntries.length] =
124         JavaCore.newSourceEntry(root.getPath());
125     javaProject.setRawClasspath(newEntries, null);
126     return root;
127   }
128
129   private void addSystemLibraries() throws JavaModelException {
130     IClasspathEntry[] oldEntries = javaProject.getRawClasspath();
131     IClasspathEntry[] newEntries =
132       new IClasspathEntry[oldEntries.length + 1];
133     System.arraycopy(oldEntries, 0, newEntries, 0,
134       oldEntries.length);
135     newEntries[oldEntries.length] =
136       JavaRuntime.getDefaultJREContainerEntry();
137     javaProject.setRawClasspath(newEntries, null);
138   }
139
140   private Path findFileInPlugin(String plugin, String file)
141       throws IOException {
142     IPluginRegistry registry = Platform.getPluginRegistry();
143     IPluginDescriptor descriptor =
144       registry.getPluginDescriptor(plugin);
145     URL pluginURL = descriptor.getInstallURL();
146     URL jarURL = new URL(pluginURL, file);
147     URL localJarURL = Platform.asLocalURL(jarURL);
148     return new Path(localJarURL.getPath());
```

```
149   }
150
151   private void waitForIndexer() throws JavaModelException {
152     new SearchEngine().searchAllTypeNames(
153         ResourcesPlugin.getWorkspace(),
154         null,
155         null,
156         IJavaSearchConstants.EXACT_MATCH,
157         IJavaSearchConstants.CASE_SENSITIVE,
158         IJavaSearchConstants.CLASS,
159         SearchEngine.createJavaSearchScope(new IJavaElement[0]),
160         new ITypeNameRequestor() {
161           public void acceptClass(char[] packageName,
162               char[] simpleTypeName, char[][] enclosingTypeNames,
163               String path) {}
164           public void acceptInterface(char[] packageName,
165               char[] simpleTypeName, char[][] enclosingTypeNames,
166               String path) {}
167         },
168         IJavaSearchConstants.WAIT_UNTIL_READY_TO_SEARCH, null);
169   }
170 }
```

C Inhalt der beiliegenden CD

Auf der beiliegenden CD befindet sich der folgende Inhalt:

Verzeichnis	**Inhalt**
Ausarbeitung	Dieses Dokument als PDF.
Implementierung/dist	Die JAR-Dateien der drei Plug-Ins Infer Type, Inject Dependency und Inject Dependency Guice.
Implementierung/javadoc	Die Quellcode-Dokumentation der beiden Plug-Ins Inject Dependency und Inject Dependency Guice.
Implementierung/src	Der Quellcode bzw. die eclipse-Projekte.
Implementierung/eclipse	Eine Installation von eclipse 3.4 mit den bereits installierten Plug-Ins Infer Type, Inject Dependency und Inject Dependency Guice.

Das eclipse-Projekt des Plug-Ins Infer Type musste geringfügig angepasst werden, damit dessen Refactoring-Mechanismus in das Inject Dependency Plug-In integriert werden konnte. Durch den notwendigen Zugriff auf den Typ `SelectedElement` des internen Package `org.intoJ.inferType3.internal.constraintModel.util` mussten hierfür in den Projekteinstellungen die internen Packages als *Exported Packages* markiert werden. Da das Inject Dependency Plug-In ausschließlich mit dieser Anpassung lauffähig ist, befinden sich auf dieser CD das modifizierte Infer Type-Projekt und eine daraus erstellte JAR-Datei.

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[Bal96] H. Balzert: **Lehrbuch der Software-Technik - Software-Entwicklung**
Spektrum Akademischer Verlag, 1996.

[Bal98] H. Balzert: **Lehrbuch der Software-Technik - Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung**
Spektrum Akademischer Verlag, 1998.

[Bau08] T. Baumann: **Ein Refaktorisierungswerkzeug zum Ersetzen von kollaborierenden Objekten in Unit-Tests durch Mock-Objekte**
Masterarbeit an der FernUniversität in Hagen, 2008.

[Bec02] K. Beck: **Test Driven Development: By Example**
Addison Wesley, 2002.

[Bec03] K. Beck: **Extreme Programming**
Addison Wesley, 2003.

[Cin00] M. Ó Cinnéide: **Automated Application of Design Patterns: A Refactoring Approach**
PhD thesis, Trinity College Dublin, 2000.

[CR06] E. Clayberg, D. Rubel: **eclipse: Building Commercial-Quality Plug-Ins**
Addison Wesley, 2006.

[Ecl08] Eclipse Foundation: **Manipulating Java code**
http://help.eclipse.org/stable/index.jsp?topic=/org.eclipse.jdt.doc.isv/guide/jdt_api_manip.htm, Juni 2008.

[Eva04] E. Evans: **Domain Driven Design - Tackling Complexity in the Heart of Software** Addison Wesley, 2004.

[Fow00] M. Fowler: **Refactoring: Improving the Design of Existing Code** Addison Wesley, 2000.

[Fow08a] M. Fowler: **Inversion of Control Containers and the Dependency Injection pattern** http://www.martinfowler.com/articles/injection.html, Juni 2008.

[Fow08b] M. Fowler: **Domain Model** http://www.martinfowler.com/eaaCatalog/domainModel.html, Juni 2008.

[Fre08] L. Frenzel: **The Language Toolkit: An API for Automated Refactorings in Eclipse-based IDEs** http://www.eclipse.org/articles/Article-LTK/ltk.html, Juni 2008.

[GB03] E. Gamma, K. Beck: **Contributing to Eclipse: Principles, Patterns, and Plug-Ins** Addison Wesley, 2003.

[GHJV95] E. Gamma, R. Helm, R. Johnson, J. Vlissides: **Design Patterns - Elements of Reusable Object-Oriented Software** Addison Wesley, 1995.

[Hof08] M. Hofmann: **Eclipse Workbench: Using the Selection Service** http://www.eclipse.org/articles/Article-WorkbenchSelections/article.html, Juni 2008.

[Keg07] H. Kegel: **Constraint-basierte Typinferenz für Java 5** Diplomarbeit an der FernUniversität in Hagen, 2007.

[Ker04] J. Kerievsky: **Refactoring To Patterns** Addison Wesley, 2004.

[Kuh08] T. Kuhn: **Abstract Syntax Tree** http://www.eclipse.org/articles/article.php?file=Article-JavaCodeManipulation_AST/index.html, Juni 2008.

[Mar02] M. Marques: **Agile Software Development. Principles, Patterns, and Practices**
Prentice Hall International, Juni 2008.

[Mar08] R. C. Martin: **Exploring Eclipse's ASTParser**
http://www-128.ibm.com/developerworks/opensource/library/os-ast/, 2002.

[Mil08] A. Miller: **Code Spelunking Techniques**
http://tech.puredanger.com/2007/09/18/spelunking/, Juni 2008.

[MG08] W. Melhem, D. Glozic: **PDE Does Plug-ins**
http://www.eclipse.org/articles/Article-PDE-does-plugins/PDE-intro.html, Juni 2008.

[Opd92] W. Opdyke: **Refactoring Object-Oriented Frameworks**
PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[Pad08] K. Padegaonkar: **Contributing Actions to the Eclipse Workbench**
http://www.eclipse.org/articles/article.php?file=Article-action-contribution/index.html, Juni 2008.

[RQZ03] C. Rupp, S. Queins, B. Zengler: **UML 2 glasklar**
Hanser Fachbuchverlag, 2003.

[Sne87] H. M. Sneed: **Software Management**
Verlagsgesellschaft Rudolf Müller GmbH, 1978.

[SM05] F. Steimann, P. Mayer:
Patterns of Interface-Based Programming
Journal of Object Technology,
Volume 4, Nr. 5, S. 75-94, 2005.

[SMM05] F. Steimann, P. Mayer, A. Meißner:
Decoupling classes with inferred interfaces
SAC 06: Proceedings of the 2006 ACM symposium on Applied computing,
ACM Press, S. 1404-1408, 2006.

[WR05] E. Woods, N. Rozanski: **Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives**
Addison Wesley, 2005.