Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Dennis Kilan

Entwicklung und Implementierung eines modellbasierten Dependency Injection Frameworks

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Dennis Kilan

Entwicklung und Implementierung eines modellbasierten Dependency Injection Frameworks

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. Stefan Sarstedt
Zweitgutachter : Prof. Dr. Olaf Zukunft

Abgegeben am 20.08.2010

Dennis Kilan

Thema der Bachelorarbeit

Entwicklung und Implementierung eines modellbasierten Dependency Injection Frameworks

Stichworte

Dependency Injection, Modellbasiert,UML 2, Komponentendiagramm, XMI, Reflection, Dependency Injection, Framework

Kurzzusammenfassung

Die meisten Dependency Injection Frameworks nutzten textuelle Beschreibungen für die Konfiguration eines Software Sytems. Diese Beschreibungen sind oft schwer leserlich und müssen zusätzlich erstellt werden. Die Erstellung von Modellen ist heutzutage ein fester Bestandteil bei dem Entwurf von Software. Diese Arbeit untersucht ob man diese Modelle nicht auch für ein Dependency Injection Framework nutzen kann. Dabei wird untersucht welche Modelle sich eignen, wie das Framework sie lesen kann und wie aus diesen Modellen ausführbarer Code entstehen kann.

Title of the paper

Design and realization of a modelbased dependency injection framework

Keywords

dependency injection, model based, UML 2, componentdiagram, XMI, Reflection

Abstract

Most Dependency Injection frameworks use textual descriptions for system configurations. Those descriptions are more than often not easy to read. These descriptions must be written in addition. Creating models is nowadays a standard task while designing software. This bachelor thesis tries to use these models as descriptions of a system configuration for a dependency injection framework instead of textual descriptions. This thesis examines which type of models are best suitable, how the framework can read these models and how executable code can be generated out of these models.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

1 Einleitung

1.1 Motivation

Komponenten zerlegen ein Software System in unabhängige Einheiten. Jede dieser Komponenten beschreibt ihr Verhalten nach außen durch Schnittstellen. Das Innenleben einer Komponente bleibt anderen Komponenten verborgen. Dadurch sind Komponenten austauschbar und wiederverwendbar. Dieses Vorgehen nennt man „Komponentenbasierte Entwicklung“ und es ist ein bewährtes Programmier-Paradigma.
Wenn eine Komponente Dienste einer anderen Komponente zur Ausführung benötigt, besitzt sie eine Abhängigkeit zu der anderen Komponente. Ein System besteht aus mehreren Komponenten und muss so zusammengesetzt werden, dass jede Abhängigkeit einer Komponente befriedigt werden kann. Das Dependency Injection Entwurfsmuster ist ein häufig gewähltes Vorgehen um ein System so zusammenzustellen, dass alle Abhängigkeiten befriedigt werden können. Im Dependency Injection Entwurfsmuster setzt ein Container die Abhängigkeiten einer Komponente.

Dependency Injection Frameworks helfen dem Anwender beim Erstellen einer Systemkonfiguration. Der Nutzer eines der Dependency Injection Frameworks schreibt die Konfiguration des Systems in eine externe Datei, z.B. im XML Format oder in der Sprache des Systems. Anhand dieser Konfigurationsdateien erstellt das Framework den Container.(Mehr zu Dependency Injection im Kapitel 2.1).
Die derzeit auf dem Markt befindlichen Dependency Injection Frameworks haben einen Nachteil. Das Schreiben der Konfiguration für ein Framework muss in einem speziellem Format erfolgen. Zum Beispiel muss die Konfiguration im XML Format geschrieben werden, so dass der Nutzer dieses Frameworks XML Kenntnisse besitzen muss. Diese Dateien sind oft schwer verständlich und wirken unübersichtlich. Gibt es beispielsweise für ein System 2 Konfigurationen, so sind die Unterschiede in den Konfigurationsdateien meist nur schwer erkennbar.

Die Konfiguration eines Systemes ist Teil des Entwurfes und liegt deshalb auf einem hohem Abstraktionslevel. Eine geeignete Darstellung auf diesem Abstraktionslevel bieten Modelle. Bilder sagen mehr als Worte, ein Modell ist deutlich lesbarer und verständlicher

als Text. So kann man zum Beispiel die beiden Konfiguration desselben Sytems, wie im oberen Abschnitt beschrieben, sehr viel schneller unterscheiden, indem man zum Beispiel die Komponenten nach Typen unterschiedlich einfärbt. Außerdem sind Modelle heutzutage Standard in dem Entwurf moderner Software Systeme:

> No one would imagine constructing an edifice as complex as a bridge or an automobile without first constructing a variety of specialized system models.
> Selic (2003)

Diese Gedankengänge führen zur Frage, ob es nicht möglich sei, ein Dependency Injection Framework zu erstellen, welches Modelle zur Erstellung des Containers nutzt. Diese Bachelorarbeit nimmt diesen Gedankengang auf und nimmt sich zur Aufgabe solch ein Framework zu entwickeln und zu implementieren.

1.2 Zielsetzung

Hauptziel dieser Bachelorarbeit ist die Entwicklung eines modellbasierten Dependency Injection Frameworks in der Programmiersprache Java. Dafür wird zunächst die Möglichkeit eines Modells als Basis für das Framework untersucht. Ziel dieser Untersuchung ist herauszufinden welche Modelle sich besonders eignen und welche Repräsentation des Modells sich am besten lesen lässt. Im Besonderen wird das Format XMI als Repräsentation untersucht.
Nachdem diese Grundlagen geklärt sind, ist das Ziel das modellbasierte Dependency Injection Framework zu entwerfen und zu implementieren. Zum Entwurf gehört die Wahl eines passenden Parsers zum Einlesen der Repräsentation des Modells, das Verarbeiten der Information aus dieser Repräsentation, sowie das Erstellen des Containers.
Diese Arbeit versucht weiterhin zu klären, welche Vorteile und welche Nachteile der modellbasierte Ansatz beim Entwurf eines Dependency Injection Frameworks besitzt.

1.3 Gliederung

Diese Bachelorarbeit beginnt mit einer theoretischen Einführung[2], in der das nötige Hintergrundwissen vermittelt wird. So erklärt die Bachelorarbeit zunächst das Dependency Injection Entwurfsmuster. Mit diesem Wissen wird ein fiktives Beispielsystem eingeführt. Das Beispielsystem dient nicht nur zum Test des in dieser Arbeit entwickelten Frameworks, sondern dient auch als Basis für eine Einführung und Vergleich von drei bekannten Frameworks: Google Guice, PicoContainer und Spring.

Außerdem untersucht die theoretische Einführung welche Modelle sich für ein modellbasiertes Dependency Injection Framework eignen. Um diese Modelle für einen Computer lesbar zu machen, wird eine Zwischenrepräsentation benötigt. Diese Arbeit stellt das XMI Format als Zwischenrepräsentation und die Möglichkeiten XMI Dateien zu Parsen in der theoretischen Einführung vor.

In einem Analyseteil(3) wird geklärt, was die Anforderungen an das in dieser Arbeit zu entwickelnde modellbasierte Framework sind. Die Analyse beschäftigt sich zunächst mit einigen Bedingungen über das Komponentendiagramm. Unter Berücksichtigung dieser Annahmen werden die Anforderungen an das modellbasierte Framework formuliert. Der Analyseteil schließt mit einem Beispielszenario ab, das darstellt wie das Framework benutzt werden soll.

Das Kapitel Design[4] stellt zunächst die Systemarchitektur vor. Dazu wird geklärt, welche Komponenten benötigt werden um die Umwandlung des Modells bis hin zum Container in der gewünschten Zielsprache zu realisieren. Der Feinentwurf dieser Komponenten erfolgt in den nächsten Abschnitten. Diese Arbeit schließt das Kapitel Design mit dem Entwurf des Containers ab.

Das Kapitel Realisierung [5] beschäftigt sich mit der tatsächlichen Umsetzung des Frameworks. Dabei liegt der Fokus auf der Fachlichkeit und interessante Aspekte der Realisierung.

In einem Testkapitel [6] werden zunächst wichtige Funktionen der Komponenten des Framework auf ihre korrekte Funktionsweise getestet. Abschließend wird in dem Kapitel das gesamte Framework einem Systemtest unterzogen.

Ein Schlußteil[7] fasst diese Arbeit noch einmal zusammen und bewertet das modellbasierte Dependency Injection Framework und die Erkenntnisse aus dieser Arbeit.

2 Theoretische Einführung

2.1 Dependency Injection Grundlagen

Dependency Injection ist ein Entwurfsmuster um Abhängigkeiten von Objekten zu setzen. Um Dependency Injection weiter zu erläutern, werden hier 3 Begriffe eingeführt:

Abhängigkeit Ein Objekt A besitzt eine Abhängigkeit zu einem anderem Objekt B, wenn Objekt A einen Service des Objektes B benötigt um seine Dienste auszuführen. Damit ein Objekt korrekt funktioniert, müssen alle seine Abhängigkeiten befriedigt werden. Um Dependency Injection nutzen zu können, müssen Objekte ihre Abhängigkeiten explizit machen, beispielsweise über spezielle Setter Methoden oder Parameter des Konstruktors. Nur so kann mittels Dependency Injection die Abhängigkeiten eines Objektes gesetzt werden.

Service Bietet ein Objekt einen Dienst über seine öffentlichen Methoden an, so nennt man dieses einen Service des Objektes. Services sollten über ein Interface definiert werden, damit derselbe Service von mehreren Objekten verschieden implementiert werden kann. Dadurch kann der Anwender das Verhalten eines Objektes durch einen Austausch der Implementierung des Services verändern. Objekte, die einen Service über eine Schnittstelle nutzen, besitzen keine direkte Abhängigkeit zum Serviceanbieter.

Container Ein Container kennt alle Abhängigkeiten eines Systems. Der Anwender kann den Container nach einem speziellem Objekt fragen. Der Container sorgt dafür, dass dieses angefragte Objekt alle seine Abhängigkeiten gesetzt bekommt und liefert ein voll funktionsfähiges Objekt zurück. Der Container *injiziert* die Abhängigkeiten in das Objekt, bevor es dieses an den Anwender zurückliefert.

Klassischerweise sorgen Objekte selbst dafür, dass alle ihre Abhängigkeiten gesetzt werden. Dadurch sind jedoch die Anbieter der Services sehr stark an die Service nutzenden Objekte gekoppelt. Durch Dependency Injection werden die Abhängigkeiten eines Objektes von außen gesetzt, zum Beispiel über Setter Methoden oder Konstruktor Argumente. Deswegen spricht man hier auch von *Inversion of Control*.

Laut Fowler Fowler (2004) ist Inversion of Control jedoch ein viel zu weit gefasster Begriff, denn dieses sei eine Eigenschaft vieler Frameworks:

> When these containers talk about how they are so useful because they implement „Inversion of Control“ I end up very puzzled. Inversion of control is a common characteristic of frameworks, so saying that these lightweight containers are special because they use inversion of control is like saying my car is special because it has wheels. Fowler (2004)

Dependency Injection ist nur eine Spezialform des „Inversion of Control“ Prinzips. Klassischerweise werden 3 Arten von Dependency Injection unterschieden:

Typ 01 IOC: Interface Injection Bei Interface Injection werden die Abhängigkeiten in einem Interface über die Methoden des Interfaces explizit gemacht. Alle Objekte, die diese Abhängigkeit injiziert haben sollen, müssen dieses Interface implementieren.
Vorteile: Durch das Interface müssen die Objekte alle Methoden implementieren, so dass es für jeden Service eine passende Methode zum injizieren gibt. Objekte können durch diese Methoden den Anbieter eines Services zur Laufzeit ändern. Zyklische Abhängigkeiten stellen für Interface Injection kein Problem dar, da die Abhängigkeiten einzeln durch die Methoden injiziert werden.
Nachteile: Ein Objekt erst dann vollständig konfiguriert ist, wenn jede Methode des Interface aufgerufen wurde. Ein vergessener Aufruf nur einer dieser Methoden kann zu fehlerhaftem Verhalten des Systems führen.

Typ 02 IOC: Setter Injection Bei Setter Injection werden die Abhängigkeiten über Setter Methoden des Objektes explizit gemacht und werden über diese Methoden gesetzt.
Vorteile: Die Implementierung des benötigten Services kann auch zur Laufzeit geändert werden, somit kann das Objekt zur Laufzeit sein Verhalten ändern. Ist eine zyklische Abhängigkeit vorhanden, so kann Setter Injection wie Interface Injection damit umgehen, da hier die Abhängigkeiten auch einzeln gesetzt werden.
Nachteile: Setter Injection hat denselben großen Nachteil wie Interface Injection: Ein Objekt ist erst nach Aufruf aller Settermethoden vollständig konfiguriert.

Typ 03 IOC: Constructor Injection Bei Constructor Injection werden die Abhängigkeiten über Argumente des Konstruktors explizit gemacht. Alle Abhängigkeiten werden beim Erstellen des Objektes gesetzt.
Vorteile: Ein Objekt nach seiner Initialisierung voll funktionsfähig. Es können keine Fehler entstehen, wenn ein Objekt einen noch nicht gesetzten Service nutzen will. Durch Constructor Injection kann eine Initialisierungsreihenfolge der Abhängigkeiten vorgegeben werden. Constructor Injection kann eine Abhängigkeit immutable machen, so dass sie nicht mehr zur Laufzeit geändert werden kann.

Nachteile: Alle Abhängigkeiten stehen im Konstruktor des Objektes. Die Parameterliste kann sehr lang werden, besonders wenn auch Vererbung betrachtet wird. In der Unterklasse müssen auch alle Abhängigkeiten der Oberklasse gesetzt werden. Lange Parameterlisten sind ein sogenannter „Bad Smell“. Lange Parameterlisten des Konstruktors lassen ein Objekt komplex erscheinen. Ein Service kann zur Laufzeit nicht geändert werden.

Außer diesen 3 Hauptarten gibt es noch weitere Arten wie z.B. Field Injection oder Mischformen. Interface Injection, Setter Injection und Constructor Injection sind jedoch die bekanntesten und am häufigsten genutzten Arten. Die anderen Arten der Dependency Injection werden deshalb hier nicht weiter erläutert.
Die folgende Abbildung vergleicht Constructor Injection mit(C) und ohne Framework(B), sowie dem Verzicht auf Dependency Injection(A):

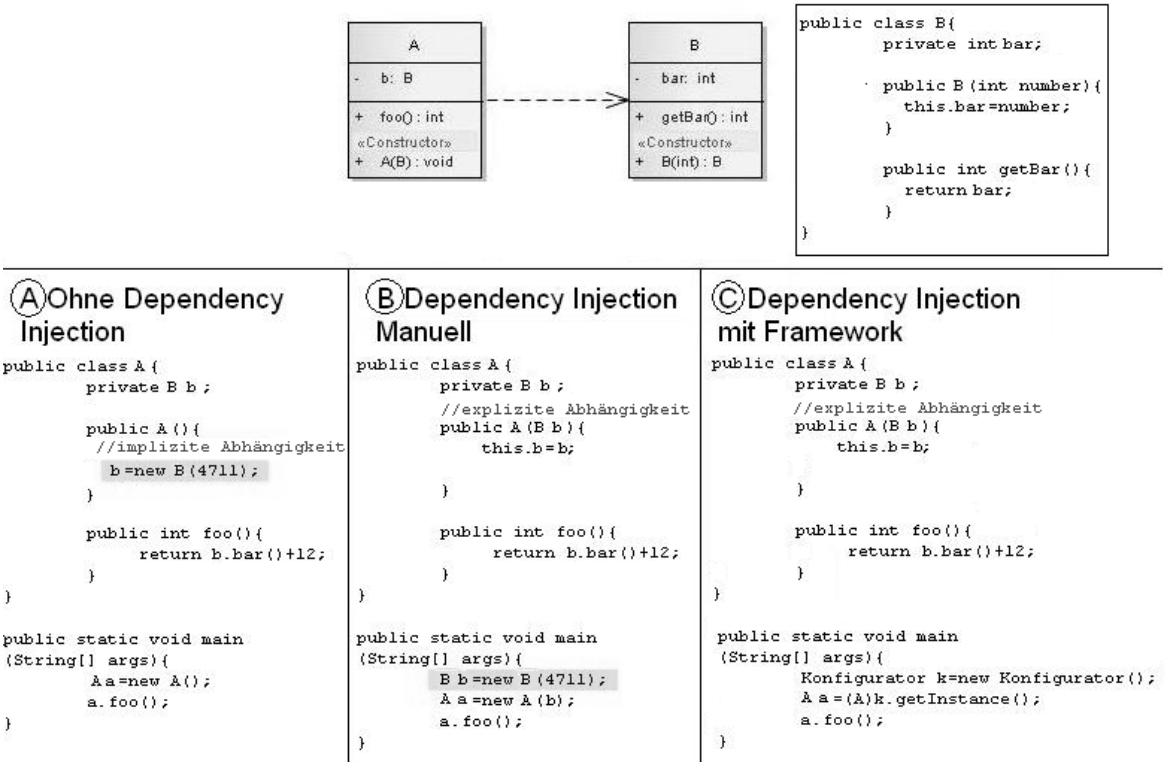

Abbildung 2.1: Umsetzung der Dependency Injection

In Teil A des Bildes wird keine Dependency Injection verwendet. Die Abhängigkeit zu Klasse B wird im Konstruktor gesetzt. Klasse A muss wissen wie Klasse B initialisiert wird. Ändert sich der Konstruktor von Klasse B, so muss auch Klasse A angepasst werden. Klasse A hat somit eine starke Kopplung zu Klasse B.

In Teil B des Bildes wird Dependency Injection ohne ein Framework genutzt. Klasse A bekommt eine Instanz der Klasse B im Konstruktor übergeben. B muss nicht mehr wissen wie Klasse A initialisiert wird. Die Kopplung aus Teil A wurde mittels Dependency Injection aufgehoben. Jedoch müssen in der main Methode alle Abhängigkeiten der Klasse A manuell erstellt werden, bevor man sie in Klasse A injizieren kann. Besitzt ein Objekt viele Abhängigkeiten, so müssen vor Nutzung dieses Objektes auch alle Serviceanbieter und deren Abhängigkeiten erzeugt werden. Dieses ist aufwändig und fehleranfällig.
In Teil C des Bildes wird ein fiktives Framework genutzt. Mit der Nutzung des Frameworks kann einfach nach einer Instanz von A gefragt werden kann, ohne dass explizit eine Instanz der Klasse B erstellt werden muss. Das Framework kümmert sich darum, dass alle Abhängigkeiten von A gesetzt werden.

Einen Überblick über die Vor-und Nachteile von Dependency Injection gibt folgende Aufzählung:

Vorteile von Dependency Injection

- Das Objekt ist nicht mehr direkt abhängig vom Anbieter des Service. Das Objekt kann dadurch in verschiedenen Kontexten eingesetzt werden.
- Änderungen an der Systemkonfiguration brauchen nur noch in der Konfigurationsdatei erfolgen, somit lassen sich zum Beispiel schnell reale Objekte durch Mock Objekten ersetzen.

Nachteile von Dependency Injection

- Die Konfiguration des Systemes wird externalisiert. Ein Blick auf den Code reicht nicht mehr aus um die Funktionsweise eines Systems zu verstehen.
- Wegen der Externalisierung kann es zu Problemen mit einigen IDEs geben, wenn diese das Framework nicht unterstützen. So kann es passieren, dass automatisierte Code Analyse oder einige Refactoring Methoden dieser IDEs nicht mehr funktionieren.

2.2 Ein Beispielsystem: „Versandhaus“

Dieses Kapitel führt ein Beispielsystem ein. Dieses System dient sowohl dem Test des in dieser Arbeit entwickelten Frameworks als auch einem Vergleich einiger Frameworks in den nächsten Kapiteln. Das Beispielsystem ist rein fiktiv und deshalb bewußt sehr simpel gehalten. Es dient zum Vermitteln des Konzeptes von Dependency Injection und soll für den Leser verständlich sein.

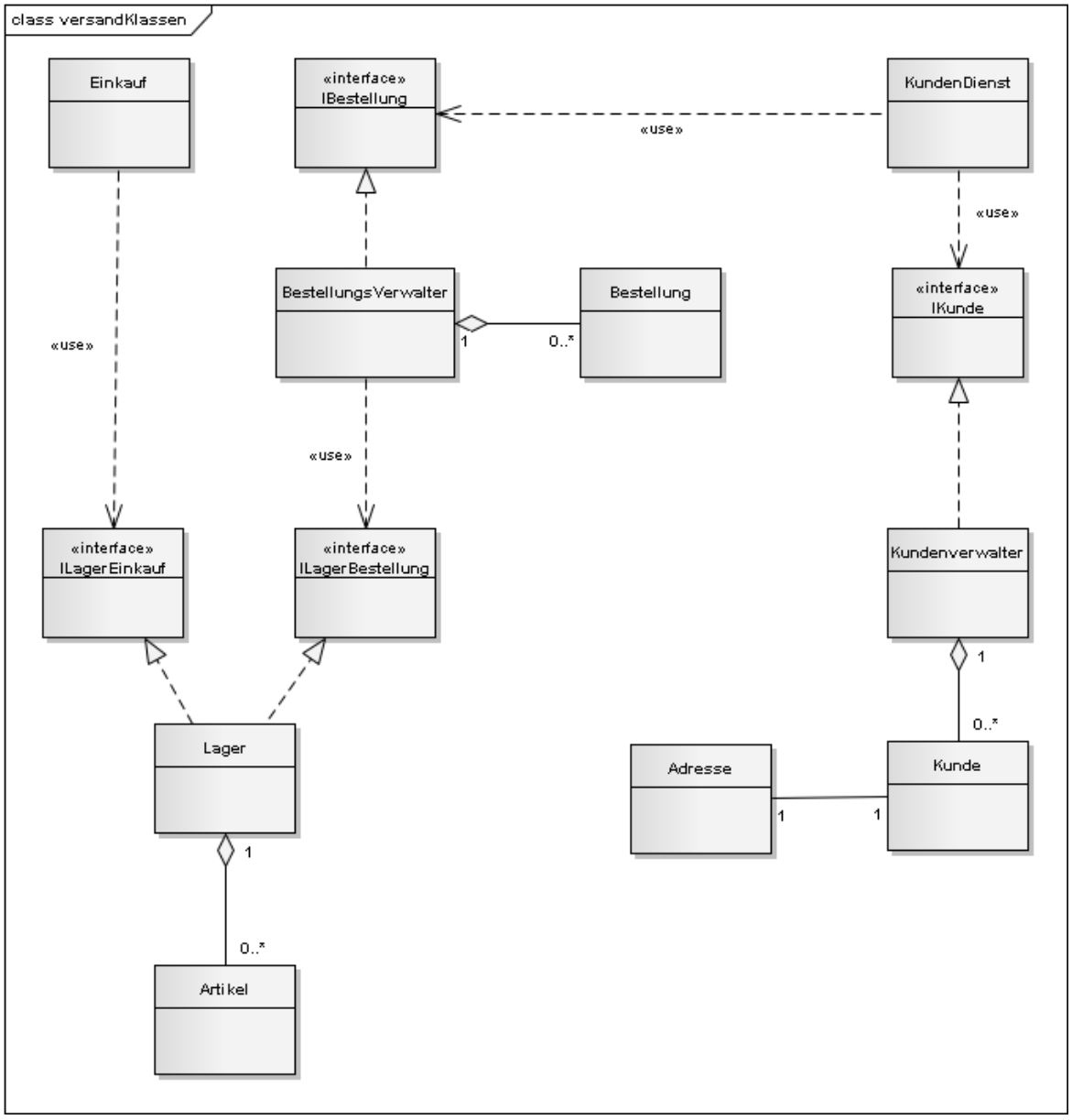

Abbildung 2.2: Klassendiagramm des Versandhauses

Das Beispielsystem ist ein einfaches Versandhaussystem. Es gibt dort Kunden, die über den Kundendienst Waren bestellen können. Der Kundendienst leitet eine Bestellung an den Bestellungsverwalter weiter, der überprüft ob es noch genügend Artikel für die Bestellung im Lager gibt. Sind noch genug Artikel vorhanden, so entnimmt der Bestellungsverwalter die Artikel aus dem Lager. Der Einkauf sorgt dafür, dass immer genug Artikel im Lager vorhanden sind.

Das Versandhaus lässt sich in folgende Komponenten einteilen, die die Architektur des Versandhauses bestimmen:

Kundenkomponente Diese Komponente verwaltet die Kunden. Sie legt zum Beispiel neue Kunden an.

Kundendienstkomponente Diese Komponente ist eine Schnittstelle für die Interaktion mit dem Anwender des Versandhauses. Der Anwender übernimmt dabei die Rolle des Kundendienstes. Es können neue Kunden angelegt werden oder Bestellungen für bestehende Kunde angelegt werden. Dafür benötigt diese Komponente Zugriff auf die Bestellungskomponente.

Bestellungskomponente Die Bestellungskomponente dient zum Anlegen neuer Bestellungen und der Überwachung bereits angelegter Bestellungen.

Lagerkomponente Diese Komponente spielt 2 Rollen. Einerseits stellt sie Dienste für die Einkaufskomponente zur Verfügung auf der anderen Seite aber auch Dienste für die Bestellungskomponente. Ihre Hauptaufgabe ist das Verwalten von Artikeln und deren Bestand im Lager. Der Einkauf erhöht den Bestand der Artikel während die Bestellungskomponente diese wiederum verringert. Damit in beiden Rollen dasselbe Lager verwendet wird, muss dieses nur einmal instanziiert werden.

Einkaufskomponente Die Einkaufskomponente ist eine Schnittstelle für die Interaktion mit dem Anwender. Der Anwender übernimmt dabei die Rolle des Einkaufs. Mit der Einkaufskomponenten kann der Bestand an Artikeln aufgefüllt werden und auch neue Artikel zum Sortiment hinzugefügt werden.

Die Kundendienst- und die Einkaufskomponente sind beides Schnittstellen zwischen dem System und dem Anwender des Systems. In einer speziellen Startklasse zum Starten des Systemes müssen diese beiden Komponenten erzeugt werden. Alle anderen Komponenten sind direkte oder indirekte Abhängigkeiten des Kundendienstes oder des Einkaufs. Deren Erzeugung ist die Aufgabe des Frameworks. Durch Dependency Injection sollten diese Komponenten einfach austauschbar sein, solange die Abhängigkeiten das Protokoll der jeweilig angebotenen Interfaces erfüllen. Das Versandhaus hat neben der Release Konfiguration auch eine Testkonfiguration mit Mock Objekten.

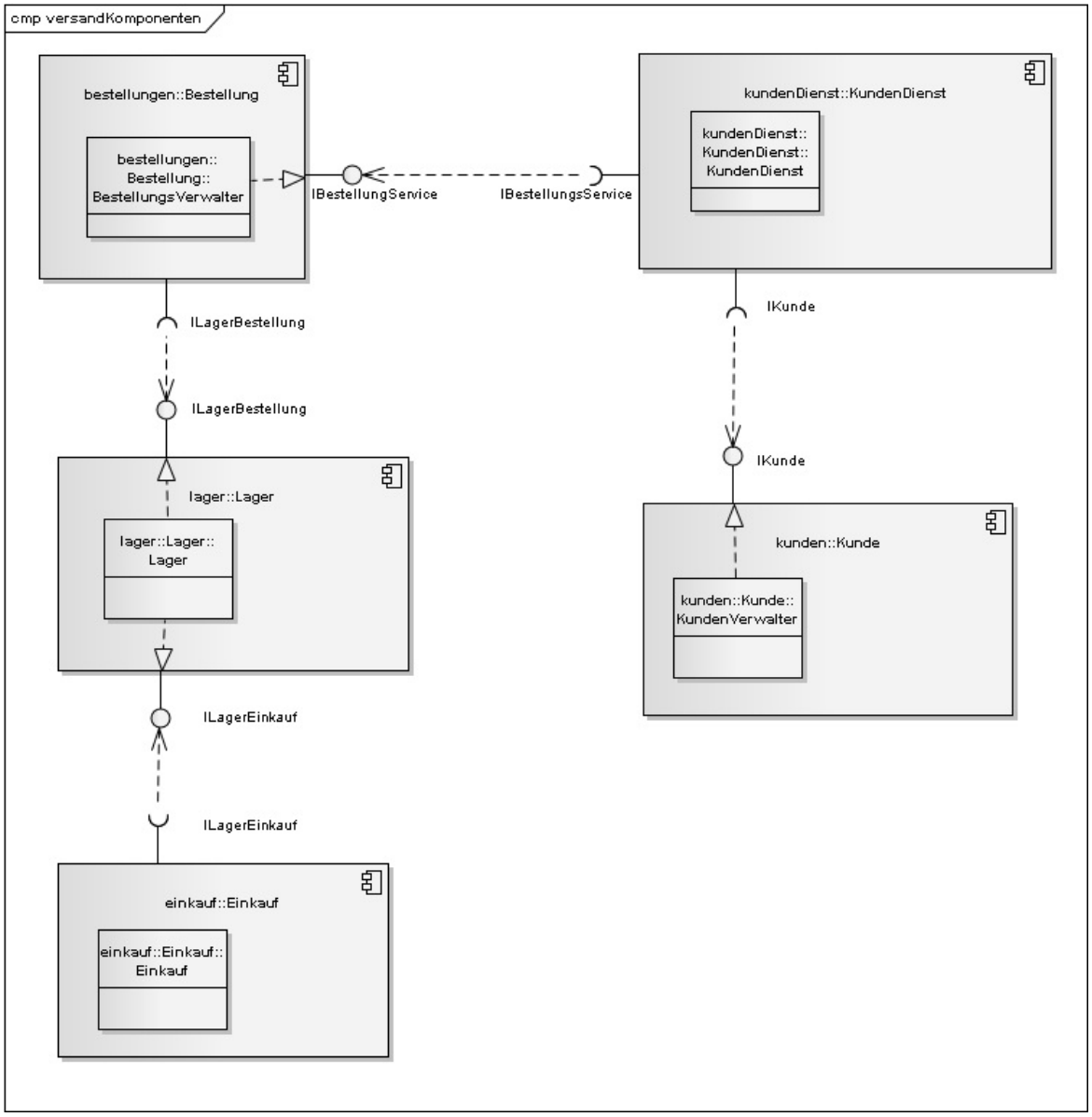

Abbildung 2.3: Komponentendiagramm des Versandhaussytems (Release Konfiguration)

2.3 Dependency Injection Frameworks

Wenn Dependency Injection manuell genutzt wird, muss es mindestens eine Klasse geben, die das System zusammensetzt. Wird in dieser Klasse ein Objekt eines bestimmten Types erstellt, so müssen in dieser Klasse auch alle Abhängigkeiten dieses Objektes erzeugt werden (siehe auch Abbildung 2.1 Abschnitt B). Haben die Abhängigkeiten des Objektes wiederrum Abhängigkeiten, so müssen diese zusätzlich auch noch in dieser Klasse erzeugt werden. Moderne Softwaresysteme haben hunderte bis tausende von Objekten, somit werden diese Klassen groß, unübersichtlich und das Schreiben dieser Klasse fehleranfällig.

Der Nutzer eines Frameworks schreibt die Konfigurationsdatei in einem Framework spezifischen Dialekt und das Framework erstellt daraus automatisch diese Klasse. In der Konfigurationsdatei werden alle Objekte, die das Framework erzeugen soll, sowie deren Abhängigkeiten festgehalten. Meist muss, wenn eine Abhängigkeit zu einem Interface besteht, auch die implementierende Klasse für dieses Interface in die Konfigurationsdatei geschrieben werden.

Das Wissen, wie ein Objekt konfiguriert wird, steckt bei Nutzung eines Frameworks nicht mehr in der Anwendung, sondern in der Konfigurationsdatei. Benötigt die Anwendung ein bestimmtes Objekt, so fragt es das Framework nach diesem Objekt, und das Framework liefert ein vollständig initialisiertes Objekt zurück. Der Anwender des Frameworks kann einfach durch Austausch zwischen der Konfigurationsdatei die Konfiguration des Systems ändern, ohne den Quellcode der Anwendung anpassen zu müssen. Der Anwender muss nun jedoch zusätzlich in die Konfigurationsdatei schauen, um die Funktionsweise der Anwendung zu verstehen. Ein Blick in den Quellcode des Anwendungsystems reicht nicht mehr aus.

Für die Programmiersprache Java gibt es 3 bekannte Dependency Injection Frameworks: „Pico Container“,“Spring“ und „Google Guice“. Diese 3 Frameworks zwecks eines Vergleiches in den nächsten Kapiteln vorgestellt.

2.3.1 PicoContainer (Version 2.10)

PicoContainer ist ein Open Source Dependency Injection Framework von Codehaus. Im PicoContainer Framework werden die Konfigurationen in Java Code geschrieben. Das Herzstück von PicoContainer sind die Container Klassen, in denen die benötigten Konfigurationen abgelegt werden. PicoContainer bietet hierfür drei Möglichkeiten an: 1. direkt als .class Datei, 2. mit einem zusätzlichen Namen, für einfacheren Zugriff oder 3. als bereits vorinitialisiertes Exemplar, so dass PicoContainer es nicht selbst erzeugen muss. Der Anwender muss alle Objekte und deren Abhängigkeiten in der Container Klasse registrieren. Außerdem sollte für

jedes Interface eine Implementierung in der Container Klasse registriert werden. Dieses ist jedoch nicht erforderlich, wenn es für ein Interface nur eine mögliche Implementierung gibt. Das Registrieren der Objekte zeigt folgendes Codebeispiel:

```
public class TestConfiguration {
    //Der Container in denen die Konfiguration abgelegt wird
    MutablePicoContainer pico=new DefaultPicoContainer(
       parent);

    public TestConfiguration(){
        /* Hier werden die Objekte in der
         * Container Klasse registriert */
        pico.addComponent(mockObjects.KundenMock.class);
        pico.addComponent(mockObjects.LagerMock.class);
        pico.addComponent(mockObjects.BestellungsserviceMock
           .class);

        /* Der Einkauf und der Kundendienst
        *   werden mit einem Identifier
        * registriert*/
        pico.addComponent("Einkauf",einkauf.Einkauf.class);
        pico.addComponent("KundenDienst",kundenDienst.
           KundenDienst.class);
    }}
```

Listing 2.1: Registrieren der Abhängigkeiten im PicoContainer (Test Konfiguration)

Nachdem man die Konfiguration erstellt hat bzw. die Objekte registriert hat, kann man diesen Container in einer beliebigen Klasse des Projektes instanziieren.
Der Anwender kann sich dann mittels der getComponent() eine Instanz des gewünschten Types aus dem Container holen.

```
public class PicoStarter {

    public static void main(String[] args) {
        MutablePicoContainer pico;
        pico=new TestConfiguration().getContainer();

        /* Es kann auch auf die Initialisierung dieser
        * Klassen verzichtet werden, wenn man die Frames
        * beim Container registriert*/
        Einkauf einkauf=(Einkauf)pico.getComponent("Einkauf"
           );
        KundenDienst kd=(KundenDienst)pico.getComponent("
           KundenDienst");

        new KundenDienstFrame(kd);
        new EinkaufFrame(einkauf);

    }

}
```

Listing 2.2: Besorgen der Abhängigkeiten aus dem PicoContainer (Test Konfiguration)

PicoContainer beherrscht Constructor Injection, Setter Injection und noch weitere Injection Typen wie z.B. Field Injection. Die Entwickler des PicoContainer Frameworks empfehlen die Nutzung von Constructor Injection.

Standardmäßig wird beim jedem Aufruf der getComponent() Methode ein neues Exemplar des gewünschten Types erzeugt. Möchte der Anwender das bei jedem Aufruf dasselbe Exemplar zurückgeliefert wird, bietet PicoContainer das Caching an. Beim Caching wird ein fertig erstelltes Objekt im Container gehalten, und beim Aufruf eine Referenz auf dieses eine Exemplar zurückgegeben.

PicoContainer bietet zusätzlich zu Dependency Injection auch eine Lebenszyklusverwaltung von Objekten an, wenn diese im Container „gecached“ werden. Durch die Lebenszyklusverwaltung können beim Erzeugen und Zerstören von Objekten spezielle Methoden aufgerufen werden. So könnte zum Beispiel beim Initialisieren der Lagerklasse des Beispielsystems ein Methode zum Herstellen einer Verbindung zu einer Datenbank aufgerufen werden.

PicoContainer erkennt zyklische Abhängigkeiten zur Laufzeit und reagiert darauf mit einer Exception. Weiterhin erkennt PicoContainer auch mehrdeutige Abhängigkeiten und wirft hierauf wieder eine Exception. Eine mehrdeutige Abhängigkeit besteht, wenn das Framework mehrere Auswahlmöglichkeiten hat um eine Abhängigkeit zu befriedigen. Mehrdeutigkeiten kann der Anwender durch Angabe von Parametern, die die Wahl des passenden Konstruktors beeinflussen, auflösen.

Vorteile
PicoContainer ist eine Lightweight Framework. PicoContainer kann einfach in das Projekt als Bibliothek eingebunden werden. Es entstehen keine neuen Abhängigkeiten. Die Konfiguration wird in der Zielsprache geschrieben und ist dadurch für den Anwender verständlicher als eine fremde Sprache. PicoContainer bietet die Möglichkeit der Lebenszyklusverwaltung von Objekten an.

Nachteile
PicoContainer nimmt immer den längsten Konstruktor dessen Abhängigkeiten es befriedigen kann. In PicoContainer kann pro Typ immer nur eine mögliche Implementation registriert werden.

2.3.2 Google Guice (Version 2.0)

Google Guice ist ein Dependency Injection Framework von Google, die Version 1.0 wurde 2007 veröffentlicht. Guice setzt Dependency Injection durch Annotationen im Code und durch Module um. In einem Modul werden wie bei dem Container des PicoContainers die Konfigurationen erstellt. Module bieten jedoch keine Lebenszyklusverwaltung an.

Im Guice Framework markieren Annotationen die Stellen an denen Guice die Abhängigkeiten injizieren soll. Eine Annotation vor einem Konstruktor bedeutet beispielsweise, dass Constructor Injection verwendet werden soll. In den Modulen wird zu jedem Typen eine passende konkrete Klasse registriert. Guice nennt dieses auch „binden“.

Das Binden zeigt folgendes Code Beispiel:

```
public class TestModule extends AbstractModule{

    @Override
    protected void configure() {
        ...
        /* Das Lager spielt zwei Rollen, sollte aber nur
           einmal instanziiert werden*/
        bind(mockObjects.LagerMock.class).in(Singleton.class
           );
        bind(ILagerBestellung.class).to(mockObjects.
           LagerMock.class);
        bind(ILagerEinkauf.class).to(mockObjects.LagerMock.
           class);
    }
}

/* Ausschnitt aus der Einkaufklasse, in die Guice
* das Lager injizieren soll */
public class Einkauf{
    private ILagerEinkauf lager;

    /* Inject Annotation teilt Guice mit, dass es hier die
    * Abhängigkeit des Konstruktors setzen soll*/
    @Inject
    public Einkauf(ILagerEinkauf lager){
        this.lager=lager;
    }
}
```

Listing 2.3: Binden der Abhängigkeiten in Google Guice (Test Konfiguration)

Falls es pro Typen mehrere Implementierungen in einem Modul geben soll, bietet Guice ein explizites Binden mittels einer „@Named(„...“)“ Annotation an. Guice bietet sogar die Möglichkeit an, ganz auf Module zu verzichten, indem man Interfaces mit der „@implementedBy(...)“ annotiert. Dieses kann zum Beispiel genutzt werden um für ein Interface eine Standard Implementierung festzulegen.

Das Holen von Instanzen aus den Modulen erfolgt so einfach wie in PicoContainer. Es muss zuerst eine Injektor Instanz erstellt werden aus der das gewünschte Exemplar geholt werden kann.

```
public class GuiceStarter {

    public static void main(String[] args) {
        Injector inject=Guice.createInjector(new guice.
           TestModule());

        Einkauf einkauf=inject.getInstance(Einkauf.class);
        KundenDienst kd=inject.getInstance(KundenDienst.
           class);

        new KundenDienstFrame(kd);
        new EinkaufFrame(einkauf);

    }

}
```

Listing 2.4: Holen der Abhängigkeiten aus Guice (Test Konfiguration)

Guice beherrscht Constructor Injection, Setter Injection und einige Spezialformen der Dependency Injection wie z.B. Field Injection. Die Entwickler von Guice empfehlen aber keine spezielle Injection Art, wie es bei Pico Container der Fall ist.

Per Default liefert Guice bei jedem Aufruf der getInstance() eine neue Instanz. Möchte der User immer dieselbe Instanz eines Objektes beim Aufruf der getInstance() bekommen, so muss er das Objekt in den „Singleton“ Scope von Guice stecken. Die Scopes von Guice bestimmen die Sichtbarkeit eines Objekts. Beispielsweise kann durch Scopes in einer Web Anwendung ein Objekt nur während einer Session verfügbar/sichtbar sein (Session Scope).

Guice hat einen besonderen Umgang mit zyklischen Abhängigkeiten. Besteht eine zyklische Abhängigkeit, so ersetzt Guice eine der beiden Abhängigkeiten temporär durch ein Proxy Objekt. Die zyklische Abhängigkeit ist somit ausgehebelt. Das Proxy Objekt wird nach Auflösung der Abhängigkeit durch ein echtes Objekt ersetzt. Guice kann diese Technik nur anwenden, wenn die Abhängigkeit über ein Interface gekapselt ist.
Bei mehrdeutigen Abhängigkeiten wird, wie bei PicoContainer, eine Exception geworfen.

Vorteile
Guice ist ein Lightweight Framework, es muss wie bei PicoContainer lediglich die Guice Bibliothek heruntergeladen und ins Projekt eingebunden werden. Durch Annotationen können mehrere Implementation eines Types in einem Modul verwaltet werden. Durch implizites Binden (z.B. durch „@ImplementedBy“) kann eine Konfiguration schnell und mit wenig Aufwand erstellt werden. Guice kann durch Proxy Objekte mit zyklischen Abhängigkeiten umgehen.

Nachteile
Bestehende Systeme müssen bei Nutzung von Guice stark angepasst werden, so müssen an vielen verschiedenen Stellen Annotation erstellt werden. Guice bietet keine Lebenszyklusverwaltung an. Das Injizieren von primitiven Parametern gestaltet sich mit der aktuellen Version von Guice schwierig. Man kann im Modul einen konkreten Wert an einen primitiven Typen binden, dadurch wird aber an allen Stellen an denen dieser primitive Typ auftritt der im Modul zugewiesene Wert verwendet. Alternativ muss man jeden primitiven Typen im Code extra durch die @Named Annotation annotieren.

2.3.3 Spring (Version 2.56)

Das Spring Framework ist ein großes Framework von SpringSource. Spring ist kein reines Dependency Injection Framework. Es bietet noch viele andere Dienste an, wie zum Beispiel Zugriff auf entfernte Objekte. Dependency Injection ist aber einer der Kerndienste vom Spring Framework.
Spring teilt die Objekte eines Systems in Beans ein. Eine Spring Bean sollte aber nicht mit einer Java Bean verwechselt werden. Spring Beans unterliegen nicht den strengen Anforderungen an Java Beans. Eine Spring Bean, nachfolgend nur noch Bean genannt, kann auch ein einfaches Java Objekt sein. Spring nutzt Container um die Beans mit ihren Abhängigkeiten (in Spring Collaborator gennant) zusammenzusetzten.
Die Konfiguration des Systemes erfolgt im Spring Framework klassischerweise über XML Dateien. Ab Version 2.5 können die Konfiguration aber auch beispielsweise in Java Code geschrieben werden.

Der folgende Code Ausschnitt zeigt eine klassische Konfiguration über das XML Format:

```
<?xml version="1.0" encoding="UTF-8"?>
...
    <!-- Jede Komponente wird als Bean erstellt -->
    <bean id="kundenDienst" class="kundenDienst.KundenDienst
       ">
        <!-- Abhängigkeiten über Konstruktor Argumente -->
        <constructor-arg>
            <ref bean="kunde" />
        </constructor-arg>
        <constructor-arg>
            <ref bean="bestellung" />
        </constructor-arg>
    </bean>

    ...

    <!-- Singleton ist Default -->
    <bean id="lager" class="mockObjects.LagerMock" />

    <bean id="einkauf" class="einkauf.Einkauf">
        <constructor-arg>
            <ref bean="lager" />
        </constructor-arg>
    </bean>

</beans>
```

Listing 2.5: Definieren der Beans in Spring (Test Konfiguration)

Um sich ein Objekt aus der Konfiguration zu holen, muss zunächst ein ApplicationContext erstellt werden, aus denen sich das Objekt geholt werden kann. Da die Konfiguration hier in einem externen Format geschrieben wird, kann Spring keine Typüberprüfung durchführen. Die Objekte müssen auf den Zieltypen gecastet werden.

```
public class SpringStarter {

    public static void main(String[] args) {
        ApplicationContext ctx=new
           ClassPathXmlApplicationContext("testConfiguration
           .xml");

        Einkauf einkauf=(Einkauf)ctx.getBean("einkauf");
        KundenDienst kd=(KundenDienst) ctx.getBean("
           kundenDienst");

        new KundenDienstFrame(kd);
        new EinkaufFrame(einkauf);

}
```

Listing 2.6: Laden eines Objektes aus einem Spring Container (Test Konfiguration)

Spring beherrscht sowohl Constructor- als auch Setter Injection. Die Entwickler von Spring empfehlen die Nutzung von Setter Injection, da die Parameterlisten der Konstruktoren durch Constructor Injection zu lang und unübersichtlich werden können und mit Setter Injection die Objekte zur Laufzeit umkonfiguriert werden können.

Zyklische Abhängigkeiten werden von Spring erst zur Laufzeit erkannt und es wird wie in PicoContainer eine Exception geworfen. Da über das <ref> Attribut in der Konfiguration nur konkrete Objekte referenziert werden dürfen, tritt in Spring die Problematik mehrdeutiger Abhängigkeiten nicht auf.

In Spring wird per Default beim Aufruf der getBean() Methode immer dasselbe Objekt zurückgeliefert, möchte der Anwender dass bei jedem Aufruf ein neues Objekt erzeugt wird, so muss er eine „Prototype Bean“ erstellen. Eine Prototype Bean dient als Blaupause für ein Objekt. Ein Bean wird zu einer Prototype Bean, indem der Anwender einfach in der Konfiguration das singleton Attribut auf false setzt.

Spring bietet wie PicoContainer auch die Verwaltung des Lebenszyklus der erstellten Beans an. Der Anwender kann in der XML Konfiguration für eine Bean eine „init“ und eine „destroy“ Methode setzten, die beim Erstellen bzw. beim Zerstören eines Objektes aufgerufen werden soll.

Eine Besonderheit von Spring ist das Auto-Wiring. Beim Auto Wiring definiert man in der Konfiguration nur die vorhandenen Objekte, ohne dass man die Abhängigkeiten explizit setzt. Das Spring Framework sucht sich beim Auto Wiring die Abhängigkeiten eines Objektes heraus und versucht sie zu befriedigen. Dieses spart dem Schreiber einer Konfiguration viel Arbeit und einem Objekt können z.B. neue Abhängigkeiten hinzugefügt werden, ohne dass die Konfiguration angepasst werden muss. Auto Wiring ist jedoch nicht ratsam, man überlässt der Intelligenz von Spring das Konfigurieren eines Systems. Das Verhalten des Systems ist nicht mehr vorhersehbar.

Vorteile
Durch die XML Struktur (z.B. die <ref> Attribute) sind die Abhängigkeiten eines Objektes sehr gut sichtbar. Da Spring mehr als nur Dependency Injection anbietet, kann man es auch für viele weitere Aufgaben des Projektes nutzen.

Nachteile
Möchte der Anwender ein reines Dependency Injection Framework, ist Spring zu mächtig. Das Softwareprojekt wird zu stark von Spring abhängig. Das XML Format ist sehr wortreich, somit sind XML Konfiguration schwerer zu schreiben oder zu lesen als beispielsweise Java Code. Eine Bean kann kein <ref> Attribut zu einem Interface haben. Wird die Implementierung eines Interfaces geändert, so muss die implementierende Klasse an vielen Stellen geändert werden. Es ist nicht möglich, in der Konfiguration einfach nur zentral die Abbildung Interface Implementierung zu ändern.

2.3.4 Zusammenfassung des Vergleiches

Jedes der hier vorgestellten Frameworks hat seine Vor- und Nachteile in bestimmten Anwendungsbereichen. Der große Hauptunterschied zwischen diesen 3 Frameworks ist die Art wie die Konfiguration geschrieben wird. Das Holen eines Objektes aus diesen Frameworks gestaltet sich bei allen den Frameworks sehr einfach. Alle Frameworks beherrschen sowohl Constructor als auch Setter Injection, wobei sich die Meinungen der Entwickler, welches besser sei, stark unterscheiden. Typische Fehler wie die Zyklische Abhängigkeiten werden von allen 3 Frameworks zur Laufzeit erkannt, jedoch nur Guice wirft nicht in jedem Fall eine Exception. PicoContainer und Spring bieten neben der Dependency Injection zusätzlich noch eine Lebenszyklusverwaltung an.

Einen zusammenfassenden Vergleich zeigt folgende Tabelle:

	PicoContainer	Guice	Spring
Lightweight Container	Ja	Ja	Nein
Lebenszyklus Verwaltung	Ja	Nein	Ja
Einfache Konfiguration	Ja	Ja	Nein
Bevorzugter Injection Type	Constructor	Keine Bevorzugung	Setter
Zyklische Abhängigkeiten	Fehlermeldung zur Laufzeit	Auflösung über Proxy Objekte, falls möglich, ansonsten Fehlermeldung zur Laufzeit	Fehlermeldung zur Laufzeit
Mehrdeutige Abhängigkeiten	Fehlermeldung zur Laufzeit	Fehlermeldung zur Laufzeit	Durch explizites Binden an konkrete Implementierungen nicht möglich

Tabelle 2.1: Vergleich der Frameworks

2.4 UML2 Diagramme

Um ein modellbasiertes Framework zu bauen muss sich für ein bestimmtes Modell und dessen Darstellung entschieden werden. Gute Modelle sind einfach lesbar und verständlich. Ein Modell sollte optimalerweise standardisiert sein, damit es eindeutig interpretiert werden kann. In dieser Arbeit wird der UML Standard verwendet, da dieser am meisten verbreitet ist.

UML (Unified Modeling Language) ist ein Standard der OMG (Object Managment Group)[1] und gilt als die dominierende Modellierungssprache. Es gibt 2 große Diagrammarten in der UML: Strukturdiagramme und Verhaltensdiagramme. Ein Modell der Systemkonfiguration stellt die Struktur eines Systemes dar. Für diese Arbeit sind somit die Strukturdiagramme relevant.

In der UML gibt es 6 verschiedene Strukturdiagramme, von denen sich für diese Arbeit besonders das Klassendiagramm und das Komponentendiagramm eignen. Klassendiagramme sind sehr detailliert und werden bei großen Systemen mit hunderten von Klassen unübersichtlich. Komponentendiagramme teilen das System in gröberere überschaubare Einheiten, die Komponenten, ein. Komponentendiagramme eignen sich wegen des höheren Abstraktionslevels besser für ein modellbasiertes Framework. Für Komponenten gibt es viele Definition, in dieser Arbeit wird die Definition der OMG verwendet:

> „A component represents a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment. A component defines its behavior in terms of provided and required interfaces. (Object Managment Group, 2007, Seite 146)

Komponenten haben eine niedrige Kopplung, weil sie ihr Innenleben nach außen kapseln und ihre Dienste nur über Schnittstellen anbieten. Das macht sie leicht austauschbar und wiederverwendbar. Eine Komponente kann sowohl aus nur einer Klasse als auch aus mehreren Subsystemen bestehen.

Die Abhängigkeiten einer Komponente zeigen sich im Komponentendiagramm über ihre anfordenden Schnittstellen. Der Service, der diese Abhängigkeit befriedigen kann, ist über einen Konnektor mit dieser Komponente verbunden. Konnektoren liegen immer zwischen einer angebotenen und einer anfordenden Schnittstelle.

Eine Komponente muss keinem realem Objekt der Anwendung entsprechen und kann nur eine Abstraktion für die Modellierung sein. Eine Komponente kann somit nicht einfach auf ein reales Objekt des Systems abgebildet werden.

[1] Konsortium von vielen Großen IT Unternehmen wie IBM, Apple oder Sun um gemeinsame Standards zu schaffen

2.5 XMI Grundlagen

UML Diagramme sind grafische Repräsentation eines Modells. Diagramme sind deshalb für Maschinen nur schlecht lesbar und sollten in eine maschinenlesbare Zwischenrepräsentation umgewandelt werden. Die OMG hat zu diesen Zwecken das XMI Format entwickelt.

Der XMI (XML Metadata Interchange) Standard wandelt Modelle der MOF(Meta Object Facility), zu denen auch UML Modelle gehören, in das XMI Format um. Ziel des XMI Standard ist es den Austausch von Modellen zwischen verschiedenen Modellierungswerkzeugen zu ermöglichen. XMI ist ein XML Dialekt und kann deshalb auch von XML Parsern geparst werden. Viele kommerzielle Modellierungswerkzeuge unterstützen den Ex- und Import von XMI Dateien. Der Modellierer muss deshalb keine XML Kenntnisse haben um XMI zu nutzen.

Im XMI 2.1 Standard werden keine grafischen Informationen, wie z.B. die Position eines Elements im Diagramm, gespeichert. Jedes Modellierungswerkzeug kann den XMI Standard aber erweitern, damit es diese grafischen Information mit in die XMI Datei schreiben kann. Diese Erweiterungen sind werkzeugspezifisch und können deshalb nur von dem erstellenden Werkzeug interpretiert werden. Beim Austausch eines Diagramms können somit die grafischen Informationen verloren gehen und die Elemente des Diagramms müssen per Hand neu ausgerichtet werden. Einige Werkzeuge nutzen die Möglichkeit der Extensions für mehr als nur grafische Informationen, so dass XMI Dateien auch nur von diesen Werkzeugen gelesen werden können. Meist kann der Anwender das Werkzeug aber so einstellen, dass in den Erweiterungen nur die grafischen Informationen stehen und die anderen Informationen im Standard XMI gehalten werden.

Das folgende Beispiel zeigt einen Ausschnitt aus der XMI Datei des Versandhauses;

```
1  <?xml version="1.0" encoding="windows-1252"?>
2  <xmi:XMI xmi:version="2.1" xmlns:uml="http://schema.omg.org/
       spec/UML/2.1" xmlns:xmi="http://schema.omg.org/spec/XMI
       /2.1">
3  ...
4
5      <!-- Der Standard XMI Teil -->
6      <uml:Model xmi:type="uml:Model" name="EA_Model"
           visibility="public">
7          <packagedElement xmi:type="uml:Package" xmi:id="
               EAPK_3F0457DA_8377_4090_854A_F03C3B2C1527" name="
               VersandhausKompo" visibility="public">
8              <packagedElement xmi:type="uml:Component" xmi:id
                   ="EAID_1652D46E_7B06_4325_814E_0C56533EA6E8"
                   name="Kundendienst" visibility="public">
9                  <required xmi:id="
                       EAID_A4DE172C_C482_42ce_83B7_0C38DE3F7ABD
                       " name="IKundenDB"/>
10                 ...
11             </packagedElement>
12             ...
13         </packagedElement>
14     </uml:Model>
15
16
17     <!-- Ab hier beginnt der Modellierungstool spezifische
           Abschnitt -->
18     <xmi:Extension extender="Enterprise Architect"
           extenderID="6.5">
19     ...
20     </xmi:Extension>
21 </xmi:XMI>
```

Listing 2.7: Ausschnitt aus der XMI Datei des Versandhaus Beispielsystemes

In den Zeilen 6-14 befindet sich der Standard XMI Teil. Die Elemente des Modelles werden hier als <packagedElement> Elemente abgespeichert. Der Typ des Elementes steht im xmi:type Attribut. Jedes Element hat weiterhin eine in dieser Datei eindeutige ID.
Ab Zeile 17 beginnen die Erweiterungen des Standards, in diesem Falle die Erweiterungen von Enterprise Architect. Im Erweiterungsteil stehen nur grafische Informationen, die für diese Arbeit nicht relevant sind.

2.6 XML Parser Schnittstellen

XMI Dateien sind ein XML Dialekt. XMI Dateien lassen sich somit auch problemlos mit XML Parsern lesen. Um XML Dokumente zu parsen gibt es verschiedene APIs. In diesem Kapitel werden die DOM, SAX und StAX API vorgestellt.

Ab Java Version 5 ist der Xerces Parser von Apache als Implementierung dieser APIs standardmäßig in der Java Distribution enthalten. Java abstrahiert von konkreten Parsern durch die „Java API for XML Parser (JAXP)“. Mit JAXP lassen sich alle 3 hier vorgestellten APIs nutzen.

DOM

Das DOM (Document Object Model) bietet eine durch das W3C standardisierte API für den Zugriff auf XML und HTML Dokumente an. DOM durchläuft ein Dokument vollständig und baut dabei im Speicher einen Baum mit der Hierarchie des geparsten Dokumentes auf. In einem HTML Dokument wäre beispielsweise das <html> Element die Wurzel des Baumes und seine Kinder wären das <head> und das <body> Element.
Da das Dokument nach dem Parsen vollständig als Baum vorliegt, ist dieses leicht manipulierbar. Die Elemente des Baumes lassen sich beispielsweise sortieren oder Elemente können hinzugefügt und entfernt werden. Der Aufbau eines solchen Baumes ist langsam, somit ist DOM langsamer als die anderen APIs. DOM ist sehr speicherintensiv, da XML Dokumente sehr viele Elemente enthalten können und der Baum somit entsprechend groß wird.

SAX

Die SAX (Simple API for XML Processing) API wurde im Jahre 2000 von David Megginson entwickelt. Sie ist im Gegensatz zur DOM API ereignisbasiert und baut keinen Baum auf. SAX liest eine XML Datei wie einen sequentiellen Datenstrom und löst beim Treffen auf bestimmte Elemente Ereignisse aus. Ereignisse können z.B. das Auffinden eines neuen XML Elements sein. Diese Ereignisses rufen definierte Callback Funktionen auf, die dann das gefundene Element verarbeiten. SAX holt sich selbstständig die nächsten Elemente und „drückt“ die Elemente in den Parser, deshalb gilt SAX auch als Push-API.

SAX baut keinen Baum im Speicher auf und ist dadurch schneller und weniger speicherintensiv als DOM. SAX merkt sich die bearbeiteten Elemente nicht, ein späterer Zugriff über SAX ist nicht mehr möglich.

StAX

Die StAX (Streaming API for XML) API wurde entwickelt um die Vorteile von baumbasierten APIs und ereignisbasierten APIs zu vereinen. StAX nutzt beim Parsen das Prinzip des Cursors. Der Cursor befindet sich an einer Stelle im XML Dokumente und der Parser holt sich selbstständig das nächste Element. Der Parser „zieht“ das nächste Element aus dem XML Dokument. StAX gilt deshalb als Pull-API.

StAX hat dieselben Vorteile wie SAX, bietet aber als Pull-API bessere Kontrolle über den Parsing Vorgang. StAX kann wie DOM XML Dokumente verändern und XML Dokumente erstellen.

3 Analyse

3.1 Bedingung an das UML Komponentendiagramm und die XMI Dateien

In dem Kapitel „UML2 Diagramme“(2.4) hat sich das Komponentendiagramm als geeignete Darstellung einer Systemkonfiguration herausgestellt. Bei der Erstellung eines Komponentendiagrammes bietet die UML dem Modellierer viele Freiheitsgrade. So kann zum Beispiel eine Komponente im Diagramm inklusive ihres Innenlebens abgebildet werden oder nur als Blackbox mit ihren Schnittstellen. Diese Freiheitsgrade erschweren das Erstellen eines modellbasierten Frameworks, da alle die Freiheitsgrade beachtet werden müssem. Die Abdeckung aller möglichen Freiheitsgrade ist oftmals nicht realisierbar, da zum Beispiel bei sehr abstrakten Modellen wichtige Informationen für das modellbasierte Framework fehlen können. Diese Freiheitsgrade müssen in diesem Fall eingeschränkt werden. Deshalb werden folgende Bedingungen an die Diagramme gestellt:

- **Jede angebotene Schnittstelle benötigt eine Implementierung im Diagramm**
  In der Blackbox Sicht auf eine Komponente werden nur ihre angebotenen und anfordernden Schnittstellen dargestellt. Die Abhängigkeiten eines Objektes können nur von konkreten Klassen befriedigt werden. Da aber eine Schnittstelle ihre implementierenden Klassen nicht kennt, können die konkreten Klassen aus der Blackbox Sicht nicht abgeleitet werden.Im Diagramm muss also für jede angebotene Schnittstelle eine implementierende Klasse gezeichnet werden. In der Whitebox Sicht können mehrere Klassen in einer Komponente dargestellt werden, damit aber die Abbildung Schnittstelle zu Klasse eindeutig ist, sollte zusätzlich die Schnittstelle mit ihrer Implementierung durch den Realisierungskonnektor verbunden werden.
- **Alle Objekte die vom Framework erzeugt werden sollen, müssen im Diagramm gezeichnet werden** Eine Komponente ist ein abstraktes Konstrukt und muss keinem realem Objekt der Anwendung entsprechen. Das Framework kann also über die Komponenten nicht herausfinden, welche Objekte in der Komponente enthalten sind. Das Framework kennt die enthaltenen Objekte nicht, wenn sie nicht explizit dargestellt werden.

- **Alle Objekte, die vom Framework erzeugt werden sollen, müssen im Diagramm den passenden Packages zugewiesen werden** Wie unter der vorigen Bedingung erwähnt, entspricht eine Komponente nicht immer einem realem Objekt. Eine Abbildung einer Komponente auf ein Package ist nicht gegeben. Das Framework muss für die Erzeugung eines Objektes auf die Klassen des Anwendungssystems Zugriff haben. Damit das Framework die Klassen finden kann, benötigt es den voll qualifizierenden Namen der Klassen, der auch den Packagenamen enthält.
- **Verbindungen zwischen zwei Komponenten erfolgen über den Abhängigkeitskonnektor** Besteht zwischen 2 Komponenten eine Abhängigkeitsbeziehung, so müssen diese über den Abhängigkeitskonnektor erfolgen. Der UML Abhängigkeitskonnektor wird ausgehend von der anfordenden Schnittstelle zur angebotenen Schnittstelle im Diagramm gezeichnet. Alternativ gibt es in der UML auch die Möglichkeit eine Abhängigkeit zwischen 2 Komponenten über den Kompositionskonnektor darzustellen. Dort sind die anfordende und angebotene Schnittstelle direkt miteinander verbunden. Der Kompositionskonnektor wird in dieser Arbeit nicht unterstützt, da er nur eine andere Darstellungsmöglichkeit einer Abhängigkeit ist. Eine Erweiterung des Frameworks zur Unterstützung des Kompositionskonnektors soll aber möglich sein.

Ein Diagramm, dass diese Bedingungen erfüllt ist in folgender Abbildung zu sehen:

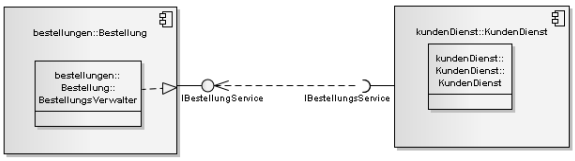

Abbildung 3.1: Ein geeignet Darstellung für das Framework

- **Diagramme sollten in XMI Version 2.1 und UML 2 exportiert werden**
  Zur Zeit dieser Arbeit ist die aktuelle XMI Version 2.1. XMI ist nicht abwärtskompatibel, da es große Unterschiede zwischen XMI in der ersten Version und der aktuellen Version gibt. Ob zukünftige XMI Versionen abwärtskompatibel zu XMI 2.1 sind, kann an dieser Stelle nicht gesagt werden. Es wird deshalb empfohlen die Diagramme im XMI 2.1 Format zu exportieren, damit sie vom Framework erkannt werden können.
  Auch die Darstellung von Komponenten hat sich von UML 1 zu UML 2 verändert. Diese Veränderung betrifft auch die aus den Diagramm erstellten XMI Dateien. Da UML 2 die aktuellste Version ist, akzeptiert das Framework auch nur in dieser Version erstellten Komponentendiagramme.
- **Diagramme müssen in Standard XMI exportiert werden**
  Wie im Kapitel XMI(2.5) erläutert kann jedes UML Modellierungswerkzeug das XMI

Format eigenständig erweitern. Durch diese Erweiterungen verlieren die XMI Dateien ihre Unabhängigkeit vom erstellenden Werkzeug. Damit das modellbasierte Framework möglichst viele Modellierungswerkzeuge unterstützt, sollte es nicht für die spezifische Erweiterungen eines Werkzeugs ausgelegt sein. Das Framework wird so konzipiert, dass es vorerst nur im werkzeugunabhängigen Standard XMI [1] exportierte Diagramme akzeptiert um eine hohe Flexibilität zu erreichen.

3.2 Anforderungen an das Framework

Aus dem Vergleich der bekannten Frameworks und eigener Gedanken werden nun einige Anforderungen aufgestellt, die für das modellbasierte Framework gelten sollten. Dabei werden, ähnlich wie im Lastenheft Anforderungen (A), Prämissen (P) und Einschränkungen (E) formuliert. Die Anforderungen werden, falls notwendig,in diesem Kapitel zur Nachvollziehbarkeit zusätzlich begründet.

- **A01** Das Framework akzeptiert als Konfiguration alle XMI Dateien, die den Bedingungen aus dem vorigen Kapitel entsprechen.
- **A02** Eine Erweiterung des Frameworks, so daß außer XMI 2.1 weitere Formate als Konfiguration akzeptieren werden, soll möglich sein. Eine feste Bindung an ein Format macht das Framework inflexibel. Ohne die Möglichkeit dieser Erweiterung kann das Framework zum Beispiel nicht angepasst werden, falls es in ein paar Jahren XMI Version 3 geben sollte.
- **A03** Eine Erweiterung des Frameworks, um weitere Diagrammarten als Basis für die Konfiguration zu akzeptieren, soll möglich sein. Wie im Kapitel Diagramme(2.4) erläutert, eignen sich zum Beispiel auch Klassendiagramme als Darstellung einer Konfiguration. Zukünftige Versionen könnten auch Klassendiagramme als Basis der Konfiguration akzeptieren.
- **A04** Das Framework unterstützt die Konfiguration von Anwendungen in der Sprache Java. Um aus einem Diagramm Objekte zu erstellen, benötigt das Framework Wissen über die Interna der Objekte, wie zum Beispiel das Wissen welche Parameter der Konstruktor des Objektes verlangt. Dieser Zugriff auf die Interna kann in Java über die Reflection API erfolgen. Es wurde sich für Java entschieden, da Java immer noch eine der beliebtesten und meist genutzten Programmiersprachen[2] ist.

[1] Spezifikation unter: Object Managment Group (2007)
[2] Siehe Quelle: Dedasys LLC (2010)

- **A05** Das Framework erstellt bei erfolgreicher Verarbeitung ein Container in der Sprache Java. Unter der Anforderung A04 wurde sich dafür entschieden, dass nur Java Anwendungen unterstützt werden. Damit die Java Anwendung den Container nutzen kann, sollte es auch in Java geschrieben werden.
- **P01** Der Quellcode der Anwendung, die konfiguriert werden soll, wird nicht vom Framework erstellt. Das Erstellen des Quellcodes ist keine Aufgabe eines Dependency Injection Frameworks. Das Framework erzeugt nur den Container.
- **P02** Die ".class“ Dateien der Anwendung, die konfiguriert werden sollen, stehen dem Framework zur Verfügung. Wie unter A04 erläutert benötigt das Framework Zugriff auf die Interna der Objekte einer Anwendung um damit arbeiten zu können. Die benötigten Informationen sind in den .class Dateien enthalten.
- **A06** Das Framework ist unabhängig von einer bestimmten Entwicklungsumgebung. Das Framework soll kein Plugin für Entwicklungsumgebungen wie Eclipse oder Netbeans sein.
- **A07** Das Framework unterstützt Constructor Injection. Die Entscheidung für Constructor Injection fiel aus folgenden Gründen: Nach Aufruf eines Konstruktors ist jede Abhängigkeit gesetzt und das Objekt vollständig konfiguriert. Konstruktoren haben den eindeutigen Namen des Objektes, während Setter Methoden verschiedene Namen haben können. Wird Setter Injection genutzt, so müssen auch alle Setter Methoden im Komponentendiagramm erscheinen, was zusätzlichen Aufwand bedeutet.
- **A08** Eine Erweiterung, so dass weitere Injection Arten unterstützt werden, soll möglich sein.
- **A09** Der Anwender des Frameworks wird beim Erstellen des Containers durch eine grafische Oberfläche unterstützt. Die Unterstützung mit einer grafischen Oberfläche ist sinnvoll, da sich einige Aspekte einer Konfiguration, wie die Wahl eines bestimmten Konstruktors, nur sehr schlecht in einem Komponentendiagramm darstellen lassen. Der Anwender hat auf der grafischen Oberfläche mindestens folgende Optionen:
  1. Setzen der primitiven Parameter und der String Parameter des Konstruktors
  2. Wahl des Konstruktors, so dass beim Vorhandensein mehrerer Konstruktoren der passende gewählt werden kann.

3. Für jedes Objekt in der Konfiguration, die Wahl, ob das Objekt „unique“ sein soll oder nicht. Ein Objekt ist unique, wenn der Container nur eine Referenz auf das Objekt zurückgibt. Das Objekt ist einmalig in der Anwendung. Wenn ein Objekt nicht unique ist, so erzeugt der Container bei jedem Aufruf eine neue Instanz des Types. Der Name „unique“ wurde gewählt, damit es keine Verwechslung mit dem Singleton Pattern gibt. Unique Objekte haben beispielsweise im Gegensatz zu Singletons keine globale Sichtbarkeit.

- **A11** Für primitive Parameter und Strings setzt das Framework Default Werte. Die Defaultwerte für primitive Parameter entsprechen den Defaultwerten von Java, wie in Sun Oracle (2009) erläutert. Ausnahme hiervon sind Strings die vom Framework den leeren String anstatt null als Defaultwert gesetzt bekommen. Durch Defaultwerte können Objekte initialisiert werden, auch wenn der Anwender das explizite Setzen dieser Parameter vergisst oder nicht setzen möchte.
- **A12** Das Framework erkennt zyklische Abhängigkeiten.
- **P03** Die Objekte in der Konfiguration haben einen öffentlichen Konstruktor.
- **E01** Eine Lebenszyklusverwaltung der Objekte in der Konfiguration wird vom Framework nicht vorgenommen. Lebenszyklusverwaltung gehört nicht zu den Kernaufgaben eines Dependency Injection Frameworks und liegen deshalb nicht im Fokus dieser Arbeit.
- **E02** Ports zur Sammlung von Schnittstellen einer Komponente werden nicht unterstützt. Eine Erweiterung des Frameworks, so dass diese erkannt werden soll möglich sein.

3.3 Beispielszenario: Konfigurieren des Versandhaussystemes

Dieser Abschnitt zeigt das gewünschte Vorgehen bei Nutzung des Frameworks anhand des Versandhaus Systems aus der theoretischen Einführung (2.2). Dabei soll die Release Konfiguration des Versandhauses, wie in Abbildung 2.3 gezeichnet, als Konfigurationsdatei dienen.

1. Der Anwender erstellt die gewünschte Konfiguration in dem Modellierungswerkzeug seiner Wahl, welches einen Export in Standard XMI 2.1 erlaubt. Dabei entspricht das erstellte Modell den Bedingungen des Kapitels 3.1
2. Der Anwender exportiert das so erstellte Diagramm im XMI Format. Dabei muss der Anwender darauf achten, dass der Export im Standard XMI erfolgt.

3. Der Anwender startet das modellbasierte Framework.
4. Bevor der Anwender die XMI Datei importieren kann, stellt der Anwender dem Framework über dessen grafische Oberfläche den Pfad zu den .class Dateien zur Verfügung. Dabei reicht der Top Level Ordner.
5. Der Anwender importiert die XMI Datei über die grafische Oberfläche in den Parser.
6. Bei erfolgreichem Import zeigt das System den Anwender in der grafischen Oberfläche die gefundenen Objekte mit ihren Konstruktoren an und belegt die primitiven Parameter und Strings mit Default Werten.
7. Der Anwender setzt die primitiven Parameter nach Wunsch neu und wählt die Konstruktoren für jedes Objekt.
8. Der Anwender wählt über die grafische Oberfläche das Verzeichnis in welches die Konfigurator Datei geschrieben werden soll.
9. Die so exportierte Datei kann der Anwender in seine Anwendung integrieren, indem er die Konfigurator Datei in sein Projekt importiert.

4 Design

4.1 Grobes Design

Die Hauptaufgabe des modellbasierten Frameworks ist es XMI Dateien einzulesen, zu verarbeiten und einen Container in Java zu erstellen. Das Framework liest also eine Datei in einem Quellformat ein, verarbeitet dessen Informationen und erstellt daraus eine neue Datei in einem bestimmten Zielformat. Diese Aufgabe ist vergleichbar mit der eines Compilers, wo zum Beispiel menschlesbarer Quellcode in Maschinencode umgewandelt wird. Die Architektur eines Compilers kann somit als Referenz für das Framework verwendet werden.

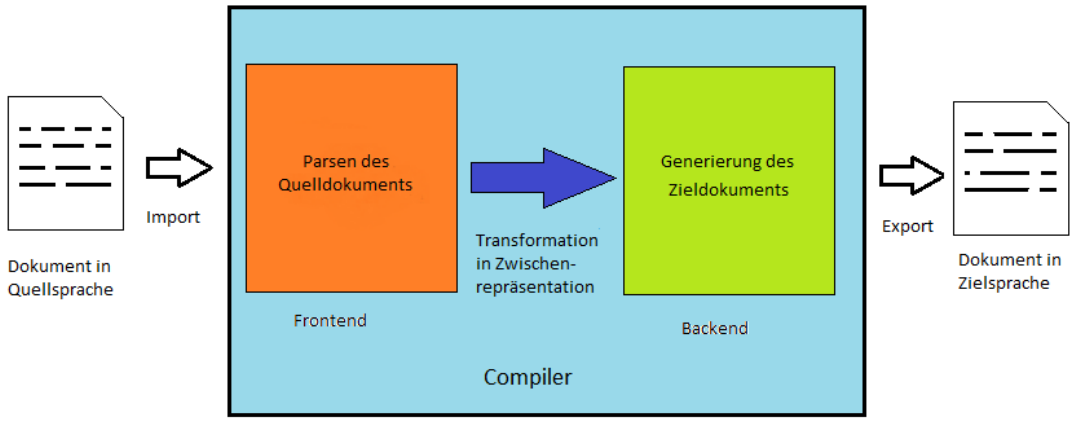

Abbildung 4.1: Architektur eines Compilers

Das Frontend des Frameworks parst die XMI Dateien und filtert dabei die für das Framework relevanten Informationen aus der Datei. Relevante Informationen sind zum Beispiel welche Komponenten es überhaupt gibt und in welcher Beziehung die Komponenten zueinander stehen. Eine zusätzlicher Validator im Frontend übernimmt eine einfache semantische Analyse, wie zum Beispiel die in Anforderung A12 geforderte Überprüfung nach zyklischen Abhängigkeiten. Der XMI Parser und der Validator bilden zusammen die Parserkomponente, die dem Frontend entsprechen.

Nachdem die Parserkomponente die XMI Datei eingelesen hat, beginnt die eigentliche Konfiguration. Hierbei wird der Anwender, wie unter Anforderung A09 beschrieben, durch eine grafische Oberfläche unterstützt. Der Anwender wählt den gewünschten Konstruktor eines Objekts, setzt seine Parameter und wählt aus ob ein Objekt unique sein soll. Mit dieser grafischen Oberfläche beeinflusst der Anwender die Übersetzung der XMI Datei in den Container. Dadurch entscheidet sich das Framework von der klassichen Compiler Architektur. Nachdem der Anwender die Systemkonfiguration nach seinen Wünschen angepasst hat, werden die Objekte zur Generierung des Containers vorbereitet. Die Anpassung der Konfiguration und das Vorbereiten der Objekte für die Generierung sind die Hauptaufgabe der Konfiguratorkomponente.

Sobald die Konfiguration abgeschlossen ist, muss nur noch der Container generiert werden. In der Generierung werden die in der Konfigurationskomponente vorbereiteten Objekte in eine Schablone für den Container übertragen. Die Generierung des Containers entspricht dem Backend eines Compilers und bildet die Generatorkomponente.

Das Framework besteht also im groben aus den 3 Komponenten: Parser, Konfigurator und Generierung. Diese Komponenten bilden die Architektur des Systems:

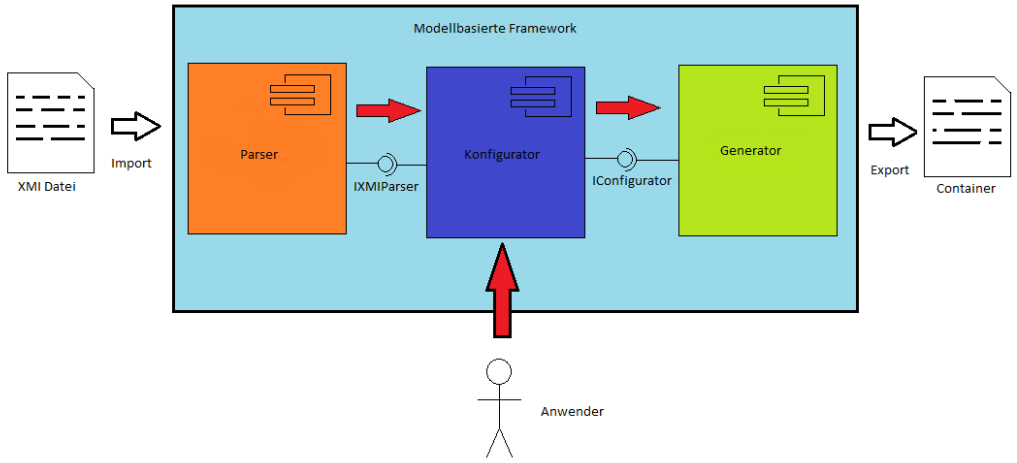

Abbildung 4.2: Architektur des modellbasierten Frameworks

Das Framework wird in einer speziellen Starterklasse konfiguriert. Der Anwender entscheidet sich innerhalb dieser Klasse beispielsweise für den Parser der verwendet werden soll. Nachdem der Anwender die Konfiguration des Frameworks festgelegt hat, setzt die Starterklasse die Komponenten zusammen und startet das Framework. Die Starterklasse liegt außerhalb der Fachlichkeit des Frameworks und gehört somit zu keiner Komponente des Frameworks.

Das Dokument muss innerhalb des Frameworks in einer festgelegten Reihenfolge verarbeitet werden. Die Parserkomponente muss das Dokument eingelesen haben, bevor der Anwender die Objekte in der Konfiguratorkomponente anpassen kann. Die Erstellung des Containers kann erst erfolgen, nachdem die Konfiguratorobjekte die Konfiguration festgehalten hat. Der korrekte Ablauf wird vom Anwender über die grafische Oberfläche gesteuert. Durch einen Knopfdruck lädt er die XMI Datei und startet damit das Parsen. Erst nach dem Laden kann der Anwender die Objekte in dieser Oberfläche anpassen. Die Generierung des Containers wird ebenfalls vom Anwender durch einen Knopfdruck gestartet.

4.2 Verfeinerung des Designs

Das vorherige Kapitel hat die Architektur des modellbasierten Dependency Injection Frameworks vorgestellt. Das Framework besteht aus den 3 Komponenten: Parser, Konfiguration und Generator. Das Design dieser drei Komponenten soll nun verfeinert werden.

4.2.1 Parser

XMI ist ein spezieller XML Dialekt, somit können XMI Dateien auch von XML Parsern verarbeitet werden. Das Framework sollte keine Abhängigkeiten zu Third Party Bibliotheken haben, die nicht standardmäßig in der Java Distribution enthalten sind. Java unterstützt standardmäßig das Parsen von XML Dateien mittels der unter 2.6 vorgestellten APIs: DOM, SAX und StAX API. Diese Technologien können für das Framework verwendet werden.

Die XMI Datei soll nur einmal geparst werden um die relevanten Informationen auszulesen. Der Parser muss keinen Baum der XMI Datei im Speicher aufbauen, da die Hierarchien innerhalb der XMI Elemente für das Framework nicht relevant sind. Es sollen auch keine neuen Elemente hinzugefügt oder entfernt werden. Der Baumbasierte Ansatz, wie in der DOM API, ist für das Framework somit nicht geeignet.

Durch die StAX API hat das Framework mehr Kontrolle über den Parsing Vorgang als bei der Verwendung der SAX API. Der Parser kann durch die StAX API die XMI Elemente einzeln lesen und bearbeiten. StAX Code gilt aufgrund der besseren Kontrollierbarkeit des Parsing Vorgang lesbarer als SAX Code. Aus diesen Gründen nutzt das Framework die StAX API zum Parsen der XMI Dateien.

Die Parser Komponente soll nicht nur die Informationen aus den XMI Dateien lesen sondern auch eine semantische Validierung durchführen. Die Validierung umfasst folgende Punkte:

- **Ob es für jede angebotene Schnittstelle im Diagramm eine Implementierung gibt, die über den Realisierungskonnektor mit der Schnittstelle verbunden ist**
  Ohne eine konkrete Implementierung der angebotenen Schnittstelle kann das Framework die Abhängigkeiten zu dieser Schnittstelle nicht befriedigen. Da es mehrere Klassen im Diagramm geben kann, die dieselbe Schnittstelle implementieren, muss jede Schnittstelle über den Realisierungskonnektor mit ihrer gewünschten Implementierung verbunden sein. Die Abbildung Schnittstelle zur Implementierung muss eindeutig sein. Wird bei der Validierung eine angebotene Schnittstelle ohne Implementierung gefunden, so wird eine Fehlermeldung generiert, die den Anwender warnt. Das Lesen der Konfiguration wird jedoch nicht abgebrochen, da eine Schnittstelle nicht zwangsweise genutzt werden muss. Wenn innerhalb der Anwendung nicht auf diese Schnittstelle zugegriffen wird, ist eine Implementierung nicht notwendig.

- **Ob für jede Klasse und jede angebotene Schnittstelle im Diagramm eine .class Datei in der Anwendung vorhanden ist.**
  Die Hauptaufgabe eines Dependency Injection Frameworks ist das Erzeugen von Objekten auf Anfrage. Das Framework benötigt für die Erstellung von Objekten Zugriff auf die Konstruktoren, um das Objekt mit den richtigen Parametern zu versorgen.
  Mit der Reflection API von Java kann das Framework auf die Interna von Klassen zugreifen, also auch auf die Konstruktoren der Klassen. Dazu benötigt das Framework jedoch Zugriff auf die .class Dateien.
  Damit der Anwender nicht für jedes Objekt den Pfad zur .class Datei angeben muss, kann der Anwender dem Framework ein Verzeichnis angeben, von dem aus es nach den .class Dateien suchen soll. Das Framework durchsucht dabei nicht nur das angegebene Verzeichnis sondern auch alle Unterverzeichnisse.
  Das Framework bricht das Parsen ab, wenn es eine .class Datei nicht finden kann. Ohne den Zugriff auf die Konstruktoren über die .class Datei kann das Framework die Objekte nicht erzeugen.

- **Ob es zyklische Abhängigkeiten gibt.**
  Ein Komponentendiagramm kann auch als gerichteter Graph betrachtet werden. In diesem Graphen sind die Komponenten die Knoten und die Kanten entsprechen der Abhängigkeitsbeziehung zwischen 2 Komponenten. Die Richtung der Kanten geht dabei von der angebotenen Schnittstelle zu den nutzenden anfordenden Schnittstellen. Führt nun ein beliebiger Weg durch diesen Graphen von einem Startknoten wieder zum Startknoten zurück, so besteht ein Zyklus in dem Graphen. Mindestens 2 Komponenten eine zyklische Abhängigkeit.

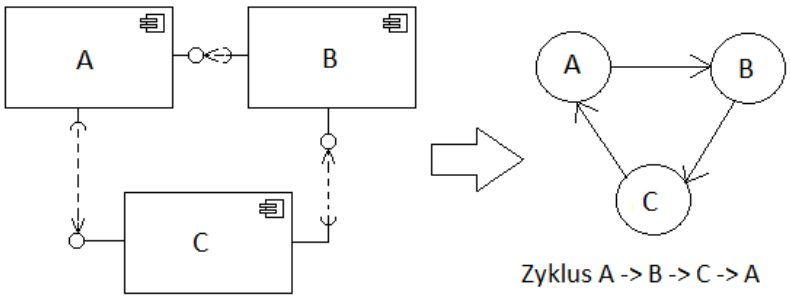

Abbildung 4.3: Zyklische Abhängigkeit im Komponentendiagramm

Zyklische Abhängigkeiten bedeuten für einen Dependency Injection Framework ein Henne-Ei Problem. Benötigt zum Beispiel Komponente A eine Instanz von B als Parameter, während Komponente B eine Instanz von Komponente A erwartet, so kann das Framework nicht entscheiden, welche Komponente zuerst erzeugt werden muss.
Der Validator durchläuft den ganzen Abhängigkeitsgraphen des Komponentendiagramms und sucht dabei auch nach zyklischen Abhängigkeiten zwischen mehreren Komponenten. Da eine zyklische Abhängigkeit das Framework vor eine unentscheidbare Aufgabe stellt, wird das Parsen sobald eine zyklische Abhängigkeit gefunden wurde, beendet und der Anwender wird auf diesen Fehler hingewiesen.

Der Validator kann in späteren Versionen um weitere Punkte erweitert werden. Es wurde sich in dieser Arbeit auf diese drei Punkte beschränkt, weil diese auf jeden Fall erfüllt sein müssen.

Die Validierung kann bei größeren System sehr aufwändig sein. Beispielsweise kann die Suche nach zyklischen Abhängigkeiten in einem System mit hunderten von Komponenten sehr lange dauern. Die Validierung des Komponentendiagramms ist deshalb optional und kann bei Bedarf an- bzw. abgeschaltet werden.

Die andere Hauptaufgabe des Parsers ist es die für die Dependency Injection relevanten Informationen aus der Konfigurationsdatei zu lesen. Für die Umsetzung des Dependency Injection Musters muss das Framework wissen welche Objekte es gibt und welche Klassen die angebotenen Schnittstellen implementieren. Die Information über die Abhängigkeiten stehen implizit in den Konstruktoren der Objekte und werden daher nicht aus der Konfiguration gelesen.

Da der Kontrollfluss des Frameworks über die Konfiguratorkomponente gesteuert wird, lässt sich über die Konfigurator Schnittstelle der Parsing Vorgang starten. Über diese Schnittstelle kann außerdem die Validierung an- bzw. ausgeschaltet werden.

Die Semantische Validierung benötigt andere Informationen als die Konfiguratorkomponente. Zum Beispiel sind Komponenten oft nur Abstraktionen, die keinen realen Objekten entsprechen. Die Komponenten sind für die Konfiguration irrelevant. Jedoch können die Komponenten in der Validierung zum Suchen nach zyklischen Abhängigkeiten genutzt werden. Damit der Validator und die Konfiguration nur Zugriff auf die für sie relevanten Informationen bekommen, spielt der XMI Parser zwei Rollen, die über verschiedene Schnittstellen definiert sind.

Die folgende Abbildung zeigt die Architektur der Parserkomponente

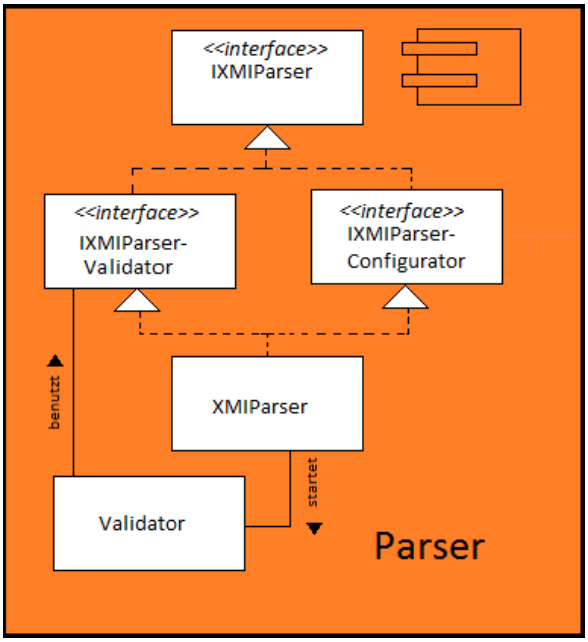

Abbildung 4.4: Architektur der Parserkomponente

4.2.2 Konfigurator

Im Komponentendiagramm lassen sich nicht alle für ein Dependency Injection Framework relevanten Informationen darstellen. Zum Beispiel benötigt der Container für jeden Parameter des Konstruktors einen gesetzten Wert um ein Objekt zu erzeugen. Die Zuweisung von Werten an bestimmte Parameter lässt sich zwar im Komponentendiagramm darstellen, bedeutet aber deutlich höheren Modellierungsaufwand. Es müssen die Konstruktoren mit ihren Parametern und den Werten der Parameter in das Diagramm gezeichnet werden.

Der Vorteil das ein Modell einfacher zu erstellen und zu lesen ist, als bei einer textuellen Darstellung, verringert sich stark, wenn das Modell mit Informationen überladen wird. Um diesen Vorteil zu behalten ist das Framework so konzipiert, dass im Modell nur die wenige Informationen dargestellt werden müssen. Die Konfiguratorkomponente hilft dabei das Modell einfach zu halten, aber trotzdem alle benötigten Informationen zu erhalten.

Die fehlenden Daten des Modells können über eine grafische Oberfläche hinzugefügt und verändert werden. Die grafische Oberfläche erlaubt eine Feinkonfiguration. Der Anwender kann hier die Parameter eines Konstruktors mit Werten belegen, bei einer Auswahl von mehreren öffentlichen Konstruktoren den gewünschten heraussuchen und wählen ob ein Objekt unique sein soll oder nicht. Im Modell müssen nur die Informationen über die vorhandenen Objekte und die Informationen über die Realisierungsbeziehungen zwischen Schnittstelle und Implementierung dargestellt werden.

Die grafische Oberfläche ist in folgender Abbildung zu sehen:

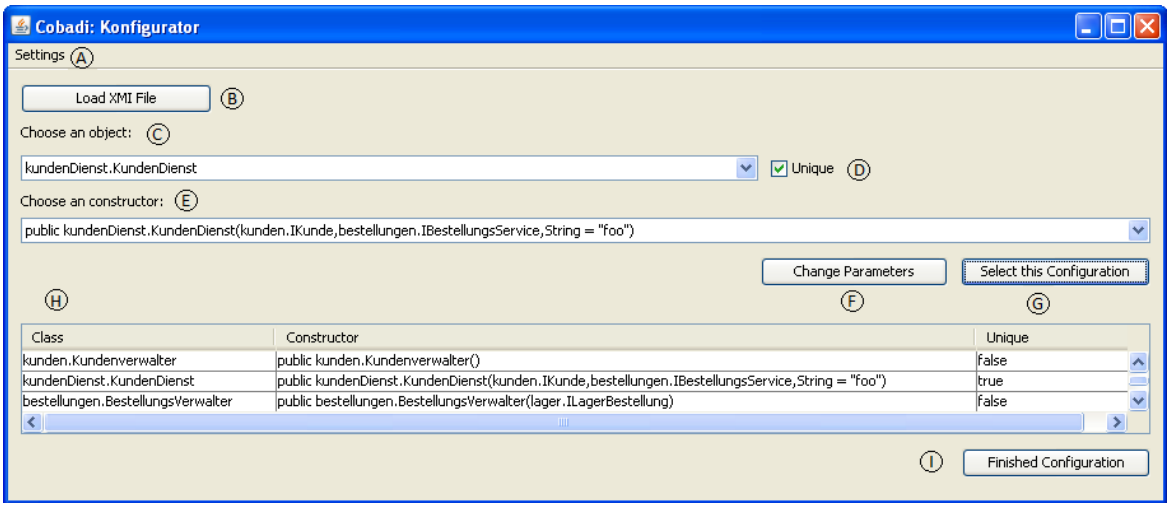

Abbildung 4.5: Das Hauptfenster des Konfigurators

Markierung in der Abbildung 4.5	Funktion
A	Das Settingsmenü erlaubt die Wahl folgender 4 Punkte: 1. Die Wahl des Verzeichnisses indem die Suche nach den .class Dateien beginnt 2. Die Wahl des Verzeichnisses in welches der Container erstellt wird 3. Die Wahl eines Namens für den Container 4. Ob beim Parsen der Konfiguration eine Validierung durchgeführt werden soll
B	Der Anwender lädt mit diesem Button die Konfigurationsdatei und startet dadurch das Parsen
C	Hiermit kann der Anwender wählen, welches Objekt er konfigurieren möchte
D	Das unter C gewählte Objekt kann hiermit als unique gekennzeichnet werden
E	Der Anwender kann hier einen Konstruktor des unter C gewählten Objektes aussuchen
F	Dieser Button öffnet ein Fenster, welches die Manipulation der primitiven Parameter des unter E gewählten Konstruktors ermöglicht
G	Dieser Button setzt die geänderte Konfiguration als gewünschte Konfiguration
H	Diese Vorschau zeigt die zur Zeit genutzte Konfiguration an. Dieses ist die Konfiguration die in den Container übertragen wird
I	Mit diesem Button bestätigt der Anwender seine Wahl der Konfiguration und startet die Generierung des Containers

Tabelle 4.1: Optionen im Hauptfenster

Die Konfiguratorkomponente besorgt sich selbstständig alle Konstruktoren eines Objektes über deren .class Dateien. Es müssen somit im Diagramm nicht alle Konstruktoren eines Objektes angegeben werden. Das heißt jedoch das das Framework den Zugriff auf diese .class Dateien benötigt. Damit der Anwender nicht für jedes Objekt den Pfad zu der Klassendatei angeben muss, kann der Anwender dem Framework, wie in der Validierung nur das Oberverzeichnis angeben, indem alle Dateien liegen. Das Framework sucht automatisch in diesem Verzeichnis und allen Unterverzeichnissen nach den .class Dateien. Der Anwender kann sogar die Wurzel des gesamten Verzeichnisbaumes wählen, was jedoch die Suchzeit nach den .class Dateien deutlich erhöht.

Der Konfigurator setzt für alle primitiven Parameter, sowie für Strings, Default Werte. Dabei setzt der Konfigurator die in der Java Spezifikation vorgesehenen Werte (siehe Sun Oracle (2009)) als Default. Für Strings setzt der Konfigurator als Default Wert den leeren String. Das Setzen der Defaultwerte bedeutet aber auch, dass im Diagramm gesetzte Werte vom Framework ignoriert werden. Die Default Werte können über die grafische Oberfläche verändert werden.

Außer der Feinkonfiguration hat die Konfiguratorkomponente die Aufgabe die Objekte für die Generierung vorzubereiten. Dafür speichert der Konfigurator nach Beenden der Feinkonfiguration die Konfiguration jedes Objektes in ConfigurationObjects. Ein ConfigurationObject ist ein spezieller Datentyp des Frameworks. Ein ConfigurationObject merkt sich den gewählten Konstruktor, die gewünschten Werte für die Parameter und die Information ob ein Objekt unique sein soll.

Damit das Framework die Objekte korrekt initialisieren kann, ist die Reihenfolge der Initialisierung relevant. Bevor ein Objekt erzeugt werden kann, müssen bereits alle seine Abhängigkeiten initialisiert sein. Die Objekte die maximal einen Service anbieten müssen zuerst initialisiert werden. Als nächstes sollten die Objekte initialisiert werden, die nur die bereits erzeugten Services nutzen. Die Konfiguratorkomponente sortiert die Objekte nach ihrer Initialisierungsreihenfolge, so dass der Generator diese auch in einer dieser Reihenfolge in den Container schreibt.

Die Konfiguratorkomponente übernimmt außerdem die Aufgabe den Kontrollfluss der Anwendung zu steuern. Diese Aufgabe wird über die grafische Oberfläche realisiert. Das Parsen der Konfiguration wird über das Laden der Konfigurationsdatei in der grafischen Oberfläche gestartet. Im nächsten Schritt findet die Feinkonfiguration in dieser Oberfläche statt. Zu letzt aktiviert der Anwender die Generierung des Containers über einen Knopfdruck in der grafischen Oberfläche.

Die folgende Abbildung zeigt die Architektur der Konfiguratorkomponente:

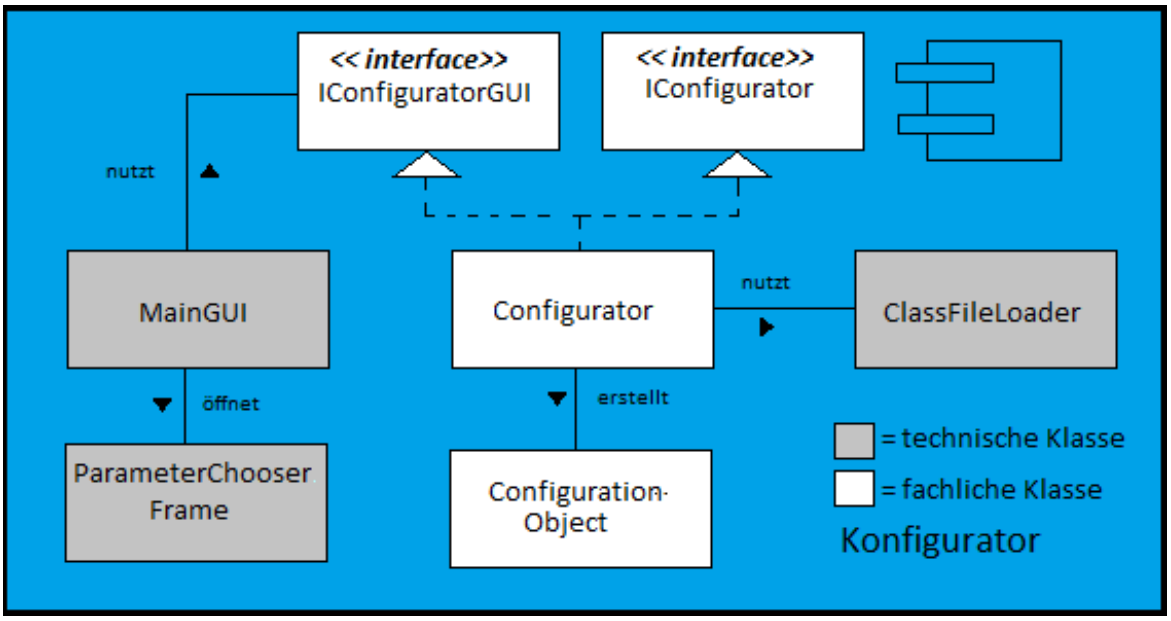

Abbildung 4.6: Architektur der Konfiguratorkomponente

4.2.3 Generator

Die Generatorkomponente erstellt den Container. Dafür muss sie die in der Konfiguratorkomponente erstellten ConfigurationObjects lesen und in eine Zieldatei schreiben. Die Zieldatei soll ein für eine Java Anwendung lesbares Format haben, da das Framework für Java Anwendungen konzipiert wurde. Diese Datei kann zum Beispiel eine Java Source Datei sein oder eine XML Konfigurationsdatei für das Spring Framework sein. In dieser Arbeit werden zunächst nur Java Source Dateien als Container erstellt.

Anwendungen die Dateien erstellen sind oft aufwändig. Für jede Zeile in der Zieldatei gibt es eine entsprechende Zeile im Code der Anwendung. Diese Codezeilen schreiben dabei nur eine Zeile in die Zieldatei. Der Code der Anwendung wird dadurch stark aufgebläht.

Die Zieldatei besteht aus einem statischen Teil, der bei jeder Generierung der Zieldatei gleich bleibt, und einem dynamischen Teil, der durch die Logik der schreibenden Anwendung generiert wird. Durch den dynamischen Teil unterscheiden sich die erzeugten Dateien.

Die Generatorkomponente nutzt diese Aufteilung indem sie zur Erstellung des Containers Templates benutzt. Diese Templates enthalten bereits den statischen Teil des Containers. Zum Beispiel sieht die öffentliche Methode zur Anfrage nach Objekten im Container immer gleich aus. Diese Methode steht in der Template Datei und wird vom Generator 1:1 in den Container übertragen. In diesen Template Dateien stehen außerdem spezielle Schlüsselwörter. Diese Schlüsselwörter markieren den Teil an denen der Generator die Informationen aus den ConfigurationObjects in den Container überträgt.

Trifft die Generatorkomponente beim Durchlaufen der Template Datei auf ein Schlüsselwort, so wird eine Callbackfunktion ausgelöst. Diese Callbackfunktion ersetzt das Schlüsselwort durch die Informationen aus den ConfigurationObjects.

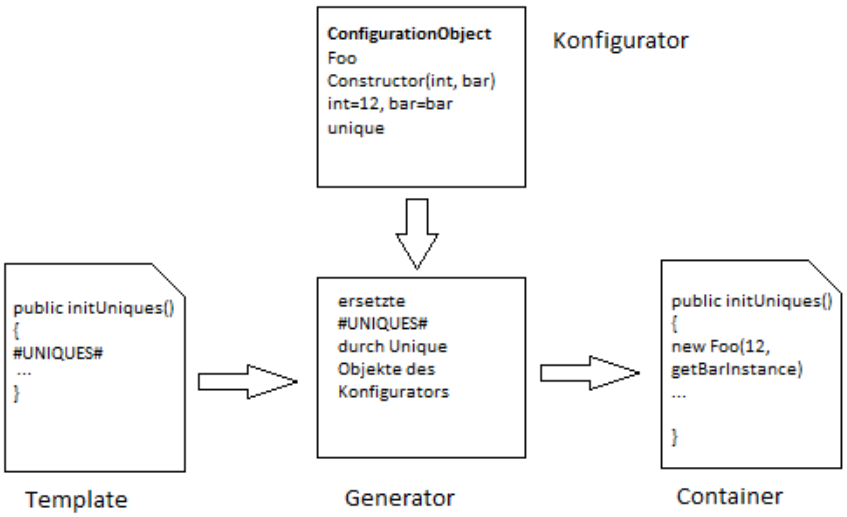

Abbildung 4.7: Funktionsweise der Generatorkomponente

Die Generatorkomponente besitzt eine abstrakte Generator Klasse, so dass nicht nur Java Source Dateien erstellen werden können. Diese abstrakte Klasse ist verantwortlich für das Lesen des Templates und den Aufruf der Callbackfunktionen. Die Callbackfunktionen werden von Unterklassen des Generators implementiert. Durch die Delegation des Aufrufes einer Callbackfunktion an die Unterklassen kann das Framework Dateien in verschiedenen Formaten erzeugen.

Die Architektur des Generators ist in folgender Abbildung zu sehen:

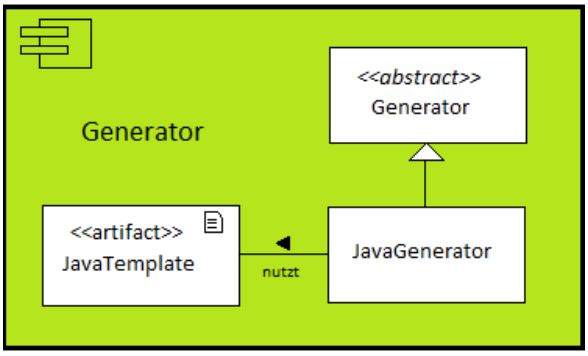

Abbildung 4.8: Architektur der Generatorkomponente

4.3 Design des Containers

Die Hauptaufgabe eines Containers ist das Bereitstellen von Objekten auf Anfrage. Der Container kann bei der Anfrage entweder eine Referenz auf das Objekt zurückliefern oder bei jeder Anfrage eine neue Kopie des Objektes erstellen und zurückliefern.

Soll der Container bei einer Anfrage die Referenz auf das Objekt zurückliefern, so muss sich der Container dieses Objekt nach der erstmaligen Erzeugung merken. Diese Objekte entsprechen den unique Objekten des Frameworks. Der in dieser Arbeit erstellte Container instanziiert diese Objekte direkt beim Erzeugen des Containers. Der Container lädt diese Objekte „eager“, so dass auf diese Objekte direkt nach der Erzeugung zugegriffen werden kann. Der Container hält die so erzeugten Objekte in einer Abbildung von den Namen des Objektes auf die Referenz des Objektes. Somit kann eine Anfrage einfach mit einem Zugriff auf diese Abbildung erfüllt werden.

Soll der Container bei jeder Anfrage ein neues Objekt des gewünschten Types erzeugen, so sollte sich der Container die erzeugten Objekte nicht merken. Der Container erstellt die Objekte erst zur Anfragezeit und vergisst das erstellte Objekt wieder. Im Container wird für jedes dieser Objekte der Konstruktor des Objektes in einer Abbildung abgelegt, so dass bei jeder Anfrage dieser Konstruktor aufgerufen werden kann. Zusätzlich muss sich der Container für jedes dieser Objekte auch die Parameter des Konstruktors merken. Bei einer Anfrage nach diesem Objekt lädt der Container zunächst über die Abbildung den Konstruktor und dessen Parameter. Nach dem Laden ruft der Container den Konstruktor auf und gibt das so erzeugte Objekt direkt an den Anfrager zurück.

Der Container muss vor dem Erstellen eines Objektes alle Abhängigkeiten des Objekts bereits erstellt haben. Die Erstellung der Objekte muss in einer bestimmten Reihenfolge erfolgen. Zuerst sollten die Objekte erstellt werden die selber keine Abhängigkeiten besitzten. Diese Objekt sind typischerweise Serviceanbieter. Erst nachdem die Serviceanbieter erstellt wurden, können die Objekte erstellt werden, die diese Services in Anspruch nehmen. Eine Analogie ist der Bau eines Hauses: Bevor das Dach gebaut werden kann, müssen zuerst die Wände des Hauses gemauert werden. Bevor die Wände gemauert werden können, muss zunächst das Fundament gegossen werden. Als erstes muss also das Fundament gegossen werden, welches den „Service“ stabiler Untergrund anbietet.

In einer gut entworfenen Anwendung werden die Services eines Objektes über Schnittstellen gekapselt. Ein Objekt hat damit keine Abhängigkeit zu konkreten Objekten sondern zu einer Schnittstelle. Zur Erstellung eines Objektes muss diese Abhängigkeit jedoch durch ein konkretes Objekt befriedigt werden. Der Container muss wissen wie er die Abhängigkeit zu einer Schnittstelle befriedigen kann. Dafür merkt sich der Container zu jeder Schnittstelle eine konkrete Implementierung. Der Container wandelt die Anfrage nach einer Schnittstelle intern in eine Anfrage nach der konkreten Implementierung um und liefert diese zurück.

5 Realisierung

Das Kapitel Design stellt die Komponenten und ihre Funktionsweisen nur prinzipiell dar. Dieses Kapitel beschäftigt sich mit der Umsetzung dieser Konzepte und beleuchtet dabei vorrangig die interessanten Aspekte der Realisierung.

5.1 Parser

Der Parser nutzt zum Parsen der XMI Konfigurationen die StAX API, welche das Prinzip eines Cursors bzw. Iterators zum Laden der XML Elemente nutzt. Der Parser fragt StAX in einer Schleife nach dem nächsten XML Element und setzt dabei den Cursor auf die nächste Position. Wenn das geholte XML Element für das Framework relevant ist, wird es vom Parser verarbeitet andernfalls ignoriert.

5.1.1 Abbildung der XMI Elemente auf die Elemente des Frameworks

Unter den Annahmen im Kapitel 3.1 wurde festgelegt, dass die Komponentendiagramme im Standard XMI Format exportiert werden müssen. Der Parser durchsucht die so erstellten XMI Dateien und sucht dabei nur die relevanten Informationen heraus. Dieser Abschnitt klärt welche XMI Elemente vom Parser verarbeitet werden und warum sie wichtig für das Framework sind. Alle XMI Elemente die in diesem Abschnitt nicht erwähnt sind, werden vom Framework nicht erkannt, da sie entweder für das Framework irrelevant sind oder nicht zum Standard XMI Format gehören.

Die für das Framework relevanten XMI Elemente sind entweder vom Typ „packagedElement“, oder ein Unterelement eines packagedElement. Die Unterscheidung dieser Elemente erfolgt über das „xmi:type“ Attribut.

Der Parser erkennt folgende Typen:

- **uml:Package**
  Ein Element vom Typ Package zeigt an, dass sich alle folgenden XML Elemente in diesem Package befinden. Der Name des Package wird im Framework für den vollen Namen eines Objektes verwendet. Die Suche nach .class Dateien benötigt den vollen Namen, um in den passenden Verzeichnissen zu suchen. Das einzige relevante Attribut dieses Elementes ist xmi:name für den Namen des Packages.

- **uml:Realization**
  Dieses Element entspricht dem Realisierungskonnektor im Diagramm. Über dieses Element kann das Framework die Zuweisung einer angebotenen Schnittstelle zu einer Implementierung herstellen, so dass es keine Mehrdeutigkeiten geben kann. Die Realisierung hat 2 relevante Attribute: supplier für die ID der angebotenen Schnittstelle und client für die ID der Implementierung.

- **uml:Dependency**
  Dieses Elemente entspricht dem Abhängigkeitskonnektor im Diagramm. Dieser Konnektor zeigt die Abhängigkeit einer Komponente zu dem Service einer anderen Komponente an. In dieser Arbeit werden die Abhängigkeiten für die Zyklenerkennung genutzt. Die Dependency hat 2 relevante Attribute: supplier für die ID der angebotenen Schnittstelle und client für die ID der anfordenden Schnittstelle.

- **uml:Component**
  Dieses Element entspricht einer Komponente im Diagramm. Aus diesem Element kann das Framework den Namen und die ID der Komponente lesen. Diese Informationen werden für die Zyklenerkennung genutzt. Die relevanten Attribute der Komponente ist der xmi:name für den einfachen Namen der Komponente und xmi:id für die ID der Komponente.
  Das Element vom Typ uml:Component besitzt außerdem relevante Unterelemente:

  - **provided**
    Das Unterelement provided kennzeichnet die angebotenen Schnittstellen einer Komponente. Dieses Element wird unter anderem für die Überprüfung, ob es für jede angebotene Schnittstelle eine Implementierung gibt, verwendet. Dieses Element hat 2 relevante Attribute: xmi:id für die ID der angebotenen Schnittstelle und xmi:name für den einfache Namen der Schnittstelle.

- **required**
  Das Unterelement required kennzeichnet die anfordenden Schnittstellen einer Komponente. Dieses Element wird in der Zyklenerkennung genutzt. Für dieses Element ist nur die xmi:id als Attribut für die ID der anfordenden Schnittstelle relevant. Der Name ist irrelevant, da es anfordenden Schnittstellen nur visuelle Hilfsmittel im Diagramm sind und nur auf der angebotenen Seite exisitieren.

- **nestedClassifier**
  Dieses Element kennzeichnet die inneren Elemente einer Komponente. Innere Elemente sind Schnittstellen und Klassen. Die inneren Schnittstellen werden wie die angebotenen Schnittstellen behandelt. Die Klassen entsprechen den Objekten, die der Container auf Anfrage zurückliefern kann. Diese Elemente haben 3 relevante Attribute: xmi:name für den einfachen Namen des Objektes, xmi:id für die ID des Elements und xmi:type für die Unterscheidung ob es eine Schnittstelle oder eine Klasse ist.

5.1.2 Erkennen von zyklischen Abhängigkeiten

Das Komponentendiagramm kann, wie in Kapitel 4.2.1 erläutert, als ein Graph betrachtet werden. Um in einen Graphen nach Zyklen zu finden kann ausgehend vom einem Startknoten eine Tiefensuche gestartet werden. Während der Tiefensuche wird ein Suchpfad aufgebaut bis alle möglichen Wege durchsucht wurden. Taucht in diesem Suchpfad ein Knoten mehrfach auf, so besteht eine zyklische Abhängigkeit im Graphen.

Der Validator in der Parserkomponente nutzt diese Tiefensuche. Der Algorithmus im Validator läuft folgendermaßen ab:

1. Solange nicht alle Komponenten als besucht markiert sind, nimm eine zufällige Komponente, die noch nicht besucht wurde, und markiere diese als Startpunkt des Suchpfades.

2. Suche ausgehend von dieser Komponente alle Nachfolger. Nachfolger sind die Komponenten, die einen Service der Komponente aus Punkt 1 in Anspruch nehmen.

3. a) Gibt es keine Nachfolger, so ist die Suche von dieser Komponente aus beendet. Die Komponente wird als besucht markiert und der Algorithmus startet von vorne mit der nächsten Komponente.
   b) Falls es Nachfolger gibt, dann überprüfe jeden Nachfolger ob er schon im Suchpfad vorhanden ist. Falls ja, dann melde eine zyklische Abhängigkeit und beende das Parsen. Falls nicht dann mache weiter bei Schritt 4

4. Solange es Nachfolger gibt, nimm einen Nachfolger, füge den Nachfolger zum Suchpfad hinzu und mache bei Schritt 5 weiter. Gibt es keine Nachfolger mehr, dann markiere alle Komponenten im Suchpfad als besucht, lösche den Suchpfad und starte wieder bei Punkt 1.

5. Überprüfe ob der Nachfolger bereits besucht wurde. Ist dieses der Fall, dann braucht der Nachfolger nicht mehr betrachtet werden, da er bereits überprüft wurde. Springe in diesem Fall zu Schritt 4 zurück und nehme dir dort den nächsten Nachfolger. Ist der Nachfolger noch nicht besucht worden, dann mache in Punkt 6 weiter.

6. Ist die Komponente noch nicht besucht worden, dann wähle diese als nächste Komponente und mache bei Schritt 3 mit den Nachfolgern der neuen Komponente weiter.

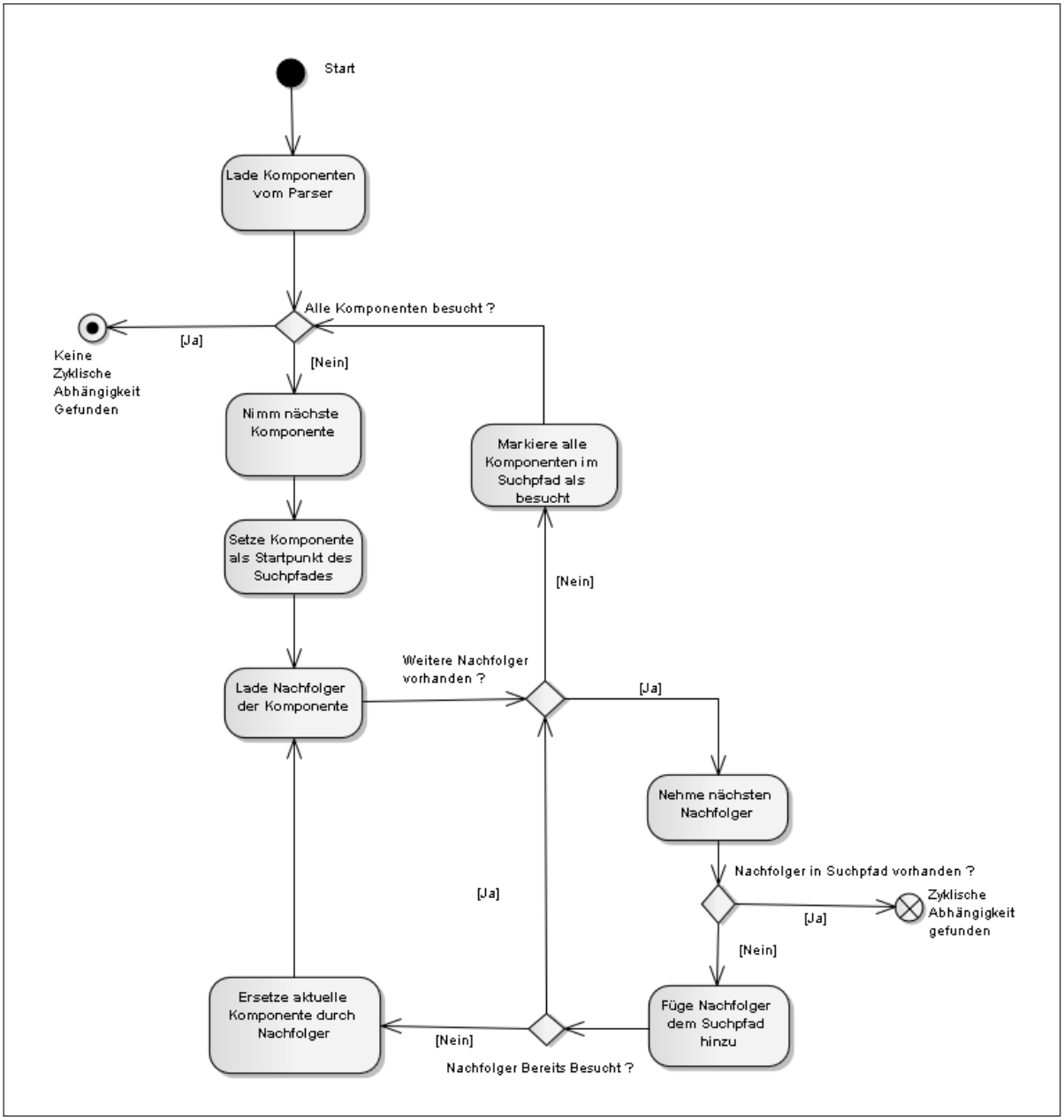

Abbildung 5.1: Ablauf des Algorithmus zur Zyklenerkennung

5.2 Konfigurator

Der Konfigurator ermöglicht die Feinkonfiguration der Objekte, wie zum Beispiel die Wahl des Konstruktors oder das Setzen der Parameter. Das Framework erlaubt die Feinkonfiguration, damit die Diagramme so einfach wie möglich gehalten werden können. So muss der Modellierer nicht für jedes Objekt des Diagramms alle Konstruktoren angeben. Um dennoch an die Konstruktoren der Objekte zu kommen, nutzt diese Arbeit die Reflection API von Java. Die Reflection API kann anhand einer .class Datei ermitteln, welche Variablen und Methoden, und somit auch welche Konstruktoren, eine Klasse besitzt.

Der Konfigurator fragt einen Classloader nach den .class Dateien für alle Objekte und lädt über diese die Konstruktoren. Die Objekte einer Anwendung sind üblicherweise in mehreren Packages verteilt, der Standard Classloader sucht aber nicht in mehreren Verzeichnissen. Dieses Framework nutzt deswegen einen speziell angepassten Classloader der ausgehend von einem Verzeichnis auch in dessen Unterverzeichnissen nach den .class Dateien sucht.

5.2.1 Abhängigkeiten zu nicht fachlichen Objekten

Die Objekte einer Anwendung haben oft Abhängigkeiten zu Objekten außerhalb der Anwendung. Die Objekte können zum Beispiel auch Abhängigkeiten zu technischen Objekten aus den Java Bibliotheken besitzen. Diese technischen Objekte gehören nicht in das Komponentendiagramm, da sie das Komponentendiagramm unübersichtlich machen und von den fachlichen Objekten ablenken. Aber dennoch müssen diese Abhängigkeiten beim Erstellen eines fachlichen Objektes befriedigt werden.

Der Konfigurator umgeht dieses Problem mit der Reflection API. Über die Reflection API lädt sich der Konfigurator die Konstruktoren als Constructor Objekte. Diese Constructor Objekte bieten Zugriff auf die Parameter des Konstruktors, so dass sich über die Constructor Objekte alle Abhängigkeiten eines Objektes auslesen lassen können. Wenn die so gefundenen Parameter Objekte sind, werden sie der Konfiguration hinzugefügt und können wie die fachlichen Objekte in der grafischen Oberfläche konfiguriert werden.

Es gibt 2 Sonderfälle die das Framework zur Zeit noch nicht unterstützt:

1. Abhängigkeiten zu technischen Schnittstellen
   Wenn ein Objekt eine Abhängigkeit zu einer Schnittstelle hat, muss diese Abhängigkeit beim Erzeugen eines Objektes durch eine konkretes Objekt befriedigt werden. Wenn diese Realisierungsbeziehung nicht explizit durch den Anwender angegeben wird, kann der Konfigurator nicht herausfinden welche Klasse für diese Schnittstelle verwendet werden soll. Bei fachlichen Objekten steht diese Realisierungsbeziehung über den Realisierungskonnektor im Komponentendiagramm. Bei technischen Objekten gibt es keine explizite Angabe für diese Beziehung. Der Konfigurator kann bei technischen Schnittstellen nicht entscheiden, welche Implementierung verwendet werden soll.

2. Abhängigkeiten zu Collections und Arrays
   Besitzt ein Objekt Abhängigkeiten zu einer Collection oder einem Array, so müssen diese in den meisten Fällen schon beim Setzen der Abhängigkeit mit Daten befüllt sein. Das Framework muss bei der Unterstützung von Collections und Arrays dem Anwender die Möglichkeit geben diese mit Daten zu befüllen. Die so angegeben Daten müssen dann auch in den ConfigurationObjects gespeichert werden und so dem Generator übergeben werden.
   Collections bieten zusätzlich die Schwierigkeit, dass man sie mit Generics typisieren kann. Bei der Eingabe der Daten muss also zusätzlich auch noch eine Typüberprüfung durchgeführt werden.
   Arrays sind sehr schwierig zu erstellen, da Arrays ein Sondertyp in der Java Sprache sind, sie sind weder Objekte noch primitive Typen. Ein Array gehört zu keiner Klasse, so liefert die Anfrage nach der Klasse eines Arrays kryptischen Code (z.B. [B; für ein byte Array). Arrays lassen sich kaum in ihrer Klassenform darstellen, was zu Schwierigkeiten in der Generatorkomponente führt.
   Weiterhin muss bei Arrays zusätzlich zum Typ auch die initiale Größe angeben werden, und die Einhaltung dieser Größe bei der Eingabe der Daten überwacht werden.
   Da die Unterstützung von Collections und Arrays sehr viel Aufwand bedeutet, werden diese in dieser Arbeit nicht unterstützt.

5.2.2 Sortierung der Objekte in ihrer Initialisierungsreihenfolge

Die Objekte müssen, wie unter 4.2.2 beschrieben, in einer definierten Reihenfolge initialisiert werden. Die Konfigurationskomponente sortiert dafür die Objekte bevor die Generierung des Containers gestartet wird. Dabei müssen die Serviceanbieter vor den Servicenutzern initialisiert werden. Der Konfigurator nutzt zur Sortierung der Objekte folgenden Algorithmus:

1. Zuerst werden alle Objekte herausgesucht die keine Abhängigkeit zu anderen Objekten haben. Dazu wird in jedem Objekt der Konstruktor betrachtet. Hat der Konstruktor entweder nur primitive Parameter (inklusive Strings) oder keine Parameter, so hat das Objekt keine Abhängigkeiten zu anderen Objekten. Diese Objekte werden im Container zu erst erstellt.
2. Alle so gefundenen Objekte sind Serviceanbieter. Sie bieten sowohl sich selbst als auch alle Schnittstellen die sie implementieren als Service an. Die so gefundenen zur Verfügung stehenden Services werden zu einer Sammlung aller zu diesem Zeitpunkt verfügbaren Services hinzugefügt.
3. Anhand der Liste der zur Zeit verfügbaren Services wird jedes noch offene Objekt überprüft, ob mit diesen Services seine Abhängigkeiten befriedigt werden können. Ist dieses der Fall kann das Objekt als nächstes initialisiert werden. Mit diesen nun verfügbaren Objekten wird der Algorithmus ab Schritt 2 weitergeführt, da die nun verfügbaren Objekte wiederrum neue Services anbieten. Wenn in diesem Schritt kein neues Objekt mehr initialisiert werden kann, sind alle Objekte nach ihrer Initialisierungsreihenfolge sortiert.

5.3 Generator & Container

Der Generator liest über einen InputStream die Template Datei zeichenweise ein. Solange der Generator dabei nicht auf die Zeichen # oder $ trifft, überträgt er die gefundenen Zeichen 1:1 in den Container. Trifft der Generator beim Lesen auf das # Zeichen, so hat er ein Schlüsselwort gefunden. Der Generator liest daraufhin alle Zeichen bis zum nächsten # Zeichen als Schlüsselwort. Dieses Schlüsselwort bedeutet, dass der Generator das Schlüsselwort an dieser Stelle durch generierten Code ersetzten soll. Das $ Zeichen ist ein spezielles Zeichen, welches am Ende von Methoden steht, die bei bestimmten Konfigurationen, zum Beispiel wenn es keine unique Objekte gibt, leer sein können.
Der Generator überliest in diesem Falle alle Zeichen in der Template Datei von einem Schlüsselwort am Anfang der Methode bis zu dem $ Zeichen.

5.3.1 Schlüsselwörter

Beim Lesen der Template Datei trifft der Generator auf einige Schlüsselwörter die bestimmte Aktionen auslösen. In der aktuellen Versionen werden folgende Schlüsselwörter erkannt:

- #PACKAGE#
  An dieser Stelle setzt der Generator die package Zeile für den Generator. Diese steht als erste Zeile in einer Source Datei. Das Package des Containers entspricht dabei den Namen des Verzeichnisses in welches der Container erzeugt wird.
- #CLASSNAME#
  An dieser Stelle setzt der Generator den Namen des Containers. Der Name des Container wird über die grafische Oberfläche gesetzt. Der Name wird unter anderem für den Klassennamen in der Source Datei des Containers benötigt.
- #UNIQUECONS#
  Der Generator erzeugt an dieser Stelle für jedes unique Objekt ein Constructor Objekt. Die Constructor Objekte werden benötigt um später in der Methode diese unique Objekte zu erzeugen. Wenn es keine unique Objekte gibt, überträgt der Generator keine Zeichen mehr in den Container bis er das $ Zeichen einliest.
- #INITUNIQUECONS#
  Der Generator erzeugt an dieser Stelle die Zuweisung der unter #UNIQUECONS# erstellten Constructor Objekten zu den Constructor Objekten die der Anwender in der Konfiguratorkomponent ausgewählt hat. Mit dieser Zuweisung können diese Objekte später im Code erzeugt werden.
- #CREATEUNIQUES#
  Der Generator erzeugt an dieser Stelle sowohl den Code zur Erzeugung der unique Objekte als auch den Code um diese frisch erzeugten Objekte in einer Map zu speichern. Diese Map hat den vollen Namen des Objektes als Key und das Objekt als Value, so dass bei jeder Anfrage an diese Map dieses Objekt zurückgeliefert wird.
- #INITPARAMS#
  An dieser Stelle werden alle Parameter für jedes nicht unique Objekt erstellt und in einer Map mit den Klassennamen und einem Array der Parameter gespeichert. Die so erstellten Paramter Arrays werden zur Erstellung neuer Exemplare auf Anfrage benötigt.

- #IMPLEMENTATIONS#
  An dieser Stelle erzeugt der Generator die Abbildung von Schnittstelle zu ihrer Implementation. Dabei wird der volle Name der Schnittstelle und der volle Name der Implementation in einer Map abgelegt, so dass diese Map auf Anfrage eine Implementierung für die Schnittstelle angibt, die dann geladen werden kann.
- #NONUNIQUES#
  An dieser Stelle erzeugt der Generator die Abbildung von den Namen eines nicht unique Objekts zu dessem gewähltem Constructor Object und speichert diese in einer Map. Durch diese Abbildung kann der Container bei einer Anfrage auf ein nicht unique Objekt, dessen Constructor Objekt auffordern eine neues Exemplar des Objektes zu erstellen.

5.3.2 Aufbau des Containers

Das Herzstück des Containers ist die getInstance(String objectName) Methode. Diese Methode liefert, falls vorhanden das Objekt mit den angegeben Objektnamen entweder in einer Kopie oder über eine Referenz zurück.

Wird diese Methode aufgerufen, wird zuerst überprüft ob das gewünschte Objekt ein unique Objekt ist. Unique Objekte werden in einer Map mit dem Namen des Objektes und dem eigentlichen Objekt in einer Map abgelegt. Die Anfrage nach einem unique Objekt wird einfach an diese Map delegiert, welche die Referenz auf das Objekt zurückliefert. Das Erstellen dieser Map erfolgt bereits während der Initialisierung des Containers, so dass alle Objekte sofort bereit stehen.

Wird das angefragte Objekt nicht in dieser Map gefunden, so überprüft der Container ob das Objekt als nicht unique Objekt bekannt ist. Nicht Unique Objekte stehen ebenfalls in einer Map mit dem Namen des Objektes als Key. In dieser Map stehen aber keine fertig initialisierten Objekte sondern deren Constructor Objekte als Value. Wird der Container nach einem nicht unique Objekt gefragt, so holt sich der Container das Constructor Objekt des gewünschten Objektes aus dieser Map. Zusätzlich lädt sich der Container aus einer weiteren Map die Parameter des Objektes als Object Array. Mit dem Constructor Objekt und dem Object Array, ruft der Container die newInstance Methode des Constructor Objekts. So wird bei jedem Aufruf ein neues Exemplar des Objektes erstellt.

Wird das angefragte Objekt immer noch nicht gefunden, so prüft die getInstance Methode ob es sich bei dem Objekt um eine Schnittstelle handelt. Dafür durchsucht die Methode eine Map mit dem Schnittstellen Namen als Key und den Namen der implementierenden Klasse als Value nach dem gewünschtem Objekt. Wenn in dieser Map der Name der Schnittstelle gefunden wird, so holt sich die getInstance Methode den Namen der Implementierung und ruft sich mit diesem Namen selber auf. Wenn als die getInstance Methode mit dem Namen einer Schnittstelle aufgerufen wird, so liefert das Framework ein Objekt zurück, welches dem Typen der Schnittstelle entspricht.

Der Container nutzt zum Setzen der Abhängigkeiten eines Objektes intern selbst die getInstance Methode, da die Abhängigkeiten als Objekte dem Constructor Objekt übergeben werden müssen. Dieser interne Aufruf ist durch die Sortierung der Objekte in ihrer Initialisierungsreihenfolge (siehe 5.2.2) möglich.

Die getInstance Methode liefert bei bekannten Objekten immer ein Objekt von dem Java Typen Object zurück. Die nutzende Anwendung muss also diese Objekte nach dem Holen aus dem Container immer auf den gewünschten Typen casten.

Bugfix des Containers

Beim Testen des Frameworks hat sich ein konzeptioneller Fehler (siehe 6.5) beim Design des Containers herausgestellt. Die Reihenfolge der Initialisierung gilt für alle Objekte, ob unique oder nicht. Die Trennung in unique und nicht unique Objekte sorgt dafür, dass entweder zuerst alle unique Objekte oder alle nicht unique Objekte zuerst erstellt werden. Werden zum Beispiel zuerst die nicht unique Objekte erstellt und hat eines dieser Objekte eine Abhängigkeit zu einem nicht unique Objekt, so kann diese Abhängigkeit zum Zeitpunkt der Erstellung des nicht unique Objektes nicht befriedigt werden.

Der Container umgeht nach diesem Bugfix diesen Fehler in dem er sich die Ob-

jekte, die aufgrund fehlender Abhängigkeiten noch nicht vollständig erstellt werden können, mit dessen Parametern in einer Map merkt. Außerdem merkt sich das Framework bei einer fehlenden Abhängigkeit den Namen der Abhängigkeit und alle Objekte die diese Abhängigkeit gesetzt bekommen müssen. Sobald das Objekt, das eine fehlende Abhängigkeit befriedigen kann, erstellt wird, versucht der Container alle Objekte, die eine Abhängigkeit zu diesem Objekt haben zu erstellen.

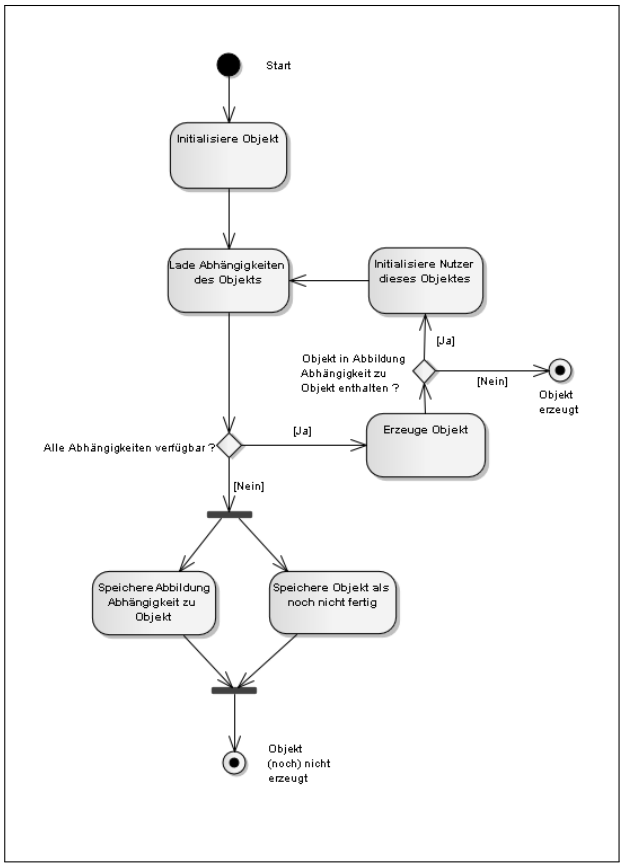

Abbildung 5.2: Neuer Ablauf der Initialisierung der Objekte im Container

6 Test

Ob das entwickelte Framework den Anforderungen entspricht und korrekt funktioniert, klärt dieses Kapitel. Dabei konzentriert sich das Kapitel nur auf die wichtigsten Funktionen des Frameworks. Für alle Test wird das Versandhaussystem der Einführung (siehe 2.2) verwendet.

6.1 Test 1: Erkennen aller XMI Elemente

Die Grundvorraussetzung für ein modellbasiertes Framework ist die Möglichkeit Modelle einlesen zu können. Das in dieser Arbeit entwickelte Framework liest dabei über einen Parser Modelle im XMI Format ein. Dieser Test überprüft ob der hier entwickelte XMI Parser die XMI Dateien korrekt einliest. Korrekt bedeutet, dass der Parser alle Elemente mit ihren relevanten Attributen findet.

Vorbereitung des Tests
Dieser Test nutzt die Releasekonfiguration des Versandhauses (siehe 2.3). Das Diagramm dieser Konfiguration ist in dem Modellierungswerkzeug Enterprise Architect erstellt und wird dort als Standard XMI Datei exportiert. Dafür muss im Enterprise Architect die Option „Enable Full Roundtrip“ abgewählt und die Option „Exclude EA Tagged Values“ gewählt werden.

Erwartete Ergebnisse
Der XMI Parser soll in der gewählten Konfiguration folgende Elemente finden:

- Klassen mit den Namen:
  kunden.KundenVerwalter, kundenDienst.KundenDienst,bestellungen.BestellungsVerwalter, lager.Lager, einkauf.Einkauf
- Angebotene Schnittstellen den Namen:
  kunden.IKunde, bestellungen.IBestellungService, lager.ILagerBestellung, lager.ILagerEinkauf

- Komponenten mit den Namen:
  Kunde, KundenDienst, Bestellung, Lager, Einkauf
- Verbindungen zwischen den Komponenten von der angebotenen zur anfordenden Schnittstelle (jeweils mit der ID):
  Kunde => KundenDienst, Bestellung => KundenDienst, Lager => Bestellung, Lager=>Einkauf
- Realisierungsbeziehung von angebotener Schnittstelle zu Implementation (jeweils mit der ID):
  kunden.IKunde=>kunden.KundenVerwalter, bestellungen.IBestellungService=>bestellungen.Bes
  lager.ILagerBestellung=>lager.Lager, lager.ILagerEinkauf=>lager.Lager

Die IDs der Elemente Klasse, Schnittstelle und Komponente werden dabei auch eingelesen. Jedoch können diese für den Test nicht genutzt werden, da die IDs von dem Modellierungswerkzeug vergeben werden, und somit nicht vorrausgesagt werden können.

Durchführung des Tests

Der Test wird mit dem JUnit Testframework anhand der XMIParser Klasse durchgeführt. Dazu wird für jede Klasse von Elementen (Klassen, Schnittstellen, ...) eine Liste mit den erwarteten Werten erstellt. Die Testklasse ruft die Methoden des Parser auf, die diese erwarteten Werte zurückliefern sollen.

Zuerst wird überprüft ob je Elementklasse die Anzahl der gefundenen Elemente mit der Anzahl der erwarteten Elemente übereinstimmt. Sind alle Elemente gefunden worden, so iteriert die Testklasse über die Liste mit den erwarteten Elementen und überprüft ob jedes Element auch in der Liste der gefundenen Elemente enthalten ist.

Der Test, ob alle Realisierungsbeziehungen und ob alle Verbindungen vom Parser gefunden werden, ist aufwändiger, da der Parser diese Beziehungen nur über die IDs der beteiltigten Elemente erkennt. Die Testklasse wandelt diese IDs zuerst in die Namen der Objekte um, bevor sie diese Beziehungen testet.

Der Test wird sowohl für den Erfolgsfall als auch für den Fehlerfall durchgeführt. Um den Fehlerfall zu provozieren, werden die Elemente in den Listen der erwarteten Elemente entweder entfernt oder deren Namen verändert.

Ergebnis des Tests
Der Test für den Erfolgsfall hat alle Elemente gefunden. Der Test für den Fehlerfall hat der JUnit Test die erwarteten Fehlermeldungen angezeigt. Beim Test wurden keine Fehler des XMIParser festgestellt.

6.2 Test 2: Erkennen von zyklischen Abhängigkeiten

Die Objekte des Versandhauses können nur korrekt erstellt werden, wenn es keine zyklischen Abhängigkeiten in der Konfiguration gibt. Bei einer zyklischen Abhängigkeit kann der Container nicht entscheiden welches der an der zyklischen Abhängigkeit beteiltigten Objekte zuerst initialisiert werden soll (siehe 4.2.1). Dieser Test überprüft die Zyklenerkennung des SemanticValidators.

Vorbereitung des Tests
Da das Versandhaussystem keine zyklischen Abhängigkeiten aufweist, wird es für diesen Test temporär so verändert, dass ein Zyklus auftritt. Dazu wird das Versandhaussystem in diesem Test um 2 weitere Abhängigkeiten erweitert: 1. Die Kundenkomponente ist abhängig von der Einkaufskomponente und 2. Die Lagerkomponente ist abhängig von der Kundenkomponente. Somit besteht nun ein Zyklus zwischen den Komponenten Lager, Kunde und Einkauf.

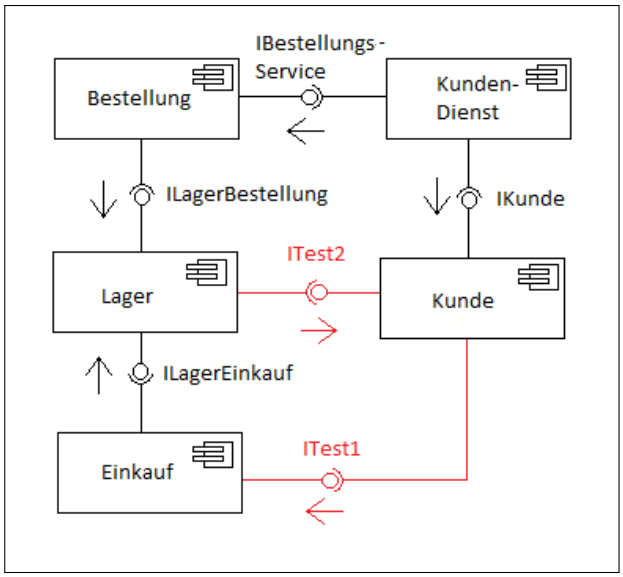

Abbildung 6.1: Veränderte Konfiguration des Versandhaussystems

Die so veränderte Konfiguration wird wie im ersten Test über die Exportfunktion von Enterprise Architect in das Standard XMI Format exportiert. Es wird zum Parsen der Konfiguration der in dieser Arbeit entwickelte XMIParser verwendet, wobei in diesem Test die Validierung eingeschaltet ist.

Erwartete Ergebnisse
Das Framework soll die zyklische Abhängigkeit zwischen dem Lager, dem Kunden und dem Einkauf finden. Sobald das Framework diesen Zyklus gefunden hat, soll es es eine Fehlermeldung ausgeben, die diesen Zyklus mit allen beteiltigten Komponenten ausgibt. Da die zyklische Abhängigkeit für das Framework eine unlösbare Aufgabe ist, soll das Framework nach Ausgabe der Fehlermeldung beendet werden.

Durchführung
Der Test initialisiert zunächst den Parser und schaltet dabei die Validierung über die entsprechende Methode ein. Nach der Initialisierung durchläuft der Parser die für diesen Test angepasste Konfiguration.

Ergebnis des Tests
Der SemanticValidator hat die zyklische Abhängigkeit mit den erwarteten beteiltigen Komponenten gefunden. Er warf die korrekte Fehlermeldung Lager => Kunde => Einkauf => Lager und hat das Framework danach erwartungsgemäß beendet.

6.3 Test 3: Vorbereitung der Konfiguration für den Generator

Nachdem der Anwender die Objekte über die grafische Oberfläche nach seinen Wünschen angepasst hat, werden diese Objekte in einem speziellem Datentyp, dem ConfigurationObject, gespeichert. Das ConfigurationObject enthält den vollen Namen des Objektes, dem vom Anwender gewähltem Konstruktor inklusive der gewählten Werte für seine Parameter und einem boolean Wert, der anzeigt ob das Objekt unique sein soll oder nicht.

Dieser Test dient dazu herauszufinden ob das Framework die ConfigurationObject Objekte korrekt erstellt. Weiterhin wird bei diesem Test auch die Sortierung der Objekte nach ihrer Initialisierungsreihenfolge getestet (siehe 5.2.2).

Vorbereitung des Tests
In diesem Test wird folgende Konfiguration des Versandhauses gewählt:

- Lager
  Das Lager nutzt den leeren Konstruktor und soll unique sein
- Einkauf
  Der Einkauf nutzt den Konstruktor mit der Abhängigkeit zu der ILagerEinkauf Schnittstelle, so wie einem int und einem char als primitive Parameter. Die Werte für diese Parameter sind zufällig gewählt. Das Einkaufsobjekt soll unique sein.
- KundenVerwalter
  Der Kundenverwalter nutzt ebenfalls den leeren Konstruktor, soll aber nicht unique sein
- KundenDienst
  Der Kundendienst nutzt den Konstruktor in denen die Abhängigkeiten zu der IKunde und der IBestellungservice Schnittstelle stehen. Der Kundendienst soll nicht unique sein.
- BestellungsVerwalter
  Der Bestellungsverwalter nutzt den Konstruktor mit seiner Abhängigkeit zu der ILagerBestellung Schnittstelle und soll nicht unique sein.

Diese Konfiguration wurde bewußt gewählt. So muss zum Beispiel bei den beiden unique Objekten Einkauf und Lager, das Lager vor dem Einkauf initialisiert werden.

Erwartete Ergebnisse
Der Test soll das Lager und den Einkauf als unique Objekte, sowie den KundenVerwalter, den Kundendienst und den BestellungsVerwalter als nicht unique Objekte erkennen. In der Liste der unique Objekte soll das Lager vor dem Einkauf stehen. In der Liste der nicht unique Objekte müssen der KundenVerwalter und der BestellungsVerwalter vor dem Kundendienst stehen.

Außerdem sollen alle Konstruktoren der Objekte die unter der Vorbereitung genannten Werte für ihre Parameter besitzen.

Durchführung des Tests
Der Test nutzt das JUnit Test Framework. Die Erstellung der Konfiguration erfolgt in diesem Test nicht über die grafische Oberfläche, sondern wird textuell in dem für die Konfigurationskomponente erforderlichem Format erstellt.

Bevor der Test beginnt, parst der XMIParser die Konfigurationsdatei des Versandhauses, so dass beim Erstellen der ConfigurationObject Objekte überprüft werden kann, ob die Objekte des Versandhauses die gewünschten Konstruktoren besitzten.

Zur Überprüfung lädt sich der Test die unique bzw. nicht unique Objekte über die entsprechenden Methoden der Configurator Klasse. Im ersten Schritt wird überprüft, ob die Anzahl der gefundenen Objekte der erwarteten Anzahl entspricht. Im nächsten Schritt überprüft der Test die Reihenfolge der Objekte in den Listen der Objekte. Zuletzt wird noch überprüft ob die Parameter der gefundenen Objekte den erwarteten Parametern entsprechen.

Ergebnis des Tests
Es wurden alle Objekte gefunden und korrekt der unique und der nicht unique Liste zugeordnet. Alle Objekte wurden von der Configurator Klasse korrekt sortiert. Die gefundenen Parameter entsprachen den gewünschten Parametern. Die Erstellung der ConfigurationObject Objekte war erfolgreich.

6.4 Test 4: Test des Generators

Der Generator liest eine Template Datei ein und überträgt diese in eine Zieldatei. Dabei ersetzt der Generator eingelesene Schlüsselwörter durch die Informationen aus der Konfigurationsdatei. Dieser Test überprüft ob der Generator beim Lesen der Template Datei alle Schlüsselwörter findet und die Zieldatei korrekt erstellt.

Vorbereitung des Tests
Für den Test gibt es eine spezielle Template Datei die genutzt werden soll. In dieser Template Datei stehen alle dem Generator bekannten Schlüsselwörter und einen Abschnitt der nicht in die Zieldatei übertragen werden soll. Der Test benötigt ein Exemplar der Configurator Klasse um den Container in ein Zielverzeichnis erstellen zu können.

Erwartete Ergebnisse
Die Schlüsselwörter sind komplett in Großbuchstaben geschrieben und sind von dem Zeichen # umgeben. Der Testgenerator soll alle Schlüsselwörter in ihr Pendant in Kleinbuchstaben ohne das # Zeichen umwandeln. Beispielsweise soll das Schlüsselwort „#PACKAGE#“ in „package“ umgewandelt werden.
Ausgenommen hier von ist das Schlüsselwort „#INITUNIQUECONS#“. Wenn der Generator auf dieses Schlüsselwort trifft, so soll ab dieser Stelle bis hin zum $ Zeichen

alle Zeilen in der Template Datei überlesen werden.

Durchführung des Tests
Der Test nutzt eine für den Test angepasste Unterklasse der abstrakten Generatorklasse. Diese Klasse ersetzt in ihren Callback Methoden die Schlüsselwörter in ihr Pendant in der Kleinschreibung. Ausgenommen hiervon ist die Callback Methode des Schlüsselwortes „#INITUNIQUECONS#“. Ab diesem Schlüsselwort überliest der Test alle Zeilen der Template Datei bis zum $ Zeichen

Bei einem zweiten Testdurchlauf wird der Template Datei ein dem Framework unbekanntes Schlüsselwort hinzugefügt, um zu Testen wie der Generator auf unbekannte Schlüsselwörter reagiert.

Die Auswertung des Testes erfolgt über einen manuellen Vergleich der Template Datei mit der erzeugten Datei.

Ergebnis des Tests
Im ersten Testdurchlauf wurden alle Schlüsselworter gefunden und diese korrekt ersetzt. Außerdem wurden alle Zeilen ab dem Schlüsselwort „#INITUNIQUECONS#“ bis zum $ Zeichen erwartungsgemäß ignoriert.

Im zweiten Testdurchlauf wurde das unbekannte Schlüsselwort erkannt und ohne Veränderung in die Zieldatei übertragen.

6.5 Systemtest des Frameworks

Dieser Test überprüft abschließend das Zusammenspiel aller Komponenten. Der Systemtest testet das gesamte Framework vom Einlesen der XMI Datei des Versandhauses bis zum Erstellen des Containers.

Vorbereitung des Tests
Vor dem Test wird das Framework in der Release Konfiguration des Versandhauses (siehe 2.3) durch Enterprise Architect in eine XMI Datei exportiert. Die Feinkonfiguration entspricht der Konfiguration aus dem drittem Test (siehe 6.3). Das Versandhaus System ist vor dem Test bereits erstellt und der Test hat Zugriff auf dessen .class Dateien.

Das Framework selbst ist so konfiguriert, dass nur die in dieser Arbeit entwickelten Objekte genutzt werden. Die XMIParser Klasse übernimmt das Parsen der XMI Datei, die Konfiguration wird mit dem SemanticValidator überprüft und der Container wird von der JavaGenerator Klasse mit der JavaTemplate Datei erstellt.

Um zu testen, ob der Container alle Objekte korrekt erstellt, wird ein Exemplar des Containers in einer speziellen Starterklasse des Versandhauses erstellt. Die Starterklasse erzeugt über den Container ein Exemplar des Kundendienstes und ein Exemplar des Einkaufs um damit das Versandhaus zu starten. Diese Starterklasse entspricht den Starterklassen bei dem Vergleich der Frameworks in der Einleitung (siehe 2.3).

Erwartete Ergebnisse
Der Parser soll alle 5 Objekte des Versandhauses in ihrer gewählten Konfiguration erkennen. Der SemanticValidator soll in dieser Konfiguration keine Fehler finden.

In der grafischen Oberfläche sollen alle 5 Objekte mit all ihren Konstruktoren auswählbar sein. Wenn ein Konstruktor primitive Parameter und Strings enthält so sollen diese in gültige Werte änderbar sein.
Der Generator erzeugt einen Container, mit dem über die grafische Oberfläche gewählten Namen, in einer Java Source Datei in dem vom Anwender gewähltem Zielverzeichnis.

Der so erzeugte Container soll folgendermaßen in die Starterklasse eingebunden werden:

```
1
2  public static void main(String[] args) {
3
4          /*
5          * VersandConfig ist der vom Anwender
6          * gewählte Name des Containers
7          */
8          VersandConfig config=new VersandConfig();
9
10         Einkauf einkauf=(Einkauf)config.getInstance("
               einkauf.Einkauf");
11         KundenDienst kd=(KundenDienst)config.getInstance
               ("kundenDienst.KundenDienst");
12
13         new KundenDienstFrame(kd);
14         new EinkaufFrame(einkauf);
15
16 }
```

Listing 6.1: Erzeugen des Containers

Nachdem Starten des Versandhauses über die obige Starterklasse, sollen alle Funktionen des Versandhauses korrekt funktionieren.

Durchführung des Tests

Der Test startet über die Starterklasse des Frameworks oder über das Ausführen der Mobadi.jar Datei. Bevor die XMI Datei eingelesen wird, schaltet der Tester die Validierung ein. Nach dem Einlesen setzt der Tester das Lager Objekt und das Einkauf Objekt als unique Objekte. Außerdem wählt er für das Einkaufsobjekt den Konstruktor mit dem int und dem char Parameter. Diese Parameter werden durch den Tester auf neue Werte gesetzt, wobei auch ungültige Werte getestet werden sollen.

Der Tester exportiert nach der Konfiguration den Container in sein gewünschtes Zielverzeichnis. Nach der Exportierung des Containers kann das Framework beendet werden. Der Tester bindet die exportierte Container Datei in das Projekt des Versandhauses ein, so dass die Starterklasse darauf Zugriff hat.

Der Tester startet daraufhin das Versandhaus. Nach erfolgreichem Start erstellt

der Tester im Versandhaus einen neuen Kunden und gibt für den neuen Kunden eine Bestellung auf. Der Tester überprüft den Bestand der Artikel der Bestellung vor und nach der Bestellung. Außerdem fügt der Tester einen neuen Artikel dem Lager hinzu.

Ergebnis des Tests

Der erste Systemtest schlug fehl. Das Framework konnte erfolgreich gestartet werden. Das Laden der XMI Datei bei aktivierter Validierung verlief problemlos. Die Anpassung der Konfiguration sowie der Export des Containers erfolgte erwartungsgemäß.

Das Versandhaus ließ sich über den Container starten und ein neuer Kunde wurde erfolgreich angelegt. Jedoch konnte keine Bestellung aufgegeben werden. Da im Container die nicht unique Objekte zuerst initialisiert werden, kann die Abhängigkeit des Bestellungsverwalter zu dem Lager bei der Erstellung des BestellungsVerwalters nicht befriedigt werden.

Nach der Behebung dieses Fehlers (siehe 5.3.2) konnten alle Aufgaben beim Test des Versandhauses mit dem Container erfolgreich erledigt werden.

7 Schluss

Nachdem das modellbasierte Dependency Injection Framework entwickelt und implementiert ist, fasst dieses Kapitel die bei dieser Arbeit gewonnenen Erkenntnisse zusammen.

7.1 Zusammenfassung

Die theoretische Einführung (2.1) klärt zunächst, was Dependency Injection überhaupt ist und warum dieses Entwurfsmuster bei dem Design von Softwaresystemen so wichtig ist. Außerdem stellt die theoretische Einführung PicoContainer, Guice und Spring als bereits auf dem Markt befindliche Frameworks vor, um zu erläutern wie klassische Dependency Injection Frameworks dieses Entwurfsmuster umsetzen. Um dabei eine Vergleichsmöglichkeit zu haben, wird ein fiktives Versandhaussystem eingeführt.

Weiterhin beschäftigt sich die theoretische Einführung mit der Problemstellung der geeigneten Darstellung von Softwaresystemen, so dass diese von dem Framework eingelesen werden können. Dabei stellten sich UML 2 Komponentendiagramme und dessen Zwischenrepräsentation im XMI Format als geeignet heraus.

Die Einführung schließt mit einer Vorstellung der XML Parser APIs SAX, DOM und StAX ab, die zum Lesen von Dateien im XMI Format genutzt werden können.

Nachdem die Grundlagen gelegt sind, formuliert diese Arbeit in einem Analyseteil (3) mögliche Anforderungen an ein modellbasiertes Dependency Injection Framework . Die Erstellung von Modellen bieten dem Modellierer viele Freiheitsgerade. Ein Modell kann zum Beispiel sehr abstrakt oder sehr detailliert sein oder in verschiedenen Formaten erstellt werden. Damit das Framework korrekt arbeiten kann, werden zunächst einige Bedingungen an das Modell und dessen Format gestellt.

Das Analyse Kapitel führt daraufhin die Anforderungen, die Prämissen und die Ausgrenzungen im Stile eines Lastenhefts ein. Dabei werden diese Punkte im Gegensatz zum Lastenheft zur Nachvollziehbarkeit zusätzlich auch begründet.

Die Analyse schließt mit der Vorstellung eines Beispielszenarios für das Konfigurieren des Versandhaus Beispielsystems ab, um aufzuzeigen wie das Framework bedient werden soll.

Im Kapitel Design (4) wird festgestellt, dass die Aufgabe des Frameworks ähnlich der Aufgabe eines Compilers ist, da das Framework XMI Dateien als Quellformat in Java Source Dateien als Zielformat umwandelt. Somit ähnelt auch die Architektur des Frameworks der eines Compilers, mit dem Parser des Frameworks als Frontend zum Einlesen der XMI Datei, der Konfiguratorkomponente zum Bearbeiten der Objekte und der Generatorkomponente als Backend für die Generierung des Containers. Das Kapitel Design erläutert diese 3 Komponenten anschließend im Detail.

Das Design Kapitel schließt mit dem Entwurf des Containers ab. Dieser liefert auf Anfrage entweder jedesmal dasselbe Objekt zurück (unique Objekt) oder erzeugt bei jeder Anfrage ein neues Objekt.

Das Kapitel Realisierung (4) zeigt besondere Aspekte bei der Umsetzung des Designs auf. Zuerst wird erläutert welche XMI Elemente vom Framework erkannt werden. Außerdem klärt dieses Kapitel wie das Framework nach zyklischen Abhängigkeiten sucht.

In einem Abschnitt über die Generatorkomponente wird erläutert, wie das Framework Abhängigkeiten zu nicht fachlichen Klassen findet und warum alle Objekte in einer bestimmten Reihenfolge initialisiert werden müssen. Abschließend erklärt das Realisierungs Kapitel wie aus der Konfiguration ein Container als Java Source Datei erstellt wird.

Ob das Framework wie erwartet funktioniert, klärt das Kapitel Test (6). Dabei werden die relevanten Funktionen des einzelnen Komponenten an dem Beispiel Versandhaus System aus der Einführung getestet. Beispielsweise wird getestet ob der Parser alle Elemente der Konfigurationsdatei findet oder ob die Umwandlung der Konfiguration in einen speziellen Datentyp des Frameworks erwartungsgemäß funktioniert. Dabei wird zur Nachvollziehbarkeit erläutert wie der Test erstellt und ausgeführt wird.

Abschließend wird das gesamte Framework in einem Systemtest getestet. Dabei wird eine gewünschte Konfiguration des Versandhaussystems festgelegt und überprüft ob das Framework diese Konfiguration in den Container übertragen kann. Außerdem wird überprüft, ob der Container die Objekte vollständig initialisiert zurückliefert, indem die Funktionen des Versandhauses ausgeführt werden.

7.2 Bewertung

Einer der großen Vorteile von guten Modellen ist ihre bessere Lesbarkeit und Übersichtlichkeit gegenüber textuellen Beschreibungen. Diese beiden Vorteile werden stärker je abstrakter ein Modell ist. Um ein rein modellbasiertes Dependency Injection Framework zu entwickeln, also ein Framework, dass direkt das Modell in den Container umwandelt, müssen die Modelle sehr detailliert sein. Die beiden genannten Vorteile sind dadurch nicht sehr groß. Deswegen ist die Entwicklung eines rein modellbasierten Dependency Injection Frameworks durchaus möglich, aber der Vorteil gegenüber klassichen Dependency Injection Frameworks ist nicht so groß. Besser ist der in dieser Arbeit genutzte Mittelweg, wo der Anwender in den Erstellungsprozess über eine grafische Oberfläche eingreifen kann.

Optimalerweise lässt ein modellbasiertes System dem Modellierer sehr viele Freiheitsgrade bei der Erstellung des Modells. Jedoch muss ein Modell für das System eindeutig interpretierbar sein um deterministische Ergebnisse zu liefern. Deswegen muss für die Entwicklung eines modellbasierten System die Freiheitsgrade stark eingeschränkt werden was zu Kosten der Flexibilität geht.

Ein weitere Schwierigkeit beim Erstellen eines Dependency Injection Frameworks ist die große Flexibilität der Programmiersprache für die das Framework entwickelt wird. So können in Java die Parameter eines Konstruktors primitiv, Objekte, generische Collections oder Arrays sein. Diese Flexibilität macht das automatische Erzeugen von Objekten in einem Container schwieriger als erwartet.

Das Ziel dieser Arbeit wurde dennoch erreicht. Das Framework liest XMI Dateien aus UML 2 Diagrammen ein und erzeugt einen funktionierenden Java Container. Die noch fehlende Flexibilität kann in zukünftigen Versionen des Frameworks hinzugefügt werden.

Literaturverzeichnis

[Dedasys LLC 2010] DEDASYS LLC: *Programming Language Popularity*. 2010. – URL http://langpop.com/. – Zuletzt aufgerufen am 04.07.2010

[Fowler 2004] FOWLER, Martin: *Inversion of Control Containers and the Dependency Injection pattern*. 2004. – URL http://martinfowler.com/articles/injection.html. – Zuletzt Aufgerufen am 15.06.2010

[Grose u. a. 2002] GROSE, Timothy J. ; DONEY, Gary C. ; BRODSKY, Stephen A.: *Mastering XMI*. John Wiley and Sons, 2002. – ISBN 0-471-38429-1

[Hitz u. a. 2005] HITZ, Martin ; KAPPEL, Gerti ; KAPSAMMER, Elisabeth ; RETSCHITZEGGER, Werner: *UML @ Work*. 3.Auflage. dpunkt.verlag, 2005. – ISBN 3-89864-261-5

[Jenkov] JENKOV, Jakob: *What is Dependency Injection*. – URL http://tutorials.jenkov.com/dependency-injection/index.html. – Zuletzt Aufgerufen am 12.03.2010

[Johnson u. a. 2004 –2008] JOHNSON, Rod ; HOELLER, Juergen ; ARENDSEN, Alef ; SAMPALEANU, Colin ; HARROP, Rob ; RISBERG, Thomas ; DAVISON, Darren ; KOPYLENKO, Dmitriy ; POLLACK, Mark ; TEMPLIER, Thierry ; VERVAET, Erwin ; TUNG, Portia ; HALE, Ben ; COLYER, Adrian ; LEWIS, John ; LEAU, Costin ; FISHER, Mark ; BRANNEN, Sam ; LADDAD, Ramnivas ; POUTSMAT, Arjen: *Spring java/j2ee Application Framework*. 2004 -2008. – URL http://static.springsource.org/spring/docs/2.5.x/spring-reference.pdf. – Zuletzt aufgerufen am 25.06.2010

[Kecher 2007] KECHER, Christoph: *UML 2.0, Das umfassende Handbuch*. 1.Auflage. Galileo Computing, 2007. – ISBN 978-3-89842-738-8

[Object Managment Group 2007] OBJECT MANAGMENT GROUP: *UML 2.1.2 Specification*. 2007. – URL http://www.omg.org/spec/UML/2.1.2/. – Speziiell: http://www.omg.org/spec/UML/2.1.2/Superstructure/PDF/ und http://www.omg.org/spec/UML/20061001/Superstructure.cmof

[Selic 2003] SELIC, Bran: *The Pragmatics of Model-driven Development*. 2003. – URL http://www.dcs.shef.ac.uk/~ajhs/remodel/papers/SelicPragmatics.pdf

[Sun Oracle 2009] SUN ORACLE: *Primitive Data Types*. 2009. – URL http://java.sun.com/docs/books/tutorial/java/nutsandbolts/datatypes.html. – Zuletzt aufgerufen am 03.07.2010

[Ullenboomr 2009] ULLENBOOMR, Christian: *Java ist auch eine Insel*. 8.Auflage. Galileo Computing, 2009. – ISBN 3836213710

[Unbekannt 2003 –2008] UNBEKANNT: *What is PicoContainer ?* 2003 -2008. – URL http://www.picocontainer.org/. – Und alle Unterseiten, Zuletzt aufgerufen am 21.06.2010

[Unbekannt 2005] UNBEKANNT: *Why StaX ?* 2005. – URL http://java.sun.com/webservices/docs/1.6/tutorial/doc/SJSXP2.html. – Zuletzt aufgerufen am 28.06.2010

[Unbekannt 2007 –2010] UNBEKANNT: *Google Guice Wiki*. 2007 - 2010. – URL http://code.google.com/p/google-guice/wiki/Motivation?tm=6. – Und alle Unterseiten, Zuletzt aufgerufen am 17.06.2010

[Vanbrabant 2008] VANBRABANT, Robbie: *Google Guice*. 1.Auflage. Apress, 2008. – ISBN 1-59059-997-7

[Walls und Breidenbach 2008] WALLS, Craig ; BREIDENBACH, Ryan: *Spring in Action*. illustrated Edition. Manning, 2008. – ISBN 1-93239-435-4

Glossar

API Eine API (Application Programming Interface) ist eine Schnittstelle eines Systemes nach außen, um den Zugriff zu dem System zu ermöglichen.

Bad Smell Ein Bad Smell ist bestimmtes Muster im Quellcode, das auf tiefere Probleme hinweist. Bad Smells sollten durch Refactoring aufgelöst werden. Wird häufig auch als Code Smell bezeichnet.

Container Stellt auf Anfrage eine Implementierung eines ihm bekannten Objektes zur Verfügung. Ist in der Regel in der Zielsprache geschrieben.

Entwurfsmuster Bewährtes Lösungsmuster für oft auftretende Entwurfsprobleme in der Software Entwicklung

Framework Stellt einen Gerüst zur Entwicklung von Software Lösungen zur Verfügung, so dass der Anwender nur die Anwendungslogik hinzufügen muss.

JUnit JUnit ist ein Testframework von Kent Beck und Erich Gamma zum Durchführen von Komponententests auf Java Objekte.

Konfigurationsdatei Eine Datei in der die Konfiguration eines Software Systemes festgehalten wird. Die Datei ist oft im XML Format oder im Format der Zielsprache verfasst.

Mock Objekt MockObjekte sind Attrappen für ein echtes Objekt. Sie implementieren das Protokoll des echten Objektes, arbeiten aber mit Testdaten, deshalb werden sie hauptsächlich für Unit Tests verwendet

Programmierparadigma Ein grundlegender Programmierstil. Ein bestimmte Vorgehensweise um programm basierte Problemstellungen zu lösen.

Proxy Ein Proxy Objekt ist ein Stellvertreter Objekt, welches ein echtes Objekt mimt. Somit kann zum Beispiel eine Zugriffskontrolle auf das echte Objekt vorgenommen werden.

UML UML steht für Unified Modeling Language. UML ist eine nach ISO standadisierte Sprache zur Modellierung von Softwaresystemen

W3C Das W3C (World Wide Web Consortium) legt Standards für das Internet betreffende Technologien fest. Das W3C legte beispielsweise den HTML,XML oder SVG Standard fest.

XMI Die Abkürzung XMI steht für XML Metadata Interchange. XMI ist ein Standard der OMG (Object Managment Group) und dient zum Austausch von Objekten auf Basis des MOF Formates (wie z.B. UML Modellen) zwischen mehreren Werkzeugen

XML Die XML (eXtensible Markup Language) Sprache stellt Daten in einer maschinenlesbaren Form dar. XML Dokumente sollen den einfachen Austausch von Dokumenten zwischen mehreren verschiedenen Anwendungen durch Standardisierung ermöglichen.

Anhang A

Die beigelegte CD besteht aus 3 Verzeichnissen:

1. Bachelorarbeit
   In dem Bachelorarbeit Verzeichnis befindet sich die eigentliche Bachelorarbeit als PDF Datei.
2. Mobadi (Modelbased Dependency Injection Framework)
   Das Mobadiverzeichnis enthält:
   - Das Source Verzeichnis mit den Java Source Dateien (src Verzeichnis), den zugehörigen Class Dateien (bin Verzeichnis), der JavaDoc (JavaDoc Verzeichnis), sowie allen Dateien die zur Ausführung des Frameworks, wie der Template Datei benötigt werden.
   - Die Mobadi.jar zum Ausführen des Frameworks
   - Eine Anleitung zur Bedienung des Frameworks
3. Versand
   Das Versand Verzeichnis enthält:
   - Das Source Verzeichnis mit den Java Source Dateien (src Verzeichnis), den zugehörigen Class Dateien (bin Verzeichnis), der JavaDoc (JavaDoc Verzeichnis), sowie allen Bibliotheken die zur Ausführung des Versandsystems mit Spring, Guice und PicoContainer benötigt werden (lib Verzeichnis)
   - Das Diagramme Verzeichnis enthält die Enterprise Architect Projektdatei des Versandhauses und die XMI Dateien für die Releasekonfiguration (Versand.xmi), der Testkonfiguration mit Mockobjekten(VersandMock.xmi) und der Konfiguration für den Test des Frameworks (VersandZyklus.xmi).
   - Einer Anleitung zum Starten des Versandsystems

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §24(5) ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg,der 19.08.2010	
Ort, Datum	Unterschrift