Masterarbeit

Contribution Funktionalität

Techniken zur Servicekonfiguration für modulare Java Software durch erweitertes Spring IOC

Masterarbeit zur Erlangung des akademischen Grades
Diplomingenieur der Angewandten Informatik an der Paris Lodron
Universität Salzburg, Naturwissenschaftliche Fakultät,
Fachbereich Computerwissenschaften

Eingereicht von

Ortwin Probst

Betreuer

Ao.Univ.Prof. Mag.Dr. Helge Hagenauer

Salzburg, Jänner 2014

Diese Arbeit beschäftigt sich mit *Dependency Injection* als einer Methode zur Unterstützung von modularer Softwareentwicklung. Im Detail werden die Vorteile des erweiterten *Dependency Injection* Mechanismus aus dem Java-Frameworks *Tapestry* beleuchtet und als Inspiration für eine praktische Erweiterung des Java-Frameworks *Spring* herangezogen. Das praktische Resultat der Arbeit ist die konkrete Umsetzung dieser Erweiterung in einem eigenen Spring-Modul. Darin werden parallel zwei unterschiedliche Konfigurationsmechanismen mit äquivalentem Funktionsumfang angeboten, um die Anforderungen innerhalb des Spring-Frameworks zur Gänze erfüllen zu können. Beide Konfigurationsmechanismen werden, im Detail auf deren Unterschiede in der Implementierung, verglichen und in ihrer Anwendung gegenübergestellt.

Widmung

Ich widme diese Arbeit allen, die es mir ermöglicht haben in meinem Studium bis zu diesem Punkt zu kommen. Meinen Eltern für die Freiheit, die sie mir geschenkt haben. Meinem Betreuer Ao.Univ.Prof. Mag.Dr. Helge Hagenauer von der Universität Salzburg für seine sehr geschätzte Unterstützung und die hilfreiche Kritik. Meinem Kollegen und Betreuer Dipl. Ing. Christian Köberl auf Seiten der Porsche Informatik, für die Hilfe bei der Themenfindung, die Fachliche Unterstützung und die Entwicklung der Codebasis. Und meinem Studienkollegen Stefan Esterer für die Austauschmöglichkeit und die regelmäßige Portion Motivation.

Inhaltsverzeichnis

1 Einführung

Motivation

Meine Diplomarbeit fällt thematisch in den Bereich Modulare Softwareentwicklung für Java Enterprise Anwendungen. Im Detail stehen Konfigurationsmöglichkeiten für über Module verteilte Services (distributed configuration) zur Diskussion . Dabei beschäftige ich mich im Speziellen mit der sogenannten "Contribution" Funktionalität [4]. Dies ist eine Service Konfigurationsmöglichkeit des IOC-Containers, der das Kernstück des *Apache* Tapestry-Frameworks [1], einem Java Framework für Web-Anwendungen, bildet. Ziel ist es, den Contribution-Mechanismus auf das Spring-Framework, einem weiteren auf "Inversion Of Control" (IOC) basierenden Java Framework, zu übertragen.

Software Module werden im Folgenden nicht als Werkzeug zur Anwendungsmodellierung, sondern mehr als konkretes Mittel für die Programmierung mit einer Objektorientierten Programmiersprache betrachtet. Im Speziellen kann diese Art der Verwendung von Modulen auch als weiteres Mittel zur Kapselung (siehe Abschnitt 2.2) in der Objektorientierten Entwicklung betrachtet werden.

Theorieteil

Der theoretische Teil der Arbeit soll sich, ausgehend von einer Betrachtung zu modularem Softwaredesign im Bereich von Java Enterprise Anwendungen, genauer mit den zur Verfügung stehenden Techniken für Modulare Entwicklung auseinandersetzen. Dabei wird einerseits die klassische IOC Technik der *Dependency Injection*, sowie auch die Möglichkeit der Contributions und deren Nutzen erklärt und gezeigt. Im Detail wird dabei auf die IOC-Konfiguration der beiden Frameworks *Tapestry* und *Spring* eingegangen.

Praxisteil

Mein Ausgangspunkt für die praktische Arbeit ist das Interesse der Porsche Informatik, eine bestehende, modular entworfene und auf dem Apache Tapestry-Framework basierende Java Webapplikation, auf das in der Firma zum Standard erklärte IOC-Framework *Spring* umzustellen. Um einen hohen Grad an Modularität zu erreichen, wurde die Webapplikation ursprünglich auf dem Open Source Framework *Tapestry* aufgebaut. Diese Framework bietet einige Techniken die ein modulares Programmieren erleichtern.

Konkret bietet das Framework Methoden, die es ermöglichen, die Konfiguration eines Services aus verschiedenen, von einander unabhängigen Modulen heraus zu erstellen. Dadurch können die gesamte Anwendung und all ihre Services, nur durch das Einhängen

eines weiteren Moduls in den Classpath, um neue Funktionalität erweitert werden. Diese Technik wird in *Tapestry* mit "Contribution" bezeichnet.

Als anschauliches Beispiel kann man sich den Menü-Service einer Webanwendung vorstellen. Das Service dient dazu, automatisch ein Auswahlmenü über alle zur Verfügung stehenden Funktionen im Benutzerinterface bereit zu stellen. Dieses Service wird in *Tapestry* aus der zusammengeführten Servicekonfiguration aller Module aufgebaut. Das heißt jedes Modul "contributet" so zu sagen seinen Teil der Menü Service Konfiguration. Durch das Ein- oder Aushängen eines Moduls, wird die Anwendung dann automatisch um dessen Menüpunkte erweitert oder verringert.

Das Ziel ist es nun, zuerst die in *Tapestry* vorhandene Technik des Contribution Mechanismus für das Spring-Framework zu adaptieren und danach auf dieser Basis zusätzlich das Tapestry "Strategy Builder Service" ebenfalls für *Spring* bereit zu stellen. Das Ergebnis dieser Arbeit soll ein eigenes Java Modul mit dem Namen SpringContributions[1] sein, mit dem die bestehende Webapplikation so umgestellt werden kann, dass anstelle des Tapestry Contribution-Systems das neue SpringContributions System verwendet wird.

Da es in *Spring* zwei unterschiedliche Wege der Konfiguration gibt, ist es auch Ziel des Praxisteils, je eine Lösung für diese beiden unterschiedlichen Wege umzusetzen. Es wird ein Augenmerk darauf gelegt werden, im Bezug auf den Funktionsumfang und die Art der Anwendung, auf Gleichheit bzw. Ähnlichkeit der beiden Lösungen zu achten. Ebenso soll die Ähnlichkeit zum Vorbildsystem berücksichtigt werden und besonders auf Unterschiede, die sich aus der Implementierung oder konzeptionellen Abweichungen von *Tapestry* zu *Spring* ergeben, eingegangen werden. Probleme, die sich auf dem Weg zu diesen Zielen ergeben, sollen aufgezeigt und erläutert werden.

[1] Das SpringContributions Projekt wird auf Googlecode gehostet und ist unter folgender URL zu finden (10.12.2013): https://code.google.com/p/spring-contributions/

2 Modulare Programmierung

In allen hoch automatisierten Industriellen Fertigungsbereichen, ist der Ansatz der Modularisierung heute nicht mehr wegzudenken. Ein Paradebeispiel hierfür ist die Autoindustrie und ihr erfolgreicher Einsatz der Komponentenbauweise. Ohne die Komponentenbauweise, wäre die Nachfrage nach Automobilen wahrscheinlich schon seit längerem nicht mehr zu decken.

Seit vielen Jahren wird auch in der Informatik versucht, das Konzept der Modularisierung auch in der Software Entwicklung anzuwenden. Das Ziel war es auch hier, Softwarekomponenten zu bauen, die sich durch ihre Eigenschaften dazu eignen, in unterschiedlichsten Kompositionen als neue gesamte Anwendung miteinander zusammen zu arbeiten. Mittlerweile sind Softwarekomponenten in der Form von Frameworks sehr verbreitet und erfreuen sich auch großer Beliebtheit und Verwendung.

Liest man von Modularer Software Entwicklung, so ist der Blickwinkel meist der eines Anwendungsentwicklers. Hier hat man eine Sichtweise die, ausgehend von der Applikation als Große Einheit, versucht deren modularisierten inneren Aufbau zu betrachten. Ich verwende hier mit "Modularer Programmierung" absichtlich einen leicht anderen Begriff für den inhaltlich gleichen Bereich der Informatik, um auf den etwas anderen Fokus dieser Arbeit hinzuweisen. In beiden Fällen beschäftigen wir uns mit Software Modulen, wenn ich aber Modulare Programmierung sage, dann meine ich nicht den Blickwinkel des Anwendungsentwicklers, sondern den des Modulentwicklers. Diese Sichtweise beschäftigt sich in ihrer Betrachtung mit der kleinen Einheit des Moduls, und deren Zusammenspiel mit anderen Modulen. Der Fokus liegt also auf den Modulen und nicht auf der Applikation.

2.1 Software Modul bzw. Komponente

Der Vergleich mit Komponenten aus der Autoindustrie lässt vielleicht einen intuitiven Begriff davon entstehen, was eine Komponente bzw. ein Modul in der Softwareentwicklung sein kann. Um bei diesem Kernbegriff und Ausgangspunkt der Arbeit, nicht nur eine intuitives Verständnis vorauszusetzen, folgt hier eine Definition des Begriffes.

Definition (siehe [20]) Ein Software Modul ist eine nach außen hin über Schnittstellen definierte und gekapselte Einheit, die zur Kombination mit anderen Modulen vorgesehen ist. Die Kapselung bezieht sich dabei sowohl auf eine Zusammenfassung des Moduls in einer binären Einheit, als auch auf einen thematisch abgegrenzten Bereich. Die konkrete innere Struktur und Implementierung ist für den Benutzer irrelevant. Es kommt natürlich vor, dass ein Modul die zur Verfügung gestellte Funktionalität nur in Zusammenarbeit

mit anderen Modulen bewerkstelligen kann. Diese Abhängigkeiten sind jedoch nicht auf konkrete Implementierungen bezogen, sondern werden auch über Schnittstellen abgebildet. Das heißt ein Software Modul ist für sich genommen eine unabhängige Einheit. Die positiven Eigenschaften, die Softwaremodule (siehe Abschnitt 2.1.2) mit sich bringen, entstehen großteils aus diesem Aspekt der Unabhängigkeit von anderen konkreten Implementierungen. Da sich Module, wie sie im folgenden Kontext verwendet werden, in ihren Eigenschaften, nach Szyperskis Definition, nicht von Komponenten unterscheiden, werden im Folgenden beide Begriffe als Synonyme verwendet.

2.1.1 Service, Service Provider und Consumer

Definition

In dieser Arbeit wird der Begriff "Service" bzw. "Dienst" als Zusammenfassung für, über Schnittstellen definierte und zur Verfügung gestellte Funktionalität verwendet. In der Informatik wird der Begriff häufig auch im Zusammenhang mit Verteilten Systemen gebraucht und assoziiert eventuell einen verteilten Kontext. Deshalb möchte ich hier explizit darauf hinweisen, dass im Bereich dieser Arbeit ein Service keine Eigenschaften für eine verteilte Anwendung mit sich bringt.

Da der Kontext in dem wir uns hier befinden die objektorientierte Programmierung ist, verbirgt sich hinter einem Service im Grunde immer ein mehr oder weniger komplexes Objekt. Die Unterscheidung von einem Service zu einem normalen Objekt ist allerdings, dass die Funktionalität eines Service Objekts immer über ein Interface definiert sein muss.

Soll ein Service nicht nur innerhalb eines Moduls, sondern auch über Modulgrenzen hinweg, zur Verfügung gestellt werden, so muss natürlich die Interface Definition für andere Module sichtbar sein. Zur Darstellung der Verwendungsbeziehung eines Service, werden im Folgenden die Begriffe Service Provider (Service) und Service Consumer (Konsument) verwendet.

2.1.2 Positive Eigenschaften von Software Modulen

Ein Softwaresystem in einem Modularen Design zu entwerfen, ist ein gutes Mittel um Wiederverwendbarkeit (reusability), Austauschbarkeit und Erweiterbarkeit (plugability), erleichterte Wartung (maintainability) und Testbarkeit (testability) zu ermöglichen.

Aus den folgenden Vorteilen, die durch modulare Programmierung entstehen können, werden die Punkte 1-3 auch von David Parnas [15] hervorgehoben.

1. Verringerung der Entwicklungszeit, da Entwicklergruppen ohne viel Kommunikation getrennt an Modulen arbeiten können.

2. Steigerung der Flexibilität, durch die Möglichkeit, Änderungen in einem Modul vor zu nehmen, ohne dabei andere Module anpassen zu müssen.

3. Erleichterung der Verständlichkeit des Gesamtsystems, und somit auch die Unterstützung eines besseren Designs.

4. Erleichterung im Testen durch die Trennung von Interface, Implementierung und Konfiguration.

2.1.3 Modulentwicklung vs. Zusammenstellung modularer Anwendungen

Betrachtet man die modulare Softwareentwicklung, so muss man dabei zwei Bereiche klar voneinander abgrenzen. Zum einen die Entwicklung eines Softwaremoduls und zu anderen der Aufbau einer Modularen Anwendung.

Die Entwicklung eines Moduls ist die Aufgabe eines Modulentwicklers. Dieser ist zuständig für die interne Implementierung und die Definition der Modulschnittstellen.

Der Aufbau einer modularen Anwendung hingegen wird durch einen Anwendungsentwickler durchgeführt und beschäftigt sich nicht mehr mit der Implementierung, sondern mit der Auswahl, der Kombination und der Konfiguration mehrerer Komponenten zu einer bestimmten Anwendung.

2.2 Grundvoraussetzungen für Modularisierung

Auf Implementierungsebene gibt es mehrere Voraussetzungen, die zur Entwicklung von Komponenten notwendig sind. Dabei handelt es sich um die "Lose Kopplung", die "Kapselung", um "Schnittstellen" und das "Bereitstellen von Konfiguration". Diese vier Teilaspekte werden benötigt, will man eine Modularisierung von Software erreichen.

Lose Kopplung

In der klassischen Objektorientierung entstehen schon zum Zeitpunkt der Programmierung Abhängigkeiten zwischen Objekten. Benötigt zum Beispiel der Konstruktor bzw. die Initialisierungsmethode einer Klasse eine konkrete andere Klasse, so entsteht hier eine feste Kopplung [11] hin auf eine ganz bestimmte Implementierung. Für die Entwicklung von Services einer Komponente muss es aber möglich sein, Klassen zu schreiben, deren Laufzeitverhalten nicht schon zum Implementierungszeitpunkt durch ihre Abhängigkeiten festgelegt ist [6]. Es ist also nötig, die Implementierung auf Schnittstellendefinitionen aufzubauen, um die Funktionalität zwar zu definieren, aber von einer konkreten Implementierung unabhängig zu halten. So kann die Voraussetzung geschaffen werden, damit Komponenten erst zum Zeitpunkt der Softwareverteilung (deployment), also zum Installationszeitpunkt der Anwendung, oder sogar erst zum Ausführungszeitpunkt, eine konkrete Verbindung untereinander aufbauen. So ist es auch für eine Anwendungsentwickler, der keinen Einfluss auf die Implementierung einzelner Komponenten hat, möglich mehrere Softwarekomponente zu einer Anwendung zu kombinieren. Wie die lose Kopplung auch zu einer Erleichterung beim Testen von Komponenten führt wird in Abschnitt 2.3.5 beschrieben.

Kapselung

Um eine Lose Kopplung zu ermöglichen, benötigt man Mittel zur Kapselung von Programmeinheiten. In der objektorientierten Programmierung ist der Begriff Kapselung für das Zusammenfassen von Funktionalität, in von außen nur über Schnittstellen definierte und angreifbare Einheiten, definiert [18]. Wenn ich hier im Zusammenhang mit der modularen Programmierung von Kapselung spreche, so meine ich die gleiche Definition. Im modularen Kontext verschiebt sich nur die Betrachtungsebene der Kapselung von der Objektebene auf die des Moduls. Module kapseln also ihre Funktionalität in sich und bilden somit eine zusammengehörige, nur durch Interfaces nach außen definierte und angreifbare Einheit.

Schnittstellen

Betrachtet man die Lose Kopplung und die Kapselung als Grundvoraussetzung für Modularität, so folgt daraus die Möglichkeit von Schnittstellendefinitionen als weitere Grundvoraussetzung. Die Möglichkeit von Schnittstellen bzw. Interfaces ist auf Klassenebene prinzipiell in allen Objektorientierten Sprachen gegeben. Mit diesen ersten drei Grundvoraussetzungen ist es in der objektorientierten Programmierwelt möglich, Softwaremodule zu entwickeln.

Bereitstellung und Konfiguration

Sind die nötigen Schnittstellen einmal definiert und sind die dazugehörigen Modulimplementierungen vorhanden, so ist das nächste zu lösende Problem die Bereitstellung der Services für andere Module. Es muss also ein Weg gefunden werden, eine bestimmte Interface Implementierung überall dort zur Laufzeit zur Verfügung zu stellen, wo das Interface verwendet wird.

Dieses Problem kann durch Konfigurationsmechanismen gelöst werden. Man benötigt also eine von den Modulimplementierungen losgelöste Konfiguration, in der im einfachsten, aber nicht unbedingt elegantesten Fall, direkt alle Service Instanzen modulübergreifend, für die gesamte Anwendung, angelegt und initialisiert werden. Diese Konfiguration bildet dann zusammen mit allen zugehörigen Modulen eine Anwendung.

Es gibt verschiedenste Ansätze zur Realisierung dieser eigenständigen Konfiguration. Einige davon werden im Abschnitt 3.2.1 noch genauer beleuchtet werden. Alle Ansätze haben dabei aber die Gemeinsamkeit, dass sie sich auf einen zentralen Dienst stützen. Dieser Dienst bietet zum einen eine Möglichkeit zur Serviceregistrierung, mittels der jedes Modul die Zuordnung von Service Interface zur moduleigenen Implementierung, selbst als Konfiguration durchführen kann. Zum anderen stellt er eine Möglichkeit dar, die registrierten Services abzurufen und somit zu verwenden. Dienste die diesen Zweck erfüllen werden oft mit dem Begriff "*Service Registry*" bezeichnet.

Je nach Flexibilitätsgrad ergeben sich unterschiedliche Zeitpunkte, zu denen die Anwendungskonfiguration zur Verwendung kommt. Beginnend bei der einfachsten Lösung, über direkte programmatische Instanziierung und Initialisierung, die also schon zur Kompilierzeit die Anwendungszusammenstellung genau festlegt. Die konkrete Serviceland-

schaft der Anwendung erst bei der Anwendungsinstallation festzulegen, kann zum Beispiel über interne, also sich innerhalb des physischen Anwendungspakets befindlichen, Konfigurationsfiles oder Klassen, im Zusammenspiel mit Dynamischen Assemblerdiensten (siehe Abschnitt 2.3.2) gelöst werden. Liegen die Konfigurationsfiles oder der Konfigurationscode außerhalb des physischen Anwendungspakets, so verschiebt sich der Zeitpunkt der Auflösung auf den Systemstart.

Es gibt sogar Ansätze zur Konfiguration von modularen Anwendungen, die eine Einflussnahme auf Modulabhängigkeiten und Service Konfigurationen noch zur Laufzeit ermöglichen. Ein Komponentensoftwaresystem zur Laufzeit anzupassen, d.h. neue Komponenten einzufügen, andere zu entfernen und gewisse Verbindungen umzustellen, ist eine Eigenschaft die wegen des Aufwands nicht alle komponentenorientierten Entwicklungsansätze verfolgen. Als Beispiel für die Anpassbarkeit zu Laufzeit kann hier *OSGI*, als Spezifikation einer Modul und Serviceplattform für Java, erwähnt werden. *OSGI* ist unter anderem im später noch genauer beschriebenen Spring-Framework (siehe Abschnitt 3.1.1) umgesetzte. Die Möglichkeit der Anwendungskonfiguration zur Laufzeit, soll hier aber nur der Vollständigkeit wegen erwähnt sein und wird im Weiteren nicht mehr genauer behandelt.

2.3 Modularisierung mit "*Inversion of Control*"

Moderne Java Frameworks zur Unterstützung von modularer Softwareentwicklung bieten oftmals IOC-Container an. Diese Container sind spezielle Formen von Serviceregistries (siehe Abschnitt 2.2) mit deren Hilfe eine Entkopplung von Consumer und Provider realisiert werden kann.

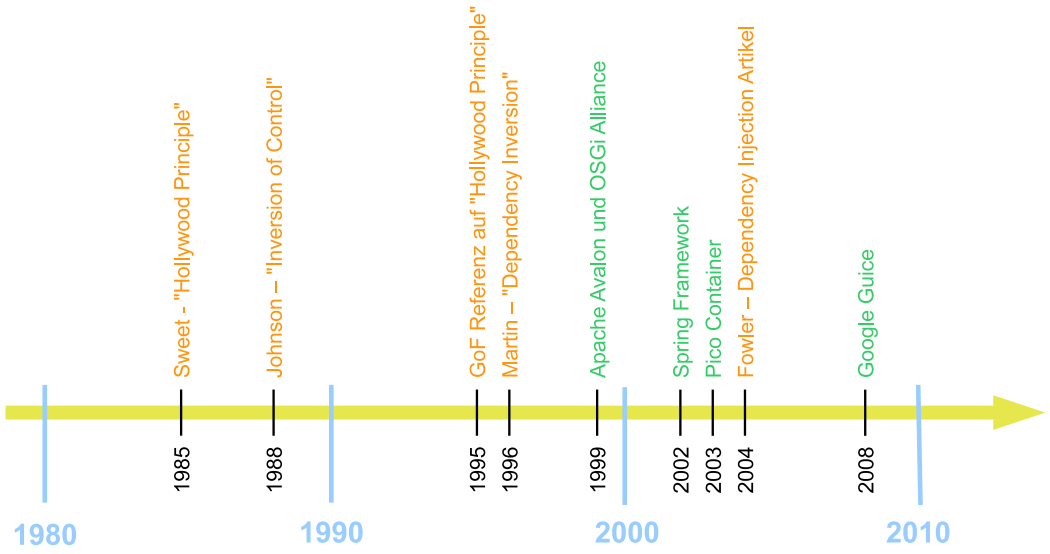
2.3.1 "*Inversion of Control*"

„Inversion of Control“ kurz IOC bezeichnet ganz allgemein die Umkehrung eines Kontrollflusses. Kontrollflussumkehrung ist aber ein sehr allgemeines Prinzip und wird somit auch in verschiedensten Ausprägungen in der Softwareentwicklung verwendet. Die für den aktuellen Kontext wichtige Ausprägungsform von IOC als "*Dependency Injection*" wird im nächsten Abschnitt 2.3.2 genauer beschrieben. IOC und das *Dependency Injection Pattern* sind Methoden deren Anwendung meist direkt auf Objektebene verwendet wird. Beide Prinzipien können aber ohne weiteres auch auf der Komponentenebene ihre Anwendung finden.

Geschichte: Martin Fowler [10] liefert eine eine Erklärung und Informationen zur Herkunft des Begriffs "*Inversion of Control*" in der Informatik. So findet sich bei Ralph E. Johnson [17] eine frühe Verwendung des Begriffes, wenn die Autoren selbst, wieder auf einen anderen, nicht bekannten Schöpfer des Begriffes verweisen. Mit dem "*Hollywood Principle*" [19] existierte schon seit 1985 ein weiterer Begriff für dasselbe Prinzip auf den auch Erich Gamma et.al. in ihrem bekannten Buch zu "*Design Patterns*" [11] Bezug nehmen "Don'tcall us, we'll call you".

Das *Pico Container* Framework (siehe Abschnitt 2.4) liefert auf seiner Homepage [2] weitere Erläuterungen zur Geschichte des IOC Begriffs. Als Wegbereiter des heutigen IOC Begriffs werden hier Robert C. Martin, der unter der Bezeichnung "*Dependency Inversion*" [13] über die Umkehrung von Abhängigkeiten zur Erreichung von unabhängigen Modulen schreibt, Michael Mattesson [14], Brian Foote und Joseph Yoder [7], sowie Ralph E. Johnson und Brian Foote [12] erwähnt. Weiters finden hier die ersten Umsetzungen des IOC Prinzips mit den Projekten *Apache Avalon* und *OSGi* ihre Erwähnung. Folgende Grafik soll die Historie von Fowler und die Timeline von *Pico Container* zusammenführen und ergänzen.

Abbildung 2.1: Meilensteine des IOC-Pattern



Bedeutung: Die allgemeine Bedeutung von IOC als Kontrollfluss Umkehr beschreibt Fowler [9] am Beispiel vom Wechsel von einer Kommandozeilen gesteuerten Anwendung, die den Benutzer Schritt für Schritt nach Eingaben fragt, hin zu einer Steuerung über eine Benutzeroberfläche, bei der die Benutzeroberfläche Benutzereingaben entgegen nimmt und diese an die Anwendung weiterleitet. Hier wurde also der ursprüngliche Kontrollfluss umgekehrt.

Der IOC Begriff wird aber nicht immer so allgemein betrachtet. Das Pico Container Framework zum Beispiel verwendet den Begriff als Zusammenfassung ganz konkreter Techniken.

Das allgemeine IOC Prinzip kann als eine Haupteigenschaft von Frameworks betrachtet werden. Frameworks haben im Gegensatz zu einfachen Programmbibliotheken die Kontrolle über ihre Funktionalität großteils selbst inne. Eine Programmbibliothek stellt lediglich Funktionalität für einen Konsumenten zur Verfügung, die Kontrolle über die

Funktion geht hier aber vom Konsumenten aus. Ein Framework stellt zwar auch Funktionalität zur Verfügung, im Gegensatz zur Programmbibliothek wird hier aber mit verschiedenen Methoden (Subclassing, Plugins) das eigene Programmverhalten in das Framework eingebunden. Das Verhalten des Frameworks wird so zu sagen für die eigenen Anwendungsbedürfnisse konfiguriert.

Dieses Prinzip der externen Konfiguration des Programmverhaltens und der gleichzeitigen Beibehaltung der eigenen Kontrolle des Frameworks über die Funktionen an sich, kann nicht nur auf Frameworks, sondern auch auf Module angewandt werden. Ein Modul, wie zum Beispiel ein Verrechnungsmodul, kapselt einen bestimmten Fachlichen Aspekt und die dazu bereitgestellte Funktionalität (Rechnungserstellung, Stornierung, etc.). Das konkrete Verhalten im jeweiligen Anwendungskontext des Moduls, muss jedoch von eben diesem Anwendungskontext als Konfiguration bereitgestellt werden. Das heißt die Services eines Moduls werden von außen konfiguriert, behalten selbst aber die Kontrolle über ihre primäre Funktionalität. Um derartige Services realisieren und in einem Modul kapseln zu können, muss es möglich sein die Funktionalität eines Services von dessen konkreten Konfiguration zu entkoppeln. Dieses Entkoppeln, als die Möglichkeit zum Auflösen von konkreten Abhängigkeiten, ist also eine Hauptvoraussetzung für Modulares Programmieren.

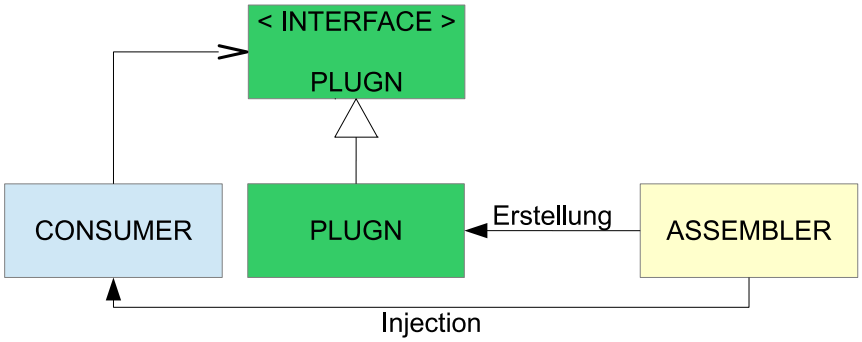
In der Objektorientierten Programmierwelt ist IOC eine der wichtigsten Techniken, um eben diese Entkopplung zu erreichen. Auch wenn hier im Allgemeinen oft von IOC gesprochen wird, so ist im Konkreten oft *Dependency Injection*, eine ganz spezielle Form von IOC gemeint.

2.3.2 "Dependency Injection"

"*Dependency Injection*" (DI) ist ein von Martin Fowler [9] genauer beschriebenes Pattern für die objektorientierte Programmierung. Mit Hilfe des Patterns wird es möglich, einen sogenannten Plugin Mechanismus zu realisieren. Das heißt ein Consumer muss nur den Interface Typ des benötigten Services kennen, die konkrete Service Implementierung wird durch einen Assembler von außen als Plugin in den Consumer injiziert oder eingehängt (siehe Abbildung 2.2). In Java gibt es für den Vorgang der Injection mehrere unterschiedliche Ansätze, die je nach Anwendungsfall vor und Nachteile haben können. In der folgenden Auflistung können die ersten drei als die klassischen Wege für *Dependency Injection* betrachtet werden. Die Annotation Injection ist eine eher neuere Ausprägung des *Injection Pattern*.

Konstruktor-Injection Bei dieser Form von Injektion werden alle nötigen Plugins über den Konstruktor des Consumer injiziert. Alle Plugins werden hier also schon bei der Erstellung des Consumer benötigt. Daraus ergeben sich in der Praxis oftmals Probleme (z.B. Abhängigkeits-Zyklus), für deren Lösung moderne IOC-Frameworks verschiedene Wege (z.B. Lazy Loading, Proxying) anbieten. Für den Assembler ist es hier wichtig einen bestimmten Konstruktor für den Injizierungsvorgang zur Verfügung zu haben. Die Auswahl des Konstruktors ist hier ein wichtiges zu lösendes Problem und kann auf unterschiedliche Arten erfolgen (siehe Absatz 2.3.2).

Abbildung 2.2: *Dependency Injection Pattern*



Setter-Injection Wie auch hier schon der Name sagt, werden bei dieser Art der DI, Plugins über, vom Consumer selbst zur Verfügung gestellte Setter, in den Consumer eingehängt. Man benötigt hier für jedes Plugin einen eigenen Setter. Ein besonderer Vorzug der Setter Injection gegenüber der Konstruktor Injection ist hier, dass der Assembler den Zeitpunkt der Injection beliebig nach der Objekterstellung wählen kann.

Interface-Injection Anstelle von einem Setter pro Plugin, gibt es bei der Interface-Injection für jede Instanzvariable eine über ein Injection Interface definierte Inject Methode zum setzen des jeweiligen Plugins. Das Injection Interface wird immer zusammen mit dem Plugininterface angeboten. Jeder Consumer, der Plugins vom Typ des Plugininterfaces verwenden möchte, muss gleichzeitig auch das Injection Interface für diesen Plugintyp implementieren, wenn er Plugins injiziert bekommen möchte. Somit stehen dem Assembler die Inject Methode für das Plugin zur Verfügung.

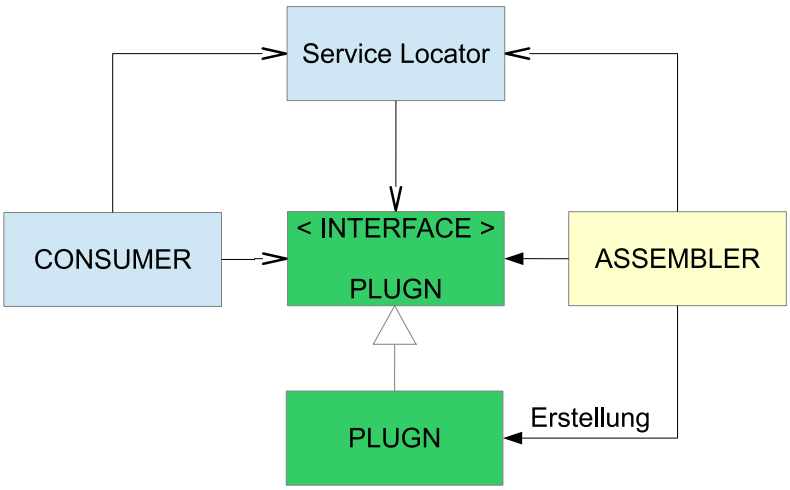
Annotation-Injection Nachdem im Java Release 5.0 Annotationen eingeführt worden waren, wurde dieses neue Feature sogleich von verschiedensten IOC-Frameworks verwendet, um eine für den Entwickler sehr komfortable, neue Art der DI zu realisieren. Dabei dienen nun Annotationen wie `@Autowired` (*Spring* IOC) bzw. `@Inject` (*JSR 330*) als Marker für das IOC-Framework, mit denen der Entwickler Instanzvariablen für eine durchzuführende Injection kennzeichnen kann. Wichtig ist hier aber, dass die Instanzvariable eben nicht über einen expliziten Setter oder einen Konstruktor injiziert wird. Der Assembler injiziert die Instanzvariable direkt über Java Reflection in das Objekt.
Natürlich kann und wird die Markierfunktion von Annotationen auch in Zusammenhang mit allen anderen DI Arten benutzt. In diesen Fällen werden die Annotationen aber lediglich dazu verwendet, um Mehrdeutigkeiten für eine Injection Methode aufzulösen. Für den Fall, dass mehrere Klassen vom selben Typ existieren der injiziert werden soll, bieten viele IOC-Frameworks die Möglichkeit an, die Klasse über Injektion-Annotationen und deren Parameter genauer zu spezifizieren, so dass die zu injizierende Klasse wieder eindeutig festzustellen ist. Eine weitere Markierfunktion wird oft angeboten, um einen

bestimmten Konstruktor für die Konstruktor Injection zu kennzeichnen oder, um bestimmte Methoden als Setter für die Setter-Injektion zu markieren.

2.3.3 "Service Locator"

Eine Alternative zur Entkopplung von Klassen mittels *Dependency Injection* ist die Verwendung des "*Service Locator Pattern*". Diese Methode stützt sich auf die zentrale Klasse des Service Locator, der Methoden anbietet, über die man Objekte von einem gewünschten Typ erhält. Anstatt einer Klasse bei ihrer Erstellung nun all ihre Abhängigkeiten mit zu geben, gibt man diese Klasse initial nur den Service Locator mit. Die Kontrolle über das Instanziieren von Instanzvariablen obliegt bei diesem Pattern nun wieder dem Objekt selbst. Benötigt das Objekt nun ein anderes Objekt, so kann es dieses über den Service Locator bekommen. Hier kommt es also nicht zu einer Kontrollfluss Umkehrung wie bei der zuvor beschriebenen DI.

Abbildung 2.3: *Service Locator Pattern*



2.3.4 Plugin Konfiguration

Egal welche Art der DI verwendet wurde um einen Consumer für Plugins vorzubereiten, man benötigt zudem eine Plugin Konfiguration, wenn man den Code in einem Programm zur Laufzeit nutzen möchte. Unabhängig vom Typ der DI ist also diese Konfiguration essentiell, denn darin wird festgelegt, welche Plugins überhaupt zum konsumieren zur Verfügung stehen und welches Plugin von welchem Consumer verwendet werden soll. Natürlich können nur genau die Plugins von einem Consumer verwendet werden, für deren Verwendung er auch über die Angabe des Interfaces vorbereitet ist. Solange es nur ein konfiguriertes Plugin zu einem Interface gibt, solange könnte auch die Injection, also die Referenzierung von Consumer zu Plugin, durch den Assembler ganz automatisch durchgeführt werden. Da es aber vorkommen kann, dass man mehrere Plugins des

selben Interfacetyps in einer Konfiguration angelegt hat, ist es dann auch notwendig, die Beziehung zwischen den Service Consumern und den Plugins explizit in der Konfiguration abzulegen. Die Verwendung einzelne Plugins von mehreren Consumern kann natürlich ebenfalls gewünscht sein und wird, soweit dies unterstützt ist, ebenfalls in der Konfiguration festgelegt. Bei der Konfiguration müssen natürlich so manche Probleme berücksichtigt werden. Verwendet man zum Beispiel innerhalb eines Plugins ebenfalls DI für die Verwaltung der Abhängigkeiten, so ist es natürlich möglich Abhängigkeits-Zyklen zu produzieren, indem die Konfiguration so aufgebaut wird, dass innerhalb der Referenzen eine Abhängigkeit von einem Plugin zu seinem eigenen Consumer entsteht und somit bei der Auflösung der Abhängigkeiten ein Deadlock entstehen würde. Je nach der "Intelligenz" des Assemblers, gibt es für derartige Konfigurationsproblematiken auch "intelligente", automatische Problemauflösungs- bzw. Vermeidungsstrategien. Unabhängig von solchen automatischen Hilfestellungen, ist es die Konfiguration selbst, in der derartige Probleme entstehen aber eben auch vermieden werden können.

2.3.5 Testability durch IOC

Die Verwendung von IOC zur Regelung der Abhängigkeiten von Objekten, hat einen ungemein positiven Effekt auf die Testbarkeit der Objektlogik. Es spielt dabei keine Rolle, auf welche genaue Weise die DI umgesetzt wird, wichtig ist einzig die Verwendung des IOC-Pattern. Werden zum Beispiel die Abhängigkeiten in einem Service durch diesen selbst geregelt, so hält das Service so zu sagen die eigene, Konfiguration unveränderbar in sich verborgen. Möchte man das Service nun in einem Unit-Test auf seine korrekte Funktionsweise testen, so ist man gezwungen, nicht nur die Einheit (Unit), also das Service selbst, sondern auch alle Abhängigkeiten des Service und deren Verhalten mit zu berücksichtigen. Über das Verhalten dieser Abhängigkeiten hat man im Test aber keinerlei Kontrolle.

Wenn das Service so geschrieben ist, dass es seine Konfiguration durch IOC von außen übergeben bekommt, so kann man dies in einem Unit-Test für eine spezielle Testkonfiguration nutzen. Es wird also möglich, zu jeder Abhängigkeit des Service sogenannte Mock-Objekte zu schreiben und diese dann für den Test zu übergeben (siehe [9]). Diese Mock-Objekte sind spezielle Implementierungen mit denen man die vom Service benötigten Abhängigkeiten ersetzen kann und, wie Dave Thomas und Andy Hunte zusammenfassen [21], über deren Verhalten man nun die volle Kontrolle hat. Es ist dann also zum Beispiel möglich, in einem Mock-Objekt genau festzulegen, welchen Rückgabewert dieses liefern wird, wenn das getestete Service eine Methode auf dem Mock aufruft. Dadurch ist nun also das Service als unabhängige Einheit testbar.

2.4 IOC-Frameworks für modulares Entwickeln

Die in Abschnitt 2.3.2 beschriebenen Varianten des "*Inversion of Control Pattern*" sind heute Kernstücke unterschiedlicher IOC-Frameworks (siehe Listing 2.1)

Tabelle 2.1: IOC-Frameworks

Framework	IOC-Varianten	Konfiguration
Spring	Konstruktor, Setter, Annotation, Service Locator	statisch (XML, Annotationen) programmatisch
Tapestry (Apache)	Konstruktor, Setter, Annotation, Service Locator	statisch (Annotationen) programmatisch
Guice (Google)	Konstruktor, Setter, Annotation	Annotationen
Java-CDI (Oracle)	Konstruktor in Kombination mit Annotation, Annotation	statisch (Annotationen, XML)
Silk DI	Konstruktor	programmatisch
Pico-Container	Konstruktor, Setter, Annotation	statisch (Annotationen) programmatisch

Durch die Eigenschaft der Entkopplungsmöglichkeit die diese Frameworks bieten, können sie sehr hilfreich als Basis für eine modulare Softwareentwicklung eingesetzt werden. In der Praxis setzt mittlerweile ein Großteil der namhaften Softwareentwickler IOC-Frameworks zur leichteren Modularisierung ihrer Software ein. Mit dem selbst entwickelten IOC-Framework *Google Guice* setzt z.B. auch der Suchmaschinen Anbieter ganz stark auf IOC. Neben *Guice* gibt es weiter IOC-Frameworks wie *Pico Container*, *Silk DI*, *Apache Tapestry* oder *Oracle CDI*. Manche dieser IOC-Frameworks sind sehr spezialisiert. So ist z.B. das Tapestry-Framework rein auf die Entwicklung von Webanwendungen ausgerichtet.

Der wahrscheinlich bekannteste Vertreter aus der Gruppe der IOC-Frameworks ist unter dem Namen "*Spring*" bekannt. Mit *Spring* ist die Zusammenfassung einer Vielzahl von Modulen gemeint, deren Einsatzbereich sehr breit gestreut ist. Einen Überblick über die Modularität und die Vielfalt der Module kann unter [5] gefunden werden. Bei aller Vielfalt der durch *Spring* zur Verfügung gestellten Funktionalität, bleibt aber der IOC-Container mit seiner *Dependency Injection* Funktionalität das Herzstück von *Spring*. Dieses Herzstück ermöglicht zum einen den eigenen modularen Aufbau des Spring-Frameworks, kann aber auch für die Entwicklung von eigenständiger modularer Java-Software eingesetzt werden.

3 IOC Konfiguration

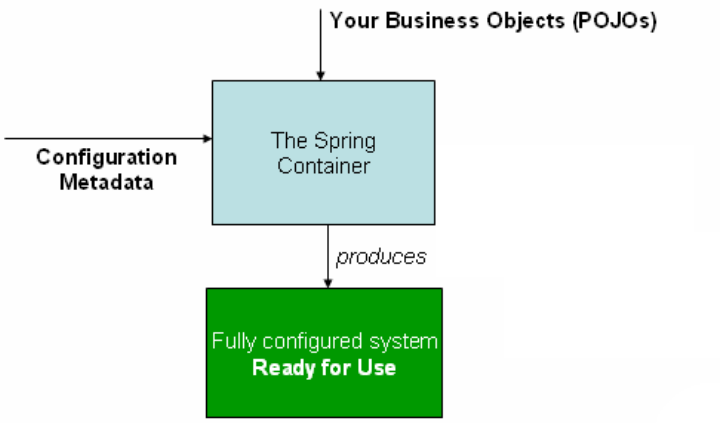
Wie bereits im Abschnitt 2.3.4 erklärt wurde, muss man bei der Verwendung des *Dependency Injection Pattern* immer auch eine konkrete Konfiguration für die zur Verfügung stehenden und benötigten Plugins bereit stellen. Wie diese Konfiguration für bestimmte IOC-Container konkret aussieht soll im Folgenden beleuchtet werden.

3.1 *Spring*

Spring[1] ist ein mittlerweile sehr weit verbreitetes, und äußerst umfangreiches Java Framework. Ungeachtet der zahlreichen Erweiterungen die das Framework seit seinem ersten Release erfahren hat, ist der Kern des Spring-Frameworks immer noch dessen IOC-Container. Die zentrale Rolle des IOC-Containers führte auch dazu, dass im laufe der Framework Entwicklung mehrere verschiedene Arten zur IOC-Konfiguration entstanden sind und aus Gründen der Flexibilität parallel angeboten werden.

Das Java Interface *org.springframework.beans.factory.BeanFactory* repräsentiert den *Spring* IOC-Container. Über dieses Interface werden alle notwendigen Aktionen abgedeckt. Eine *FactoryBean* Implementierung kann also zum Registrieren von Objekten (Plugins), zum Instanziieren von registrierten Objekten, sowie zur Aufbewahrung und Rückgabe dieser Instanzen verwendet werden. Darüber hinaus übernimmt eine *BeanFactory* die Aufgabe des Assemblers aus dem DI-Pattern.

Abbildung 3.1: *Spring* IOC Konfiguration [3]



[1] Wenn nicht anders angegeben, beziehen sich im gesamten Text Referenzen zum Spring-Framework auf *Spring* in der Releaseversion 3.1.

Eine Klasse deren Lebenszyklus durch den *Spring* IOC-Container verwaltet wird, nennt man ein "Bean". Innerhalb einer *Spring* Anwendung existieren sowohl Beans als auch ganz normale Java-Klassen nebeneinander. Beans sind ebenso ganz normale Java-Klassen, nur eben mit dem Zusatz, dass sie durch den IOC-Container verwaltet werden. Um deren Verwaltung überhaupt durchführen zu können, benötigt der IOC-Container eine Konfiguration für die Menge der Beans. Diese Konfiguration wird als "Configuration Metadata" bezeichnet und enthält Metadaten, die Aussage darüber geben welche Beans überhaupt in einer Anwendung zur Verfügung stehen und welche Abhängigkeiten diese Beans untereinander haben. *Spring* bietet verschiedenste Implementierungen des *FactoryBean* Interfaces an. Dabei besteht unter anderem die Möglichkeit, die Configuration Metadata direkt über Javacode, über Annotationen mit "Component scanning", über Java-Konfigurationsklassen, oder über XML-Konfigurationsfiles, die aus der historischen Entwicklung heraus in *Spring* als Standardkonfigurationsmethode gelten, festzulegen. Abbildung 3.1 gibt einen Überblick zur Verwendung des *Spring* IOC-Containers.

3.1.1 Konfiguration

Dieser Abschnitt gibt eine kurze Einführung in die beiden Möglichkeiten der Konfiguration (XML-Konfiguration und Java-Konfiguration)[2], die in *Spring* zu Verfügung stehen. Die beiden Listings (3.1, 3.2) zeigen die einfache Definition des Service *MyServiceImpl* als Bean. In beiden Varianten wird dasselbe Bean konfiguriert. Um die jeweilige Konfiguration zur Anwendung zu bringen, muss sie nur dem *Spring* IOC-Container übergeben werden.

XML-Variante der *Spring* Konfiguration

Ein *Spring* XML-Konfiguration besteht aus mindestens einer XML-Datei. Innerhalb dieser XML-Datei können beliebig viele Beandefinitionen konfiguriert werden. Diese sind durch den XML-Tag **<bean>** gekennzeichnet. Alle Beandefinitionen werden innerhalb des Wurzelelements **<beans>** angelegt. Die Beankonfiguration benötigt dabei mindestens zwei Angaben. Zum einen muss im XML-Attribut "class" der voll qualifizierte Name der Java-Klasse und zum anderen im XML-Attribut "id" eine eindeutige Identifikation für das Bean festgelegt werden.

[2] Die beiden Konfigurationsmethoden werden in der gesamten Arbeit abwechselnd behandelt. Um in Überschriften anschaulich zu kennzeichnen welche Methode gerade gemeint ist, wird der Überschrift entweder (XML) bzw (Java) vorangestellt.

Listing 3.1: *Spring* Beandefinition in XML-Datei

```
<?xml version="1.0" encoding="UTF-8" ?>
    <beans
        xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation=
            "http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/
            spring-beans-3.0.xsd">

        <bean id="MyBeanId" class="MyServiceImpl">
            <!-- weitere Metadaten zum jeweiligen Bean -->
        </bean>

    </beans>
</xml>
```

Java-Variante der Spring Konfiguration

Eine Spring Java-Konfiguration besteht aus mindestens einer Java-Klasse, die mit der Annotation `@Configuration` über dem Klassennamen als Spring-Konfiguration markiert ist. Um nun in dieser Java-Klasse ein Bean zu definieren, wird eine mit der Annotation `@Bean` gekennzeichnete Methode benötigt, die eine Instanz der gewünschten Bean-Klasse zurück gibt. Der Name des Bean kann innerhalb der `@Bean` Annotation über den Parameter "**name**" angegeben werden. Wird der Name nicht explizit definiert, so wird der Methodenname als Name für das Bean herangezogen.

Listing 3.2: *Spring* Beandefinition in Java-Klasse

```
import org.springframework.context.annotation.Configuration;

@Configuration
public class MyModulConfiguration
{
    @Bean
    public static MyServiceInterface myBeanId()
    {
        return new MyServiceImpl();
    }
}
```

Im folgenden Kapitel wird das Tapestry-Framework beschrieben werden. Dieses diente als Inspiration und Vorbild für die SpringContributions. Dabei kann die gerade beschriebene *Spring* Java-Konfiguration, durch den Umstand, dass diese ebenfalls in Java-Klassen durchgeführt wird, wohl etwas einfacher mit der Tapestry-Variante verglichen werden. Dasselbe gilt auch für alle späteren Abschnitte.

3.2 *Tapestry*

Apache Tapestry [3] ist ein Framework zur Entwicklung von Webanwendungen mit Java. *Tapestry* unterstützt dabei auch besonders den modularen Aufbau einer Webanwendung. Auch hier bildet ein Framework eigener IOC-Container zusammen mit eine Reihe von Services das Kernstück des Frameworks. Darüber wird die Konfiguration der Module und ihrer Dienste ermöglicht. Durch die Verwendung eines IOC-Containers findet die *Dependency Injection*, in ihren verschiedensten Ausprägungen (siehe Abschnitt 2.3.2), eine umfassende Verwendung innerhalb aller Teile eines Tapestry-Moduls.

Ein mit *Tapestry* entwickeltes Modul wird als Java Archiv (.jar) gepackt und lässt sich in dieser Form mit anderen Tapestry-Modulen kombinieren. Innerhalb des Archivs wird ein Tapestry-Modul durch eine Modul-Klasse repräsentiert. Diese Klasse stellt dem Framework Informationen über die im Modul vorhandenen Services zur Verfügung. Der Inhalt der Modul Klasse entspricht hier dem der Konfiguration-Metadaten des Spring-Frameworks (siehe Abschnitt 3.1.1).

3.2.1 Konfiguration

Die Konfiguration eines Tapestry-Moduls besteht aus reinem Javacode. *Tapestry* bietet für das Festlegen der Modulkonfiguration innerhalb der Modul-Klasse eigene Namenskonventionen und Annotationen an. Das Framework erkennt dann zum Beispiel am Anfang eines Methodennamen, dass es sich hier um eine Servicedefinition handelt. In der Modul-Klasse stehen mehrere Konfigurationsmöglichkeiten zur Verfügung. Folgende vier Arten zeigen einen Großteil der Möglichkeiten, wobei wir uns in der Folge nur noch mit den ersten beiden beschäftigen werden.

- "Bind": Das Definieren eines Services, durch das Zuweisen einer Service Implementierung zu dem zugehörigen Service Interface
- "Contribute": Das Bereitstellen von Service Konfigurationsdaten
- "Build": Das Definieren und Bauen eines Services durch expliziten Javacode
- "Decorate": Das Dekorieren eines Services, wie es aus dem "*Decorator Pattern*" [11] bekannt ist

[3] Wenn nicht anders angegeben, beziehen sich im gesamten Text Referenzen zum Tapestry-Framework auf *Tapestry* in der Releaseversion 5.0 auch bezeichnet als *Tapestry 5*.

Eine Servicedefinition in *Tapestry* entspricht in etwa einer Beandefinition in der Spring-Konfiguration. In beiden Fällen handelt es sich um Konfigurationsdaten die dem jeweiligen IOC-Container die nötigen Informationen geben, um dem System eine, über ein Interface definierte Funktionalität, bereitstellen zu können.

In *Tapestry* gibt es verschiedenen Möglichkeiten, ein Service zu definieren. Im Allgemeinen existiert für ein Tapestry-Service ein Interface und mindestens eine zugehörige Implementierung. Um das Service dem System bekannt zu geben wird die *"bind"* Methode in Zusammenspiel mit dem *org.apache.tapestry5.ioc.ServiceBinder* verwendet. Über den *ServiceBinder* wird das Framework informiert, welche Serviceimplementierung für ein Service Interface zur Verwendung hinterlegt werden soll (siehe Listing 3.3).

Listing 3.3: *Tapestry*: Registrierung eines Dienstes "binding"

```
public class MyModulConfiguration
{
    public static void bind(ServiceBinder binder)
    {
        binder.bind(MyServiceInterface.class, MyServiceImpl.class);
    }
}
```

Eine weitere Möglichkeit der Service Definition passiert über eine sogenannte *"build"* Methode, die mit dem Service Interface als Rückgabewert definiert wird. Bei dieser Art der Servicedefinition können auch gleich Setupmethoden auf dem Serviceobjekt durchgeführt werden (siehe Listing 3.4).

Listing 3.4: *Tapestry*: Definition einer Serviceimplementierung innerhalb der Modulklasse

```
public class MyModulConfiguration
{
    public static MyServiceInterface build()
    {
        return new MyServiceInterface()
        {
            //service methods and implementation
        };
    }
}
```

Der Konfigurationsmechanismus im Tapestry-Framework unterstützt aber nicht nur das Definieren von Services. Eine der besonderen Stärken des Frameworks liegt in der Möglichkeit via IOC einem Service eine Liste von Objekten des selben Interfacetyps zu

übergeben. Diese Art der Konfiguration wird in *Tapestry* mit "Contribution" bezeichnet. Eine genaue Beschreibung des Tapestry Contribution-Mechanismus mit allen möglichen Übergabeformen wird im nächsten Kapitel gegeben.

4 Tapestry-Contributions

Die meisten Services können die Dienste die sie anbieten nicht komplett selbstständig erledigen. Also stehen Services oft in Abhängigkeit zu anderen Services. Es gibt viele Fälle in denen eine solche Abhängigkeit nicht nur in einer 1:1 Beziehung besteht. Es kommt vor, dass sich ein Service einer ganzen Liste an Implementierungen eines anderen Serviceinterfaces bedient, um seine Aufgaben zu lösen. In einer herkömmlichen Java Anwendung würde diese Liste, bei der Initialisierung des Dienstes, an diesen mitgegeben werden. In einer durch einen IOC-Container verwalteten Anwendung, benötigen wir die Möglichkeit, diese Liste als Konfiguration für das Service bereit zu stellen. In *Tapestry* erledigen wir diese Aufgabe über eine *"contribute"* Methode in der Konfiguration. Laut Namenskonvention beginnt diese Konfigurationsmethode mit der Bezeichnung "*contribute*", gefolgt von dem Namen des Services, für den die Konfiguration und somit die Liste gedacht ist (siehe Listing 4.1). Dadurch entsteht eine eindeutige Zuordnung zwischen der Liste und dem Service das sie konsumiert.

Ein großer Vorteil, den diese Art der Servicekonfiguration durch die Indirektion über den IOC-Container mit sich bringt, ist die Möglichkeit jener Liste, derer sich der Service bedienen soll, modulübergreifend weitere Implementierungen hinzuzufügen.

4.1 "Configuration"

Wenn man nun einem Service eine Liste von Objekten übergeben möchte, so bedient man sich innerhalb der *"contribute"* Methode eines Konfiguration-Objekts (*org.apache.tapestry5.ioc.Configuration<T>*), in dem man die einzelnen Objektdefinitionen ablegen kann (siehe Listing 4.1).

Listing 4.1: *Tapestry*: Einfache Listenübergabe

```
public class MyModulConfiguration
{
    public static void contributeMyServiceInterface(Configuration<
       MyServiceDependencyElement> configuration)
    {
        configuration.add(new MyServiceDependencyElementImplOne());
        configuration.add(new MyServiceDependencyElementImplTwo());
        // ... more Elements to be added to the list
    }
}
```

Der Konfigurationsmechanismus im Tapestry-Framework unterstützt aber nicht nur eine einfache Listenübergabe über das Interface

org.apache.tapestry5.ioc.Configuration<T> als Servicecontribution, sondern bietet noch weitere, komplexere Übergabeformen.

4.2 "Ordered-Configuration"

Es gibt Anwendungsfälle in denen es nicht ausreicht, einem Service eine einfache Liste von Objekten zu übergeben. Bei der Verwendung der übergebenen Liste wird diese üblicherweise sequenziell abgehandelt. Man kann sich nun leicht Fälle vorstellen, bei denen die Reihenfolge der Objekte in der Liste nicht beliebig sein kann. Wenn zum Beispiel Abhängigkeiten zwischen den einzelnen Diensten aus der Liste bestehen, so hat die Reihenfolge ihrer Abarbeitung Auswirkungen auf das Ergebnis. Für den Fall, dass man also die Reihenfolge der Dienste in der Liste beeinflussen möchte bzw. muss, bietet *Tapestry* die Übergabeform einer geordneten Liste über das Interface

org.apache.tapestry5.ioc.OrderedConfiguration<T> an. Wie eine derartige "Ordered Contribution" als Konfiguration durchgeführt wird zeigt das Codebeispiel in Listing 4.2.

Listing 4.2: *Tapestry*: Sortierte Listenübergabe

```
public class MyModulConfiguration
{
    public static void contributeMyServiceInterface(
        OrderedConfiguration<MyServiceDependencyElement> configuration
        )
    {
        configuration.add("last", new LastService(), "after:*");
        configuration.add("second", new SecondService(), "after:first
            ");
        configuration.add("first", new FirstService(), "before:*");

        // ... more Elements to be added to the ordered list
    }
}
```

In Listing 4.2 wird eine Servicekonfiguration durchgeführt, in der festgelegt ist, dass einem Service, der das Interface *MyServiceInterface* implementiert, eine sortierte Liste von Services übergeben werden soll. Die Sortierung der Liste kann dabei durch die Schlüsselwörter "before" und "after" in Zusammenhang mit dem Wildcardzeichen "*" oder dem direkten Bezug auf ein anderes Element der Liste beschrieben werden. Um den direkten Bezug auf andere Elemente zu ermöglichen, müssen den einzelnen Elementen eindeutige Namen zugeordnet werden. Die Sortierung der Liste ist im Beispiel so eingestellt, dass

das Service *FirstService* an erster Stelle, vor allen anderen kommt, gefolgt von dem Service *SecondService* und dem Service *LastService*, der das Ende der Liste bildet.

4.3 "Mapped-Configuration"

Es lassen sich aber auch Anwendungsfälle finden, in denen die Datenstruktur einer Liste, sei sie geordnet oder ungeordnet, als Übergabeform für ein zu konfigurierendes Service nicht ausreicht, um die ihm gestellten Aufgaben zu bewältigen. In manchen Fällen ist es notwendig, Services anhand von Schlüsselelementen zu identifizieren. Für diese Zuordnung kann eine Listenstruktur nicht mehr verwendet werden. Sehr wohl lässt sich aber eine derartige Anforderung in Java durch den Objekttyp einer *Map* realisieren. Damit es nun auch möglich ist, Services mit Schlüsseln in Beziehung zu bringen, und dies auch als Konfiguration im IOC-Container abzulegen, bietet *Tapestry* das Interface *org.apache.tapestry5.ioc.MappedConfiguration<K,V>* an.

Listing 4.3: *Tapestry*: Serviceübergabe mit Schlüsselmapping

```
public class MyModulConfiguration
{
    public static void contributeMyServiceInterface(
        MappedConfiguration<String, MyServiceDependency> configuration
        )
    {
        configuration.add("keyFoo", new FooService());
        configuration.add("keyFooFoo", new FooFooService());

        // ... more key/value pairs to be added to the map
    }
}
```

In Listing 4.3 wird die Verwendung einer "Mapped Contribution" gezeigt. Dabei wird eine Servicekonfiguration angelegt, die einem Service, das das Interface *MyServiceInterface* implementiert, eine Map übergibt. Über die Schlüssel "keyFoo" und "keyFooFoo" kann das Service dann Zugriff auf die zwei, den Schlüsseln zugeordneten, Objekte erhalten.

5 Bean-Listen und Bean-Maps in *Spring*

In *Spring* gibt es von Haus aus die Möglichkeit, unterschiedliche Beans vom selben Interfacetyp in Listen oder in Maps zusammenzufassen und via *Dependency Injection* an andere Beans weiterzugeben. Dieser Dienst des IOC-Containers ist in seiner Form etwas wie eine leichtgewichtige Minimalform des Tapestry Contribution-Services. *Spring* stellt diese Funktionalität automatisch zur Verfügung und sieht dafür keine eigenständige Konfigurationssyntax vor. Der Mechanismus läuft dabei folgendermaßen ab. Sobald der IOC-Container ein Bean instanziieren muss, dessen Konstruktor eine Liste von Objekten erwartet, die dasselbe Interface implementieren, werden alle im IOC-Container befindlichen Beans auf die diese Eigenschaft zutrifft, in eine Liste verpackt und dem konsumierenden Bean bei der Erstellung übergeben.

Dasselbe Prinzip wird auch für Map-Parameter angewandt, wobei die einzelnen Beans nun als Werte in einer Map zusammengefasst werden. Als Schlüssel wird jeweils der in der Konfiguration festgelegte Name des Beans herangezogen. Zur Beschreibung dieses Mechanismus zeigt Listing 5.1 zwei Java Klassen als Konsumenten und Listing 5.2 die zugehörige, benötigte Spring-Beankonfiguration.

Listing 5.1: *Spring*: "leichtgewichtige" Standard-Contribution Konsumenten

```
// Java-Implementierung eines Listen-Konsumenten
public class ListValueHolder
{
    public ListValueHolder(List<String> contributedStringList)
    {
        //do something with the list
    }
}

// Java-Implementierung eines Map-Konsumenten
public class MapValueHolder
{
    public MapValueHolder(Map<String, String> contributedStringMap)
    {
        //do something with the map
    }
}
```

Listing 5.2: *Spring*: "leichtgewichtige" Standard-Contribution Konfiguration

```
<!-- (XML) Bean Konfig des Listen-Konsumenten -->
<bean name="listValueHolder" class="org.springframework.contributions
    .ListValueHolder" />

<!-- (XML) Bean Konfig des Map-Konsumenten -->
<bean name="mapValueHolder" class="org.springframework.contributions.
    MapValueHolder" />

<!-- (XML) Konfiguration der Elemente für die Liste und die Map-->
<bean name="string1" class="java.lang.String" value="String 1" />
<bean name="string2" class="java.lang.String" value="String 2" />
```

Die eine Klasse *ListValueHolder* erwartet in ihrem Konstruktor eine Liste von Strings. Da in der XML-Beankonfiguration zwei Beans vom Typ *String* angegeben sind, wird *Spring* beim Instanziieren des Beans *ListValueHolder* diese beiden Strings in eine Liste verpacken und diese Liste an den Konstruktor übergeben. Im Vergleich zur Tapestry-Variante der Konfiguration hat man hier auf die Reihenfolge der Beans in der Liste keine explizite Einflussmöglichkeit.

Die zweite Klasse *MapValueHolder* erwartet in ihrem Konstruktor eine *Map* mit *Strings* als Schlüssel und Wert. Hier würde *Spring* beim Instanziieren der Klasse automatisch eine *Map* erstellen, dieser die beiden String-Beans als Werte mit deren Beannamen als Schlüssel hinzufügen und die *Map* dem Konstruktor übergeben. Auch hier bietet die Konfiguration keine komplexeren Möglichkeiten, wie zum Beispiel das Verwenden von Enumeration-Werten als Schlüssel.

6 SpringContributions - Eine Umsetzung als Spring-Erweiterung

Das Spring-Framework ist selbst aus Komponenten aufgebaut. Aus diesem Grund bietet es auch die einfache Möglichkeit, neue Komponenten in das Framework zu integrieren. Dies hat sowohl Vorteile für die interne Weiterentwicklung, als auch für die Entwicklung von Erweiterungen durch Außenstehende. Um die aus dem Tapestry-Framework bekannte Contribution-Funktionalitäten auch in *Spring* einführen zu können, wurde genau diese Möglichkeit verwendet und die eigene Spring-Komponente SpringContributions entwickelt.

Seit den Anfängen der Spring-Framework Entwicklung gibt es auch den klassischen XML-Konfigurationsmechanismus des Frameworks. Wenn das Framework um neue Funktionalität und damit einhergehend die Konfiguration um neue Schlüsselwörter erweitert wurden, so geschah dies also immer über die Definition von XML-Elementen. Im Rahmen dieser Arbeit wurde zu Beginn genau dieser klassische Weg gewählt, um die SpringContributions Erweiterung umzusetzen.

Seit der Hauptversion 3.0 bietet *Spring*, parallel zur XML-Konfiguration, einen weiteren Weg der Konfiguration an. Die neue Möglichkeit, *Spring* zu verwenden, nennt sich "Java Configuration" und wurde aus der zuvor eigenständig entwickelten Projekt "*Spring JavaConfig*" [16] in das Spring-Framework übernommen. Dabei handelt es sich, wie der Name bereits vermuten lässt, um eine Möglichkeit, mittels der man die gesamte Spring-Konfiguration eines Projektes in Java-Klassen vornehmen kann. Es ist ein Einfaches, sich vorzustellen, dass es viele Vorteile geben kann, die Konfiguration eines Projektes, statt in einer Auszeichnungssprache wie XML, in einer Programmiersprache wie Java durchführen zu können. Daher war es auch ein Ziel bei der Entwicklung der SpringContributions Erweiterung, eben auch diese zweite Konfigurationsmöglichkeit anzubieten. Wie die Herangehensweise bei der Umsetzung des jeweiligen Konfigurationsmechanismus aussieht, wird in den Unterabschnitten 6.2.1 und 6.2.2 behandelt.

6.1 Konzeptionelle Unterschiede von *Tapestry* zu SpringContributions

Das Spring IOC-Modell unterscheidet sich in manchen Punkten deutlich vom Tapestry IOC-Modell. Einer der Unterschiede wirkt sich dabei auch maßgeblich auf die Umsetzung und die Funktionalität der SpringContributions aus. Es handelt sich dabei um den Umstand, dass in *Spring* jedes Konfigurationseinheit in einem Bean abgebildet wird. Das Bean-Konzept in *Spring* schreibt vor, dass jedes Bean auf jeden Fall eine eindeutige Iden-

tifikation besitzen muss. Dieser Umstand wurde in der Konzeption der SpringContributions miteinbezogen und konnte dazu genutzt werden, durch eine kleine Abweichung vom Vorbild des Tapestry Contribution-Mechanismus, einen Mehrwert in den Verwendungsmöglichkeiten zu gewinnen.

In *Tapestry* verhält es sich so, dass eine Contribution immer eindeutig einem Konsumenten zugeordnet ist. In der Spring-Konfiguration wird jede Contribution als eigenes Konfigurationselement und somit als eigenes Objekt bzw. Bean behandelt. Wie bereits erwähnt, erhält nun jedes Bean zur Identifikation auch einen eigenen, eindeutigen Namen. Daraus ergibt sich, dass in *Spring* eine Contribution, die ja als eigenes und über den eindeutigen Namen eindeutig referenzierbares Objekt existiert, nicht nur von einem bestimmten, sondern von vielen beliebigen Konsumenten verwendet werden kann. Ist ein Contribution-Bean also einmal definiert, so kann jedes Bean das sich nun dieser Contribution anschließen oder sich ihrer bedienen will, über deren eindeutigen Namen das tatsächliche Bean referenzieren. Dieser Vorteil, aber auch die zu beachtenden Risiken bei der mehrfachen Verwendung einer Contributions, wird im Unterabschnitt 6.5.2 noch genauer beleuchtet.

Der eben erwähnte Mehrwert in der Verwendung bringt aber auch gleichzeitig neue Herausforderungen für den Anwender. Dieser ist nämlich selbst für die Vergabe der Namen für Contributions verantwortlich und muss sich deshalb selbst um deren Eindeutigkeit kümmern. Wie der Name für ein Contribution-Bean definiert wird, ist im Detail im Unterabschnitt 6.4.4.5 beschrieben.

Zusätzlich ist es im Unterschied zur Tapestry-Vorlage, durch das Wegfallen der eindeutigen Zuordnung einer Contribution zu genau einem Konsumenten, nötig den Namen der Contribution innerhalb der Konfiguration des Konsumenten zu wissen und anzugeben. Das automatische Auflösen der Referenzen von Konsument zu Contribution durch den IOC-Container kann also in den SpringContributions nur mehr auf Basis des eindeutigen Namen funktionieren.

Ein weiterer, konzeptioneller Unterschied zur Tapestry-Vorlage betrifft die Abbildung der Listenform einer Contribution. In den SpringContributions wird nicht mehr extra zwischen unsortierten und sortierten Listen unterschieden. Dieser Weg wurde gewählt, um die Menge der unterschiedlichen Konfigurationselemente möglichst gering zu halten. Es wird also nur noch ein Konfigurationselement für eine Contribution-Liste benötigt, egal ob diese sortiert oder unsortiert sein soll. Die Angabe einer Reihenfolge der Listenelemente wird über eine optionale Sortierungsangabe ermöglicht. Demnach besteht nur noch die Unterscheidung zwischen Listenübergabe und "gemappter" Übergabeform. Im SpringContributions-Kontext werden für diese beiden Übergabeformen von nun an die Bezeichnungen Contribution-Liste und Contribution-Map verwendet.

6.2 Die zwei Wege der Spring-Erweiterung

Dieser Abschnitt soll zeigen was genau gemacht werden muss, wenn man *Spring* mit einer der beiden angebotenen Möglichkeiten um ein Modul erweitern möchte. Zuerst wird auf die Erweiterung mit der klassischen XML-Konfiguration eingegangen, danach folgt eine

ausführliche Beschreibung der Erweiterung über die Java-Konfiguration.

6.2.1 Erweiterung mittels Spring XML-Konfiguration

Entscheidet man sich, bei der Erweiterung von *Spring* dafür, den klassischen Weg der XML-Konfiguration anzubieten, so sind die Ausgangspunkte in der Entwicklung der neuen Komponente ein eigener Namensraum und das zugehöriges XML-Schema. Die für *Spring* ab der Version 3.0 entwickelten SpringContributions verwenden das XML-Schema

"http://www.springframework.org/schema/contributions/spring-contributions-3.0.xsd"

mit dem XML-Namensraum "http://www.springframework.org/schema/contributions".

Um die Grundlage für den *Spring* Contribution-Mechanismus zu legen, werden im XML-Schema einige neue Konfigurationselemente definiert. Jede einzelne dieser Konfigurationsmöglichkeiten wird in Unterabschnitt 6.3 genauer beschrieben werden. Dabei werden die zur Verwendung stehenden XML-Elemente und XML-Attribute, ausgehend von ihrer Schemadefinition und im Hinblick auf ihre Anwendung, eingeführt.

6.2.2 Spring-Erweiterung mittels Java-Konfiguration

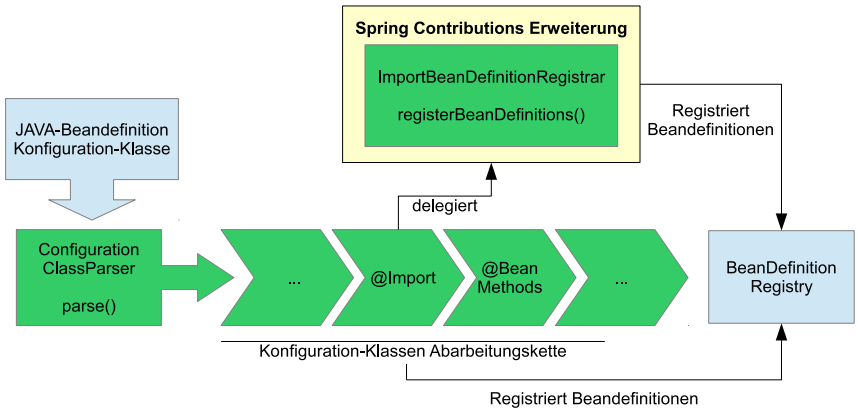
Will man für eine eigene Spring-Komponente die Konfiguration auch als Spring Java-Konfiguration (siehe Abschnitt 3.1.1) ermöglichen, so gibt es zwei unterschiedliche Wege der Umsetzung. Zum einen bietet *Spring* eine Eingriffsmöglichkeit, bei der man sich direkt in die früheste Phase der Registrierung von Beandefinitionen einschalten kann (siehe Listing 6.1). Zum anderen kann man nach Ablauf der Standardregistrierung von Beandefinitionen eingreifen und die bestehenden Definitionen verwenden, anpassen und auch neue Definitionen hinzufügen (siehe Listing 6.2). Für eine erste Umsetzung der Java-Konfiguration für die SpringContributions habe ich mich für die zweite Variante entschieden, wobei ein kleiner Teil der Umsetzung sich auch auf die erste Variante stützt. Auf die Gründe für diese Entscheidung und die Umsetzung werde ich nach der Beschreibung der beiden möglichen Varianten genauer eingehen.

Eingriff während der Beandefinition Registrierung (Variante 1): Ebenso wie ein XML-Konfigurationsfile muss auch eine Spring Java-Konfigurationsklasse zuallererst geparst werden. Danach werden alle Annotationen (@PropertySource, @ComponentScan, @Import, @Bean, etc.) und deren Metadaten in einer festen Reihenfolge abgearbeitet. Jeder Schritt in dieser Abarbeitungskette behandelt eine ganz bestimmte Annotation und den zugehörigen Konfigurationscode. Dabei bietet die Abarbeitung der **@Import** Annotationen, die noch vor der @Bean Annotation an dritter Stelle behandelt wird, den Anknüpfungspunkt für eigene Erweiterungen der Java-Konfiguration. Über diesen Weg lässt sich ein eigener *ImportBeanDefinitionRegistrar* definieren, in den die Logik zum Registrieren der Beandefinitionen kommt. Diese Klasse lässt sich dann über die @Import Annotation an eine eigens definierte Annotation (z.B. @EnableContributions) binden. Verwendet man nun diese selbst definierte Annotation innerhalb einer Java-Konfiguration,

so wird also beim Abarbeiten der Imports die *registerBeanDefinitions* Methode der *ImportBeanDefinitionRegistrar* Klasse aufgerufen und die eigenen Logik zum registrieren von Beandefinitionen angewandt.

Bei diesem Weg der Erweiterung befindet man sich zeitlich noch vor der *Spring* eigenen Beandefinition Registrierung und damit sehr früh in der Abfolge der IOC-Registry Erstellung. Das würde den Vorteil ergeben, dass alle mit `@Bean` definierten Services bereits die Contribution-Konfiguration verwenden könnten. Umgekehrt besteht natürlich nicht die Möglichkeit, bei der Abarbeitung der Contribution-Konfiguration, bereits auf die Beandefinitionen zuzugreifen. Wenn man also die Beans für die Contribution schon zu diesem Zeitpunkt benötigt, so muss man sich ebenfalls selbst darum kümmern. Das heißt, man müsste äquivalent zur `@Bean` Annotation, eine eigene Konfiguration für Beans anbieten. Darüber könnte man schließlich Beans definieren, deren Definition dann bereits zum Erstellungszeitpunkt der Contribution-Konfiguration zur Verfügung stehen würden.

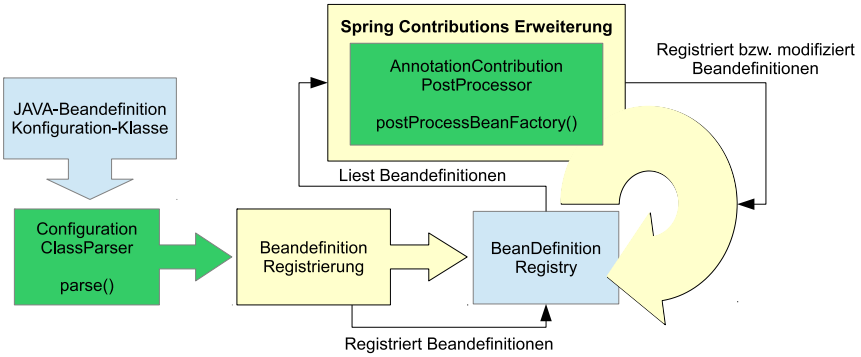
Abbildung 6.1: Spring Java-Konfiguration Erweiterung (Variante 1)



Eingriff nach der Beandefinition Registrierung (Variante 2): Bei dieser Variante kann mittels eines Postprozessors, in einer zuletzt gereihten Phase beim Erstellen des *ApplicationContext*, die vom Spring-Kern erstellte IOC-Konfiguration angepasst werden. Zu diesem späten Zeitpunkt sind bereits alle *Spring* eigenen Standardkonfigurationen als Beandefinitionen in der IOC-Registry abgelegt. Nun kann man innerhalb einer Implementierung des Interfaces *BeanDefinitionRegistryPostProcessor* den Code zur Abarbeitung der SpringContributions Konfiguration realisieren. Hier ist es dann natürlich möglich, alle bereits bestehenden Beandefinitionen in der Abarbeitung der Contribution-Konfiguration zu berücksichtigen und zu referenzieren. Man hat hier also den Vorteil, sich selbst nur um die Cotribution-Bean Erstellung und nicht um die normale Beankonfiguration Erstellung kümmern zu müssen, da diese ja bereits im Spring-Kern erstellt wurden.

Natürlich besteht bei dieser Variante nicht die Möglichkeit, dass in der normalen Beandefinition Abarbeitung bereits auf die Contribution-Definitionen zugegriffen werden kann, da diese ja erst im Postprozessor erstellt werden. Konkret bedeutet das für die Umsetzung der SpringContributions, dass für die normalen mit `@Bean` annotierten Beankonfigurationen eine Indirektion zum Zugriff auf Conribution-Beans verwendet werden muss. Dazu müssen eigene "Resolver Services" angeboten werden, die schon zum Erstellungszeitpunkt der Beans zur Verfügung stehen und später, beim Instanziieren der Beans, einen Zugriff auf die Contribution-Beans ermöglichen.

Abbildung 6.2: Spring Java-Konfiguration Erweiterung (Variante 2)



Ehemals mögliche Variante 3: Bevor die Java-Konfiguration in das Spring-Kernprojekt übernommen wurde, existierte sie als eigenes Projekt mit dem Namen "*Spring Java Configuration*". Die Entwickler dieses Projektes hatten sich einen angenehmen Weg der Erweiterung der Java-Konfiguration ausgedacht. Dabei lehnten sie sich an den aus dem Spring-Framework bekannten Namespacehandler-Mechanismus für die Erweiterung der XML-Konfiguration an. Umgesetzt war die Erweiterbarkeit über eine eigens definierte Annotation `@Plugin` und das angebotene Interface *ConfigurationPlugin*. Das Interface musste implementiert werden und enthielt dann den nötigen Code für die neue Java-Konfiguration Funktionalität. Diese Implementierung konnte dann mit der `@Plugin` Annotation an eine eigene Annotation gebunden werden. Über die eigene Annotation konnte dann die neue Funktionalität in jeder normalen Konfigurationsklasse mit eingebunden werden. Diese Art der Erweiterung war ähnlich der zuvor vorgestellten Variante 1 und wurde dann auch nicht in das Kernprojekt von *Spring* mit übernommen.

Auswahl von Variante 2 für die konkrete Umsetzung: Für die Implementierung der SpringContributions Java-Konfiguration fiel die Entscheidung auf die zweite Variante, da für die erste Variante die komplette Beankonfiguration Generierung von *Spring* erneut implementiert hätte werden müssen. Da sich aber der interne Spring-Code für die Beankonfiguration Generierung nicht zur Wiederverwendung eignet, hätte hier all zu viel recht komplexer Code dupliziert werden müssen. Dieses Unterfangen wäre nicht nur

besonders fehleranfällig gewesen, sondern hätte auch besonders intensive Wartungsarbeit erfordert, wenn sich im Zuge von Versionsänderungen die diesbezügliche Logik im Spring-Framework geändert hätte. Demzufolge wäre das gesamte SpringContributions Modul durch eine Umsetzung der ersten Variante sehr fragil geworden.

Durch die Wahl der zweiten Variante, entsteht nun aber die Notwendigkeit der Entkopplung von Referenzen auf Contribution-Beans. Dies muss durch die Verwendung von eigenen "Resolver Services" (siehe Abschnitt 6.3.2) realisiert werden. Da aber die Umsetzung dieser Services keinen großen Aufwand bedeutet, und die Verwendung von derartigen Services das Benutzen der Java-Konfiguration auch nicht erheblich komplexer gestaltet, gibt es auch deshalb keinen triftigen Grund, der ersten Variante den Vorzug zu geben. Deshalb fiel die Wahl bei der Umsetzung auf die Variante 2, und damit auf das Abarbeiten der Contribution-Konfiguration über das Postprocessing.

Probleme in der Umsetzung: Bei der Konzeption der Contribution-Map Funktionalität für die Java-Konfiguration ergaben sich einige Probleme dabei, exakt dasselbe Set an Konfigurationsmöglichkeiten zur Verfügung zu stellen, wie es bereits in der XML-Variante umgesetzt worden war. In *Spring* hat man sich bei der Java-Konfiguration dafür entschieden, Konfigurationsmetadaten über Annotationen abzubilden. Aus Gründen der Konsistenz baut die Java-Konfiguration der SpringContributions ebenfalls auf Annotationen auf. Diese Entscheidung bringt leider auch einen Nachteil gegenüber der Tapestry-Variante der Konfiguration mit sich. In *Tapestry* werden die Konfigurationsmetadaten auch ohne Annotationen, großteils in Verbindung mit eigenen Builder-Services, geregelt. Es werden die Konfigurationsmetadaten also hauptsächlich im Javacode der einzelnen Konfigurationsmethoden angegeben. Dadurch ist es in einer Tapestry-Konfigurationsklasse möglich, für die Konfiguration auch auf Objekte bzw. Variablen zuzugreifen. Dabei kann dann natürlich auch die Syntaxprüfung des Compilers genutzt werden. Verwendet man jedoch den in *Spring* gängigen Weg der Java-Konfiguration und benutzt hauptsächlich Annotationen als Konfigurations-Metadaten, dann fällt die Unterstützung der Syntaxprüfung durch den Compiler, in Zusammenhang mit verwendeten Objekten bzw. Variablen, dieser Entscheidung zum Opfer. Das liegt daran, dass es in Annotationen nicht möglich ist, Objektreferenzen als Parameter anzugeben.

Aus der ausschließlichen Verwendung von Annotationen zur Abbildung der Konfigurationsmetadaten ergab sich darüber hinaus noch eine Schwierigkeit in der Umsetzung des von der XML-Lösung bekannten Funktionsumfangs. Die momentane Implementierung bildet daher einen Kompromiss aus dem Streben nach Äquivalenz zur XML-Konfiguration und dem Bemühen, die Java-Konfiguration überschaubar und verwendbar zu gestalten. So wurde zum Beispiel die Möglichkeit weggelassen, konkrete Beans beziehungsweise Objekte als Schlüsselelement einer Contribution-Map verwenden zu können. Auf diese Möglichkeit musste verzichtet werden, da man eben in einer Annotation keine konkreten Objektinstanzen übergeben kann.

Da Enumerationen sich, seit ihrer Einführung in Java, einer sehr hohen Beliebtheit erfreuen, war es wichtig die Möglichkeiten der Verwendung von Enumerationen als Schlüssel der Contribution-Map umzusetzen. Dies konnte leider, wieder aufgrund der Eigenschaf-

ten von Annotationen, nur über die Verwendung der zwei Parameter Enumeration-Klasse und Enumeration-Wert verwirklicht werden (siehe Listing 6.14). Die direkte Angabe einer konkreten Enumeration, wie dies in der XML-Konfiguration möglich ist, konnte nicht umgesetzt werden. Daraus ergibt sich, dass sich der Vorteil der IDE-Unterstützung bezüglich der Referenzsuche im Javacode, im Falle der Konfiguration für eine Contribution-Map mit Enumeration als Schlüssel, nicht ausnutzen lässt.

Ein letzter wichtiger Punkt war die Verwendung von Klassen als Schlüssel der Contribution-Map. Da es bei der Verwendung des Strategy-Services (siehe Abschnitt 6.6) unbedingt notwendig ist, Klassen als Schlüssel in einer Contribution-Map zu verwenden, konnte auf diese Funktionalität nicht verzichtet werden.

6.3 XML- und Java-Konfiguration im Detail

In diesem Abschnitt werden die einzelnen Konfigurationselemente der beiden Konfigurationsarten (XML, Java) im Bezug auf ihre Anwendung beschrieben. Dabei wird abwechslungsweise, für jede Art von Konfiguration, zuerst die XML-Variante und darauf folgend die Java-Variante erläutert.

6.3.1 (XML) Contributions Referenzieren

Die folgenden beiden XML-Elemente können verwendet werden um Contributions der verschiedenen Typen zu referenzieren.

XML-Element "contribution-ref" und "mapped-contribution-ref" : Über das Element "contribution-ref" wird ein Contribution-Listen Objekt referenziert. Möchte man eine Contribution-Map referenzieren, so muss man das XML-Element "mapped-contribution-ref" verwenden. Für beide Typen gilt, dass in dem XML-Attribut "name" der Name des zu verwendenden Contribution-Bean angegeben wird. Beide XML-Elemente können nun an allen möglichen Stellen, also z.B. innerhalb einer Konstruktor Definition oder auch in einer Setter Definition eines Beans, als Referenzierung auf das Contribution-Bean verwendet werden (siehe Listing 6.2).

Listing 6.1: XML-Schema für "contribution-ref" und "mapped-contribution-ref"

```
<xsd:element name="contribution-ref" type="ContributionType"/>
<xsd:element name="mapped-contribution-ref" type="ContributionType"/>
<xsd:complexType name="ContributionType">
    <xsd:attribute name="name" type="xsd:string">
        <xsd:annotation>
            <xsd:documentation><![CDATA[The name of the contribution,
                beans were added to.]]></xsd:documentation>
        </xsd:annotation>
    </xsd:attribute>
</xsd:complexType>
```

Die zu den beiden XML-Elementen gehörende XML-Schemadefinition wird in Listing 6.1 gezeigt.

Listing 6.2: (XML) Contribution Referenzen bei Konstruktor- und Setter-Injection

```
<?xml version="1.0" encoding="UTF-8" ?>
    <beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:ctr="http://www.springframework.org/schema/contributions
        " xsi:schemaLocation="http://www.springframework.org/schema/
        beans␣␣http://www.springframework.org/schema/beans/spring-
        beans-3.0.xsd␣http://www.springframework.org/schema/
        contributions␣http://www.springframework.org/schema/
        contributions/spring-contributions-3.0.xsd">

        <!-- Referenzierung einer Contribution-Liste und einer "
            Mapped-Configuration" in einem Bean mit Konstruktor-
            Injektion -->
        <bean id="constructorClassName" class="
            MyConstructorInitializedService">
            <constructor-arg>
                <ctr:contribution-ref name="contributionName" />
                <ctr:mapped-contribution-ref name="
                    mappedContributionName" />
            </constructor-arg>
            <!-- weitere Metadaten zum jeweiligen Bean -->
        </bean>

        <!-- Referenzierung einer Contribution-Liste und einer
            Contribution-Map in einem Bean mit Setter-Injektion -->
        <bean id="setterClassName" class="MySetterInitializedService"
            >
            <property name="listDependency" >
                <value>
                    <ctr:contribution-ref name="contributionName" />
                </value>
            </property>
            <property name="mappedDependency" >
                <value>
                    <ctr:mapped-contribution-ref name="
                        mappedContributionName" />
                </value>
            </property>
        </bean>
    </beans>
</xml>
```

6.3.2 (Java) Contributions Referenzieren

Für die SpringContributions wurden zwei "Resolver Services" (*OrderedContributionResolver* und *MappedContributionResolver*) implementiert, mit deren Hilfe es leicht möglich ist, ein Contribution-Objekt, nur durch die Angabe des Contribution-Namen, zu erhalten. Die Notwendigkeit für diese beiden Services resultiert daraus, dass die Java-Konfiguation der SpringContributions in ihrer aktuellen Umsetzung (siehe Abschnitt 6.2.2) erst nach der Standard-Beankonfiguration durchgeführt wird. Zur Abarbeitungszeit der Standard-Beankonfiguration stehen aber noch keine Contribution-Objekte bereit und die Abhängigkeiten von Beans auf Contribution-Objeke können vom IOC-Container noch nicht aufgelöst werden.

Listing 6.3: (Java) Referenzierung von Contribution-Containern

```
@Configuration
@EnableContributions
public class MyModulConfiguration
{
    // Referenzierung einer Contribution-Liste und einer Contribution
       -Map in einem Bean mit Konstruktor
    @Bean()
    public static MyConstructorInitializedService
        constructorClassName(
         OrderedContributionResolver<MyListElement> list,
         MappedContributionResolver<MyMapElement> map)
    {
         return new MyKonstructorServiceImpl(list.resolve("
             contributionName"), map.resolve("mappedContributionName");
    }

    // Referenzierung einer Contribution-Liste und einer Contribution
       -Map in einem Bean mit Setter
    @Bean()
    public static MySetterInitializedService setterClassName(
         OrderedContributionResolver<MyListElement> list,
         MappedContributionResolver<MyMapElement> map)
    {
         MySetterInitializedService service = new MySetterServiceImpl
             ();
         service.setList(list.resolve("contributionName"));
         service.setMap(map.resolve("mappedContributionName"));
         return service;
    }
}
```

Würde man also in einer Beankonfiguration direkt auf ein Contribution-Objekt zugreifen wollen, so müsste das zu einem Fehler führen. Durch die Indirektion über die beiden

Resolver Services kann dieses Problem umgangen werden. In der XML-Konfiguration werden die beiden Services nicht benötigt, da diese gleichzeitig mit der normalen Spring XML-Konfiguration verarbeitet werden und somit Contribution-Objekte wie Beans in der selben Phase, so zu sagen "gleichzeitig", erstellt werden.

Die Resolver Services bieten, wegen der Typisierbarkeit ihrer Rückgabe, auch noch den Vorteil, dass im Vergleich zu einem Beanobjekt das man direkt aus dem Spring Applicationcontext bezieht, keine Typspezifizierung mittels "Typecast" Befehl mehr auf das gelieferte Objekt angewendet werden muss.

Zusätzlich kommt den Resolver Services noch die Aufgabe der Fehlerbehandlung zu. Für den Fall, dass ein Contribution-Objekt tatsächlich nicht vorhanden ist, wird durch sie eine Warnung ausgegeben. Anstatt des Contribution-Objekts wird dann eine leere Liste oder Map zurückgegeben. Wie die Referenzierung von Contribution-Containern über die beiden Resolver Services in einer Java-Konfiguration durchgeführt wird zeigt Listing 6.3.

Da der Beanname jedes Contribution-Containers, bei der Erstellung seiner Beandefinition, ein Namenspräfix erhält (siehe Abschnitt 6.4.4.5), muss dieser natürlich auch dem eigentlichen Namen aus der Konfiguration vorangestellt werden, möchte man das Bean über den Namen aus dem IOC-Container laden. Diese Aufgabe wird einem ebenfalls durch die beiden Resolver Services abgenommen. Die Verwendung des Namenspräfix ist somit für den Benutzer sowohl in der Vergabe als auch in der Verwendung transparent.

6.3.3 (XML) Hinzufügen von Beans zu einer Contribution-Liste

Da uns in XML sowohl Elemente als auch Attribute zur Verfügung stehen, haben wir aus Flexibilitätsgründen die Möglichkeit zum Hinzufügen von Beans zu einer Contribution-Liste auf drei Arten realisiert.

Will man also ein Bean zu einer Contribution-Liste hinzuzufügen, so kann man sich der drei folgenden Varianten bedienen:

XML-Attribut "contributeTo": Mit diesem Attribut legt man in der Konfiguration fest, dass das eben definierte Bean ein Element der Contribution-Liste, mit dem im XML-Attribut "contributeTo" definierten Namen, werden soll (siehe Listing 6.4). Die zugehörige XML-Schemadefinition wird in Listing 6.5 gezeigt.

Listing 6.4: (XML) Einfache Elementdefinition zur Listenkonfiguration über das XML-Attribut "contributeTo"

```
<!-- Auszug aus einer XML-Konfiguration -->

<bean id="className" class="classToBeContributed"
    contributeTo="contributionName">
    <!-- weitere Metadaten -->
</bean>
```

Listing 6.5: XML-Schema für das XML-Attribut “contributeTo”

```
<xsd:attribute name="contributeTo" type="xsd:string"></xsd:attribute>
```

XML-Element “contribute”: Mit dem XML-Element “contribute” erreicht man dasselbe Ergebnis wie mit dem XML-Attribut “contributeTo” und kann also ein Bean zu einer Contribution-Liste hinzufügen (siehe Listing 6.6). Zusätzlich ist es bei dieser Methode aber möglich, das “constraints“ Attribut zu verwenden, um damit die Reihenfolge des Elements in der Contribution-Liste zu beeinflussen. Die zugehörige XML-Schemadefinition wird in Listing 6.7 gezeigt.

Listing 6.6: (XML) Einfache Elementdefinition zur Listenkonfiguration über das XML-Element “contribute”

```
<!-- Auszug aus einer XML-Konfiguration -->

<bean id="className" class="classToBeContributed">
    <contribute to="contributionName" constraints="after:*"/>
    <!-- weitere Metadaten -->
</bean>
```

Listing 6.7: XML-Schema für das XML-Element “contribute”

```
<xsd:element name="contribute">
    <xsd:complexType>
        <xsd:attribute name="to" type="xsd:string">
            <xsd:annotation>
                <xsd:documentation><![CDATA[The name of the
                    contribution this bean is added to.]]></
                    xsd:documentation>
            </xsd:annotation>
        </xsd:attribute>
        <xsd:attribute name="constraints" type="xsd:string">
            <xsd:annotation>
                <xsd:documentation><![CDATA[The sort constraints for
                    this contribution.]]></xsd:documentation>
            </xsd:annotation>
        </xsd:attribute>
    </xsd:complexType>
</xsd:element>
```

XML-Element "contribution": Über das XML-Element "contribution" kann man gleich mehrere Beans zu einer Contribution-Liste hinzufügen. Die Angabe des Namens des Contribution-Objekts, zu dem die Elemente hinzugefügt werden sollen, wird auch hier über das Attribut "to" bewerkstelligt. Im Gegensatz zum Tapestry Contribution-Mechanismus unterscheiden die SpringContributions nicht zwischen der sortierten und der unsortierten Listenübergabeform. Jede Konfiguration einer Contribution-Liste kann über das XML-Attribut "constraints" optional mit Sortierungsinformationen versehen werden. Lässt man die Sortierungsinformationen weg, so bestimmt die Reihenfolge des Vorkommens im Konfigurationstext die Ordnung der Elemente in der Liste. Die Syntax für die Sortierung wird, wie innerhalb einer Tapestry-Konfiguration, über die Schlüsselwörter "before" und "after" in Verbindung mit einem konkreten Elementnamen, der Stern Wildcard "*" oder einem regulären Ausdruck festgelegt.

Das XML-Attribut "ref" wird wie auch sonst in der Spring-Konfiguration als Verweis auf ein, an einer anderen Stelle definiertes, Bean verwendet. Das bedeutet, dass zu jedem über das XML-Element "Entry" definierte Element der Contribution-Liste, eine referenzierbare Beandefinition innerhalb der Konfigurationsfiles vorliegen muss (siehe Listing 6.8). Die Beandefinition muss aber im Konfigurationsfile nicht wie im Beispiel direkt vor seiner Referenzierung platziert werden. Die zugehörige XML-Schemadefinition wird in Listing 6.9 gezeigt.

Listing 6.8: (XML) Mehrfache Elementdefinition für normale und sortierte Listenkonfiguration

```
<!-- Auszug aus einer XML-Konfiguration -->

<bean name="myContributedService" class="MyContributedService" />

<ctr:contribution to="contributionName">
    <ctr:entry name="myContributedService" ref="myContributedService"
        constraints="after:*" />
    <!-- weitere Elemente -->
</ctr:contribution>
```

Listing 6.9: XML-Schema für das XML-Element “contribution”

```
<xsd:element name="contribution">
    <xsd:annotation>
        <xsd:documentation source="java:org.springframework.beans.
            factory.config.ListFactoryBean">
             Builds a List instance, populated with the specified
                content.
        </xsd:documentation>
        <xsd:appinfo>
            <tool:annotation>
                <tool:exports type="java.util.List"/>
            </tool:annotation>
        </xsd:appinfo>
    </xsd:annotation>
    <xsd:complexType>
        <xsd:group ref="contributionElements"/>
        <xsd:attribute name="to" type="xsd:string">
            <xsd:annotation>
                <xsd:documentation><![CDATA[The name of the
                    contribution this bean is added to.]]></
                    xsd:documentation>
            </xsd:annotation>
        </xsd:attribute>
    </xsd:complexType>
</xsd:element>

<xsd:group name="contributionElements">
    <xsd:sequence>
        <xsd:element ref="beans:description" minOccurs="0"/>
        <xsd:element name="entry" minOccurs="1" maxOccurs="unbounded"
            >
            <xsd:complexType>
                <xsd:complexContent>
                    <xsd:extension base="beans:propertyType">
                        <xsd:attribute name="constraints" type="
                            xsd:string">
                            <xsd:annotation>
                                <xsd:documentation>
                                    <![CDATA[The sort constraints for
                                        this contribution.]]>
                                </xsd:documentation>
                            </xsd:annotation>
                        </xsd:attribute>
                    </xsd:extension>
                </xsd:complexContent>
            </xsd:complexType>
        </xsd:element>
    </xsd:sequence>
</xsd:group>
```

6.3.4 (Java) Hinzufügen von Beans zu einer Contribution-Liste

Die Contribution-Liste in Java: In der Java-Konfiguration gibt es nicht, wie in der XML-Konfiguration, drei Schreibweisen (siehe Abschnitt 6.3.3), sondern nur noch eine Möglichkeit, ein Bean zu einer Contribution-Liste hinzuzufügen. Dafür wird die Annotation @Contribution mit dem Parameter "*to*" verwendet (siehe Listing 6.10). Ob das Element eine festgelegte Position in der Contribution-Liste erhalten soll oder nicht, wird über das Hinzufügen oder Weglassen des Parameters "*constraints*" gesteuert.

Listing 6.10: (Java) Einfache und sortierte Listenübergabe

```
@Configuration
@EnableContributions
public class MyModulConfiguration
{

    // Ein Bean das der Contribution-Liste "elements" als Element
       hinzugefügt wird;
    // ohne Information zu Positionierung in der Liste
    @Contribution(to="elements")
    @Bean(name="MyServiceDependencyElementImplOne")
    public MyServiceDependencyElement one()
    {
        return new MyServiceDependencyElementImplOne();
    }

    // Ein weiteres Bean das der Contribution-Liste "elements" als
       Element hinzugefügt wird;
    // mit Information zur Positionierung in der Liste
    @Contribution(to="elements", constraints="after:*")
    @Bean(name="MyServiceDependencyElementImplTwo")
    public MyServiceDependencyElement two()
    {
        return new MyServiceDependencyElementImplTwo();
    }
}
```

6.3.5 (XML) Hinzufügen von Beans zu einer Contribution-Map

XML-Element "mapped-contribution": Möchte man mehrere Elemente zu einer Contribution-Map (siehe Abschnitt 4.3) hinzufügen, so muss man sich des XML-Elements "mapped-contribution" bedienen. Der innere, komplexe Datentyp dieses Elements entspricht dem einer *java.util.Map*, und benötigt daher für jedes Element auch die Angabe von Schlüssel und zugeordnetem Wert über die Attribute "*key*" und "*value*" bzw. "*key-ref*" und "*value-ref*" (siehe Listing 6.12). In den beiden Attributen "*key*" und "*value*" können di-

rekt Schlüssel und Wert angegeben werden. Über die Attribute "*key-ref*" und "*value-ref*" können Beandefinitionen, die an einer anderen Stelle der Konfiguration definiert werden, über deren Namen referenziert und somit als Schlüssel und Wert angegeben werden. Die zugehörige XML-Schemadefinition wird in Listing 6.11 gezeigt.

Listing 6.11: XML-Schema für das XML-Element "mapped-contribution"

```
<xsd:element name="mapped-contribution">
    <xsd:annotation>
        <xsd:documentation source="java:org.springframework.beans.
            factory.config.MapFactoryBean">
             Builds a Map instance of the specified type, populated
                with the specified content.
        </xsd:documentation>
        <xsd:appinfo>
            <tool:annotation>
                <tool:exports type="java.util.Map"/>
            </tool:annotation>
        </xsd:appinfo>
    </xsd:annotation>
    <xsd:complexType>
        <xsd:complexContent>
            <xsd:extension base="beans:mapType">
                <xsd:attribute name="to" type="xsd:string">
                    <xsd:annotation>
                        <xsd:documentation><![CDATA[The name of the
                            contribution this bean is added to.]]></
                            xsd:documentation>
                    </xsd:annotation>
                </xsd:attribute>
            </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>
</xsd:element>
```

Listing 6.12: (XML) Mehrfache Elementdefinition für eine Contribution-Map

```
<!-- Auszug aus einer XML-Konfiguration -->

<bean name="myContributedService" class="MyContributedService" />

<ctr:mapped-contribution to="contributionName">
    <ctr:entry key="myServiceKey" value-ref="myContributedService" />
    <!-- weitere Elemente -->
</ctr:mapped-contribution>
```

6.3.6 (Java) Hinzufügen von Beans zu einer Contribution-Map

Das Hinzufügen von Beans zu einer Contribution-Map geschieht in der Java-Variante der Konfiguration über die `@ContributionMapped` Annotation und deren Parameter "*to*". Mit diesem obligatorischen Parameter wird der Name der Contribution-Map festgelegt, zu der das Bean hinzugefügt werden soll. Auf welches Bean sich die Annotation bezieht, wird über den Parameter "*name*" festgelegt. Dieser Parameter muss nicht unbedingt angegeben werden, da dann einfach der Name der Konfigurationmethode als Name herangezogen wird. Beim Abarbeiten der Konfiguration wird über diesen Namen die zugehörige Beandefinition aus dem IOC-Container geladen und kann dann der Contribution-Map hinzugefügt werden. Die Konfiguration funktioniert natürlich nur, wenn auch eine Beandefinition mit diesem Namen existiert. Aus diesem Grund wird also entweder der "*name*" Parameter oder der Bezeichner der zur Annotation gehörenden Konfigurationsmethode, mit dem Parameter "*name*" der zugehörigen `@Bean` Annotation übereinstimmen müssen.

Da jedes Bean in einer Contribution-Map auch einen Schlüsselwert benötigt, gibt es nun auch dafür Parameter. Möchte man einfach einen *String* als Schlüssel verwenden, dann verwendet man den Parameter "*key*". Möchte man aber eine Klasse als Schlüsselwert verwenden, so muss man den Parameter "*keyClass*" dementsprechend setzen. Die bisher beschriebenen Parameter sind in der Beispielkonfiguration in Listing 6.13 zu sehen.

Listing 6.13: (Java) Beandefinition und Übergabe an eine Contribution-Map

```
@Configuration
@EnableContributions
public class MyModulConfiguration
{
    @ContributionMapped(name="myService1",
        to="stringKeyContributionMap", key="myServiceKey")
    @Bean(name="myService1")
    public MyContributedService myService1()
    {
        return new MyContributedServiceImpl();
    }

    @ContributionMapped(to="classKeyContributionMap", keyClass="
       Interger.class")
    @Bean(name="myService2")
    public MyContributedService myService2()
    {
        return new MyContributedServiceImpl();
    }
}
```

Will man eine Bean zu einer Contribution-Map hinzufügen und dabei als Schlüssel-

element eine Enumeration verwenden, so kann dies über die Verwendung der optionalen Attribute "*keyEnumClass*" und "*keyEnumValue*" erreicht werden. Dabei erwartet der Parameter "*keyEnumClass*" eine Java-Klasse vom Typ Enumeration und der Parameter "*keyEnumValue*" den Namen eines Wertes aus der angegebenen Enumeration-Klasse als *String*. Natürlich müssen diese beiden Parameter immer in Kombination auftreten. Wird nur einer der beiden Parameter angegeben, so kommt es beim Starten des Spring Applicationkontext zu einem Fehler. Eine Beispielkonfiguration zu einem Contribution-Map Element mit einer Enumeration als Schlüssel ist in Listing 6.14 angegeben.

Die Verwendung von zwei Attributen ist hier deshalb notwendig, da es nicht möglich ist direkt einen Enumeration-Wert in einer Annotation zu referenzieren. Diese Art der Umsetzung ergibt sich aus der Entscheidung für die Verwendung von Annotationen zur Angabe der Konfigurations-Metadaten. Auf diese Weise entfällt zwar der mögliche Vorteil einer Syntaxprüfung durch den Compiler, den man bei der Verwendung von Java-Klassen als Konfiguration nutzen könnte, doch man erhält, bezüglich der Verwendung von Enumerationen, auf jeden Fall die gewünschte Gleichheit zum Funktionsumfang der XML-Konfiguration.

Listing 6.14: (Java) Contribution-Map: Enumeration Schlüssel Definition

```
//Hinzufügen eines Schlüssel/Wert-Paares zu einer Contribution-Map
    mit einer Enumeration als Schlüssel

@ContributionMapped(to="myMappedContribution", keyEnumClass=KeyEnum.
    class, keyEnumValue="keyOne")
@Bean(name="enumOneKeyObject")
public Object enumOneKeyObject()
{
    return new Object();
}

//Die verwendete Enumeration-Klasse mit ihren Werten
public enum KeyEnum
{
    keyOne,
    keyTwo;
}
```

6.4 Realisierung

In den letzten Abschnitten wurden die Anwendungsmöglichkeiten der SpringContributions gezeigt. In diesem Abschnitt soll nun in drei Teilen ein Einblick in die Implementierung des Moduls gegeben werden.

Zuerst wird gezeigt, wie der traditionell über XML gesteuerte Spring Konfigurati-

onsmechanismus erweitert wird, damit die neuen Schlüsselwörter wie z.B. "contribution" erkannt und in Beandefinitionen umgewandelt werden können.

Die SpringContributions unterstützen auch die in *Spring* neu vorhandene Möglichkeit der Java-Konfiguration. Ein weiterer Unterabschnitt zeigt ebenfalls die dafür nötigen Erweiterungen im Detail.

Abschließend wird die, für beide Konfigurationsarten gleiche, Erstellung der Beandefinitionen genauer beschrieben und ein Überblick über das Zusammenspiel von *Spring* und der neu entstandene SpringContributions Konfiguration gegeben.

6.4.1 (XML) Erweiterung der Spring-Konfiguration

Spring bietet grundsätzlich eine Möglichkeit zur Erweiterung seines XML-Konfigurationsmechanismus. Konkret werden von dem Framework die abstrakte Klasse *NamespaceHandlerSupport* und die Interfaces *NamespaceHandler, BeanDefinitionParser* und *BeanDefinitionDecorator* bereit gestellt. Die Implementierung des Interfaces *NamespaceHandler,* als Erweiterung der Klasse *NamespaceHandlerSupport,* ermöglicht es, einen eigenen XML-Namensraum für die Spring-Konfiguration festzulegen. Die Umsetzung für die SpringContributions geschieht in der Klasse *ContributionsNamespaceHandler* zusammen mit dem Namensraum http://www.springframework.org/schema/contributions.

Listing 6.15: XML-Schema Auszug aus *spring-contributions-3.0.xsd*

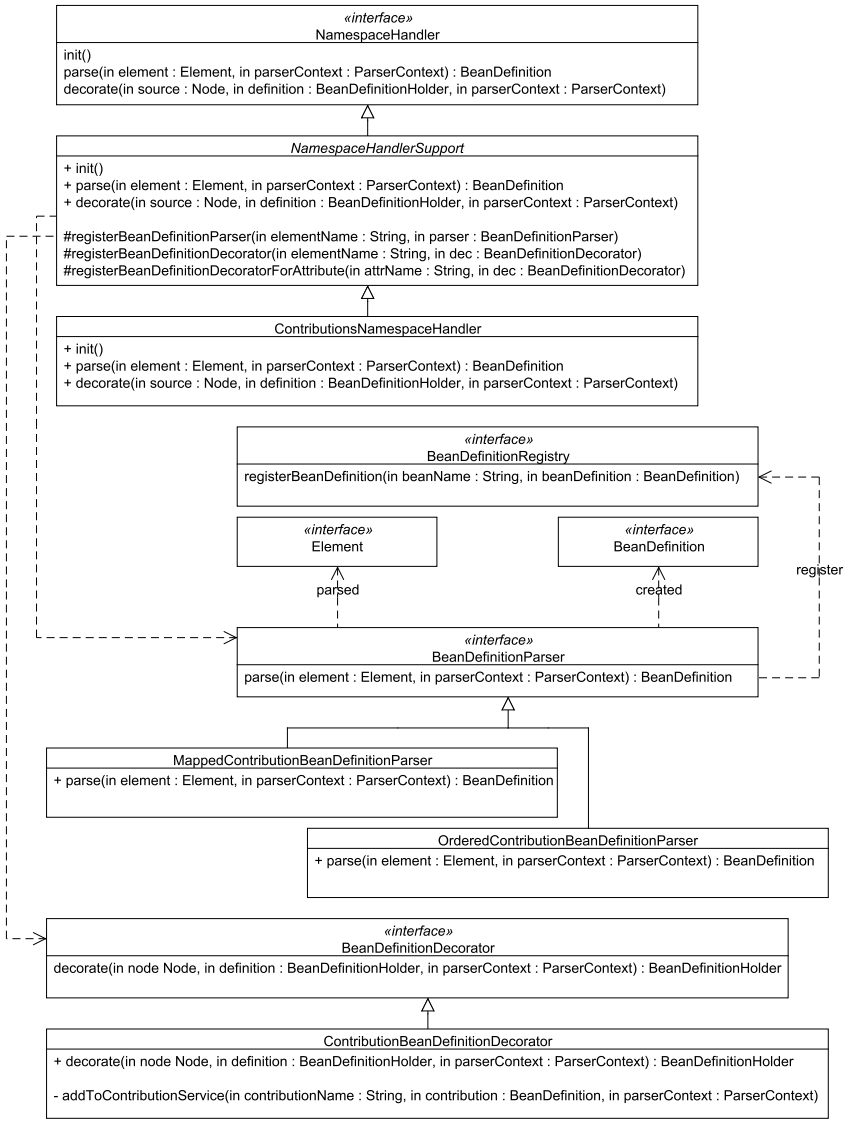
```
<xsd:attribute name="contributeTo" type="xsd:string"></xsd:attribute>

<xsd:element name="contribute">
    <xsd:complexType>
        <xsd:attribute name="to" type="xsd:string">
            <xsd:annotation>
                <xsd:documentation>
                    <![CDATA[The name of the contribution this bean
                        is added to.]]>
                </xsd:documentation>
            </xsd:annotation>
        </xsd:attribute>
        <xsd:attribute name="constraints" type="xsd:string">
            <xsd:annotation>
                <xsd:documentation>
                    <![CDATA[The sort constraints for this
                        contribution.]]>
                </xsd:documentation>
            </xsd:annotation>
        </xsd:attribute>
    </xsd:complexType>
</xsd:element>
```

Das dazugehörige XML-Schema *spring-contributions-3.0.xsd* - ein Auszug ist in Listing

6.15 zu sehen - beschreibt alle benötigten Elemente und Attribute und legt damit die Syntax der SpringContributions Konfiguration fest. *Spring* erhält die Information über den neuen Namensraum über zwei Konfigurationsfiles im META-INF Verzeichnis des SpringContributions Moduls. In dem einen "spring.schemas" wird das neue XML-Schema www.springframework.org/schema/contributions/spring-contributions-3.0.xsd, in dem anderen "spring.handlers" die dazugehörige Klasse *ContributionsNamespaceHandler* eingetragen.

Abbildung 6.3: (XML) UML zur Erweiterung der Spring-Konfiguration



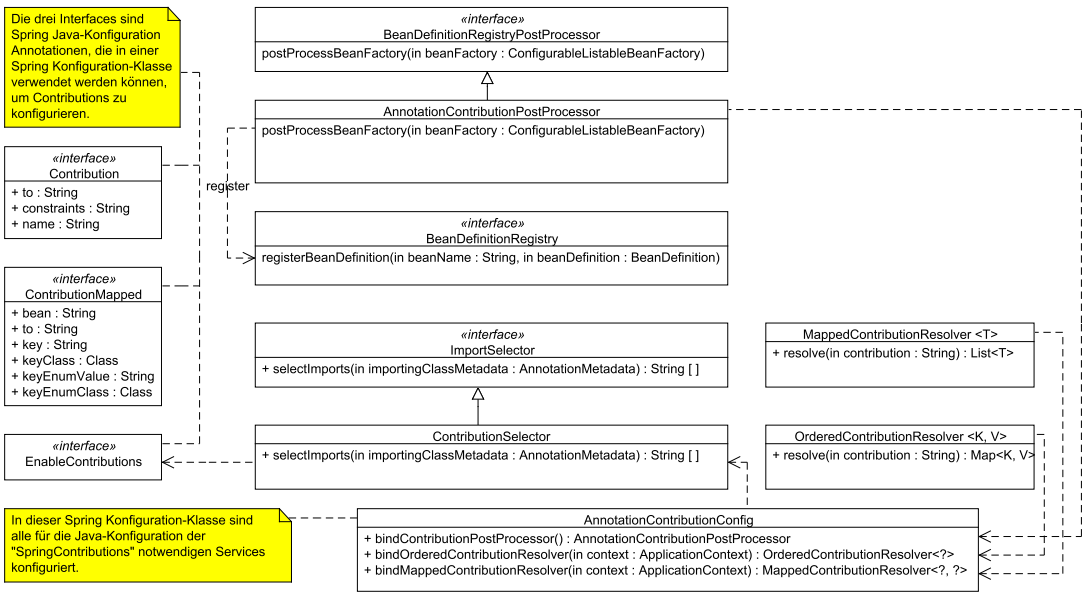
Damit der *NamespaceHandler* mit neu definierten, für *Spring* bisher unbekannten XML-Elementen umgehen und aus ihnen Beandefinitionen erstellen kann, nimmt die Klasse *NamespaceHandlerSupport* Implementierungen des Interfaces *BeanDefinitionParser* entgegen. Für die beiden Contibution-Basistypen, der Contibution-Liste und der Contibution-Map, werden hier zwei separate Parser, der *OrderedContributionBeanDefinitionParser* und der *MappedContributionBeanDefinitionParser,* verwendet. Abbildung 6.3 gibt einen Überblick zu den erwähnten Klassen.

6.4.2 (Java) Erweiterung der Spring-Konfiguration

Wie in Abschnitt 6.2.2 beschrieben, ist die Java-Konfiguration der SpringContributions über die Implementierung des Interfaces *BeanDefinitionRegistryPostProcessor* umgesetzt worden.

Dieser Postprozessor registriert, nach dem *Spring* eigenen Initialisieren der Beans, alle in Konfigurationsklassen definierten Contributions. Damit der Postprozessor beim Erstellen des IOC-Containers vom Framework automatisch verwendet wird, muss dieser selbst als Service-Bean definiert werden. Dies geschieht in der SpringContributions Basiskonfigurationsklasse (siehe Listing 6.16) für die Java-Konfiguration *AnnotationContributionConfig*. In dieser Konfigurationsklasse werden auch die beiden übrigen, notwendigen Services *OrderedContributionResolver* und *MappedContributionResolver* als Beans definiert.

Abbildung 6.4: (JAVA) UML zur Erweiterung der Spring-Konfiguration



Möchte man nun die SpringContributions Funktionalität in einer Konfigurations-

klasse verwenden, so würde es nun genügen, die Basiskonfigurationsklasse, z.B. über die Annotation @Import, in einer eigenen Konfigurationsklasse mit einzubinden. Um einen eleganteren und *Spring* üblichen Weg zur Aktivierung der SpringContributions anzubieten, bietet das Modul zusätzlich die Annotation @EnableContributions. Eine detaillierte Beschreibung hierzu folgt im nächsten Unterabschnitt 6.4.2.1. Einen Überblick zu den eben erwähnten Klassen gibt Listing 6.4.

6.4.2.1 Aktivierung der SpringContributions in Java Konfigurationsklassen

Will man die SpringContributions Funktionalität in Java Konfigurationsklassen verwenden, so müssen alle notwendigen Services natürlich selbst als Beans innerhalb des Spring Applicationcontextes zur Verfügung stehen. Dafür sind alle nötigen Dienste, wie der *AnnotationContributionPostProcessor* und die beiden Resolver Services, in einer Basiskonfigurationsklasse als Beans definiert. Zusätzlich wird ein eigener *ImportSelector* definieren. Diese Klasse lässt sich dann über die Annotation @Import an die Annotation @EnableContributions binden. Diese Annotation ist für das Aktivieren der Java-Konfiguration vorgesehen.

Listing 6.16: (Java) Basis Konfigurationsklasse

```
//Spring Java-Configuration-Klasse AnnotationContributionConfig:

@Configuration
public class AnnotationContributionConfig
{
    @Bean()
    public static BeanFactoryPostProcessor
        buildContributionPostProcessor()
    {
        return new AnnotationContributionPostProcessor();
    }

    @Bean(name="orderedContributionResolver")
    public OrderedContributionResolver<?>
        buildOrderedContributionResolver(ApplicationContext context)
    {
        return  new OrderedContributionResolver(context);
    }

    @Bean(name="mappedContributionResolver")
    public MappedContributionResolver<?, ?>
        buildMappedContributionResolver(ApplicationContext context)
    {
        return  new MappedContributionResolver(context);
    }
}
```

Verwendet man nun diese Annotation auf Klassenebene in einer Java-Konfiguration, so stehen für diese Konfiguration nun alle Dienste der SpringContributions Java-Konfiguration zur Verfügung und können somit innerhalb der eigenen Konfigurationsklasse genutzt werden. Listing 6.16 zeigt die Basiskonfigurationsklasse der SpringContributions. In dieser sind alle, für die Java-Konfiguration notwendigen, Service-Beans definiert.

Listing 6.17: (Java) Implementierung der **@EnableContributions** Annotation

```
//Java-Klasse ContributionContributionSelector:

/**
 * This {@link ImportSelector} implementation returns the
    configuration class {@link AnnotationContributionConfig}
 * and can be used in annotations which should be used for enabling
    Spring-Contributions in configuration classes.
 */
public class ContributionContributionSelector implements
   ImportSelector
{
    public String[] selectImports(AnnotationMetadata
       importingClassMetadata)
    {
        return new String[] { AnnotationContributionConfig.class.
           getName()};
    }
}

//Java-Klasse EnableContributions:

import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
import org.springframework.context.annotation.Import;

/**
 * This annotation can be used to activate Spring-Contribution
    functionality in a Spring java configuration
 * class annotated with the {@link Configuration} annotation.
 */
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Import(ContributionContributionSelector.class)
public @interface EnableContributions{}
```

In Listing 6.17 ist zu sehen, wie diese Basiskonfiguration an die Annotation @EnableContributions gebunden wird, so dass über diese Annotation die SpringContributions Funktionalität in der Spring Konfigurationsklasse aktiviert werden kann. Die Klassen aus Listings 6.16 und 6.17 und deren Zusammenhang sind zusätzlich in Abbildung 6.4 als UML-Diagramm dargestellt.

6.4.3 XML-Konfiguration VS Java-Konfiguration

Tabelle 6.1 zeigt eine Gegenüberstellung der Konfigurationselemente von XML- und Java-Konfiguration. Es ist hier zu sehen, dass für die Java-Variante weniger Elemente zur Verfügung stehen als in der XML-Variante. Dies resultiert aus der geringeren Komplexität der Java-Variante. In der Java-Variante gibt es zum Beispiel nicht die Möglichkeit, über eine Konfigurationsmethode eine ganze Liste von Beans als Elemente zu einer Contribution-Liste hinzuzufügen. Ebenso können nicht mehrere Schlüssel/Wert-Paare mit einer Konfigurationsmethode an eine Contribution-Map übergeben werden.

Tabelle 6.1: Gegenüberstellung von XML- / Java-Konfigurationschlüsselwörtern

Hinzufügen von ...	**XML-Element / -Attribut**	**Java-Annotation**
... einzelnen Beans zu einer Liste	contributeTo contribute contribution	@Contribution
... mehreren Beans zu einer Liste	contribution	
... einzelnen Beans zu einer Map	mapped-contribution	@Contribution-Mapped
... mehreren Beans zu einer Map	mapped-contribution	
Referenzieren einer ...	**XML-Element / -Attribut**	**Java-Klasse**
... Contribution-Liste	contribution-ref	OrderedContributionResolver
... Contribution-Map	mapped-contribution-ref	MappedContributionResolver

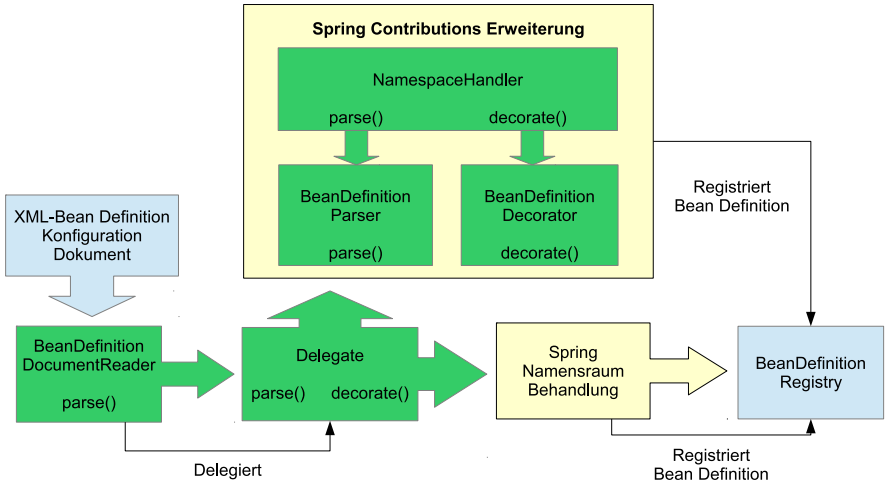
6.4.4 Von der Konfiguration zum Spring-Bean

Will man nun von einer Contibution-Konfiguration zu einer fertigen Beandefinition im IOC-Container gelangen, so sind zwei Schritte zu erledigen. Zum einen müssen die Contribution-Container, also die Beandefinitionen für Contribution-Liste oder Contribution-Map, erstellt werden. Zum anderen müssen die einzelnen Elemente der Container ebenso als Beandefinition, der Konfiguration entsprechend, zu der Beandefinition des Containers hinzugefügt werden. In den Klassen, die sich um die Abarbeitung der Konfigurationsdaten kümmern, werden die jeweils nötigen Schritte durchgeführt. Dabei wird der erste Schritt, also das Erstellen eines Contribution-Containers für jede spezielle Contribution-Liste oder Contribution-Map, nur einmal benötigt. Ist die Beandefinition für eine Contribution-Liste oder Contribution-Map bereits erstellt, so können diese Definitionen jederzeit geladen und mit Element-Definitionen befüllt werden.

6.4.4.1 (XML) Abarbeitung der Konfigurationsschlüsselwörter

Je nach gewünschten Eigenschaften eines Schlüsselwortes in der Konfiguration und auch abhängig von seinen möglichen Positionen im XML-File, bietet *Spring* unterschiedliche Interfaces und abstrakte Klassen an, mit deren Hilfe das Parsen und das Umwandeln der Schlüsselwörter in Beandefinitionen durchgeführt werden kann. Alle, für diesen Zweck entstehenden, Parser Implementierungen können dann in einem eigenen *NamespaceHandler* registriert und somit zur Behandlung der XML-Konfiguration an *Spring* übergeben werden. Der Spring-Parser identifiziert dann über den Namensraum im XML-Konfigurationsfile den *NamespaceHandler* und erhält von diesem die registrierten Parser. Für die SpringContributions wurde der *NamespaceHandler* in der Klasse *ContributionsNamespaceHandler* implementiert. Einen schematischen Überblick zur Namensraumerweiterung gibt Abbildung 6.5.

Abbildung 6.5: Spring-Namensraumerweiterung



Die Beschreibung zur Verarbeitung der Schlüsselwörter beginne ich mit den Wörtern, mit deren Hilfe Contribution-Container in einer XML-Konfiguration referenziert werden können. Die XML-Elemente "contribution-ref" und "mapped-contribution-ref" werden dafür angeboten. Sie können beispielsweise dazu dienen, einen Contribution-Container als Konstruktorargument einem konsumierenden Bean zu übergeben (siehe Absatz 6.3.1). Für die Abarbeitung der beiden XML-Elemente wurde jeweils eine Implementierung der abstrakten Klasse *AbstractSingleBeanDefinitionParser* realisiert. Diese abstrakte Klasse wird von *Spring* angeboten, um einzelne Beandefinitionen aus einer XML-Konfiguration zu gewinnen. Für unserer beiden Referenzierungselemente "contribution-ref" und "mapped-contribution-ref", wollen wir genau das erreichen, und für den Fall, dass der referenzierte Contribution-Container noch nicht existiert, diesen erstellen. Listing 6.18 zeigt als Beispiel die *"parse"* Methode für das XML-Element "mapped-contribution-ref" aus der Klasse

MappedContributionRefBeanDefinitionParser. Die Implementierung für das XML-Element "contribution-ref" in der Klasse *OrderedContributionRefBeanDefinitionParser* ist ähnlich zu dieser und wird daher nicht extra gezeigt.

Listing 6.18: Parser Implementierung für das XML-Element "mapped-contribution-ref"

```
//Auszug aus der Java-Klasse
   MappedContributionRefBeanDefinitionParser

protected void doParse(Element element, ParserContext parserContext,
    BeanDefinitionBuilder builder)
{
    String contributionName = element.getAttribute("name");
    String beanName = MAPPED_CONTRIBUTION_PREFIX + contributionName;
    BeanDefinitionRegistry registry = parserContext.getRegistry();

    if (!registry.containsBeanDefinition(beanName))
    {
        BeanDefinitionBuilder contributionBeanBuilder =
            BeanDefinitionBuilder.rootBeanDefinition(MapFactoryBean.
                class);
        contributionBeanBuilder.addPropertyValue("sourceMap", new
            ManagedMap());
        parserContext.getRegistry().registerBeanDefinition(beanName,
            contributionBeanBuilder.getBeanDefinition());
    }
    builder.addPropertyValue("targetBeanName", beanName);
}
```

Kommen wir nun zu den beiden Schlüsselwörtern "contribution" und "mapped-contribution", die als Beankonfigurationelemente der obersten Ebene im XML-File verwendet werden. Damit es möglich ist, eigene Konfiguration-Metadaten über XML-Elemente der obersten Ebene, also direkt unterhalb des Wurzeltags **<beans>**, zu definieren, bietet *Spring* das Interface *BeanDefinitionParser* an. Implementierungen dieses Interfaces ist es gestattet, während des Parsen des zugeordneten Elements, nicht nur eine sondern beliebig viele Beandefinitionen zu erstellen. Genau das ist es auch, was wir für die beiden Konfigurations-Elemente benötigen, denn in beiden können ja eines oder mehrere Beanelemente definiert werden. Für die Behandlung des Elements "contribution" mit seinem Attribut "to" (siehe Absatz 6.3.3), zur Konfiguration von Contribution-Listen, wurde das *BeanDefinitionParser* Interface in der Klasse *OrderedContributionBeanDefinitionParser* umgesetzt. Für die Konfiguration einer Contribution-Map über das Element "mapped-contribution", ebenfalls mit seinem Attribut "to" (siehe Absatz 6.3.5), wurde dasselbe Interface in der Klasse *MappedContributionBeanDefinitionParser* implementiert. Beide Parser Implementierungen fügen dem jeweils definierten Contribution-Container alle, innerhalb des Konfigurationselements befindlichen, Beandefinitionen hinzu. Zusätzlich kümmern auch sie sich um die

Listing 6.19: *"pars"* Methoden für die XML-Elemente "contribution" und "mapped-contribution"

```
//Auszug aus der Java-Klasse MappedContributionBeanDefinitionParser

public BeanDefinition parse(Element element, ParserContext
    parserContext)
{
    String contributionName = element.getAttribute("to");
    BeanDefinitionBuilder builder =
        BeanDefinitionBuilder.rootBeanDefinition(MapFactoryBean.class
            );
    element.setAttribute("merge", "true");
    Map parsedMap = parserContext.getDelegate()
        .parseMapElement(element, builder.getRawBeanDefinition());

    return MappedContributionUtils
        .addContribution(contributionName, parsedMap, parserContext.
            getRegistry());
}

//Auszug aus der Java-Klasse OrderedContributionBeanDefinitionParser

public BeanDefinition parse(final Element element,
    final ParserContext parserContext)
    {
        String contributionName = element.getAttribute("to");
        NodeList entryList = element
            .getElementsByTagNameNS(CONTRIBUTION_NAMESPACE, "entry");
        List<OrderedContributionBeenContext> beans =
            new ArrayList<OrderedContributionBeenContext>();

        for (int i = 0; i < entryList.getLength(); i++)
        {
            Element entry = (Element) entryList.item(i);
            String name = entry.getAttribute("name");
            String constraints = entry.getAttribute("constraints");
            Object beanValueOrReference = parserContext
                .getDelegate().parsePropertyValue(entry, null, name);
            beans.add(new OrderedContributionBeenContext(name,
                beanValueOrReference, constraints));
        }

    return OrderContributionUtils
        .addToContribution(contributionName, beans, parserContext.
            getRegistry());
}
```

Erstellung der Beandefinition des Contribution-Containers, falls diese noch nicht anderweitig erstellt wurde. Listing 6.19 zeigt die beiden *"parse"* Methoden, in denen jeweils nur die einzelnen Elemente identifiziert und in einer Liste, beziehungsweise in einer Map, abgelegt werden. Der Code zum Erstellen des Contribution-Containers ist unabhängig vom Konfigurationstyp und ist daher in eigenen Hilfsklassen (*OrderContributionUtils*, *MappedContributionUtils*) umgesetzt, die auch von der Java-Variante der Konfiguration verwendet werden. Einen Auszug aus einer dieser Klassen zeigt Listing 6.22.

Listing 6.20: Registrierung eines Beans zu einer Contribution-Liste: Auszug aus dem *ContributionBeanDefinitionDecorator*

```
private void addToContributionService(String contributionName,
    BeanDefinition contribution, ParserContext parserContext)
{
    final String beanName = ORDERED_CONTRIBUTION_PREFIX
        + contributionName;

    final BeanDefinitionRegistry registry =
        parserContext.getRegistry();

    if (registry.containsBeanDefinition(beanName))
    {
        BeanDefinition beanDefinition = parserContext.getRegistry()
            .getBeanDefinition(beanName);

        List list = (List) beanDefinition.getPropertyValues()
            .getPropertyValue("contributionList").getValue();

        list.add(contribution);
    }
    else
    {
        BeanDefinitionBuilder builder =
            BeanDefinitionBuilder.rootBeanDefinition(
                OrderedConfigurationFactoryBean.class
            );
        List contributionList = new ManagedList();
        contributionList.add(contribution);

        builder.addPropertyValue("contributionList",
            contributionList);

        parserContext.getRegistry().registerBeanDefinition(
            beanName, builder.getBeanDefinition()
        );
    }
}
```

Die letzten beiden Schlüsselwörter sind das XML-Attribut "contributeTo" (siehe Listing 6.4) und das XML-Element "contribute" (siehe Listing 6.6). Damit es möglich ist, selbst definierte XML-Attribute und Elemente innerhalb von allen XML-Beandefinitionen zu verwenden, bietet *Spring* das Interface *BeanDefinitionDecorator* an. Alle beiden Schlüsselwörter werden in der *BeanDefinitionDecorator* Implementierung *ContributionBeanDefinitionDecorator* behandelt. Der *ContributionBeanDefinitionDecorator* ist dann dafür zuständig, die eigentliche Beandefinition zu erstellen und in einem *OrderedContribution* Objekt abzulegen. In diesem Objekt werden auch die Parameter des "constraints" Attributs, zur Festlegung der Position des Beans innerhalb der Contribution-Liste, abgelegt. Anschließend wird das Objekt als Element zur Contribution-Liste hinzugefügt und somit in der *OrderedConfigurationFactoryBean* abgelegt. Wenn das entsprechende Bean das erste Element ist, das zur Contribution-Liste hinzugefügt werden soll, dann muss, wie auch bei den anderen Parser-Klassen, vor dem Hinzufügen zuerst noch die Beandefinition der Contribution-Liste selbst angelegt und in der *BeanDefinitionRegistry*, also dem IOC-Container, registriert werden (siehe Listing 6.20). Das heißt also, dass der *ContributionBeanDefinitionDecorator* hauptsächlich Beans zu Contribution-Listen hinzufügt, aber auch selbst Beandefinitionen zu Contribution-Listen erstellt, wenn diese noch nicht existieren und ihnen aber ein Bean als Element hinzugefügt werden soll.

6.4.4.2 (Java) Abarbeitung der Konfigurationsschlüsselwörter

Damit die SpringContributions Funktionalität auch in der Spring Java-Konfiguration verwendet werden kann, steht ein IOC-Postprozessor zur Verfügung (siehe Abschnitt 6.4.2). Dieser *AnnotationContributionPostprocessor* kümmert sich um die Abarbeitung aller Contribution-Konfigurationen aus den bereitgestellten Konfigurationsklassen. Dabei dienen die beiden Annotationen @Contribution und @MappedContribution als Konfigurationsmetainformationen, die von dem Postprozessor abgearbeitet werden. Der Postprozessor implementiert das Spring-Interface *BeanDefinitionRegistryPostProcessor*, welches eine Methode definiert, mit deren Hilfe die, nach der Standardinitialisierung in der IOC-Registry existierenden, Beandefinitionen verändert werden können. Zum Aufrufzeitpunkt dieser Methode existieren also bereits alle Beandefinitionen aus der Konfiguration, es wurde aber noch kein einziges Bean auf Basis dieser Definitionen vom IOC-Container instanziiert. Wie der Postprozessor nun, aus der Java-Konfiguration und den Metainformationen, Beandefinitionen für Contribution-Container und deren Elemente erstellt, ist auszugsweise in Listing 6.21 zu sehen und wird im Folgenden erläutert.

Als erstes werden, in der Methode *postProcessBeanFactory,* alle Konfiguration-Beans geladen. Dabei handelt es sich um Klassen, die über die Annotation @Configuration als Spring Java-Konfiguration ausgezeichnet sind. Diese Konfiguration-Beans werden nun zuerst nach Konfiguration-Methoden durchsucht, die mit der Annotation @Contribution ausgezeichnet sind. In derartigen Konfigurationsklassen sind somit Beans definiert, die zu einer Contribution-Liste hinzugefügt werden sollen. Die Abarbeitung der Annotation ist ebenso in Listing 6.21 in der Methode *handleOrderedContributions* dargestellt. In dieser Methode werden die Informationen zum Namen der Contribution-Liste ("to" Parameter) und zur Positionierung in der Liste ("constraints" Parameter) ausgelesen. Nach dem Auf-

Listing 6.21: Auszug aus dem *AnnotationContributionPostprocessor*

```
public void postProcessBeanFactory(ConfigurableListableBeanFactory
    beanFactory) throws BeansException
{
    Map<String, Object> beans =
        beanFactory.getBeansWithAnnotation(Configuration.class);
    for(String configuration : beans.keySet())
    {
        BeanDefinition beanDefinition =
            beanFactory.getBeanDefinition(configuration);
        if (beanDefinition instanceof AnnotatedBeanDefinition)
        {
            AnnotatedBeanDefinition beanDef = (
                AnnotatedBeanDefinition) beanDefinition;
            Set<MethodMetadata> orderedContributions =
                beanDef.getMetadata().getAnnotatedMethods(
                    Contribution.class.getName());
            handleOrderedContributions(beanFactory,
                orderedContributions);

            Set<MethodMetadata> mappedContributions =
                beanDef.getMetadata().getAnnotatedMethods(
                    ContributionMapped.class.getName());
            handleMappedContributions(beanFactory,
                mappedContributions);
        }
    }
}

private void handleOrderedContributions(
    ConfigurableListableBeanFactory beanFactory, Set<MethodMetadata>
        orderedContributions)
{
    for(MethodMetadata metadata : orderedContributions)
    {
        Map<String, Object> attributes = metadata.
            getAnnotationAttributes(Contribution.class.getName());
        String contributionName = (String)attributes.get("to");
        String constraints = (String)attributes.get("constraints");
        String beanName = getBeanName(metadata, attributes);
        Object beanValueOrReference = beanFactory.getBeanDefinition(
            beanName);
        OrderedContributionBeenContext beanContext =
            new OrderedContributionBeenContext(
                beanName, beanValueOrReference, constraints);
        OrderContributionUtils.addToContribution(contributionName,
            beanContext, (BeanDefinitionRegistry)beanFactory);
    }
}
```

ruf der Methode sucht der Postprozessor die Konfiguration-Methoden, die mit der Annotation @ContributionMapped ausgezeichnet wurden. Ergebnis dieser beiden Schritte sind in der IOC-Registry registrierte Beandefinitionen zu Contribution-Containern und deren Elementen. Der Code zum Erstellen des Contribution-Containers ist unabhängig vom Konfigurationstyp und ist daher in eigenen Hilfsklassen (*OrderContributionUtils*, *MappedContributionUtils*) umgesetzt, die auch von der XML-Variante der Konfiguration verwendet werden. Einen Auszug aus einer dieser Klassen zeigen Listing 6.22.

6.4.4.3 Gemeinsam verwendete Util-Klassen

Die beiden Hilfsklassen *OrderContributionUtils* und *MappedContributionUtils* werden sowohl beim Abarbeiten einer XML-Konfiguration (siehe Abschnitt 6.4.4.1), als auch beim Abarbeiten der Java-Konfiguration (siehe Abschnitt 6.4.4.2) verwendet. Diese Synergie ist möglich, da in beiden Fällen, nach dem unterschiedlichen Weg der Gewinnung der Konfigurationsinformationen, genau der selbe Schritt durchgeführt werden muss. Bei diesem Schritt handelt es sich um die finale Registrierung der Beandefinitionen für Contribution-Container und deren Elemente.

Listing 6.22: Auszug aus der Klasse *MappedContributionUtils*

```
//Auszug aus der Java-Klasse MappedContributionUtils

public static BeanDefinition addToContribution(
    String contributionName, Map map, BeanDefinitionRegistry registry
        )
{
    final String beanName =
        MAPPED_CONTRIBUTION_PREFIX + contributionName;
    BeanDefinition beanDefinition;
    if (registry.containsBeanDefinition(beanName))
    {
        beanDefinition = registry.getBeanDefinition(beanName);
        beanDefinition.getPropertyValues().addPropertyValue("
            sourceMap", map);
    }
    else
    {
        BeanDefinitionBuilder builder =
            BeanDefinitionBuilder.rootBeanDefinition(MapFactoryBean.
                class);
        builder.addPropertyValue("sourceMap", map);
        beanDefinition = builder.getBeanDefinition();
        registry.registerBeanDefinition(beanName, beanDefinition);
    }
    return beanDefinition;
}
```

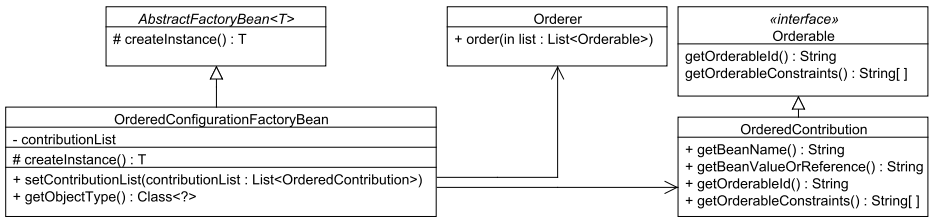
In Listing 6.22 ist zu sehen, wie zuerst versucht wird, die Beandefinition des Contribution-Containers aus der IOC-Registry zu bekommen oder diese zu erstellen, falls sie noch nicht existiert. Danach werden die einzelnen Elemente, die hier in einer *Map* enthalten sind, in dem Container abgelegt.

6.4.4.4 Sortierung von Contribution-Listen

Um die Sortierung der Beans in einer Contribution-Liste zu ermöglichen, implementiert die Klasse *OrderedContribution* das Interface *Orderable*. Dadurch kann die *OrderedConfigurationFactoryBean*, mit Hilfe der Klasse *Orderer*, die einzelnen Positionierungsangaben der Listenelemente berücksichtigen (siehe Abbildung 6.6).

Wenn nun ein Bean, das eine Contribution-Liste benötigt, von der Spring IOC-Registry instanziiert wird, so bedient sich diese der *OrderedConfigurationFactoryBean,* um die Contribution-Liste zu erhalten.

Abbildung 6.6: Contribution-Liste



6.4.4.5 Contribution-Bean Identifikation

Egal über welche Art der Konfiguration ein Contribution-Container erstellt wird, die SpringContributions versehen dessen Namen beziehungsweise dessen Identifikation in der Beandefinition mit einem bestimmten Präfix. Diese Maßnahme soll helfen, Namenskollisionen mit anderen Beandefinitionen zu vermeiden. Namenskollisionen würden entstehen, wenn zwei Beandefinitionen den selben Namen beziehungsweise die selbe Identifikation innehaben. Eine solche Namenskollision führt in *Spring* zu einem Fehler, da der IOC-Container nicht mehr zwischen den beiden Beandefinitionen unterscheiden kann. Daher gilt es, bei der Verwendung von *Spring,* auf die Verschiedenheit der Identifikation von Beandefinitionen zu achten. In kleinen Software-Projekten ist das relativ leicht zu bewerkstelligen. Mit wachsender Anzahl von Beandefinitionen erhöht sich aber generell auch die Möglichkeit von Namenskollisionen. Sobald ein Software-Projekt über seine eigenen Modulgrenzen hinaus noch fremde, auf *Spring* basierende Module mit einbindet, existiert bezüglich der Namenskollisionen eine externe, nicht beeinflussbare Fehlerquelle. Das resultiert daraus, da man auf die Namensgebung in Fremdmodulen keinen Einfluss hat.

Das Spring-Framework bietet für diese Problematik keine automatisierte Hilfestellung oder Problemlösung an. Beans erhalten im IOC-Container immer genau die Identifikation, die in der Konfiguration vom Benutzer festgelegt wurde. Die Modulzugehörigkeit oder eventuelle Packetinformationen finden hier keine Berücksichtigung. Das hat den Vorteil, dass der Benutzer sich nicht mit irgendwelchen "magischen" Namensgebungen herumschlagen muss. Auf der anderen Seite ist der Benutzer des Spring-Frameworks hier in der Pflicht, sich eigene Maßnahmen zu dieser Problematik zu überlegen. In der Regel definiert man zur Vermeidung von Namenskollisionen eigene Namenskonventionen, wie zum Beispiel das Voranstellen des Modulnamen bei jeder Beanidentifikation. Die SpringContributions gehen hier mit der automatischen Vergabe des Identifikationspräfix einen anderen Weg als es das Spring-Framework macht. Listing 6.23 zeigt die beiden Präfixdefinitionen, auf die in Verschiedenen Listings zur Konfiguration verwiesen wird.

Listing 6.23: (Java) Namenspräfix für Contribution-Container Definitionen

```
/**
* Prefix for mapped contributions.
*/
public static final String MAPPED_CONTRIBUTION_PREFIX =
    "org.springframework.contributions.mapped.";

/**
* Prefix for ordered contributions.
*/
public static final String ORDERED_CONTRIBUTION_PREFIX =
    "org.springframework.contributions.ordered.";
```

6.5 Anwendungsbeispiele

In den folgenden Anwendungsbeispielen folgt den Listings mit XML-Konfiguration sogleich ein Listing mit äquivalenter Java-Konfiguration. Dies soll der Vergleichbarkeit der beiden Konfigurationsmethoden dienen. Die beiden Konfigurationen werden danach in einem Unit-Test verwendet, in dem die Contribution in ihrer Verwendung gezeigt wird.

6.5.1 (XML / Java) ValueHolder-Service und sortierte Contribution-Liste

Ein Anwendungsbeispiel des SpringContributions Mechanismus wird in den Listings 6.24 bis 6.26 gegeben.

Dabei handelt es sich in Listing 6.24 um eine XML-Konfiguration einer Contribution-Liste und ihres Konsumenten. In Listing 6.25 wird, als Gegenüberstellung zur XML-Konfiguration, die analoge Java-Konfiguration angegeben.

Listing 6.24: (XML) Spring-Bean und Sortierte Contribution-Liste Test-Konfiguration

```
<!-- (XML) Bean Konfig des Konsumenten -->
<bean class="org.springframework.contributions.ValueHolder">
    <constructor-arg>
        <ctr:contribution-ref name="value-list" />
    </constructor-arg>
</bean>

<!-- (XML) Konfiguration der Contribution-Listen Elemente -->
<ctr:contribution to="value-list">
    <ctr:entry name="string2" value="String␣2"
        constraints="after:string1" />
    <ctr:entry name="string1" value="String␣1"
        constraints="before:*" />
</ctr:contribution>
```

Listing 6.25: (Java) Spring-Bean und Sortierte Contribution-Liste Test-Konfiguration

```
// (Java) Bean Konfig des Konsumenten
@Bean(name="valueHolder") public ValueHolder valueHolder(
    OrderedContributionResolver<String> values)
{
    return new ValueHolder(values.resolve("value-list"));
}

// (Java) Konfiguration der Contribution-Listen Elemente
@Contribution(to="value-list", constraints="after:string1")
@Bean(name="string2") public String string2()
{
    return "String 2";
}

@Contribution(to="value-list", constraints="before:*")
@Bean(name="string1") public String string1()
{
    return "String 1";
}
```

In dem Beispiel wird ein *ValueHolder* Service als Spring-Bean sowie eine sortierte Contribution-Liste mit dem Namen *"value-list"* definiert werden. Das *ValueHolder* Service erwartet die Contribution-Liste als Konstruktor-Argument. Innerhalb der Contribution-Liste sind zwei Strings ("*String 1*", "*String 2*") als Elemente definiert. Jedem Element

wird durch seine jeweiligen "*constraints*" Einstellungen eine bestimmte Position innerhalb der Liste zugewiesen.

Im Listing 6.26 findet sich zum Einen die Implementierung des *ValueHolder* Service. Zum Anderen wird die zuvor definierte Spring-Konfiguration in einem Unit-Test (*OrderedContributionIntegrationTest*) verwendet.

Listing 6.26: *ValueHolder* Service und Ordered Contribution als JUnit-Test

```
// Java-Klasse ValueHolder:

public class ValueHolder
{
    private final List<String> values;
    public ValueHolder(final List<String> values)
    {
        this.values = values;
    }

    public List<String> getValues()
    {
        return values;
    }
}

// JUnit-Test OrderedContributionIntegrationTest:

// hier wird das XML-Konfigurationsfile angegeben
// oder die äquivalente Java-Konfiguration als Klasse
// z.B. OrderedContributionIntegrationTestConfiguration.class
@ContextConfiguration("classpath:spring-contributions-ordered.xml")
@RunWith(SpringJUnit4ClassRunner.class)
public class OrderedContributionIntegrationTest
{
    @Inject
    @Named("valueHolder")
    private ValueHolder stringHolder;

    @Test
    public void testStringContribution()
    {
        assertThat(stringHolder.getValues(),
            is(Arrays.asList("String 1", "String 2")));
    }
}
```

Innerhalb des Unit-Tests wird die Konfiguration über die Spring-Annotation *@Context-Configuration* als Spring-Kontext in den Test mit eingebunden. Der Unit-Test überprüft

dann die richtige Reihenfolge der einzelnen Strings innerhalb der Contribution-Liste. Die über die Konfiguration festgelegte Reihenfolge, also "String 1" vor allen anderen Elementen (before:*) und "String 2" nach dem Bean mit der id "string1" (after:string1) wird überprüft. Das erwartete Ergebnis wird in der aktuellen Version der SpringContributions auch geliefert. Der Unit-Test zeigt nicht nur die Funktionsweise der SpringContributions, sondern stellt bei zukünftigen Adaptierungen bzw. Weiterentwicklungen des Moduls auch die Korrektheit der Änderungen sicher.

6.5.2 Contributions mehrfach konsumieren

Ein Vorteil gegenüber dem in *Tapestry* angebotenen Contribution System ergibt sich in den SpringContributions aus dem Umstand, dass jede Contribution als eigenes Bean behandelt wird. Ein Contribution-Container-Bean kann also nicht nur von einem Konsumenten verwendet werden, sondern steht, wie jedes andere Bean, zur beliebigen Verwendung bereit. Nehmen also zum Beispiel zwei Services eine Liste von Klassen des

Listing 6.27: (Java) Konfiguration mit Mehrfachverwendung einer Contribution-Liste

```
@Bean(name="singletonValueHolderOne")
public ValueHolder singletonValueHolderOne(
    OrderedContributionResolver<String> values)
{
    return new ValueHolder(values.resolve("singletonValues"));
}

@Bean(name="singletonValueHolderTwo")
public ValueHolder singletonValueHolderTwo(
    OrderedContributionResolver<String> values)
{
    return new ValueHolder(values.resolve("singletonValues"));
}

@Contribution(to="singletonValues")
@Bean
public String singletonOne()
{
    return "original";
}

@Contribution(to="singletonValues")
@Bean
public String singletonTwo()
{
    return "values";
}
```

selben Typs als Servicekonfiguration entgegen, so muss jedes Element dieser Liste, sofern sich die Liste in ihrer Zusammenstellung nicht unterscheiden soll, nur ein einziges Mal konfiguriert werden. Es ist also egal, von wie vielen Beans die Contribution anschließend verwendet wird. In *Tapestry* müsste man für dasselbe Beispiel jedem Service eine eigene Contribution-Liste übergeben. Dafür wäre es notwendig, jedes Element für jede dieser Listen zu konfigurieren. Listing 6.27 zeigt eine Java-Konfiguration in der zwei Services ein und die selbe Contribution-Liste verwenden.

Gefahren durch mehrfaches Konsumieren

Natürlich birgt die Möglichkeit der uneingeschränkten Verwendung von Contributions durch mehrere Konsumenten auch Risiken, auf die in der Verwendung acht gegeben werden muss. So muss man in der derzeitigen Umsetzung der SpringContributions beachten, dass ein Bean eines Contribution-Containers als einzelnes Objekt existiert und somit dessen Zustand, also auch die Art und Anzahl seiner Elemente, von jedem Konsumenten verändert werden kann. Siehe dazu mehr im Abschnitt 7.

Ein Beispiel für diese Problematik zeigt der Test aus Listing 6.28. Dabei sind für den Test zwei Services konfiguriert, die beide auf den selben Contribution-Container zugreifen. Die Contribution-Liste besteht dabei aus Elementen, die einen einfachen Text enthalten. Der eine der beiden Services ist ein *ValueHolderConsumer* und gibt beim Aufruf seiner *"consume"* Methode die Texte seiner Contribution-Elemente durch ein Leerzeichen getrennt zurück. Der zweite Service ist ein *ValueHolderManipulator* und fügt den Elementen der Contribution-Liste ein weiteres hinzu, das den Text "manipulated" enthält. Der

Listing 6.28: JUnit-Test zur Veränderung des Contribution-Container Zustands

```
//Auszug aus einem JUnit-Test

@Inject
@Named(value="singletonValueHolderConsumer")
private ValueHolderConsumer singletonConsumer;

@Inject
@Named(value="singletonValueHolderManipulator")
private ValueHolderManipulator singletonManipulator;

@Test
public void testUsageAndManipulationWithSingletonContribution()
{
    String beforeManipulation = singletonConsumer.consume();
    singletonManipulator.manipulate();
    String afterManipulation = singletonConsumer.consume();
    assertEquals("original values ", beforeManipulation);
    assertEquals("original values manipulated ", afterManipulation);
}
```

Test ruft nun zuerst die *"consume"* Methode auf und bekommt dabei den Text "original values " zurück. Danach wird die *"manipulate"* Methode aufgerufen. Ein erneuter Aufruf der *"consume"* Methode liefert nun den Text "original values manipulated " zurück. Der Test zeigt also, dass das Verändern des Zustands der Contribution-Liste durch einen Konsumenten auch eine Auswirkung auf alle andere Konsumenten hat. Wie diese Problematik in der Konfiguration umgangen werden könnte, zeigt der Ausblick auf mögliche Erweiterungen für die SpringContributions im Abschnitt 7.

6.6 "Strategy" Service

Im letzten Abschnitt wurde die Adaption des von *Tapestry* bekannten Contribution-Mechanismus für das Spring-Framework behandelt. Damit haben wir jetzt die Grundlage geschaffen, um uns einem weiteren Schritt zu widmen. Einer der, auf jeden Fall in der Porsche Informatik, am meisten genutzten Dienste, den das Tapestry-Framework aufbauend auf dem Contribution-Mechanismus anbietet, ist der des "*StrategyBuilder*". Aus diesem Grund ist es ein notwendiges Ziel des praktischen Teils dieser Arbeit, auch diesen Dienst innerhalb der SpringContributions anzubieten. In diesem Abschnitt sollen nun die Funktionsweise dieses Dienstes beschrieben und seine Adaption für *Spring* gezeigt werden.

Als Ausgangspunkt sei hier die Grundlage des Dienstes erwähnt. In "*Design Patterns*" [11] wurde 1995 eine Sammlung von Designmustern veröffentlicht deren Verwendung sich im Allgemeinen positiv auf die Qualität von Software auswirken. Das "*Strategy Pattern*", auf dem der Tapestry-Dienst *StrategyBuilder* basiert, ist dabei eines dieser Muster und dient dazu,

> eine Familie von Algorithmen zu definieren, wobei es einen jeden kapselt und austauschbar macht. Eine Strategy ermöglicht den Klienten, die sie verwenden, den Algorithmus unabhängig zu variieren. [11]

6.6.1 Tapestry-Variante

Tapestry bietet mit dem *StrategyBuilder* Service einen Dienst an, der das *Strategy Pattern* auf eine sehr intelligente Weise erweitert. Die Konfiguration, also welche Algorithmen in einer Strategy zur Verfügung stehen, geschieht in *Tapestry* natürlich via Contribution. Das bedeutet, dass eine gewisse Anzahl an unterschiedlichen Strategien - damit sind gekapselte Algorithmen gemeint - dem Strategy-Service übergeben werden. Die Entscheidung eines Klienten, welche Strategie er verwenden will, wird in *Tapestry* über Java-Klassen gesteuert. Das heißt, jeder Strategie ist als Schlüssel eine Java-Klasse zugeordnet, über die sie eindeutig bestimmt werden kann. Die Zuordnung von Java-Klasse und Strategie bedeutet im Detail, dass die Strategie einen Algorithmus kapselt, der Objekte vom Typ der Java-Klasse verarbeiten kann. Die Wörter Schlüssel und Contribution lassen hier gleich erkennen, dass sich der *Tapestry* *StrategyBuilder* eine Mapped-Configuration 4.3 als Konfiguration erwartet.

Die "Intelligenz" des Strategy-Services zeigt sich in einem weiteren Detail. Will ein Klient eine Strategie auf ein Objekt anwenden, für dessen Klassen-Typ eigentlich keine Strategie konfiguriert wurde, so verhält sich das Stretegy-Service in *Tapestry* folgendermaßen:

Wenn also für den konkreten Klassen-Typ des Objekts keine Strategie gefunden wurde, so wird für jede Superklasse geschaut, ob eine Strategie verfügbar ist. Die Suche wird für die komplette Hierarchie der Superklassen fortgesetzt, bis entweder eine Strategie gefunden wurde und diese auf das Objekt angewandt werden kann, oder bis man beim Klassen-Typ *Object* angelangt ist. Ist für diesen letzten Typ auch keine Strategie registriert, so wird die Suche abgebrochen.

6.6.2 SpringContributions Variante

Um für die SpringContributions eine eigene Variante des *Strategy Pattern* umsetzen zu können, war natürlich zuerst die Voraussetzung der Konfigurierbarkeit einer Contribution-Map zu realisieren. Danach musste mit der *StrategyFactoryBean* nur noch eine eigene *FactoryBean* implementiert werden, die dem Klienten die gesuchte Strategie als Bean instanziiert und zur Verfügung stellt. Implementierungen des Interfaces *FactoryBean* werden in *Spring* verwendet, um Beans zu instanziieren. Ein wichtiges Ziel des praktischen Teils dieser Arbeit war, die Tapestry-Funktionalität innerhalb des Spring-Frameworks exakt nach zu empfinden (siehe Kapite 1). Um also sicherzustellen, dass sich die SpringContributions Variante, im Bezug auf die Suche nach der passenden Strategie für ein Objekt, nicht von der Tapestry-Variante unterscheidet, wurden die dafür zuständigen Tapestry-Klassen als Kopien in das SpringContributions Projekt übernommen. Diese Klassen finden dann innerhalb der *StrategyFactoryBean* ihre Verwendung.

Beispiel: Um nun eine bessere Vorstellung vom *Strategy Pattern* in der *Tapestry* bzw. SpringContributions Variante zu erhalten, folgt hier nun ein Beispiel für dessen Anwendung anhand eines Unit-Tests. Die Listings 6.30 und 6.31 zeigen eine XML-Konfiguration und deren Java-Äquivalent. In Listing 6.32 ist der zugehörige Unit-Test zu sehen.

Das Beispiel zeigt, wie das *Strategy Pattern* für die Anwendung unterschiedlicher Strategien zur Formatierung von Strings, verwendet werden kann. Dabei werden drei von einander verschiedene Algorithmen als Strategien konfiguriert. Jede dieser Strategien benötigt nun auch eine eigene Java-Klasse, die ihr, zur Identifizierung in der Contribution-Map, als Schlüssel zugewiesen wird. Ein Objekt dieser Schlüsselklasse wird in der Anwendung dann auch den zu formatierenden String enthalten.

- Die erste konfigurierte Strategie *UpperCaseStringFormatStrategy* (siehe Listing 6.29), mit Schlüsselklasse *UpperCase,* soll einen String in eine Großbuchstabendarstellung umwandeln.
- Die zweite Strategie *BlankSeparatedStringFormatStrategy*, mit Schlüsselklasse *BlankSeparated,* ändert einen String, indem sie nach jedem Zeichen ein Leerzeichen einfügt.

- Die dritte Strategie *ReversedStringFormatStrategy*, mit Schlüsselklasse *Reversed,* dreht bei ihrer Anwendung den String so um, dass er von links nach rechts geschrieben wird.

Listing 6.29: *UpperCaseStringFormatStrategy* mit Schlüsselklasse *UpperCase*

```
// Java-Klasse StringContainer Interface:
public interface StringContainer
{
    String getStingValue();
}

// Java-Klasse UpperCase als Schlüsselklasse zur
   UpperCaseStringFormatStrategy

public class UpperCase implements StringContainer
{
    private final String value;
    public UpperCase(String value)
    {
        this.value = value;
    }

    public String getStingValue()
    {
    return value;
    }
}

// Java-Klasse StringFormatStrategy:
public interface StringFormatStrategy<T extends StringContainer>
{
    String format(T valueContainer);
}

// Java-Klasse UpperCaseStringFormatStrategy
public class UpperCaseStringFormatStrategy implements
   StringFormatStrategy<UpperCase>
{
    public String format(UpperCase valueContainer)
    {
        return valueContainer.getStingValue().toUpperCase();
    }
}
```

XML-Konfiguration In der XML-Konfiguration 6.30 wird als erstes ein Bean vom Typ *StrategyFactoryBean* konfiguriert. Dieses Bean nimmt über den Konstruktor eine Contribution-Map entgegen. Die drei Elemente der Contribution-Map, also die drei Strategien zur String-Formatierung, werden gleich darauf, im selben XML-File, zuerst als Beans konfiguriert und dann der Contribution-Map hinzugefügt.

Listing 6.30: (XML) Konfiguration einer Strategie in SpringContributions

```
<bean id="stringFormatStrategy" class="org.springframework.
    contributions.ioc.services.strategy.StrategyFactoryBean">
    <constructor-arg value="org.springframework.contributions.
        strategy.StringFormatStrategy" />
    <constructor-arg>
        <ctr:mapped-contribution-ref name="stringFormatStrategies" />
    </constructor-arg>
</bean>

<bean id="upperCase" class="org.springframework.contributions.
    strategy.strategies.UpperCaseStringFormatStrategy" />
<bean id="blankSeparated" class="org.springframework.contributions.
    strategy.strategies.BlankSeparatedStringFormatStrategy" />
<bean id="reversed" class="org.springframework.contributions.strategy
    .strategies.ReversedStringFormatStrategy" />

<ctr:mapped-contribution to="stringFormatStrategies">
    <entry>
        <key>
            <value>org.springframework.contributions.strategy.
                entities.UpperCase</value>
        </key>
        <ref bean="upperCase" />
    </entry>
    <entry>
        <key>
            <value>org.springframework.contributions.strategy.
                entities.BlankSeparated</value>
        </key>
        <ref bean="blankSeparated" />
    </entry>
    <entry>
        <key>
            <value>org.springframework.contributions.strategy.
                entities.Reversed</value>
        </key>
        <ref bean="reversed" />
    </entry>
</ctr:mapped-contribution> ...
```

Java-Konfiguration Die Java-Konfiguration 6.31 ist äquivalent zur XML-Konfiguration. Es wird ebenfalls das Bean vom Typ *StrategyFactoryBean* konfiguriert, gefolgt von der Konfiguration der benötigten Contribution-Map und ihrer Elemente. Im Unit-Test 6.32 wird nun diese Java-Konfiguration verwendet.

Listing 6.31: (Java) Konfiguration einer Strategy in SpringContributions

```
@Configuration
@EnableContributions
@EnableAspectJAutoProxy
public class StrategyFactoryBeanIntegrationTestConfiguration
{
    @Bean(name="stringFormatStrategy")
    public StrategyFactoryBean strategyCaller(
        MappedContributionResolver<Map<Class, Strategy>> strategies)
    {
        return new StrategyFactoryBean(StringFormatStrategy.class,
            strategies.resolve("stringFormatStrategies"));
    }

    @ContributionMapped(to="stringFormatStrategies", keyClass=
        UpperCase.class)
    @Bean(name="upperCase")
    public StringFormatStrategy<UpperCase> upperCase()
    {
        return new UpperCaseStringFormatStrategy();
    }

    @ContributionMapped(to="stringFormatStrategies", keyClass=
        BlankSeparated.class)
    @Bean(name="blankSeparated")
    public StringFormatStrategy<BlankSeparated> blankSeparated()
    {
        return new BlankSeparatedStringFormatStrategy();
    }

    @ContributionMapped(to="stringFormatStrategies", keyClass=
        Reversed.class, bean="reversed")
    @Bean(name="reversed")
    public StringFormatStrategy<Reversed> reversed()
    {
        return new ReversedStringFormatStrategy();
    }
}
```

Anwendung des *Strategy Pattern*

In Listing 6.32 wird nun die Spring Java-Konfiguration aus Listing 6.31 über die Referenzierung der Klasse *StrategyFactoryBeanIntegrationTestConfiguration* verwendet.

Die Instanz der benötigten *StringFormatStrategy* wird über die Annotationen @Inject und @Named über *Dependency Injection* aus dem Spring IOC-Container geholt. Dabei wird in der Annotation @Named der Name des konfigurierten Bean "stringFormatStrategy" angegeben, damit der IOC-Container exakt Bescheid weiß, welches Bean er injizieren soll. Die Anwendung der vorhandenen Strategien wird in der Testmethode gezeigt. Dabei wird der String "foobar" dreimal in einen jeweils anderen *StringContainer* gepackt, der immer als Schlüsselklasse für eine der Strategien fungiert. Mit diesem Container, wird dann der String an die *"format"* Methode *StringFormatStrategy* übergeben, wo dann die entsprechende Strategie gesucht und angewandt wird. Der Test zeigt, dass der Text je nach gewünschter Strategie formatiert wird. So wird also aus einem "foobar" im *UpperCase* Container, bei Anwendung der Strategie, ein "FOOBAR".

Listing 6.32: Strategy Anwendung als JUnit-Test

```
// hier wird die JAVA-Konfiguration als Klasse angegeben
@ContextConfiguration(
    classes={StrategyFactoryBeanIntegrationTestConfiguration.class},
    loader=AnnotationConfigContextLoader.class)
public class AnnotationStrategyFactoryBeanIntegrationTest
{
    @Inject
    @Named("stringFormatStrategy")
    private StringFormatStrategy<StringContainer>
       stringFormatStrategy;

    @Test
    public void testStringFormatStrategy()
    {
        final String value = "foobar";
        StringContainer upperCase = new UpperCase(value);
        assertThat(stringFormatStrategy.format(upperCase),
            is("FOOBAR"));

        StringContainer reversed = new Reversed(value);
        assertThat(stringFormatStrategy.format(reversed),
            is("raboof"));

        StringContainer blankSeparated = new BlankSeparated(value);
        assertThat(stringFormatStrategy.format(blankSeparated),
            is("f o o b a r"));
    }
}
```

7 Zusammenfassung und Ausblick

Im praktischen Teil zu dieser Diplomarbeit ist es gelungen mit den SpringContributions, ein Spring-Softwaremodul zu entwickeln, mit dessen Hilfe eine Spring-Anwendung um einen Contribution-Mechanismus, nach dem Vorbild des Apache Tapestry-Frameworks, erweitert wird. Dieser Mechanismus bietet die Möglichkeit, modulübergreifende *Dependency Injection* für sogenannte Container-Beans in der Form von zwei komplexen Datentypen, dem der Liste und dem der Map, anzuwenden. Die Besonderheit an dem System liegt darin, dass die Elemente dieser, für die *Dependency Injection* vorgesehenen, Container-Beans in den verschiedensten, auch durch Modulgrenzen von einander getrennten Softwarekonfigurationen definiert sein können.

Unter Verwendung der SpringContributions ist es nun also noch einfacher, im Zusammenspiel mit dem Spring-Framework, hoch konfigurierbare, multimodulare Softwareprojekte zu realisieren. Gleichzeitig besteht nun die Möglichkeit, eine auf dem Tapestry-Framework basierende Softwarearchitektur, ohne Einbußen in der Konfigurierbarkeit, mit *Spring* zu kombinieren. Darüber hinaus ist es über die Verwendung der SpringContributions natürlich auch möglich, eine in *Tapestry* entwickelte und auf dem Contribution-Mechanismus basierende Softwarekomponente von Tapestry-IOC auf Spring-IOC zu migrieren, ohne dabei die Konfigurierbarkeit der Software grundlegend anpassen zu müssen. Dies alles öffnet also den Weg dahin, zwei äußerst hilfreiche Frameworks effizient gemeinsam nutzen zu können.

Die Synergien der beiden Frameworks werden durch den Einsatz der SpringContributions in der Porsche Informatik bereits seit zwei Jahren produktiv genutzt. Die Version 1.1 der SpringContributions kommt hier innerhalb einer B2B Webapplikation zum Gebrauchtwagenverkauf, in einer Verrechnungssoftware und in der Neuentwicklung eines Händlersystems zum Einsatz. In dieser Version ist bereits die komplette XML-Konfiguration inklusive der *Strategy Pattern* Unterstützung umgesetzt. Damit war es möglich, die zuerst nur mit *Tapestry* entwickelte Webapplikation um den Funktionsumfang von *Spring* zu erweitern. Die Weiterentwicklung der SpringContributions mit der ergänzten Möglichkeit der Java-Konfiguration ist ab der aktuellen Version 1.2 im Projekt enthalten. Ein Wechsel auf die aktuelle Version ist in der Porsche Informatik bereits geplant, da in der Entwicklung der B2B Webapplikation die Verwendbarkeit von Springs Java-Konfiguration benötigt wird. Das SpringContributions Projekt ist als *Maven*[1] Projekt entwickelt und wird bei Versionsrelease in öffentlichen Mavenrepositories für die freie Verwendung zur Verfügung gestellt. Das SpringContributions Projekt wird auf

[1] *Apache Maven* ist ein Build-Management Werkzeug zum standardisierten Erstellen von Java-Programmen

Googlecode gehostet[2] und ist dort unter der *Apache License, Version 2.0*[3] veröffentlicht.

Bei der Entwicklung des SpringContributions Moduls wurde versucht, den gesamten Spielraum des aus dem Tapestry-Vorbild bekannten Konfigurationsmechanismus nachzubilden. Was die Funktionsweise betrifft wurde dieses Ziel im Rahmen des Praxisteils dieser Arbeit auch erreicht. Was den Komfort bezüglich der Schreibweise der SpringContributions Konfiguration betrifft, so besteht gerade für die Variante der Java-Konfiguration noch eine Verbesserungsmöglichkeit. Außerdem besteht generell noch Erweiterungspotential wenn es um die Sichtbarkeit von Contribution-Containern geht. Diese beiden Aspekte werden nun noch kurz beleuchtet.

Konfiguration und Übergabe mehrerer Elemente an einen Contribution-Container

In der aktuellen Version der SpringContributions, ist es in der Java-Konfiguration nur möglich ein Bean pro Konfirurationsmethode zu definieren. Das kommt daher, da hier die Basisfunktionalität von *Spring* verwendet wird und hier nicht mehrere Beans auf einmal definiert werden können. Dadurch kann natürlich auch immer nur ein Bean über die Annotierung seiner Konfigurationsmethode einem Contribution-Container hinzugefügt werden. Wollte man hier den selben Komfort, wie er aus dem Tapestry-Framework bekannt ist und wie er auch in der XML-Variante der Konfiguration in den SpringContributions umgesetzt ist, erreichen, so müsste das grundsätzliche Prinzip des Eingriffs in den Spring-Kofigurationsablauf umgestellt werden. Anstelle eines Beandefinition Postprozessing, über das die Contribution-Kofiguration im Moment umgesetzt ist, müsste man in den eigentlichen Prozess der Spring Beankonfiguration eingreifen (siehe Absatz 6.2.2). Dies würde bedeuten, dass man den gesamten bereits von *Spring* umgesetzten Prozess der Beandefinition-Erstellung für Java-Konfigurationen erneut selbst entwickeln müsste. Hierbei wäre es dann möglich, das Augenmerk darauf zu legen, eigens für die Contribution-Kofiguration geeignete Konfigurationsmethoden zu entwickeln, mit denen es auch möglich wäre, mehrere Beandefinitionen auf einmal anzulegen und dann an einen Contribution-Container zu übergeben.

Bei dieser Verbesserung müsste man aber mehrere Problemstellungen überwinden. Da der Prozess der Erstellung von Beandefinitionen nicht trivial ist, wäre es hier anzuraten, möglichst viel Funktionalität von *Spring* wiederzuverwenden, beziehungsweise zu erweitern. Leider eignet sich die derzeitige Implementierung des Fameworks in dem Bereich nicht gut zur Wiederverwendung oder Erweiterung. Da die derzeitige Codebasis im Spring-Framework hier leider nicht leicht adaptierbar ist, müsste also sehr viel Funktionalität erneut geschrieben werden. Dadurch entsteht dann aber auch die Notwendigkeit, in diesem neuen Code, mit jeder neuen Spring-Version, die von den SpringContributions unterstützt werden soll, auf eventuelle Änderungen in der *Spring* eigenen Erstellung von Beandefinitionen zu achten, und derartige Änderungen ebenso umzusetzen.

[2] Die SpringContributions Projektseite ist unter folgender URL zu erreichen (10.12.2013): https://code.google.com/p/spring-contributions/

[3] Die *Apache License, Version 2.0* ist unter folgender URL zu erreichen (10.12.2013): http://www.apache.org/licenses/LICENSE-2.0

Konfigurierbarkeit der Sichtbarkeit für Contribution-Container-Beans

In *Spring* werden alle Bean-Objekte durch sogenannte *BeanFactory* Klassen erstellt. Eine *BeanFactory* liefert also auf Wunsch eine Instanz des gewünschten Beans. Da es sich bei Beans ja auch nur um gewöhnliche Java-Objekte handelt, sind hier augenscheinlich zwei grundsätzlich unterschiedliche Strategien möglich. Zum einen kann die *BeanFactory* immer eine neue Instanz erstellen und dem jeweiligen Konsumenten übergeben, wenn eine Anfrage nach einem Bean eines bestimmten Typs gemacht wird. Zum anderen ist es möglich, dass die *BeanFactory* einmalig eine Instanz des Beans erstellt und diese Instanz bei jedem Aufruf durch einen Konsumenten an diesen weitergibt.

Ein Konsument eines Beans erhält also im ersten Fall immer eine "eigene" Instanz des gewünschten Beans. Alle Zugriffe auf dieses Bean-Objekt liegen nun einzig in der Hand des Konsumenten. Diese Art der Sichtbarkeit (Scope) ist der Standardfall in *Spring* und wird hier mit dem "Prototype Scope" bezeichnet. Bei der zweiten Möglichkeit existiert nur ein Bean-Objekt von einem bestimmten Typ im gesamten IOC-Container. Alle Konsumenten dieses Beans, haben also Zugriff auf ein und dasselbe Objekt und dessen "globalen" Zustand. Diese Variante der Sichtbarkeit wird in *Spring* als "Singleton Scope" bezeichnet.

Welchen Scope man in einer Beandefinition konfiguriert, muss je nach Anwendungsfall entschieden werden und hat damit Einfluss auf die Implementierung als auch auf die Verwendung des Beans. Bei der Verwendung der SpringContributions hat die Sichtbarkeit natürlich auch eine große Bedeutung. Hier ist es wichtig wie die Sichtbarkeit der Contribution-Container-Beans konfiguriert ist. Davon hängt es nämlich ab, ob ein Contribution-Container als einzelnes Objekt unter allen Konsumenten geteilt wird oder nicht. Der Unterschied ist deshalb wichtig, weil im Falle des Singleton Scope, ein Konsument nicht die alleinige Kontrolle über den Zustand des Containers und seiner Elemente inne hat. Es kann also vorkommen, dass die Art, als auch die Anzahl der Elemente in einem Container zur Laufzeit von anderen Konsumenten geändert werden. Dies kann, je nach Anwendungsfall, erwünscht oder auch nicht erwünscht sein.

In der derzeitigen Umsetzung der SpringContributions, wird jeder Contribution-Container mit dem Singleton Scope angelegt. Der Benutzer der Komponente hat im Moment keine Möglichkeit in diese Konfiguration einzugreifen. Ein Prototype Scope ist also derzeit nicht einstellbar.

Listing 7.1: (Java) Mögliche Scope-Konfiguration für Contribution-Listen

```
// Java-Konfiguration mit Scope-Definition:

@Contribution(to="prototypeContainer", scope="prototype")
@Bean
public SomeElement someElement()
{
    return new SomeElementImpl();
}
```

Damit der Scope doch durch den Anwender festlegbar wird, müssten beiden Konfigurationsarten (XML, Java) für beide möglichen Contribution-Container (Liste, Map) um die Einstellung des Scope erweitert werden. Listings 7.2 und 7.1 zeigen am Beispiel der Contribution-Liste wie das in den beiden Konfigurationsvarianten aussehen könnte.

Listing 7.2: (XML) Mögliche Scope-Konfiguration für Contribution-Listen

```
// XML-Konfiguration mit Scope Definition:

// Konfiguration für XML-Element 'contribution-ref'
<bean id="prototypeContributionConsumer" class="
    PrototypeContributionConsumerImpl">
    <constructor-arg>
        <ctr:contribution-ref name="prototypeContainer" scope="
            prototype" />
    </constructor-arg>
</bean>

// Konfiguration für XML-Element 'contribution'
<bean name="SomeElement" class="SomeElementImpl" />

<ctr:contribution to="prototypeContainer" scope="prototype" >
    <ctr:entry name="SomeElement" ref="SomeElement" />
</ctr:contribution>

// Konfiguration für XML-Element 'contribute'
<bean name="AnotherElement" class="AnotherElementImpl" >
    <ctr:contribute to="prototypeContainer" scope="prototype" />
</bean>
```

Literaturverzeichnis

[1] Apache tapestry home page. http://tapestry.apache.org/, 30 April 2012.

[2] Inversion of control history. http://picocontainer.org/inversion-of-control-history.html, 20 April 2012.

[3] The ioc container. http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/beans.html, 16 April 2012.

[4] Tapestry ioc configuration. http://tapestry.apache.org/tapestry-ioc-configuration.html, 30 April 2012.

[5] 1. introduction to spring framework. http://static.springsource.org/spring/docs/3.0.x/reference/overview.html, 9 September 2013.

[6] Luc Fabresse, Noury Bouraqadi, Christophe Dony, and Marianne Huchard. A language to bridge the gap between component-based design and implementation. *Comput. Lang. Syst. Struct.*, 38(1):29–43, April 2012.

[7] Brian Foote and Joseph W. Yoder. Big ball of mud. In N. Harrison, B. Foote, and H. Rohnert, editors, *Pattern Languages of Program Design*, volume 4, pages 654–692. Addison-Wesley, 2000.

[8] Martin Fowler. Module assembly. *IEEE Software*, 21:65–67, 2004.

[9] Martin Fowler. Inversion of control containers and the dependency injection pattern. http://martinfowler.com/articles/injection.html, 10 February 2012.

[10] Martin Fowler. Inversionofcontrol. http://martinfowler.com/bliki/InversionOfControl.html, 9 September 2012.

[11] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[12] Foote Johnson. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, June/July 1988.

[13] Robert C. Martin. The dependency inversion principle. June 1996.

[14] Michael Mattsson, Michael Mattsson, and Michael Mattsson. Object-oriented frameworks - a survey of methodological issues, 1996.

[15] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15:1053–1058, December 1972.

[16] Spring JavaConfig project. Spring java configuration. http://www.springsource.org/javaconfig, 29 March 2013.

[17] Brian Foote Ralph E. Johnson. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, June/July 1988.

[18] Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Conference proceedings on Object-oriented programming systems, languages and applications*, OOPLSA '86, pages 38–45, New York, NY, USA, 1986. ACM.

[19] Richard E. Sweet. The mesa programming environment. *SIGPLAN Not.*, 20(7):216–229, June 1985.

[20] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, NY, 1st edition, 1998.

[21] Dave Thomas and Andy Hunte. Mock objects. *IEEE Softw.*, 19(3):22–24, May 2002.