Freie Universität Berlin

Bachelorarbeit am Institut für Informatik der Freien Universität Berlin,

Arbeitsgruppe Software Engineering

Entwicklung eines Konfigurationsvalidierers

Christian Cikryt

Matrikelnummer: 4285814

c.cikryt@fu-berlin.de

Berlin, 29.09.2011

Eingereicht bei:	Prof. Dr. Lutz Prechelt
Zweitkorrektor:	Prof. Dr. Elfriede Fehr
Betreuer :	Dipl.-Inf. Florian Wallner (tarent solutions GmbH)

Zusammenfassung

Diese Arbeit ist in enger Zusammenarbeit mit der tarent solutions GmbH und im Hinblick auf die dort eingesetzten Portalsysteme entstanden. Das Ziel ist es, ungültige Konfigurationen von Artefakten zurückzuweisen, um eine aufwendige Fehlersuche während des Betriebs von Anwendungen zu vermeiden. Dazu wird – nach ausführlicher Evaluation existierender Technologien – eine Beschreibungssprache zum Erfassen von Konfigurationsanforderungen sowie ein Validierungsprogramm, der so genannte *Konfigurator*, entwickelt. Die erstellten Beschreibungen werden mittels eines Parsergenerators eingelesen und auf Erfüllbarkeit und Widersprüche geprüft. Anschließend können konkrete Konfigurationen anhand dieser Kriterien validiert werden. Fehler werden dabei mit Verweis auf die entsprechenden Codestellen in den Beschreibungen erklärt.

Abstract

This thesis has been developed in close cooperation with tarent solutions GmbH and is geared at the local enterprise portal systems. It aims at detecting invalid configurations of artifacts immediately in order to avoid costly debugging at runtime. After thorough evaluation of existing technologies a language for collecting requirements and a validation program, the *Konfigurator*, are being developed. The resulting description files are read using a parser generator and tested for satisfiability and contradictions. Thereafter, the *Konfigurator* validates concrete configurations with regard to the declarations. Occurring errors are explained to the user by referring to specific passages of the description files.

Eidesstattliche Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen wurden, sind als solche gekennzeichnet. Die Zeichnungen oder Abbildungen sind von mir selbst erstellt worden oder mit entsprechenden Quellennachweisen versehen. Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner Prüfungsbehörde eingereicht worden.

Berlin, den 27. September 2011

(Christian Cikryt)

Inhaltsverzeichnis

Eidesstattliche Erklärung ii

1 Einführung **1**
1.1 Motivation 1
1.2 Begriffsdefinitionen 2
1.2.1 Konfigurieren 2
1.2.2 Weitere Begrifflichkeiten 2
1.3 Beschreibung der Zielumgebung 3
1.4 Anspruch und Grenzen dieser Arbeit 3
1.5 Aufbau der Arbeit 4

2 Verwandte Arbeiten **5**

3 Beschreibung des Ansatzes **7**

4 Evaluation **9**
4.1 Suche nach fertigen Lösungen 9
4.1.1 Fragestellung 9
4.1.2 Methode und Resultate 9
4.1.3 Diskussion einer Erweiterung 11
4.2 Erforderliche Ausdrucksmöglichkeiten 12
4.2.1 Die häufigsten Fehlkonfigurationen und die Gegenmaßnahmen im Modell 13
4.2.2 Domänenmodell 14
4.3 Sprachtechnologien 19
4.3.1 Fragestellung 19
4.3.2 Domänenspezifische Sprachen 19
4.3.3 Sprachwahl für den Prototyp der internen DSL 20
4.3.4 Interne DSL in Scala 20
4.3.5 Evaluation der Technologien für die externe DSL 22
4.4 Skizze eines Installationsablaufs 25

5 Realisierung **27**
5.1 Die erschaffene Sprache 27
5.1.1 Lexer 27
5.1.2 Konstruktion und Auslesen des Abstrakten Syntaxbaumes . . . 29
5.1.3 Fehlerbehandlung 30
5.1.4 Nutzerfreundliche Features 30

5.2 Domänenmodell im Detail 31
5.2.1 Schablonenparameter 32
5.2.2 Unveränderlichkeit und Builder-Klassen 33
5.2.3 Diskussion Systemmodellierung 34
5.3 Erfüllbarkeitsüberprüfung und Validierungsalgorithmus 36
5.3.1 Evaluation von Constraint-Solvern 36
5.3.2 Grundprinzip der Erfüllbarkeitsüberprüfung 36
5.3.3 Implementierung 38
5.4 Fehlerbehandlung 42
5.5 Bemerkungen zu Funktionstests und Releasebau 43

6 Bewertung durch Benutzungstests 44
6.1 Aufgabe und Methode 44
6.2 Ergebnis 44

7 Ausblick 46

1 Einführung

1.1 Motivation

Die tarent solutions GmbH erstellt u. a. Enterprise-Portal-Lösungen, in denen dutzende selbst geschriebener Anwendungen sowie freie Software Dritter zum Einsatz kommen. Jede dieser Komponenten bringt eigene Konfigurationen mit sich, die aufeinander abgestimmt werden müssen, so dass sich das Konfigurieren des Gesamtsystems äußerst komplex gestaltet.

Fehlerhafte Einstellungen stellen deshalb keine Seltenheit dar, ziehen aber mitunter eine aufwendige Fehlersuche nach sich, weil sie in der Regel erst während der Laufzeit bemerkt werden. Es reicht ein Tippfehler in einer Server-URL, ein „htpt“ statt „http“, um eine Anfrage mit einer Fehlermeldung zu beantworten. Kostspielig wird es, weil für die Qualitätssicherung nicht erkennbar ist, ob es sich um einen Bug oder eine Fehlkonfiguration handelt, so dass über die üblichen Prozesse ein Fehlerbericht bei den Entwicklern landet und nicht bei den Administratoren, die für das Aufsetzen der Testsysteme verantwortlich sind. Selbst wenn das Problem beim Hochfahren auffällt, verzögert sich durch den Neustart ggf. des kompletten Verbundes (Clusters) die Bereitstellung der Systeme merklich. Anders als bei Mängeln im Code existieren noch keine automatischen Tests für Konfigurationen, die unmittelbar Rückmeldung liefern.

Dabei wären die meisten Fehleinstellungen im Vorfeld erkennbar, weil das Wissen über die Konfigurationsanforderungen in der Firma vorhanden ist. Allerdings ist dieses Wissen auf die Wiki-Seiten der einzelnen Projekte verstreut, teilweise redundant, widersprüchlich und häufig veraltet. Mit anderen Worten: Es ist fern vom Einsatzort und nicht automatisch auswertbar.

An diesen beiden Punkten setzt die in dieser Arbeit vorgestellte Lösung an. Konfigurationen sollen noch vor dem Einspielen automatisch überprüft und die dafür benötigten Informationen mit der Software ausgeliefert werden, so dass sie in gesammelter Form verfügbar sind. Die einzelnen Projekte schreiben die Anforderungen an die Konfiguration ihrer Software in einer zu entwickelnden Beschreibungssprache nieder. Die resultierenden Beschreibungen werden vom so genannten *Konfigurator* auf Erfüllbarkeit und Widersprüche getestet, um Boden für weitere Fehler zu nehmen. Nach dieser Vorarbeit kann der *Konfigurator* konkrete Konfigurationen validieren und Rückmeldung über aufgetretene Probleme geben.

1.2 Begriffsdefinitionen

1.2.1 Konfigurieren

In dieser Arbeit gehe ich von folgender Definition einer Konfigurationsaufgabe (des Konfigurierens) aus, die an [MF89] angelehnt ist. Meine Definition ist auf den Kontext der Softwarekonfiguration mittels Parametern spezialisiert, während sich [MF89] auf die Konfiguration von generischen Produkten bezieht.

Konfigurieren Als Ausgangspunkt sei gegeben: (A) eine feste und definierte Menge an Komponenten, die jeweils durch eine Menge an Parametern beschrieben werden, und (B) eine Menge an Bedingungen, die die möglichen Belegungen der Parameter einschränken. Dabei werden alle Optionen, deren Werte den Zustand oder das Verhalten des Systems beeinflussen, als Parameter bezeichnet.
Durch den Akt des Konfigurierens wird unter diesen Vorgaben eine Konfiguration (aus mehreren möglichen) erstellt, die alle genannten Anforderungen erfüllt. Die Konfiguration besteht aus einer Beschreibung der Wertebelegung der Parameter.
Die Menge an Parametern und Bedingungen wird durch die Konfigurationsbeschreibungen erfasst.

1.2.2 Weitere Begrifflichkeiten

Die im Folgenden eingeführten Begriffe erläutern deren Verwendung in dieser Arbeit und sollen Missverständnissen vorbeugen. Dabei gilt es besonders, den Unterschied zwischen einer Konfiguration mit konkreten Werten und einer Konfigurationsbeschreibung, die die Menge aller möglichen Werte eines Parameters einschränkt, hervorzuheben.

Beschreibungssprache ist die zu erstellende Sprache, in der die Deklaration von Konfigurationsbeschreibungen erfolgt.

Konfigurationsbeschreibung ist eine in der Beschreibungssprache verfasste Definition des gültigen Wertebereichs für eine Menge von Parametern.

Parameter wird benutzt, um jede beliebige Einstellung zu bezeichnen, die durch ihren Wert den Zustand oder das Verhalten des betrachteten Systems beeinflussen kann. Die Summe aller Parameter ergibt die Konfiguration eines Systems.

Konfigurator ist das zu entwickelnde Programm zur Validierung von konkreten Konfigurationen anhand von Konfigurationsbeschreibungen.

1.3 Beschreibung der Zielumgebung

Zwar wird der *Konfigurator* universell für die Konfigurationsvalidierung von Software entwickelt, aber sein derzeitig primäres Einsatzgebiet sind die Portalsysteme in der tarent.
Ein derartiges Portalsystem besteht aus einem Frontend- und einem Backendverbund. Die einzelnen Verbünde bestehen jeweils aus mindestens zwei Anwendungsservern, denen pro Verbund ein Lastverteilungsserver vorangestellt ist. Auf den Frontendsystemen laufen die Portalanwendungen. Konkret ist dies ein *JBoss*-Anwendungsserver[1] in Verbindung mit einem *Liferay*-Portal[2]. Auf den Backendsystemen arbeitet ebenfalls ein *JBoss*, aber ohne Portalkomponenten. Diese Basis nutzen die verschiedenen Projekte, um ihre Entwicklungen (größtenteils *Portlets* [Pro08]) einzusetzen. Weil diese Entwicklungen auf Java basieren, eignet sich die *JVM* (Java Virtual Machine) besonders als Zielplattform für den *Konfigurator*.
Ein wichtiges Konzept, das bei der Konfiguration berücksichtigt werden muss, stellen Portalinstanzen dar. So kann ein Anwendungsserver mehrere Instanzen beherbergen, in denen voneinander getrennt unterschiedliche Portal-Installationen zum Einsatz kommen. Dabei gibt es globale Einstellungen, die für alle Instanzen gelten, sowie instanzspezifische, die jede Instanz für sich neu definieren kann.
Es gibt drei Szenarien in den Portalprojekten, in denen der *Konfigurator* zum Einsatz kommen soll, die sich u. a. in den ausführenden Personen unterscheiden:

1. Interne Entwicklungssysteme, die von jedem Software-Entwickler selbstständig nach den Vorgaben des jeweiligen Projekts aufgesetzt und verwaltet werden.

2. Interne Testsysteme, die von den Administratoren eingerichtet werden.

3. Kundensysteme, die ebenfalls von den Administratoren aufgesetzt werden, aber zusätzlichen Einschränkungen bzgl. Softwareumgebung und Befugnissen unterliegen, so kann dort beispielsweise außer der *JVM* keine Laufzeitumgebung vorausgesetzt werden.

1.4 Anspruch und Grenzen dieser Arbeit

Das erklärte Ziel ist es, Konfigurationen automatisch validieren zu lassen und ungültige noch vor dem Einspielen abzuweisen. Den Benutzern soll dazu ein einfach zu bedienendes Werkzeug an die Hand gegeben werden, um ihre Kriterien zu formulieren. Dieser Ansatz erlaubt es natürlich nur, Konfigurationen auszusortieren, die gegen angegebene und somit bekannte Regeln verstoßen. Da ein Großteil der Software in der tarent geschrieben wird und auch mit den eingesetzten Anwendungs- und Portalservern langjährige Expertise gesammelt wurde, liegt das Hauptproblem nicht an fehlendem Wissen, sondern an dessen Verstreuung.

[1] `http://www.jboss.org/jbossas`
[2] `https://www.liferay.com/`

Weiterhin wird die Beschreibungssprache, realistisch betrachtet, nicht alle Verknüpfungen und Abhängigkeiten zwischen Parametern in allen ihren Details wiedergeben können, so dass vor allem Wert darauf gelegt wird, die häufigsten und schwersten Fehlkonfigurationen auszuschließen. Es ergibt sich zudem ein Konflikt zwischen der Mächtigkeit der Sprache und der gewünschten leichten Erlernbarkeit, der erst einmal zugunsten der Einsteigerfreundlichkeit aufgelöst wird. Falls nach Einführung und Akzeptanz der Sprache noch Sprachelemente fehlen, können diese auf Grundlage der bereits vorhandenen Kenntnisse leichter ergänzt werden.
Einen Schwerpunkt dieser Arbeit stellt die Evaluierung vorhandener Technologien zum Einsatz bzw. zur Erweiterung dar. Diese bezieht sich sowohl auf die zu erstellende Sprache als auch auf das Validierungswerkzeug. In diesem Zusammenhang wird auch die Eingliederung in vorhandene Werkzeuge und Prozesse betrachtet, aber mögliche Lösungsstrategien, wie ein kompletter Konfigurationsablauf, nur exemplarisch skizziert.
Die Implementierung des *Konfigurators* bleibt bei der Nutzerinteraktion auf Prototypniveau, so wird kein spezielles Augenmerk auf die Erstellung einer grafischen Benutzeroberfläche oder das Aufsuchen von Konfigurationsbeschreibungen in einer Dateisystemstruktur gelegt.

1.5 Aufbau der Arbeit

Nachdem in Kapitel 2 eine Abgrenzung zu verwandten Arbeiten erfolgt ist, erläutert Kapitel 3 den grundsätzlichen Ansatz. Kapitel 4 widmet sich der ausführlichen Evaluation der einzusetzenden Technologien und wägt dabei auch eine Erweiterung gegen eine Neuentwicklung ab. Die Implementierung wird in Kapitel 5 ausgeführt und anschließend in Kapitel 6 bewertet. Kapitel 7 bietet einen Ausblick auf zukünftige Entwicklungsfelder.

2 Verwandte Arbeiten

White et al. gehen in [WSC+07] von der Konfiguration als dezentralem Prozess aus, in dem Entscheidungen einer Rolle die Konfigurationsmöglichkeiten anderer Rollen einschränken. Sie entwickeln ein Konfigurationswerkzeug namens *Fresh*, das anhand von dezentralen Modellen an Konfigurationsregeln aktivierte Features und Entscheidungen erkennt und gültige Konfigurationsdateien erzeugt. Es konzentriert sich auf die Auswirkungen von unterschiedlichen Infrastrukturentscheidungen, wie z. B. die Wahl einer anderen Datenbank, während der Fokus dieser Arbeit auf der Überprüfung von Konfigurationsparametern liegt. Um den Unterschied zu verdeutlichen: Ihnen geht es darum, u. a. anhand von Hardwaremerkmalen die Auswahl an Features einzuschränken und entsprechende, vorgefertigte Konfigurationsdateien auszuwählen. Mein Ziel ist dagegen, konkrete Konfigurationsparameter auf Grundlage beschriebener Beschränkungen zu überprüfen. Gemeinsam ist beiden Ansätzen die Definition einer Beschreibungssprache, wenn auch mit sehr unterschiedlichem Inhalt, sowie die Einschränkung des Wertebereichs durch *Constraints*. Ihr geschaffenes Programm ist zudem an spezielle Technologien gebunden, von denen ich unabhängig bleiben will. So verlangt es zur Modellierung das *Eclipse Modeling Framework*[1] und erfordert zur Laufzeit eine *Spring*[2] Umgebung. Weiterhin ist der Quelltext nur im *Eclipse* Plug-in *GEMS*[3] verbaut veröffentlicht worden.
[FFJ00] erstellt in der Domäne der Produktkonfiguration (konkret gewähltes Beispiel ist die Zusammenstellung von PC-Komponenten) eine Sprache basierend auf *UML* und *OCL* (Object Constraint Language), um eine Konfigurationsaufgabe zu beschreiben und daraus eine Wissensdatenbank zu generieren, mit deren Hilfe gültige Konfigurationen berechnet werden können. Als Ergebnis halten die Autoren durch prototypische Implementierung validiert fest, dass sich *UML* in Verbindung mit *OCL* als Beschreibungssprache für die beschriebenen Anwendungsfälle eignet. Genau wie diese Arbeit beschäftigen sie sich mit der Beschreibung von Konfigurationsmöglichkeiten. Der Ansatz, bestehende graphische Notationen zur Beschreibung zu verwenden, ist interessant und wird in Abschnitt 4.3.5 noch einmal aufgegriffen. Jedoch scheidet zumindest *OCL* aufgrund mangelnder Kenntnisse innerhalb der tarent und seiner Komplexität zur Beschreibung aus, da einfache Erlernbarkeit als ein wichtiges Merkmal der Software gefordert ist.
Im Bereich der Produktkonfiguration und der Konfiguration als *Constraint-Satisfaction-Problem* (CSP) gibt es eine Reihe von Arbeiten (stellvertretend seien [SF96] und [Stu97] genannt), die sich alle in einem wesentlichen Punkt von dieser unterscheiden:

[1] `http://www.eclipse.org/modeling/emf/`
[2] `http://www.springsource.org/`
[3] `http://www.eclipse.org/gmt/gems/`

Mich interessiert nicht, welche Lösungen es für den eingeschränkten Wertebereich gibt, da es in der Regel sowieso unendlich viele sind, sondern lediglich, ob eine Lösung existiert und ob eine gegebene Belegung alle Anforderungen erfüllt. Die konkreten Belegungen werden als Konfigurationen vom Benutzer eingegeben und müssen lediglich überprüft werden.
Auch wenn diese Arbeit definitiv in den Bereich des *Softwarekonfigurationsmanagements* (SCM) gehört, unterscheidet sich bereits der Konfigurationsbegriff erheblich. Während nach [Buc96] bei SCM alle Funktionen der Software kombiniert mit der Hardware eine Konfiguration ausmachen, verstehe ich unter Konfiguration nur den Teil, der vom Anwender veränderbar ist. Ferner beschäftigt sich SCM stark mit Prozessen, die u. a. eine hohe Software-Qualität sicherstellen und Änderungen an der Software überwachen sollen. Der *Konfigurator* kann als automatisches Werkzeug in SCMs Unterbereichen *Softwarekonfigurationskontrolle* und *Softwarekonfigurationsüberprüfung* ([BHS78]) unterstützend wirken.

3 Beschreibung des Ansatzes

Da in einem typischen System das Wissen über die Konfigurationsanforderungen auf mehrere Personen, im Fall des Portalsystems sogar über viele Projekte, verteilt ist, wird ein dezentraler Ansatz zur Konfigurationsbeschreibung gewählt. Konkret kann jedes noch so kleine Artefakt eine eigene Beschreibung besitzen, die vom *Konfigurator* gesammelt[1] und ausgewertet werden. Im Gegensatz zu einer zentralen Beschreibung können so die Anforderungen nicht nur unabhängig voneinander formuliert, sondern auch im Einsatz beliebig untereinander kombiniert werden. Für den Fall, dass Konflikte zwischen den einzelnen Definitionen auftreten, wird eine genaue Positionsangabe den Beteiligten helfen, den Widerspruch aufzulösen.
In der Praxis werden immer wieder gesetzte Konfigurationsoptionen auftauchen, über die keine Aussagen bekannt sind. Im Wesentlichen gibt es zwei Möglichkeiten, mit ihnen umzugehen:

1. Sie werden zurückgewiesen. Infolgedessen verhalten sich die Beschreibung und die tatsächliche Konfiguration kongruent ([TB02]) zueinander, wie auf Bild 1 in Abbildung 3.1 zu sehen ist. Diese an sich erstrebenswerte Eigenschaft würde die Einführung des *Konfigurators* erschweren, weil zunächst für jedes Projekt alle Parameter definiert werden müssten. Somit wird von dieser Variante vorerst Abstand genommen, zumal der Implementierungsaufwand für eine spätere Umstellung sehr gering scheint.

2. Sie werden akzeptiert. In diesem Fall weichen tatsächliche Konfiguration und Beschreibung im Laufe der Zeit immer stärker voneinander ab (vgl. Bild 2 in Abbildung 3.1). Dies ergibt sich aus der Tatsache, dass die Menge an Einstellungen zunimmt. Diese Divergenz ([TB02]) werde ich zugunsten einer früheren Erprobung in Kauf nehmen und will ihr mit einer wachsenden Wissensbasis entgegenwirken, bis schlussendlich ausreichend Zuversicht in die Erzwingung kongruenten Verhaltens existiert. Diesen Verlauf stellt die dritte Grafik von Abbildung 3.1 dar.

Bis zu einer vollständigen Implementierung des Prototyps sind die folgenden Schritte geplant.
Um auszuschließen, dass vermeidbarer Aufwand in die Neuerfindung bereits existierender Technologien fließt, steht zu Beginn eine umfassende Evaluierungsphase an, in der auch die Erweiterung bestehender Teillösungen betrachtet wird. Als Basis für die Wahl der Beschreibungssprache werden die Anforderungen an die Ausdrucksstärke

[1] Der hier entwickelte Prototyp wird die Beschreibungen nicht suchen, sondern erwartet sie in seiner Argumentenliste.

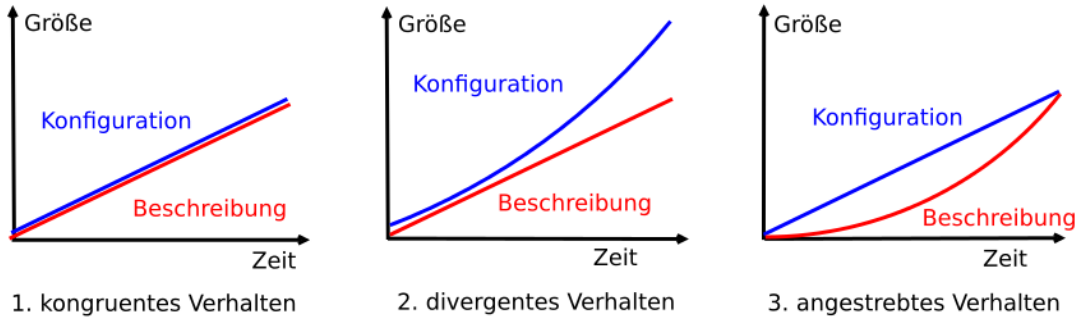

Abbildung 3.1: Drei verschiedene Verhalten von Konfiguration zu Beschreibung (1. und 2. nach [TB02])

anhand der Analyse der derzeitigen Portalkonfiguration ermittelt und in einem sogenannten Domänenmodell (siehe [Eva03]) festgehalten. Dieses kann nicht nur unabhängig von der letztendlich benutzten Sprache befüllt werden, sondern dient auch als Arbeitsgrundlage für den Erfüllbarkeits- und Validierungsalgorithmus. Bei dem Erfüllbarkeitsalgorithmus gilt es ebenfalls abzuwägen, ob auf fertige *Constraint-Solver* zurückgegriffen wird oder ein maßgeschneiderter Algorithmus bessere Ergebnisse liefert.

4 Evaluation

Dieses Kapitel beschreibt die durchgeführte Evaluation, mit der verhindert werden sollte, dass Aufwand in das Neuschreiben bereits existierender Software fließt. Weiterhin sollten mit ihr die am besten geeigneten Technologien aufgefunden werden.

4.1 Suche nach fertigen Lösungen

4.1.1 Fragestellung

Gibt es Systeme, die die folgenden Anforderungen zumindest größtenteils erfüllen?

- Beschreibungsmöglichkeit für Konfigurationsanforderungen.
- Fähigkeit, Konfigurationen zu validieren.
- Lauffähigkeit unter GNU/Linux Systemen ohne Forderung nach speziellen Laufzeitumgebungen; lediglich die *JVM* darf vorausgesetzt werden.
- Veröffentlichung unter einer freien Lizenz.

Falls aussichtsreiche Kandidaten nicht alle Funktionen bieten, wird eine Weiterentwicklung in Erwägung gezogen.

4.1.2 Methode und Resultate

Mögliche Kandidaten wurden sowohl über die Suchmaschinen *google* und *google scholar* als auch durch Anfrage auf der Plattform *stackoverflow.com*[1], die zuvor nach verwandten Themen durchsucht wurde, und auf der tarent-internen Mailingliste aller Entwickler ermittelt. Die genannten Suchmaschinen wurden u. a. nach den Begriffen „validate configurations“, „describe configuration requirements/constraints“ und „java verify configuration“ sowie nach Synonymen und Kombinationen dieser Worte (auch auf Deutsch) gefragt. Dabei kamen die im Folgenden vorgestellten Programme in die nähere Auswahl.

Konfigurationsmanagementsysteme erschienen zunächst aussichtsreich, weil sie versprachen, für die Einhaltung von Konfigurationen zu sorgen. Konkret wurden die Fähigkeiten der prominentesten Vertreter *puppet*[2], *cfengine*[3], *SmartFrog* ([GGL+03])

[1] `http://stackoverflow.com/questions/5270872/which-solutions-are-available-for-meta-configuration-or-configuration-verificatio`
[2] `http://www.puppetlabs.com/`
[3] `http://cfengine.com/`

und *bcfg2*[4] detailliert untersucht. Ihr Einsatzgebiet ist die Konfiguration und Verwaltung einer Menge von Rechnern durch die Errichtung von Master-Knoten, die Konfigurationsvorgaben an die Slave-Knoten weitergeben. Sie bieten alle lediglich die Möglichkeit, die konkrete Konfiguration eines Softwaresystems zu definieren; sie setzen also implizit voraus, dass die Validität der Konfiguration zuvor geprüft wurde, was genau den Kern dieser Arbeit darstellt. Dafür sind sie in der Lage, die Einhaltung einer Konfiguration fortwährend zu garantieren. Entwickelt wurden sie mit dem Hauptziel unixoide Systeme zu verwalten, so dass nur der Umgang mit einfach strukturierten Textdateien gut unterstützt wird; insbesondere die im Java-Enterprise-Umfeld weit verbreiteten *XML*-Dateien werden nicht nativ unterstützt. Außerdem ist ihnen das Konzept von Rechnerverbünden mit mehreren Portalinstanzen, wie es bei den Portalen der tarent solutions GmbH zum Einsatz kommt, unbekannt. Zusammengefasst erfüllen sie essentielle Anforderungen nicht, so dass sie nicht direkt verwendet werden können, aber da sie alle eine freie Lizenz besitzen, kommt eine Erweiterung prinzipiell in Frage. Die Abwägung hierzu findet sich in Unterabschnitt 4.1.3.

Config Checker Es handelt sich hierbei um eine kommerzielle Lösung, die laut Beschreibung der Webseite[5] einen sehr ähnlichen Ansatz wie ich verfolgt. Der spärlichen Dokumentation ist zu entnehmen, dass sich Wertebereiche für Parameter spezifizieren lassen, deren Einhaltung überprüft werden kann. Allerdings scheidet sie sofort aus, weil sie unter keiner freien Lizenz steht. Es konnten auch keine Anzeichen auf eine Verbreitung der Software gefunden werden.

Config-Model Auch wenn hinter diesem Werkzeug eine andere Motivation steht, nämlich unerfahrenen Benutzern bei der Konfiguration von Programmen zu helfen sowie Entwickler zu unterstützen, ihre Software leichter konfigurierbar zu machen, besitzt es ein interessantes Konzept, das meinem ähnlich ist. So können Konfigurationen in begrenztem Umfang beschrieben werden und aus diesen Beschreibungen wird anschließend ein Modell erzeugt, mit dem Konfigurationen überprüft werden können. Die Beschreibungsmöglichkeiten sind auf den unixoiden Paketierungsprozess zu geschnitten und insgesamt für unsere Zwecke zu gering. Denn es werden im Wesentlichen nur primitive Datentypen und konkrete Aufzählungen als Werte unterstützt, während wir u. a. eine Erkennung von URLs und Ports benötigen. Die Tatsache, dass *Config-Model* in Perl geschrieben ist, erschwert eine Weiterentwicklung erheblich, da weder bei mir und in der tarent nennenswerte Perl-Kenntnisse vorhanden sind. Endgültig ausgeschlossen wurde Config-Model, weil auf Kundensystemen kein Perl-Interpreter vorausgesetzt werden kann.

Fresh ist das Ergebnis einer wissenschaftlichen Arbeit, die bereits in Kapitel 2 vorgestellt wurde. Obwohl es unter einer freien Lizenz steht, besitzt es weder eine aktive

[4] `http://bcfg2.org/`
[5] `http://www.configchecker.com/`

Entwicklung noch eine Community. Das einzige Release ist für eine Wiederverwendung denkbar ungünstig gepackt, da es neben anderen Komponenten in das *Eclipse* Plug-in *GEMS* geschnürt ist. Die starke Bindung an die *Eclipse*-Plattform und die Abhängigkeit zum *Eclipse Modeling Framework* will ich nicht eingehen.

Augeas dient zur sicheren Manipulation von Konfigurationsdateien und zur Abbildung von Konfigurationen in ein Augeas-Baummodell. Die Definition dieses Mappings erscheint auf den ersten Blick umständlich. *Augeas* könnte auch maximal bei der Abbildung von Konfigurationen in das Domänenmodell helfen, weil sowohl die Unterstützung für eine Beschreibung von Konfigurationsmöglichkeiten als auch ein Überprüfungsmechanismus fehlt. Da es frei lizenziert ist und zumindest über Java-Bindings verfügt, sollte es nicht völlig abgeschrieben werden und, falls die Implementierung Bedarf erkennen lässt, noch einmal detaillierter betrachtet werden. Aber mehr als eine Teilkomponente der Lösung kann es in keinem Fall darstellen.

Keines der untersuchten Programme kann allen Anforderungen gerecht werden. Daher gilt es nun abzuwägen, ob ein bestehender Ansatz erweitert werden soll oder eine Neuentwicklung ansteht.

4.1.3 Diskussion einer Erweiterung

Da die anderen Programme im vorherigen Abschnitt ausgeschlossen wurden, kommt als Ausgangspunkt für eine Erweiterung nur eines der Konfigurationsmanagementsysteme in Frage. Für eine Erweiterung spricht:

- Die Administration von Systemkomponenten – dafür wurden diese Werkzeuge entwickelt – und Java-Anwendungen würde vereinheitlicht und infolgedessen der Verwaltungsaufwand reduziert werden. Zudem liefern sie eine Infrastruktur mit, um fortwährend Konfigurationsänderungen einzuspielen und die Einhaltung der Vorgaben zu garantieren.

- Die Wahrscheinlichkeit, Unterstützung für eine Weiterentwicklung außerhalb der tarent zu finden ist größer als bei einer Eigenentwicklung, da bereits aktive Entwicklergemeinschaften existieren.

- Zwar bieten die vorhandenen Lösungen keine Erleichterung für die Erstellung der Beschreibungssprache und der Validierungsalgorithmen, aber im Bezug auf die zukünftige, einheitliche Verwaltung von Systemen ergibt es Sinn, auf etablierte Mechanismen zurückzugreifen. Denn das Ergebnis dieser Arbeit wird in jedem Fall nur einen Teil im Konfigurationsprozess darstellen.

Gegen eine Erweiterung spricht:

- Von den in Frage kommenden Systemen ist lediglich *SmartFrog* in Java geschrieben, *puppet* verwendet Ruby, *cfengine* C und *bcfg2* Python. Die drei letztgenannten Sprachen zählen nicht zum Standardrepertoire eines tarent-Mitarbeiters,

was eine Weiterentwicklung und Pflege erschwert. *SmartFrog* hat von diesen vier Werkzeugen die kleinste Community und wurde im Gegensatz zu *cfengine* und *puppet* noch nie in die nähere Auswahl für eine Einführung in der tarent gestellt.

- Unabhängig von der Sprache erfordern alle eine lange Einarbeitung, um herauszufinden an welchen Stellen sie sinnvoll erweitert werden können bzw. ob dies überhaupt mit vertretbarem Aufwand möglich ist.

- Je nach Tiefe des Eingriffs ist es auch fraglich, ob man die Änderungen überhaupt zu den offiziellen Releases beisteuern darf. Falls nicht, wäre der Pflegeaufwand wahrscheinlich genauso hoch wie bei einer Neuentwicklung[6].

- Die benötigte Infrastruktur ist nicht in allen Einsatzszenarien gegeben. So kann bei der Konfiguration von Kundensystemen unter Umständen kein Einfluss auf die dort verwendeten Konfigurationsmanagementsysteme genommen werden. Im Zweifel darf noch nicht einmal eine Ruby-Installation vorausgesetzt werden, die beispielsweise *puppet* benötigt. Zurzeit werden auch die Entwicklungsmaschinen unabhängig voneinander von einzelnen Entwicklern verwaltet. Diesen könnte zwar die Verwendung eines Managementsystems nahegelegt werden, aber der Einarbeitungs- und Verwaltungsaufwand würde den ohnehin komplizierten Installations- und Konfigurationsprozess weiter erschweren.

- Zuletzt würde die enge Kopplung an ein bestimmtes System – was eine Erweiterung zur Folge hätte – die Einsatzmöglichkeiten des *Konfigurators* erheblich einschränken und die Wahl des Systems tarent-weit festlegen. Für das Fällen einer solchen weitreichenden und langfristigen Entscheidung fehlen im Moment die nötigen Parameter, z. B. steht eine umfassende Evaluation der Kandidaten noch aus.

Als Konsequenz sieht das weitere Vorgehen wie folgt aus. Der *Konfigurator* wird unabhängig von bestehenden Konfigurationsmanagementsystemen in einer auf der *JVM* lauffähigen Programmiersprache entwickelt. Dabei wird eine Definitionsprache für die Beschreibung der Konfigurationsmöglichkeiten entwickelt, für die eventuell auf bereits vorhandene Technologien zurückgegriffen werden kann. Insbesondere ist sicherzustellen, dass die Produkte des *Konfigurators* mit Konfigurationsmanagementsystemen harmonieren, sobald eines davon in der tarent eingeführt wird.

4.2 Erforderliche Ausdrucksmöglichkeiten

Zunächst werden die Anforderungen an die Ausdrucksstärke der Beschreibungssprache ermittelt und in einem Domänenmodell ([Eva03]) festgehalten, so dass auf dieser Grundlage die Sprachsyntax skizziert und damit Technologien auf ihre Eignung untersucht werden können. Das entstandene Domänenmodell stellt eine solide Schnittstelle

[6] Dies hängt davon ab, wie viele Mitstreiter sich für einen Fork finden würden.

zwischen Sprache und Validierungslogik dar und bleibt von Änderungen an der Beschreibungssprache unbeeinträchtigt. Gleichzeitig lässt sich eine konkrete DSL daran testen, ob sie alle Objekte des Modells mit Daten befüllen kann. Als Ausgangspunkt für die benötigten Sprachkonstrukte sollen die bisher häufigsten Ursachen für Fehlkonfigurationen im Portalprojekt dienen, die durch eine Umfrage unter den beteiligten Administratoren und Entwicklern gesammelt wurden. Die jetzige Portalkonfiguration dient als zusätzliche Quelle.

4.2.1 Die häufigsten Fehlkonfigurationen und die Gegenmaßnahmen im Modell

Die Konfiguration eines Portals geschieht mittels einer Liste von Parametern (in der Größenordnung von hundert Stück[7]). Da sich die meisten Installationen nur in wenigen Punkten unterscheiden, besteht die gängige Praxis darin, Vorlagen bzw. andere Konfigurationen zu kopieren, durchzugehen und die entsprechenden Einstellungen anzupassen. Dieses Vorgehen erklärt die meisten der nachfolgenden Fehlerklassen, lässt sich aber nicht einfach ändern, da es bei der Anzahl an Optionen niemandem möglich ist, an alle benötigten Parameter zu denken und diese nacheinander aufzuschreiben. Aus diesem Grund bieten sich Vorlagen, die alle möglichen Konfigurationsoptionen enthalten, die dann gelöscht bzw. deren Werte angepasst werden, für ein effizientes Konfigurieren an. Eine naheliegende Abwandlung bestünde darin, die Schablonen zielgerichteter erzeugen zu lassen, so dass beispielsweise nur tatsächlich benötigte Parameter aufgeführt werden. Diese Arbeit stellt eine Vorstufe dar, denn für die automatische Generierung von Konfigurationsdateien benötigt man Wissen über die Wechselwirkung und Eigenschaften von Parametern, das mit der Beschreibungssprache erfasst werden soll. Gleichzeitig kann der *Konfigurator* ungültige Konfigurationen zurückweisen, unabhängig davon, ob sie wie bisher über naive Vorlagen oder über generierte Zuschnitte entstanden sind.

1. **Fehlende Parameter:** Trotz der Auflistung aller bekannter Konfigurationsoptionen werden manche fälschlicherweise entfernt, weil das Löschen in der Regel abschnittsweise geschieht, oder auf ihren leeren Standardwerten belassen. Um dies zu verhindern, sollen für jedes Artefakt alle benötigten Parameter aufgeführt werden, so dass diese auf Vorhandensein überprüft werden können.

2. **Verstoß gegen formale Typkriterien:** Darunter fallen die Flüchtigkeitsfehler, die beispielsweise zu missgebildeten URLs oder *LDAP Distinguished Names* führen und somit leicht automatisch erkennbar sind. Natürlich tauchen ebenso viele falsche Werte auf, die den formalen Kriterien genügen. Viele von diesen lassen sich auf Wunsch aber, basierend auf ihrer Typangabe, auf Plausibilität überprüfen, z. B. kann die Auflösbarkeit eines Hostnames festgestellt werden. Zusätzlich arbeiten wir gerade bei Pfaden und URLs viel mit aus mehreren Parametern zusammengesetzten Attributwerten, so dass doppelte Schrägstriche

[7] Diese Zahl schwankt sehr mit den verwendeten Komponenten

und dergleichen besonders häufig auftreten. Als Gegenmaßnahme wird vorgeschlagen, dass jeder Parameter mit seinem Typ definiert wird.

3. **Ungeänderte Platzhalterwerte:** Diese Kategorie ist eng mit den beiden vorangehenden verwandt und der Tatsache geschuldet, dass in den Musterkonfigurationen oft keine leeren Werte, sondern offensichtlich zu überschreibende (wie „changeme.example.com“) verwendet werden, um die konkrete Struktur anzudeuten und sie mit einfacher Suche auffindbar zu machen. Die Praxis zeigt aber, dass die Auffindbarkeit nicht ausreicht, weil der momentane Portal-Installer auch für die aktuelle Komponentenauswahl nicht-benötigte Parameter mit einliest, die dann natürlich „changeme“-Werte besitzen. Um diese Werte aufzufinden, müssen keine zusätzlichen Informationen erfasst werden, wohl aber soll eine Zuordnung zwischen Komponenten und Parametern festgehalten werden, so dass nur die relevanten Werte überprüft werden.

4. **Inkonsistenzen:** Hier kristallisierten sich zwei Formen heraus. Zum einem gibt es Parameterblöcke, die sich auf optionale Funktionalitäten beziehen, zum anderen existieren verschiedene Ausprägungen einer Komponente, die in sich schlüssig konfiguriert werden müssen. Die Misskonfigurationen sind dementsprechend fehlende Parameter bei aktivierter Funktionalität bzw. das Mischen von Parametern zweier inkompatibler Varianten, z. B. werden im Portal *Oracle*- und *Postgres*-Datenbanken über anders benannte Java-Properties konfiguriert.

   In der Beschreibungssprache sollen sowohl Ausprägungen als auch optionale Funktionalitäten direkt als Konzept unterstützt werden, so dass sich Gruppen mit gegenseitigem Ausschluss bzw. Parameterregeln, die durch eine Bedingung aktiviert werden, definieren lassen.

Im nächsten Abschnitt werden die hier erarbeiteten Anforderungen detailliert zusammengefasst und erweitert, da bisher nur direkte Reaktionen auf die häufigsten Fehleinstellungen berücksichtigt wurden.

4.2.2 Domänenmodell

Attribute von Parametern

Ein Parameter definiert seinen Typ, z. B. Port oder URL, und gibt an, ob er optional oder verpflichtend gesetzt werden muss. Über ihre ID sind Parameter global eindeutig bestimmbar. Diese Eindeutigkeit wird für alle tarent-Projekte gefordert und notfalls durch ID-Präfixe für einzelne Komponenten erreicht. Neben dem Typ kann auch ein fester Wert vorgegeben werden; dies ergibt speziell bei bedingten Definitionen in optionalen Blöcken bzw. bei unterschiedlichen Ausprägungen Sinn. Außer diesen direkt für die Validierung relevanten Informationen, muss für jeden Parameter sein Ort und sein Gültigkeitsbereich (instanzspezifisch oder global) aufgeschrieben werden. Die Ortsangabe ist notwendig, um das Schreiben von Konfigurationsoptionen und in einer späteren Entwicklungsstufe auch das Auslesen zu ermöglichen. Das Grundprinzip

des *Konfigurators* besteht nämlich darin, eine übergebene Menge von Parametern zu validieren und sie im Erfolgsfall an die richtigen Stellen zu schreiben, so dass sie aktiv werden. Hier spielt auch der Gültigkeitsbereich mit hinein, weil instanzspezifische Einstellungen für jede Portalinstanz in andere Dateien geschrieben werden müssen. Weiterhin kann man einen Standardwert oder einen kurzen erklärenden Kommentar an Parameter-Definitionen anhängen, so dass nicht-gesetzte Werte gegebenenfalls automatisch gefüllt werden können bzw. der konfigurierende Benutzer über den Zweck der Option unterrichtet werden kann.

Schablonenparameter

Wie bei den Ursachen für Fehlkonfigurationen bereits angesprochen wurde, sind viele Parameter unterteilt. So besitzt die URL „`http://${Hostname}:${Port}`“ beispielsweise zwei variable Teile, von denen der erste ein Hostname und der zweite ein Port ist. Durch die Unterstützung derartiger Schablonenparameter (Templates) können Werte präziser überprüft werden, da nun nicht nur gefordert ist, dass es sich um eine URL handelt, sondern auch, dass sie mit „`http://`“ beginnt und nach dem Hostbezeichner mit einer Portangabe endet. Eine ausführliche Diskussion dieses Themas befindet sich in der Implementierungsbeschreibung in Unterabschnitt 5.2.1
Die syntaktische Umrandung mit „`${`“ und „`}`“ ist den *Ant*[8]-Properties entnommen, so dass sie tarent Entwicklern bereits vertraut ist.
Da aus mehreren Teilen zusammengesetzte Parameter große Fehleranfälligkeit zeigen, scheint der zusätzliche Implementierungsaufwand gerechtfertigt.

Referenzierende Parameter

Die in den Schablonen enthaltenen Teilparameter können entweder durch Angabe der ID auf eine andere Parameterdefinition verweisen oder selbst Typ und Wert angeben. Auch bei regulären Werten soll eine Referenz auf andere Parameterwerte möglich sein, um auszudrücken, dass beide stets identisch sind.
Es ist zwar nur eine Stelle in der aktuellen Portalkonfiguration bekannt, an der dies zum Problem wurde, dafür war die Fehlersuche langwierig und das Problem trat mehrfach auf. Neben dem erlangten Wissen der Beteiligten ist ein warnender Kommentar in der Konfigurationsvorlage alles, was diesen Fehler von einem erneuten Auftreten abhalten will, so dass das Referenzkonzept des *Konfigurators* besseren Schutz anbietet.
Zur Verdeutlichung sind alle bisherigen Anforderungen in Abbildung 4.1 als Klassendiagramm dargestellt.

Komponenten

Wie in Unterabschnitt 4.2.1 erläutert, ist es wichtig zu erfassen, welche Komponente welche Parameter benötigt, so dass auch Komponenten im Modell berücksichtigt

[8] `http://ant.apache.org/`

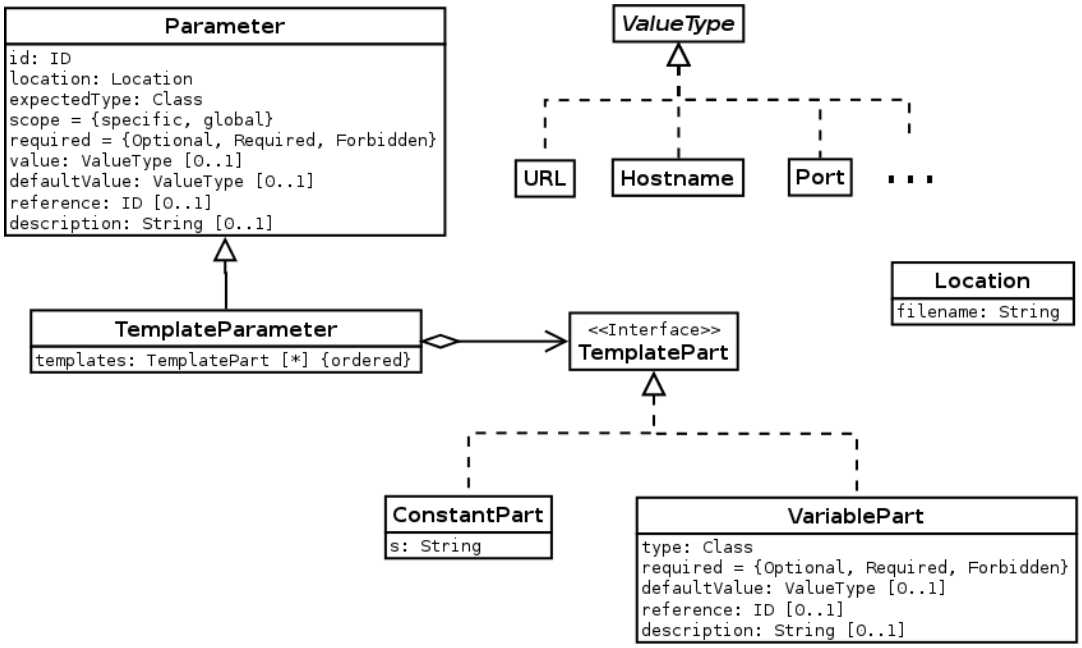

Abbildung 4.1: Ausschnitt aus dem Domänenmodell: Parameter

werden müssen. Der Begriff Komponente bezeichnet an dieser Stelle allgemein Softwaremodule jedweder Größe und Komplexität, ich unterscheide dabei zwei Arten von Komponenten: Artefakte und Systeme (deren Einführung am Ende des Abschnitts motiviert wird). Während sich ein Artefakt auf die jeweils kleinste Einheit bezieht, setzt sich ein System aus mehreren Komponenten zusammen. Für unsere Zwecke zeichnet sich ein Artefakt dadurch aus, dass es mehrere Parameter umfasst und gemeinsame Abhängigkeiten und Inkompatibilitäten definieren kann. Dabei ist zu betonen, dass dies nur in Bezug auf die Konfiguration interpretiert wird, d. h. eine Abhängigkeit zur Komponente A bedeutet, dass alle Regeln von Komponente A erfüllt sein müssen. Eine Inkompatibilität bedeutet, dass die entsprechenden Komponenten nicht gemeinsam ausgewählt werden dürfen. Es bedeutet folglich nicht, dass abhängige Komponenten heruntergeladen werden. Denn für diese Aufgabe existiert bereits eine Reihe an Werkzeugen, von denen *maven*[9] in der tarent verwendet wird. Für eine reibungslose Anbindung an *maven* wird nahegelegt, dass die Grenzen (gemeint sind die umfassten Parameter) der beschriebenen Artefakte mit den Grenzen der *maven*-Artefakte übereinstimmen. In diesem Fall kann die *maven*-Gruppen- und Artefakt-ID zusammengenommen zur Komponenten-Identifikation verwendet werden.
In dem Portalsystem kann man eine Reihe von Subsystemen identifizieren, über die man gemeinsame Aussagen treffen möchte. Um dies zu unterstützen wird das System als Konzept eingeführt; es verhält sich wie ein Artefakt, definiert aber zusätzlich, welche Komponenten es umfasst.

[9] `http://maven.apache.org/`

Für eine ausführliche Diskussion über alternative Modellierungen von Komponenten, sei auf Unterabschnitt 5.2.3 verwiesen.

Optionale Funktionalitäten und Ausprägungen

An dieser Stelle wird Punkt 4 „Inkonsistenzen“ der häufigsten Fehlkonfigurationen wieder aufgegriffen und in das Domänenmodell überführt. Nach Analyse der jetzigen Portalkonfiguration wurde nämlich festgestellt, dass Implikationen praktisch nur in Gruppen auftreten, z. B. wenn A auf *true* gesetzt ist, dann müssen die Parameter B, C und D mit bestimmten Beschränkungen gesetzt werden. Hierbei handelt es sich um eine optionale Funktionalität, wie beispielsweise einen serverseitigen Virenscanner.
Ähnlich zu Funktionalitäten verhalten sich verschiedene Ausprägungen von Komponenten. Die Besonderheit hierbei ist, dass sich unterschiedliche Ausprägungen, die sich auf das selbe Merkmal beziehen, gegenseitig ausschließen.
In Quelltext 4.1 sind drei Ausprägungen definiert, die sich alle auf das Merkmal „Datenbank“ beziehen. „Postgres“ und „Oracle“ verlangen jeweils den Parameter „`db.url`“ vom Typ URL, während „MySQL“ die Optionen „`db.hostname`“ und „`db.port`“ benötigt. Der Anspruch, dass keine der Anforderungen einer anderen Ausprägung erfüllt sein darf, muss in diesem Beispiel gelockert werden, da sonst „Postgres“ und „Oracle“ gleichzeitig „`db.url`“ erwarten und verbieten würden. Konkret wird nur gefordert, dass bei aktiver „Oracle“- oder „Postgres“-Ausprägung weder „`db.hostname`“ noch „`db.port`“ gesetzt sein dürfen und umgekehrt verbietet „MySQL“ den Parameter „`db.url`“. Für den Benutzer ändert sich durch diese Lockerung nichts, er muss sich lediglich keine Gedanken machen, dass seine Angaben durch den gegenseitigen Ausschluss unerfüllbar werden.

```
Postgres {
    db.url of type URL
}
Oracle {
    db.url of type URL
}
MySQL {
  db.hostname of type Hostname
  db.port defaultsTo '3306'
}
```

Quelltext 4.1: Beispiel für drei Ausprägungen bzgl. der Datenbank

Falls keine der Ausprägungen aktiviert ist, gilt dies auch als ein korrekter Zustand. Eine Komponente kann mehrere Ausprägungsgruppen besitzen, die sich jeweils auf unterschiedliche Merkmale beziehen, beispielsweise wird in manchen Komponenten neben der Datenbank noch zwischen einer „Master“- und einer „Slave“-Konfiguration unterschieden.
Es sei darauf hingewiesen, dass die hier vorgestellte Semantik nicht die einzig denkbare ist, aber auf Grundlage der Portalkonfiguration die Bedürfnisse am treffendsten widerspiegelt.

Implikationen zwischen nur zwei Parametern der Form „A ⇒ B“ lassen sich als optionale Funktionalität modellieren mit A als Aktivierungsbedingung und B als Konsequenz. Dies ist zwar für diesen Spezialfall umständlicher, aber kommt dem Beschreibungsaufwand im Regelfall zugute. Somit lässt sich das Klassendiagramm in Abbildung 4.2 vervollständigen.

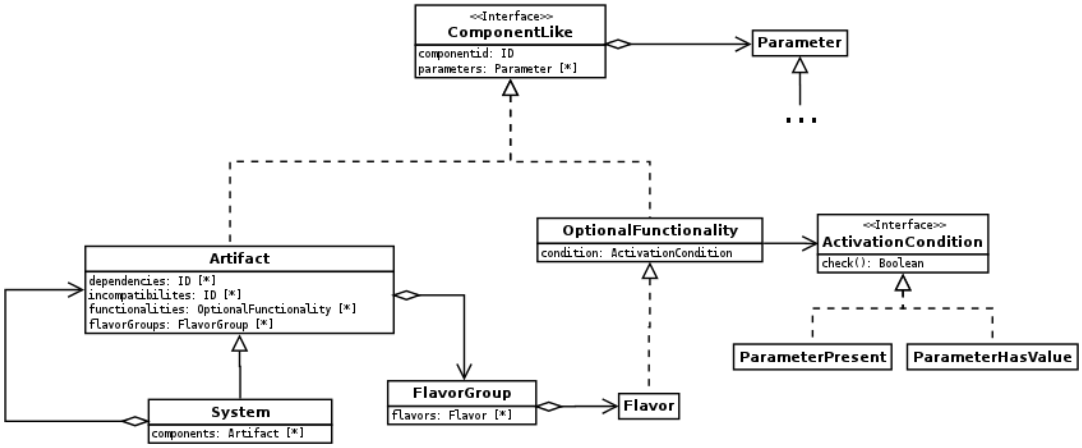

Abbildung 4.2: Ausschnitt aus dem Domänenmodell: Komponenten

Auf Basis dieses Modells werden im nächsten Abschnitt Sprachtechnologien auf ihre Eignung untersucht. Dazu wird eine Skizze (Quelltext 4.2) einer möglichen Syntax erstellt, die versucht, alle erforderlichen Elemente möglichst verständlich und kompakt darzustellen.

```
component.id {
  dependencies: artifact1, artifact2
  incompatibilities: artifact3
  components: subartifact1
  parameters {
    in config.file {
      parameter.id[Bool] defaultsTo 'false' optional specific,
      parameter2.id[URL] = 'http://${Hostname}:${Port}',
      parameter3.id[URL] = parameter2.id # 'describing comment'
    }
  }
  flavors {
    db1 :
      activatedWhen parameter.id = 'true'
      parameters {
        db.url[URL] in config.file2
      }
    db2 :
      activatedWhen ...
  }
```

```
  functionality virusscan {
    activatedWhen ...
  }
}
```

Quelltext 4.2: Skizze einer möglichen Syntax der zu erstellenden Sprache

4.3 Sprachtechnologien

4.3.1 Fragestellung

Welche Technologien bieten sich für die Erstellung der Beschreibungssprache an? Dabei sollten folgende Anforderungen erfüllt werden:

- Sie soll einfach erlernbar sein und eine präzise und kompakte Beschreibung erlauben. Daher sind eine verständliche und konsequente Syntax sowie eine intuitive Semantik Pflicht; als Anhaltspunkt dient hierzu die Skizze aus Quelltext 4.2. Als Hauptanwender werden dabei nicht nur Software-Entwickler, sondern auch Projektleiter erwartet.
- Sie soll für zukünftige Anforderungen erweiterbar sein.
- Eine gute IDE-Unterstützung mit Syntaxvervollständigung und -hervorhebung ist wünschenswert, aber bei der angestrebten Einfachheit der Sprache nicht zwingend nötig.
- Der Aufwand für das Parsing soll möglichst gering ausfallen.

4.3.2 Domänenspezifische Sprachen

Man bezeichnet eine derartige Sprache aufgrund ihrer Beschränktheit auf eine Anwendungsdomäne auch als Domänenspezifische Sprache (Domain-specific Language/DSL). Im Gegensatz zu generischen Programmiersprachen beschränkt sich eine DSL auf die für den Anwendungsfall minimal benötigten Funktionen, z. B. verzichten DSLs für gewöhnlich auf Kontrollstrukturen und Variablen. Für einen Überblick über die Publikationen in diesem Bereich, sei auf [vDKV00] verwiesen. Dabei wird zwischen internen und externen DSLs unterschieden. Interne DSLs sind in der Hostsprache geschrieben, während externe DSLs in einer separaten Sprache verfasst sind und geparst werden müssen.
Interne DSLs haben den Vorteil, dass keine zusätzliche Einarbeitung in externe Technologien erforderlich ist und man die gewohnte IDE-Unterstützung in Form von Syntaxvervollständigung und -hervorhebung während des Verfassens erhält. Dagegen ist man in der Syntaxwahl je nach verwendeter Sprache mitunter sehr eingeschränkt.
Externe DSLs besitzen in der Regel ein stark beschränktes Typsystem und erschweren Fehlermeldungen. Während interne automatisch auf die vollen Fähigkeiten der Hostsprache zurückgreifen können, muss man bei externen selbst dafür Sorge tragen, dass

Zeileninformationen zu den Stellen gelangen, an denen sie mit hilfreichen, inhaltlichen Informationen angereichert werden können[10].
Wegen des wegfallenden Parsing-Aufwands und den besprochenen Features wird zunächst auf eine zusätzliche Technologie verzichtet und eine prototypische interne DSL entwickelt, um abschätzen zu können, ob die mögliche Syntax den Anforderungen genügt. Diese DSL wird wahrscheinlich später für kleine Integrationstests nützlich sein, da sie Testbeschreibungen auf kompakte Art erzeugen lässt. Falls sie als ungenügend beurteilt wird, wird eine externe DSL geschaffen.

4.3.3 Sprachwahl für den Prototyp der internen DSL

Da die *JVM* als Zielplatform vorgegeben ist, kommen als Sprache für die interne DSL nur Java, Groovy, Scala, JRuby oder Clojure in Frage.
Java, das derzeit die überwiegend eingesetzte Sprache in der tarent darstellt, hat den entscheidenden Nachteil, dass sie einen hohen syntaktischen Overhead und keine speziellen DSL-Fähigkeiten besitzt (viel mehr als die Verkettung von wohl benannten Funktionsaufrufen ist nicht möglich). JRuby und Clojure müssen aufgrund fehlender Kenntnisse – sowohl bei mir als auch in der tarent – ausgeschlossen werden. Groovy hat zwar eine sehr ähnliche Syntax zu Java, aber zwei deutliche Nachteile gegenüber Scala. Erstens ist aufgrund der dynamischen Natur der Sprache die IDE-Unterstützung nur sehr mäßig und zweitens beherrsche ich Scala, während mir Groovy neu ist.
Scala bietet unter anderem mit impliziten Konvertierungen, faktisch überladbaren Operatoren und Semikolon-Interferenz viele DSL-freundliche Features an[11]. Es ist zudem komplett Bytecode-kompatibel zu Java und erlaubt die Verwendung sämtlicher Java-Klassen. Auch außerhalb des DSL-Kontextes profitiert man von Scalas Ausdrucksstärke durch funktionale Elemente, wie z. B. Funktionen höherer Ordnung und Typinterferenz. Den größten Nachteil im Vergleich zu Java stellt die geringe Verbreitung innerhalb der tarent dar, aber es wurde von mehreren Seiten Interesse an Scala bekundet und bereits von der Geschäftsleitung eine Schulung in Aussicht gestellt. Zusätzlich gibt es eine engagierte Lerngruppe, so dass für die spätere Pflege außer mir auch andere Personen gewählt werden können. Ein kleines, internes Projekt bietet sich zudem besonders gut an, um eine neue Sprache einzuführen.

4.3.4 Interne DSL in Scala

Auf Grundlage der Domänenklassen wurde nun eine interne DSL in Scala erstellt, die es ermöglicht, Beschreibungen in der in Quelltext 4.3 zu sehenden Form anzugeben.

```
ArtifactDescription("component.id") {
  dependencies = "artifact1" + "artifact2"
  incompatibilities = "artifact3"
```

[10] Es versteht sich, dass auch interne Sprachen nicht von Haus aus immer verständliche Fehlerinformationen liefern, sondern hier manuelles Nachbessern erforderlich werden kann.
[11] Für näheres Interesse sei [Gho11] empfohlen.

```
  parameters("config.file")(
    oneOfType[Bool]("parameter.id") defaultsTo false optional
        specific,
    oneTemplateOfType[URL]("parameter2.id") hasToBe "http://" +
        a[Hostname] + ":" + a[Port],
    oneOfType[URL]("parameter3.id") sameAs "parameter2.id"
  )
  flavors(
    "db1"
    activatedWhen("parameter.id" is true)
    parameters("config.file2")(
      oneOfType[URL]("db.url")
    ),
    "db2"
    activatedWhen ...
  )
  functionality(
    "virusscan"
     activatedWhen ...
  )
}
```

Quelltext 4.3: Ausschnitt aus der Beschreibung in der internen DSL

Dabei zeigte sich, ungeachtet der Syntax, die sich mit mehr Aufwand glatter gestalten ließe, ein unvorhergesehenes Problem, nämlich die mit acht Sekunden sehr lange Startzeit des Scala Interpreters. Umgehen könnte man dies, indem man die Beschreibungsklassen nach der Erstellung kompiliert und den *Konfigurator* direkt den Bytecode einlesen lässt. Dieses Vorgehen hat drei nicht unerhebliche Nachteile. Zum einen muss man die Beschreibungen mit „import“ Anweisungen und Objektgerüsten versehen, was überflüssigen Ballast darstellt, und zum anderen ist die eingelesene Version nicht mehr menschenlesbar. Man kann zwar noch den Quelltext dem Archiv beilegen, muss sich aber dann darauf verlassen, dass er der aktuellen Version entspricht. Der dritte und schwerwiegendste Nachteil ist, dass jedes Projekt Scala als zusätzlichen Schritt in seine Buildkette aufnehmen muss, was den Beschreibungsprozess unnötig erschwert und die Akzeptanz mindert.
Als Konsequenz dessen wird die interne DSL als Medium für den Anwender fallengelassen, wird aber weiterhin für das kompakte Erzeugen von Testdaten verwendet. Stattdessen werden nun die möglichen Technologien für eine externe DSL evaluiert.
Trotz des Scheiterns der internen DSL wird wegen der in Unterabschnitt 4.3.3 genannten Gründe an Scala zur restlichen Implementierung festgehalten.
An dieser Stelle möchte ich noch eine Erkenntnis festhalten, zu der ich während der Entwicklung der internen DSL gelangte und die eines der Argumente für diese abschwächt. Zwar schreibt man in einer vertrauten Sprache, doch unterscheidet sich die Arbeitsweise erheblich, so dass man sich neue Entwicklungsmuster erarbeiten muss und nicht auf vertraute zurückgreifen kann. Es gestaltet sich bisweilen als eine Art Knobelaufgabe, bei der anderweitig geltende Regeln über Bord geworfen werden

müssen.

4.3.5 Evaluation der Technologien für die externe DSL

Bei der externen DSL steht man vor der Wahl, die Beschreibung entweder in einer vorhandenen Sprache zu formulieren oder eine eigene Sprache mit angepasster Struktur zu entwickeln.

Bereits existierende Sprachen

Es werden ausschließlich gängige Formate betrachtet, da man andernfalls vor eben so unbekannten Technologien und Sprachelementen steht, wie bei einer eigenen DSL und darüber hinaus in der syntaktischen Struktur eingeschränkt ist.

Datenaustauschformate *XML* ist der wahrscheinlich bekannteste Vertreter dieser Kategorie und kann nicht nur mit sehr guter IDE-Unterstützung, sondern auch durch Vertrautheit punkten. Nichtsdestotrotz ist es eine sehr geschwätzige Sprache und durch seine strikte, umständliche Syntax nicht für die Bearbeitung durch Menschen geschaffen. Vor allem die negativen Erfahrungen mit *Ant*[12] haben in der tarent zu starken Vorbehalten gegen handgeschriebene *XML*-Dateien geführt. Es sei auch an dieser Stelle noch einmal betont, dass dieses Projekt auf die Akzeptanz der Mitarbeiter angewiesen ist, da es von der Güte der Beschreibungsdaten lebt. Aber auch auf technischer Ebene werden die Schablonenparameter enorme Schwierigkeiten bereiten und man wird wahrscheinlich auf einen zusätzlichen Parser ausweichen müssen. *XML* wird aufgrund seiner Syntax ausgeschlossen und stattdessen *YAML* als Alternative betrachtet.
YAML[13] ist eine Umsetzung von *XML* in einer kompakteren, freundlicheren Syntax, welche aber die Sprache ebenfalls durch seine strikte Struktur beschränkt. So ist in Quelltext 4.4 gut erkennbar, wie die Parameterdefinition aufgebläht wird und der Schablonenteil gar als ungeparster String angegeben werden muss, um eine zusätzliche Schachtelungsebene zu vermeiden.

```
component:  component.id
  dependencies:  artifact1, artifact2
  incompatibilities:  artifact3

  parameters:
    location:  config.file
      - id:  parameter.id
        type:  Bool
        default:  true
        optional
        specific
```

[12] http://ant.apache.org/
[13] http://www.yaml.org/

```
      - id:  parameter2.id
        type:  URL
        value: "http://${Hostname}:${Port}"
  ...
```

Quelltext 4.4: Skizze der Beschreibung in YAML

Das gleiche trifft auch auf *JSON*[14] zu, wie Quelltext 4.5 demonstriert. Zusätzlich fallen die Anführungszeichen, die auch um eigentliche Schlüsselworte der Beschreibungssprache gesetzt werden müssen, negativ auf. Die `location` Angabe wurde in die einzelnen Parametereinträge geschoben, um einer weiteren Listenstruktur (`[{},{}]`) zu entgehen.

```
{
  "component" : "component.id",
  "dependencies" : ["artifact1", "artifact2"],
  "incompatibilities" : ["artifact3"],
  "parameters" :
  [{
    "id" : "parameter.id",
    "location" : "config.file",
    "type" : "Bool",
    "default" : true,
    "optional" : true,
    "specific" : true
  },
  {
    "id" : "parameter2.id",
    "location" : "config.file",
    "type" : "URL",
    "value" : "http://${Hostname}:${Port}"
  },
  ...
  ]
  ...
}
```

Quelltext 4.5: Skizze der Beschreibung in JSON

Im Endeffekt eignet sich keine der betrachteten Sprachen für unsere externe DSL. Beim Verfassen dieser Beispiele wurde mir noch einmal deutlich, dass die prinzipiell klare Struktur der beiden Sprachen nicht intuitiv genug ist, um sie ohne mehrfaches Nachschlagen niederzuschreiben.

Grafische Beschreibung Der Vollständigkeit halber sei erwähnt, dass auch grafische Notationen in Erwägung gezogen wurden. Konkret würde sich *UML* anbieten, da eine gewisse Vertrautheit vorausgesetzt werden darf. Bei der zu erwartenden listenförmigen

[14] http://www.json.org/

Beschreibungen wird allerdings kein Vorteil gegenüber einer textuellen Repräsentation gesehen. Zudem ist ungewiss, ob es brauchbare, freie Werkzeuge gibt, mit denen *UML*-Diagramme gezeichnet und anschließend daraus Java-Objekte erzeugt werden können.

Parser Generatoren und Parser Kombinatoren

Da die syntaktischen Einschränkungen existierender Sprache nicht in Kauf genommen werden, bleibt als letzte Alternative noch, eine eigene Sprache zu definieren und einen Parser für diese zu schreiben. Mit einer selbst geschriebenen Grammatik sind wir auf der sicheren Seite, was zukünftige Erweiterungen anbelangt und müssen keine Sorge haben, dass eine vorgegebene Struktur uns einschränkt. Zu diesem Zweck existieren ausgereifte Werkzeuge, so dass man keinen Parser mühsam von Hand implementieren muss, sondern sogenannte Parser Generatoren oder Parser Kombinatoren verwenden kann.
Parser Kombinatoren implementieren die Grammatik mit Hilfe von Parserobjekten in der Hauptsprache, in diesem Fall Scala. Dies stellt einen Vorteil gegenüber Parser Generatoren da, die ebenfalls aus einer Grammatik einen Parser erzeugen, dabei aber auf eine externe Sprache bzw. Bibliothek zurückgreifen. Andererseits stellt die Parser-Kombinator-Bibliothek in Scala eine eigene DSL dar, so dass auch hier eine Einarbeitung notwendig ist. Aufgrund des ineffizienten Backtracking Algorithmus ([OSV10]) und der fehlenden Unterstützung für anspruchsvolle Parsing-Probleme und Fehlerbehandlung ([FP11], Kapitel 22) wird ein Parser Generator gewählt.
Eine Internetrecherche lieferte eine Auswahl an Parser Generatoren, die Java Code erzeugen können: ANTLR[15], javacc[16] und Coco/R[17]. In dieser Auswahl wurden nur unter Linux lauffähige und kombinierte Parser und Lexer Generatoren berücksichtigt, da die Einführung einer separaten Lexertechnologie vermeidbaren Zusatzaufwand bedeutet. Die Wahl fiel wegen seiner weiten Verbreitung und der ausführlichen Dokumentation auf das ausgereifte ANTLR . Zudem besitzt es eine IDE, die bei der Erstellung und dem Debugging von Grammatiken hilft.

xText

Da die mit ANTLR entwickelte Sprache, weder auf Syntaxvervollständigung noch auf -hervorhebung oder ein existierendes Typsystem zurückgreifen kann, wurde untersucht, ob xText[18] dagegen Abhilfe verschaffen kann. Zwar wären die zusätzlichen Features an die *Eclipse*-Plattform gebunden, aber solange, bei Verzicht auf diesen Komfort, der *Konfigurator* auch ohne *Eclipse* läuft, ist dies akzeptabel. Denn xText scheint kaum mehr Aufwand als ANTLR allein zu verursachen.
Jedoch musste ich mich gegen xText entscheiden, da es inkompatibel mit unseren auf *maven* basierenden Buildsystemen ist. Es lässt sich nämlich quasi nicht außerhalb

[15] `http://www.antlr.org/`
[16] `http://javacc.java.net/`
[17] `http://ssw.jku.at/coco/`
[18] xText verspricht ausgereifte IDE-Unterstützung für selbst geschriebene Sprachen (`http://www.xtext.org/`).

von *Eclipse* bauen, da ein Großteil der Abhängigkeiten nur in der *Eclipse*-Umgebung gebündelt und nicht einzeln releast wird[19]. Man müsste also für jede Abhängigkeit ein eigenes *maven*-Projekt erstellen und pflegen, um beständige Builds zu erhalten, was vom Aufwand her nicht vertretbar ist.

4.4 Skizze eines Installationsablaufs

Dieser Abschnitt schließt mit einer Skizze eines möglichen Installationsablaufs (dargestellt in Abbildung 4.3) die Evaluation ab und deutet an, wie der *Konfigurator* mit Konfigurationsmanagementsystemen harmonieren würde. Er zeigt also, dass sich Gedanken um die Integration der im nächsten Kapitel vorgestellten Lösung gemacht wurden.

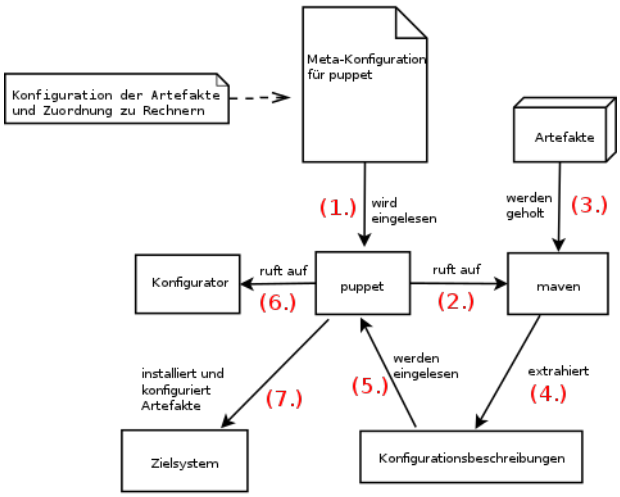

Abbildung 4.3: Skizze der Einbindung des *Konfigurators* in den Installationsprozess

Jeder Rechner des Portalsystems benötigt eine eigene Konfiguration, die möglicherweise aus anderen Konfigurationen abgeleitet wird, d.h. sie könnte aus einer vom Anwender erstellten Meta-Konfigurationsdatei erzeugt werden. Ein Konfigurationsmanagementsystem, z. B. *puppet*, verwaltet diese Datei (1.), in der u. a. steht, welche Artefakte auf welchem System installiert werden sollen und welche Konfiguration für welches System verwendet werden soll. Zu den Artefakten zählen auch Datenbanksoftware, LDAP Clients usw., denn schließlich soll die Verwaltung von Systemkomponenten und selbst geschriebenen Anwendungen vereinheitlicht werden. Die bisherigen Schritte beherrscht *puppet* bereits grundsätzlich. Nun werden, von *puppet* überwacht, die zu installierenden Portalprogramme per *maven* geholt (2. und 3.), in deren Archiven sich auch die Konfigurationsbeschreibungen befinden (4. und 5.). An dieser Stelle kann der *Konfigurator* eingebunden werden, denn *puppet* besitzt jetzt sowohl

[19] Teilweise gibt es inoffizielle Versionen Dritter.

konkrete Konfigurationen für die einzelnen Hosts (aus der Meta-Konfigurationsdatei) als auch die Beschreibungsdateien (aus den *maven*-Artefakten), mit denen der *Konfigurator* aufgerufen wird (6.). Damit kann der *Konfigurator* ein Validierungsergebnis liefern und ggf. Konfigurationsdateien erzeugen, die *puppet* samt der Artefakte an die zu konfigurierenden Rechner ausliefert (7.).

5 Realisierung

Dieses Kapitel stellt die Besonderheiten und Abwägungen bei der Implementierung vor. Bevor in den einzelnen Abschnitten spezielle Aspekte herausgegriffen werden, sei in einem Überblick das Zusammenspiel der Teile umrissen.
Für das Einlesen der Beschreibungsdateien wird mit ANTLR ein Parser aus einer kontextfreien Grammatik generiert. Der vom Parser erzeugte Abstrakte Syntaxbaum wird von sogenannten „Baumläufer"-Klassen (*tree-walk*[1]) in die Domänenklassen umgewandelt. Hierbei findet dank der Bytecode-Kompatibilität ein stufenloser Übergang zwischen dem in Java geschriebenen Parser und den in Scala verfassten Baumläufer-Klassen statt. Nachdem die Beschreibungsdatei ggf. mit weiteren Beschreibungen vereinigt und in das Validierungsmodell überführt wurde, findet die Erfüllbarkeitsprüfung statt, so dass auf der resultierenden Grundlage anschließend Konfigurationen eingelesen und validiert werden können. Bei der Überführung in das Validierungsmodell wird in erster Linie die Semantik von Ausprägungen und optionalen Funktionalitäten in ausführbare Regeln übersetzt und die Bedingungen und Parameter werden zur leichteren Überprüfung gruppiert.

5.1 Die erschaffene Sprache

Da ich bei dieser Sprache frei in der Syntaxwahl war, ist das Resultat nahezu identisch mit der in Quelltext 4.2 auf Seite 18 skizzierten Sprache. Lediglich ein Versionsnummerfeld der Form „`version: 0.1`" wurde als Kopfzeile hinzugefügt, so dass zwischen verschiedenen Versionen der Beschreibungssprache differenziert werden kann, da erwartet wird, dass sich die Syntax erst während der Praxiserprobung festigt. Zwar könnte die Versionsnummer für die erste Revision theoretisch ausgelassen werden und alle Beschreibungen ohne Angabe implizit dieser zugeordnet werden, aber aus den folgenden zwei Gründen habe ich mich für diesen Overhead entschieden. Erstens ist dem Nutzer so explizit bewusst, welche Version und damit welche Syntax er verwendet, und zweitens gewöhnt er sich so von Anfang an dieses Datum, dessen spätere Notwendigkeit erwartet wird.

5.1.1 Lexer

Der Lexer erzeugt aus dem Eingabetext einen Strom an Tokens (*Tokenstream*), die als Grundbausteine für die Grammatik des Parsers verwendet werden.
Wie in Abschnitt 4.2.2 bereits erwartet, verursachen die Schablonenparameter zusätzlichen Aufwand bei der Erstellung der Tokens. Dies rührt daher, dass Werte-Strings

[1] Siehe dazu auch Kapitel 24 in [FP11].

nicht als einfache Strings markiert werden können, sondern erneut geparst werden müssen. Zwar könnte man sie auch nicht-analysiert an die Anwendung weiterreichen und dort einen eigenen Parser implementieren, aber um genau diese fehleranfällige Arbeit zu vermeiden, wurde ANTLR gewählt. Dass die Parsing-Aufgabe nicht trivial ist, zeigt die allgemeinste Variante[2] eines Schablonenstrings in Quelltext 5.1. Zusätzlich kann der String natürlich noch einen gewöhnlichen Wert darstellen.

```
template: (STRING | templatePart)+;

templatePart : '${'
  (ID ('optional')?)
| (TYPE ('defaultsTo' '=' STRING)? ('optional')?
  ('#' STRING)?)
  '}';
```

Quelltext 5.1: Schablonenparameter in der ANTLR-Grammatik

Je nachdem, ob sich der Lexer gerade außerhalb von Anführungszeichen, zwischen Anführungszeichen, aber nicht in einer „${}“-Gruppe oder innerhalb von „${...}“ befindet, gelten andere Regeln für die Tokenerstellung. Beispielsweise dürfen in Zeichenketten „parameters“ oder „URL“ nicht als Schlüsselwort bzw. Typ markiert werden, während in den Schablonenabschnitten eine eingeschränkte Menge an Schlüsselworten und Typen als solche ausgezeichnet werden müssen.
Einen naheliegenden Implementierungsansatz[3] zeigt [FP11] in Kapitel 28, bei dem über durch bestimmte Zeichen ausgelöste Code-Aktionen Zustandsfelder des Lexer verändert werden und bestimmte Lexer-Regeln zustandsabhängig aktiv werden. Aber selbst bei den zwei mindestens benötigten Zustandsvariablen wächst der generierte Java-Bytecode für die Parsing-Methode auf über 64 KB an und löst den Java-Kompilationsfehler „`code too large`“[4] aus, weil die Anzahl der Schlüsselworte einschließlich der Datentypen recht hoch ist und der Bedingungscode in jede Regel dupliziert wird. Dieses Problem lässt sich prinzipiell umgehen, indem man den Lexer vom Parser separiert oder den Parser auf mehrere Dateien aufteilt. Bei der Aufteilung des Parsers, die nach Zustandsvariablen sortiert stattfindet, tritt ein ungelöster Bug[5] in ANTLR auf, der zustandsgesteuerte Regeln betrifft und ohne erkennbares Muster viele Verteilungen mit einem Kompilationsfehler quittiert. Insgesamt kann man es damit zwar schaffen knapp unter der 64 KB Grenze zu bleiben, aber selbst kleinste Änderungen oder Ergänzungen an der Grammatik verändern die Situation wieder, so dass kein vernünftiges Arbeiten möglich ist.
Als Alternative wird für jeden Zustand ein separater Lexer verwendet. Durch den Wegfall der Zustandsabfragen und durch das folgliche Nicht-Auftreten des oben beschriebenen Bugs können Parser und Lexer beliebig separiert werden. Der Code wird

[2] Die ausführliche Erklärung befindet sich in Abschnitt Unterabschnitt 5.2.1
[3] ANTLR unterstützt nämlich Lexer-Zustände nicht direkt ([FP11] Kapitel 28.1.2).
[4] Laut Kapitel 4.8.1 der JVM-Spezifikation [LY99] muss eine kompilierte Methode kleiner als 65536 Byte sein.
[5] `http://www.antlr.org/jira/browse/ANTLR-315`

als Nebeneffekt nicht nur viel übersichtlicher und verständlicher, sondern jeder Lexer lässt sich auch komplett unabhängig testen. Die Herausforderung bei dieser Variante stellt die Koordination unter den Lexern dar.
Bei der letztendlich gewählten Lösung werden die Lexer verschachtelt aufgerufen, d. h. sobald der erste den Beginn eines Strings erkennt, übergibt er an den nächsten Lexer, der wiederum bei „${“, dem Beginn eines Schablonenabschnitts, dem dritten Lexer die Analyse überlässt. Stoßen die Lexer auf das Ende ihres Abschnitts, senden sie `Token.EOF` und der vorherige Lexer übernimmt wieder. Die Rückgabetoken werden an die eigenen Token angefügt und zurückgereicht, so dass der Parser auf einem kontinuierlichen Strom arbeiten kann. Weil standardmäßig nur genau ein Token pro Anfrage gelesen und an den Parser weitergegeben werden kann, mussten die Lexerklassen um eine Queue erweitert werden, in der die zusätzlichen Token zwischengespeichert werden, und die `emit()` und `nextToken()` Methoden überschrieben werden. Durch dieses Vorgehen behalten alle Token ihre Zeilen- und Positionsinformationen, was bei der Erzeugung eines neuen Tokenstroms für jeden Abschnitt (mein erster Ansatz) sehr aufwendig nachträglich hinzugefügt hätte werden müssen. Als netter Nebeneffekt der verschachtelten Lexer können überall einfache und doppelte Hochkommata zur Markierung von Strings verwendet werden, also insbesondere auch einfache Hochkommata innerhalb einer mit einfachen Hochkommata umschlossenen Schablone. Die Information, welches der beiden einen konkreten Bereich abschließt, wird dem entsprechenden Lexer beim Aufruf übergeben.

5.1.2 Konstruktion und Auslesen des Abstrakten Syntaxbaumes

Der Parser erzeugt auf Grundlage der definierten Grammatik aus dem Tokenstrom einen Abstrakten Syntaxbaum (AST). ANTLR bietet dafür besondere Sprachmittel an, so dass die Baumerzeugung ohne nennenswerten Mehraufwand direkt in den Parsing-Regeln geschieht[6]. Dieser Baum wird im Scala-Code durchlaufen und die Elemente in die Klassen des Domänenmodells, das im kommenden Abschnitt 5.2 näher beschrieben wird, umgewandelt.
In dem Zusammenhang wurde untersucht, ob es Sinn ergibt, die interne DSL hinter die externe DSL zu setzen, weil sie sowieso vorhanden ist und noch für Testzwecke genutzt wird (vgl. Unterabschnitt 4.3.4). Mit der Hintereinanderschaltung würde sie aktiv benutzt werden, so dass eine Veraltung oder etwaige Fehler schnell auffallen würden. Des Weiteren müssen interne und externe DSL beide die Modellobjekte erzeugen, so dass es sinnvoll scheint, diese Aufgabe an einer Stelle zu kapseln. Dagegen spricht aber, dass die interne DSL noch recht unausgereift ist und sich somit oft ändern wird, gerade wenn sie stets zur externen passen soll. Schlussendlich ist der Anspruch an die interne wesentlich geringer, so dass eine Angleichung an die Fähigkeiten der externen einen zu hohen Aufwand verursachen würde. Als Konsequenz findet die Kapselung gemeinsamen Verhaltens in den Builder-Klassen statt, die in Unterabschnitt 5.2.2 im Zusammenhang mit den entstandenen Schichten noch einmal näher beleuchtet

[6] Dem interessierten Leser sei für weitere Informationen [Par07] empfohlen.

werden.

5.1.3 Fehlerbehandlung

Es stellt eine wichtige Frage dar, wo auf welche Fehler geprüft und wo welche Fehlermeldung erstellt werden soll. Einerseits lautet eine Grundregel, Fehler so früh wie möglich abzufangen, um in der folgenden Logik unter klaren Voraussetzungen agieren zu können, andererseits ist es bei vielen Fehlern umständlich bis unmöglich, diese im Parser abzufangen. Zum Beispiel kann in einer kontextfreien Grammatik nicht ermittelt werden, ob ein Parameter bereits mit Typ- und Ortsangabe definiert wurde. Weiterhin fehlt dem Parser der semantische Kontext, um für den Benutzer hilfreiche Meldungen herauszugeben. So ist es wenig sinnvoll, einem Parameter zugleich einen Standardwert und einen festen Wert zuzuweisen, weil der Standardwert nur bei ungesetztem Wert relevant ist. Falls man dies in der Grammatik verbietet, würde die Fehlermeldung in etwa „Unerwartetes Token bei 'default'." lauten, ohne weitere Begründung oder Hinweise.
Daher sind die ANTLR-Regeln eher locker gehalten und fehlerhafte Beschreibungen werden größtenteils erst bei der Baumerzeugung abgefangen. Genauer gesagt werden dort Fehler der Kategorie `InvalidDescriptionException` abgefangen, wozu in erster Linie widersprüchliche und fehlende Parameterdefinitionen gehören. Inhaltliche Logikfehler können erst durch die Validierungsklassen erkannt werden und werden als `UnsatisfiableDescriptionException` geworfen. Um diese Ausnahmen so informativ wie möglich zu gestalten, werden die Positionsinformationen der einzelnen Worte aus dem AST mit in die Domänenklassen aufgenommen. Ergänzende Informationen zur Fehlerbehandlung bei der Validierung finden sich in Abschnitt 5.4.

5.1.4 Nutzerfreundliche Features

Da einfache Erlernbarkeit und Benutzerfreundlichkeit die Hauptgesichtspunkte waren, nach denen diese Sprache entwickelt wurde, seien die Ergebnisse dieser Bemühungen hier zusammengefasst.
Neben einer möglichst intuitiven und klaren Syntax und hilfreichen Fehlermeldungen, können als drittes Feature bei Parametern zuvor definierte Anforderungen weggelassen werden. Dies bedeutet nicht nur weniger Schreibaufwand und dient der Übersichtlichkeit, sondern begrenzt auch Änderungen auf eine Stelle. Dabei gelten die Parameteranforderungen, die in einem bedingten Block definiert wurden, nur für diesen Block, während im globalen `parameters{...}`-Block angegebene in der restlichen Datei gelten. Quelltext 5.2 verdeutlicht den Unterschied.

```
component.id {
   flavors {
    one :
      activatedWhen one[Bool] in config.file specific optional
         = 'true'
      parameters { two[IP] in hosts.file = '42.42.42.42' }
```

```
    two :
      activatedWhen one[Bool] in config.file specific optional
         = 'false'
      parameter { two[IP] in hosts.file = '5.5.5.5' }
   }
}

component.id {
  parameters {
    one[Bool] in config.file specific optional,
    two[IP] in hosts.file
  }
  flavors {
    one :
      activatedWhen one = 'true'
      parameters { two = '42.42.42.42' }
    two :
      activatedWhen one = 'false'
      parameters { two = '5.5.5.5.' }
  }
}
```

Quelltext 5.2: Gegenüberstellung von redundanter und kompakter Beschreibung

Die Definitionen werden beim Durchlaufen des ASTs von den beiden Klassen `ParameterCollector` und `ConditionalParameterCollector` gesammelt und Widersprüche mit einer `InvalidDescriptionException` beantwortet (siehe auch Unterabschnitt 5.1.3). Da die Beschreibungen einzelner Komponenten voneinander unabhängig sein sollen, werden keine anderen Beschreibungsdateien zur Auflösung fehlender Typangaben o. Ä. herangezogen. Es wird aber geprüft, dass in verschiedenen Dateien keine widersprüchlichen Aussagen getroffen werden. Das Festlegen von „benötigt“ und „global“ als Standardwerte erhöht die Übersichtlichkeit zusätzlich.
Als weitere Erleichterung kann unabhängig vom Kontext frei zwischen einfachen und doppelten Hochkommata entschieden werden, es wird auch, wo immer möglich, ganz auf sie verzichtet.
Zum Schluss sei noch angemerkt, dass bei diesen kleinen Beispielen der syntaktische Overhead verhältnismäßig groß erscheint, aber in realistischen Szenarien mit längeren Parameterlisten kaum ins Gewicht fallen wird. Mit der selbst entwickelten DSL ist es aber auch einfach möglich, beispielsweise eine kompaktere Schreibweise für Ein-Parameter-Blöcke einzuführen, falls es gewünscht wird. Im Moment liegt der Fokus auf einer klaren Struktur ohne Spezialfälle, um die Einarbeitung nicht zu erschweren.

5.2 Domänenmodell im Detail

Die Entstehung des Klassenmodells wurde bereits in Unterabschnitt 4.2.2 dargelegt. Deshalb werden in diesem Abschnitt lediglich besondere Aspekte der Implementierung hervorgehoben sowie wichtige Abwägungen diskutiert.

5.2.1 Schablonenparameter

Wie bereits in Abschnitt 4.2.2 angesprochen, dienen Schablonenparameter dazu, aus mehreren Teilen bestehende Parameter zu beschreiben. Dabei wird zwischen festen und variablen Teilen unterschieden; die festen verlangen eine exakte Zeichenfolge, während die variablen Teile einen eigenen eingebetteten Parameter definieren können. Weil dieser Parameter nicht außerhalb der Schablone verwendet wird, ist weder eine Angabe der ID noch des Orts, wozu auch der Gültigkeitsbereich (Instanzbezogenheit) gehört, notwendig. Ebenso fällt die Angabe eines festen Werts weg, da dieser Fall durch den festen Teil abgedeckt wird, so dass nur noch der Typ, die Optionalität, ein Standardwert und eine Beschreibung übrig bleibt. Alternativ kann auch eine Referenz auf einen anderen Parameter gesetzt werden, dessen Wert dynamisch übernommen wird. Abbildung 5.1 zeigt den Klassendiagrammsausschnitt mit den Schablonenparametern.

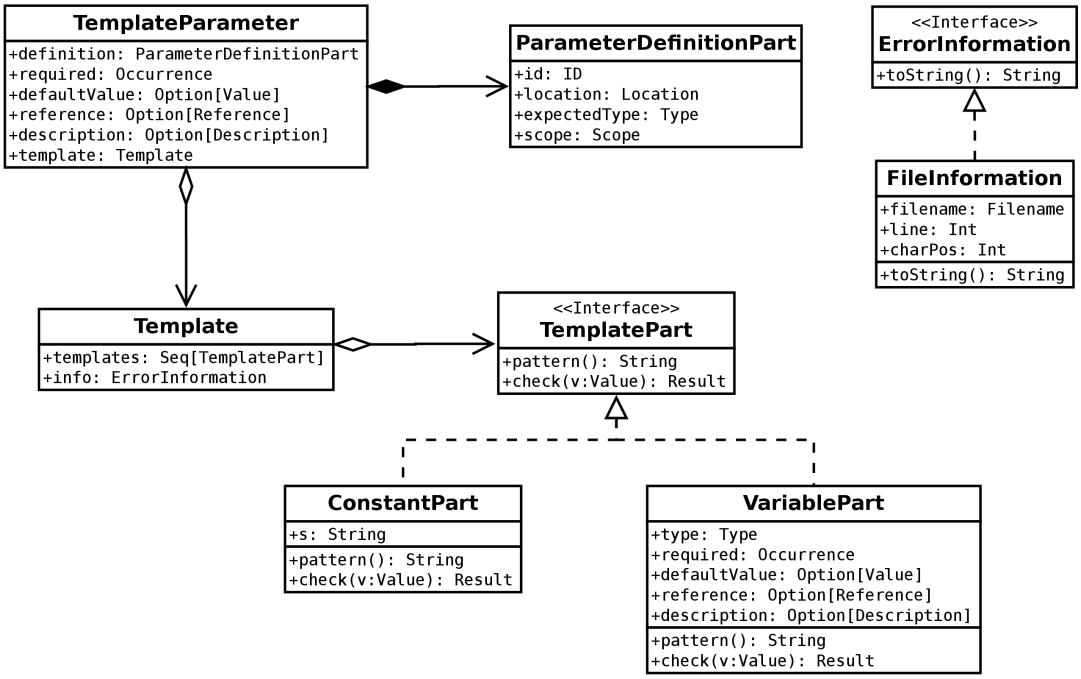

Abbildung 5.1: Klassendiagramm der Implementierung von Schablonenparametern

In dem Klassendiagramm sind bereits, wie in Scala üblich, optionale Parameter mit `Option[Typ]` versehen[7]. Die Typen `Value`, `Occurrence`, `Scope`, `Reference`, `Location` und `Description` enthalten alle, wie exemplarisch an der Klasse `Template` zusehen ist, zusätzlich zu dem eigentlich Wert noch Fehlerinformationen, die Aufschluss darüber geben, wo der Wert gesetzt wurde.

[7] Im Quellcode bringt dies mehr Sicherheit, da man vom Compiler gezwungen wird diese optionalen Werte auf Vorhandensein zu überprüfen.

Die zusehenden Methoden `checkValue` und `pattern` dienen zur Validierung eines Werts bezogen auf die Schablone. Dabei iteriert die `Template`-Klasse zuerst über alle Teile und setzt die erhaltenen Muster zu einem regulären Ausdruck zusammen. Anschließend wird der zu testende Wert anhand dieses Ausdrucks in Abschnitte eingeteilt, die jeweils den zuständigen `ConstantPart`- und `VariablePart`-Objekten zur Überprüfung übergeben werden.
Bei den Schablonenparametern ist es wichtig, die Reihenfolge, mit der feste und variable Teile aufeinanderfolgen, zu speichern. Weil sich Schablonenparameter nicht zwangsweise abwechseln müssen, sondern auch zwei variable Abschnitte hintereinanderstehen können, wird eine `Seq`[8] als Datenstruktur die Reihenfolge speichern.
Die Klasse `ErrorInformation` kapselt alle Informationen, die aus der externen DSL übertragen werden. Durch diese Kapselung und den Zugriff über die `toString`-Methode ist eine spätere Erweiterung transparent möglich.

5.2.2 Unveränderlichkeit und Builder-Klassen

Der *Konfigurator* ist in drei Schichten unterteilt, die in Abbildung 5.2 dargestellt sind.

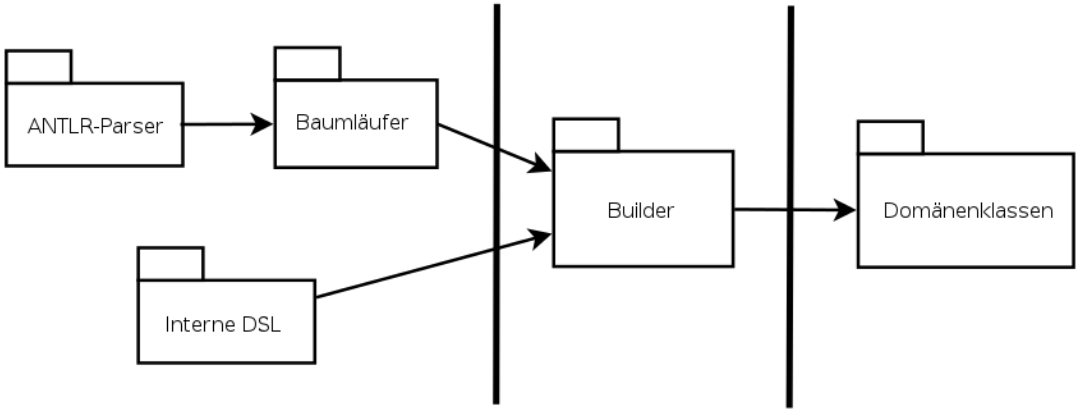

Abbildung 5.2: Schichten-Architektur

Das Domänenmodell wird mit Hilfe der *Builder*-Klassen (nach [GHJV95]) gefüllt, die wiederum entweder von der internen Scala-DSL oder über die so genannten Baumläufer-Klassen mit Daten versorgt werden. Es gibt zwei Gründe, die Modellklassen über die *Builder* abzukapseln.

1. Der Parser bzw. die interne DSL können sich unabhängig vom Modell entwickeln. Bei zwei sich verändernden Eingabetechnologien[9] sind einheitliche Schnittstellen besonders nützlich, weil sie verhindern, dass sich Änderungen in einer der beiden Sprachen direkt auf das Domänenmodell auswirken und umgekehrt.

[8] Sequenz. Einen ausführlichen Überblick über Scalas *Collections* liefert z. B. [OSV10]
[9] Beide werden natürlich mit anderer Zielsetzung und Priorität entwickelt.

2. Die Klassen des Domänenmodells sind unveränderlich, so dass in jedem Fall eine Art *Builder*-Klasse benötigt wird. Mit der Unveränderlichkeit wird das gedankliche Konzept, dass die Klassen einmal beim Durchlaufen des ASTs befüllt und danach nicht mehr geändert werden, im Code manifestiert. Zudem sind unveränderliche Klassen weniger anfällig für Fehler, weil klare Annahmen über den Zustand eines Objekts getroffen und an einer Stelle, bei der Erzeugung, erzwungen werden können[10].

5.2.3 Diskussion Systemmodellierung

Dieser Abschnitt diskutiert alternative Modellierungen der Komponenten und zeigt auf, wieso sie der jetzigen Lösung unterlegen sind.

Explizite Hierarchie

Eines der gesteckten Ziele war es, nicht nur einzelne Portalserver zu beschreiben, sondern auch einen kompletten Serververbund, der eine klare Hierarchie aufweist. So hätte die rekursive Struktur aus Artefakten und Systemen, die wiederum Systeme beinhalten können, explizit formuliert werden können, wie Abbildung 5.3 zeigt.

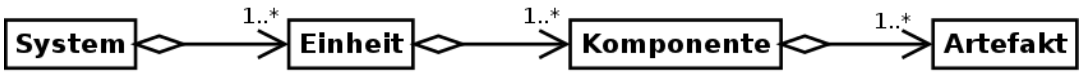

Abbildung 5.3: Alternative Hierarchie zur Modellierung von Komponenten

So hätten *Artefakte* ebenfalls *maven*-Artefakten entsprochen, mehrere *Artefakte* würden zu einer *Komponente* zusammengefasst, mehrere *Komponenten* zu einer *Einheit*, was dann genau einem Server entspräche und schließlich würden mehrere Server, als ein *System*, das komplette Cluster abbilden. Der Vorteil ist, dass die eindeutige Struktur, deren Einhaltung über Konventionen verlangt wird, zusätzliche Aussagen über die einzelnen Bausteine impliziert, z. B. wäre klar, dass mit *System* der Gesamtverbund gemeint ist und nicht eine unbestimmte Menge an Subsystemen.
Der Nachteil, der letztendlich auch zum Ausschluss führte, ist die starre Struktur, die zwar Portalsysteme sauber untergliedert, aber nur bedingt auf andere Szenarien übertragbar ist. So würde es ausreichen, dass eine Zwischenstufe benötigt wird, um das Modell zu brechen.

Beschreibung von Rechnerverbunden

Eine weitere Idee war, zwar die rekursive Unterteilung in Artefakte und Systeme beizubehalten, aber zusätzliche Klassen einzuführen, um beschreiben zu können, aus welchen Servern ein Verbund aufgebaut ist und welche Einschränkungen diesbezüglich

[10] Weitere Gründe und eine umfangreiche Diskussion findet sich in [Blo08] (Item 15).

gelten. So besteht er im Regelfall aus einem getrennten Frontend- und Backendverbund, die jeweils aus einem Server bestehen oder aus einer Master- und Slave-Konstellation, für die dann ein Lastverteiler benötigt wird. Quelltext 5.3 beschriebt diese Aussage in einer um zusätzliche Elemente erweiterten Notation.

```
Portalsystem { components: Backendsystem, Frontendsystem }

Backendsystem { components: OneOf(BackendMitLastverteiler,
   BackendOhneLastverteiler) }

BackendMitLastverteiler { components: Lastverteiler, BackendMaster,
    BackendSlave }

BackendMaster { components: JBoss, Liferay, ... }

...
```

Quelltext 5.3: Beschreibung der Rechnerkonstellation in einem Portalverbund

Im Prinzip wird die Semantik der optionalen Funktionalität bzw. der Ausprägung (bei mehreren Komponenten mit gegenseitigem Ausschluss) auf Komponentenlevel durch das Schlüsselwort `OneOf` noch einmal kopiert. Um diese Redundanz zu vermeiden, werden die Rechner durch Parameter modelliert und die Rechnerklasse mittels einer Konvention aus der Parameter-ID bestimmt. Quelltext 5.4 zeigt, wie die Beschreibung in der bereits eingeführten Notation aussieht. Dieses Szenario könnte auch mit einer Ausprägung modelliert werden, falls man bei nur einem Host den Lastverteiler verbieten will.

```
Portalsystem { components: Backendsystem, Frontendsystem }

Backendsystem {
  parameters{
    in host.config { backend.master[URL],
                     backend.slave[URL] optional
                   }
  }

  functionality BackendMitLastverteiler {
    activatedWhen backend.slave present
    parameters { backend.lastverteiler[URL] }
  }
}

BackendMaster { components: JBoss, Liferay, ... }
...
```

Quelltext 5.4: Beschreibung der Rechnerkonstellation in einem Portalverbund in der externen DSL

Diese Lösung ist elegant, da die Beschreibung im Kleinen (Parameter) und im Großen (Rechner) identisch abläuft und sie auf den eigentlichen Kern der Konfiguration, nämlich den Parametern, fokussiert bleibt. Denn letztendlich interessiert uns nicht der Rechner als solches, sondern lediglich die Adresse, unter der er verfügbar ist und welche Konfiguration er haben soll. Die Zuweisung einer konkreten Konfiguration an einen Rechner, bezeichnet durch seine Adresse, findet beim Konfigurieren und nicht in der Beschreibungssprache statt. Einziger Nachteil dieser Lösung ist, dass man das Konzept der Rechner nicht explizit in der Klassenhierarchie findet und eine Abbildung zwischen Bezeichner und Aufgabe implizit durch eine Konvention stattfindet.

5.3 Erfüllbarkeitsüberprüfung und Validierungsalgorithmus

Die erstellten Beschreibungen werden auf Erfüllbarkeit überprüft, um auszuschließen, dass die Konfiguration aufgrund von widersprüchlichen Anforderungen scheitert. Die nächsten Abschnitte beschreiben den dafür entworfenen Algorithmus sowie den Mechanismus der Validierung, der auf dieselben Methoden zurückgreift.

5.3.1 Evaluation von Constraint-Solvern

Wie in Kapitel 2 erläutert, interessiert uns im Gegensatz zum *Constraint-Satisfaction-Problem* keine konkrete Lösung, sondern lediglich die Frage nach ihrer Existenz. Da dies ein Teilproblem darstellt, wird die Implementierung bewährter Algorithmen evaluiert.
Die eigene Implementierung eines CSP-Algorithmus wird als zu aufwendig verworfen, zumal bereits fertige Lösungen existieren. Für Java steht mit *choco*[11] eine ausgereifte und gut dokumentierte Bibliothek zur Verfügung (*JaCoP*[12] wäre eine Alternative mit minimal weniger Funktionsumfang). Nach Studieren der API und einigen kleinen Testprogrammen wurde deutlich, dass *choco* nur in der Lage ist, „erfüllbar“ oder „unerfüllbar“ ohne weitere Anhaltspunkte als Ergebnis zu liefern. Dies ist zu wenig, um sinnvoll ein Problem beheben zu können. Weil der *Constraint-Solver* nicht die Validierung von Konfigurationen beherrscht, hätte dafür noch zusätzlicher Code geschriebenen werden müssen.
Folglich wurde ein eigener, im nächsten Abschnitt erklärter Algorithmus implementiert, der nicht nur konkrete Fehlerbeschreibungen liefert, sondern auch Erfüllbarkeitsprüfung und Validierung mit denselben Methoden durchführen kann.

5.3.2 Grundprinzip der Erfüllbarkeitsüberprüfung

Eine Konfiguration ist einzig und allein durch die Parameter bestimmt[13], deshalb ist eine Konfigurationsbeschreibung genau dann erfüllbar, wenn sie keine widersprüchliche Anforderungen an dieselben Parameter stellt, z. B. inkompatible Typen definiert.

[11] `http://www.emn.fr/z-info/choco-solver/`
[12] `http://jacop.osolpro.com/`
[13] siehe: Definition eines Parameter in Abschnitt 1.2.2.

Dabei muss zwischen bedingten Anforderungen (Ausprägungen und optionalen Funktionalitäten) und unbedingten differenziert werden. Die Semantik von Ausprägungen und optionalen Funktionalitäten lässt sich auf die Implikation, „Aktivierungsbedingung ⇒ Parameteraussagen“, zurückführen. Während es trivial ist, die unbedingten Aussagen auf Widersprüche zu überprüfen, wird bei Bedingungen folgendes Vorgehen angewendet, das am Beispiel von Abbildung 5.4 illustriert wird. Dabei seien `A`, `B`, `C`, `D`, `E`, `Y` und `Z` Aussagen über Parameter.

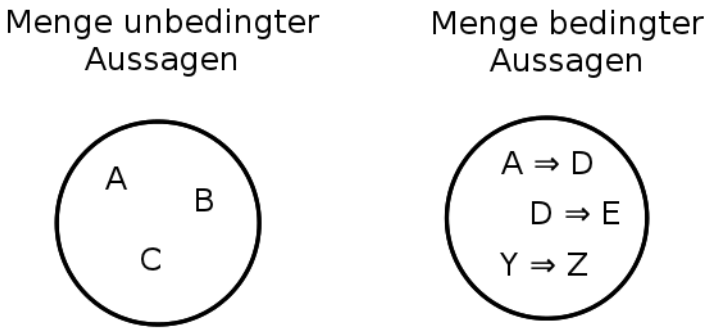

Abbildung 5.4: Beispielmengen für abstrakte Parameteraussagen

Die Gesamtmenge an Aussagen ist in definitiv zu erfüllende und in unter einer gewissen Voraussetzung zu erfüllende unterteilt. Im abgebildeten Szenario müssen `A`, `B` und `C` gelten, während `D` es nur muss, falls `A` gilt, usw..
Nun werden aus der Menge an Implikationen diejenigen herausgesucht, deren Bedingung aufgrund des vorhandenen Wissens erfüllt sein muss. Dazu wird jede Implikation betrachtet und bei Gültigkeit der Bedingung wird die zugehörige Parameteraussage in die Menge der unbedingt geltenden aufgenommen und anschließend die Implikation entfernt. Bei jeder Aufnahme wird auf Widersprüche geprüft. Dies wird solange durchgeführt, bis bei einem Durchgang keine gültigen Bedingungen gefunden wurden. Im Beispiel müssen zum Schluss `A`, `B`, `C`, `D` und `E` gelten, während `Z` nur unter der Bedingung `Y` erfüllt sein muss. Somit wurde sicher gestellt, dass für alle unbedingt geltenden Parameter konkrete Werte eingesetzt werden können, weil sie keine widersprüchlichen Anforderungen stellen. Weiterhin können alle verbleibenden Implikationen durch Nicht-Erfüllung der Bedingungen *wahr* werden. Es existiert folglich eine Konfiguration, die allen beschriebenen Anforderungen genügt. Falls bei Parametern mit endlichem Wertebereich alle Werte verboten sind, wird dies ebenfalls als Widerspruch erkannt (eine ausführliche Erklärung findet sich in Abschnitt 5.3.3).
Auch wenn es auf den ersten Blick so aussehen mag, dass Aktivierunsbedingungen, deren Erfüllung an anderer Stelle gefordert wird, unnötig seien, kann dies bei unabhängig erstellten Konfigurationsbeschreibungen mehrerer Komponenten durchaus vorkommen und daher nicht von vornherein abgewiesen werden.
Während der präsentierte Algorithmus lediglich sicherstellt, dass es eine Lösung gibt,

führt die tatsächliche Implementierung weitere Prüfungen durch, um sinnfreie Aussagen, wie folgende Ergänzungen zu Abbildung 5.4 „$\mathtt{Y} \Rightarrow \neg\mathtt{A}$“ oder „$\mathtt{X} \Rightarrow \mathit{false}$“, zu erkennen. Eine ausführliche Beschreibung der Realisierung und der dabei zu berücksichtigenden Sonderfälle liefert der nächste Abschnitt.

5.3.3 Implementierung

Validierungsklassen

Die Implementierung ist ganz auf das Erkennen von Konflikten an einem Parameter ausgerichtet, wie dem Klassendiagramm in Abbildung 5.5 zu entnehmen ist.

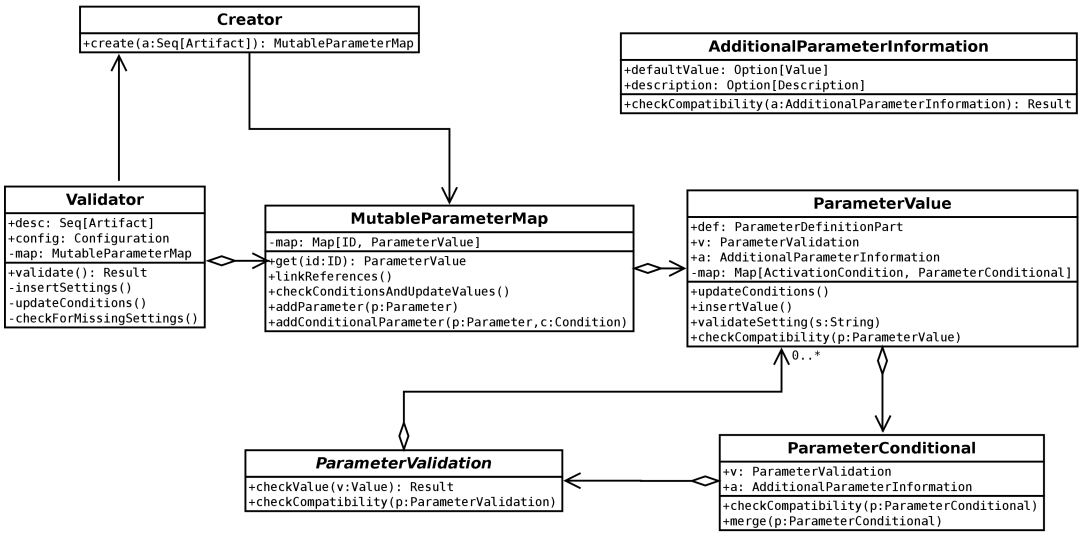

Abbildung 5.5: Klassendiagramm der Validierungslogik

Alle Informationen über einen Parameter werden im selben `ParameterValue`-Objekt gesammelt, so dass beim Befüllen der Felder Widersprüche sofort erkannt werden können. Die bedingten Anforderungen sind dabei innerhalb des zugehörigen `Parameter-Value`-Objekts in einer Hashtabelle (`HashMap`) nach Aktivierungsbedingung getrennt gespeichert. Durch das Zusammenfassen von Informationen mit gleichen Bedingungen können eindeutig fehlerhafte Beschreibungen zurückgewiesen werden, die eine Erfüllbarkeitsprüfung bestehen würden. Denn $\mathtt{A} \Rightarrow \mathit{false}$ ist erfüllbar, solange keine weitere Regel besagt, dass `A` gilt.
Um jeder Klasse eine klare Verantwortlichkeit zu geben, wurden alle für die Überprüfung relevanten Attribute in die entsprechende `Validation`-Unterklasse verlagert, die die Aufgabe der Konfliktüberprüfung und als Teil davon die der Wertprüfung übernimmt. Abbildung 5.6 zeigt die zugehörige Klassenhierarchie[14].
Jede Unterklasse besitzt dabei verschiedene Invarianten bzgl. ihrer Attribute, z. B. fasst `TemplateParameterValidation` alle Parameter mit Schablone aber ohne fes-

[14] Zugunsten der Übersichtlichkeit sind nicht alle Methoden dargestellt.

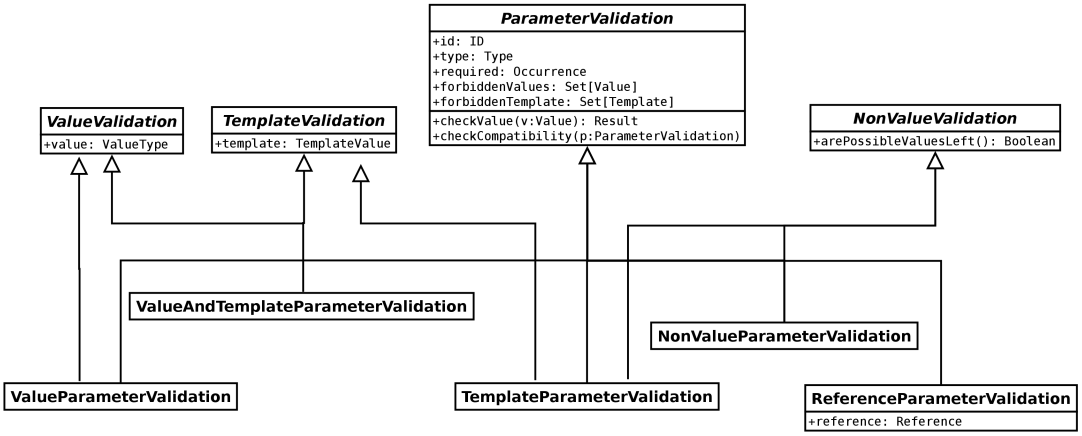

Abbildung 5.6: Klassendiagramm der Validierungshierarchie

ten Wert oder Referenz zusammen. Durch diese Invarianten reduziert sich die Überprüfungslogik in den einzelnen Klassen auf ein Minimum. Die Auswahl und Erzeugung der richtigen Subklasse wird von einer statischen Fabrikmethode übernommen[15]. Da jede Subklasse bei der Kompatibilitätsprüfung die gleichen Schritte durchläuft, definiert die abstrakte Klasse `ParameterValidation` eine Schablonenmethode (Template Method [GHJV95]) und implementiert das Standardverhalten. Weiterhin sind drei Kategorien von Subklassen erkennbar: solche mit einem festen Wert, solche mit einer Schablone und solche ohne Wert oder Schablone. Die Gemeinsamkeiten der einzelnen Kategorien sind in den drei *Traits*[16] `ValueValidation`, `TemplateValidation` und `NonValueValidation` gebündelt und werden von Unterklassen eingebunden. So greift `TemplateAndValueParameterValidation` beispielsweise auf `ValueValidation` und `TemplateValidation` zurück. Durch diese *Mixins* [BC90], die, wo nötig, das Standardverhalten überschreiben, konnte die Redundanz aus den Subklassen eliminiert werden.

`ParameterValue` kümmert sich in erster Linie um die Verwaltung und Aktualisierung der bedingten Aussagen und delegiert die eigentliche Validierung. Diese Klasse ist bewusst veränderlich gehalten, da ein Parameter ein Referenzobjekt (nach [Eva03]) darstellt, d. h. ein Parameter-Objekt ist durch seine ID eindeutig bestimmt und alle Informationen bzgl. dieses Parameters werden in genau einem Objekt zusammengetragen. So wird sichergestellt, dass ein von einem anderen Parameter referenzierter Parameter immer im aktuellen Zustand gesehen wird. Die Speicherung unter der ID als Schlüsselwert in einer `HashMap` garantiert die Existenz genau eines `ParameterValue`-Objekts für jeden Parameter. Für diese `HashMap` wurde eine eigene Klasse geschaffen, um die konkrete Datenstruktur zu verbergen und spezielle Zugriffsmethoden anbieten zu können (vgl. *Encapsulate Collection* in [Fow99]).

[15] Für zusätzliche Informationen sei auf [Fow99] (*Extract Hierarchy* in Kapitel 12) und [Blo08] (Item 1) verwiesen.

[16] Interfaces in Scala, die konkreten Implementierungscode erlauben (siehe Kapitel 12 in [OSV10]).

Man mag sich fragen, wieso dann die Klasse `Parameter` im Domänenmodell (siehe Unterabschnitt 5.2.2) als unveränderlich entworfen wurde. Dort stellt ein `Parameter`-Objekt lediglich ein Abbild aus einer Beschreibungsdatei dar. Das Zusammenfügen mehrerer Beschreibungen und die daraus resultierende Eindeutigkeit der Parameter-ID geschieht erst bei der Überführung in die `ParameterValue`-Klasse. Infolgedessen findet auch die Verlinkung von Referenzparametern erst nach Aufbau der Validierungstabelle statt.

Ablauf einer Validierung

Die Validierung findet in drei Schritten statt.

1. **Aufbau der Parametertabelle:** Die Informationen aus den Domänenklassen werden nach und nach in die Tabelle gefüllt, dabei erzeugen die Klassen `Flavor-Group`, `Flavor` und `OptionalFunctionality` die Implikationslogik. Somit ist das Wissen über ihre Semantik ausschließlich in den betreffenden Klassen vorhanden. Nach der Verknüpfung der Referenzparameter mit der entsprechenden `ParameterValue`-Klasse ist das Modell vollständig erstellt.

2. **Erfüllbarkeitsüberprufung:** Wie in Unterabschnitt 5.3.2 beschrieben, werden nun alle Bedingungen geprüft und eventuell aktualisiert. Falls dabei Konflikte auftreten, werden diese entweder mit einer `InvalidDescriptionException` (vor Prüfung der Bedingungen) oder mit einer `UnsatisfiableDescriptionException` (ab Aktualisierung der Bedingungen) quittiert.

3. **Konfigurationsvalidierung:** Nun, da alle Anforderungen gesammelt und auf Erfüllbarkeit überprüft sind, kann die entstandene Datenstruktur zur Prüfung von konkreten Konfigurationen wiederverwendet werden. Dazu werden die Parameterwerte aus der Konfigurationsdatei ausgelesen und in die Parametertabelle eingetragen. Erneut wird auf erfüllte Bedingungen und Konflikte geprüft, die nun nicht mehr als Ausnahme zurückgeworfen werden, sondern gesammelt und als `Result` (ausgeführt in Abschnitt 5.4) gemeldet werden. Dies hat zur Folge, dass der konfigurierende Nutzer ggf. mehrere Fehler auf einmal beseitigen kann, anstatt dass er nur ein Problem pro Aufruf angezeigt bekommt. Nach einem fehlerfreien Einsetzen weiß man, dass alle gesetzten Werte valide sind. Es bleibt letztlich nur noch zu verifizieren, dass auch alle erforderlichen Parameter gesetzt sind.

Behandlung von Sonderfällen

Aktivierungsbedingung mit endlicher Wertemenge Der in Quelltext 5.5 dargestellte Spezialfall muss bei der Erfüllbarkeitsprüfung berücksichtigt werden; da ein Boolescher Parameter nur genau zwei Werte annehmen kann und er obligatorisch ist, muss eine der beiden Ausprägungen „one“ oder „two“ aktiv sein.
Folglich müssen bei der Erfüllbarkeitsprüfung alle Aktivierungsbedingungen mit obligatorischen Parametern endlicher Wertemengen gesucht werden und falls jeder dieser

möglichen Werte in einer Bedingung verwendet wird, muss durch konsequentes Einsetzen getestet werden, ob eine Belegung zu keinem Widerspruch führt. Da dieser Backtracking-Ansatz ein Kopieren der Parametertabelle erforderlich macht, wurde eine `deepCopy`-Methode gemäß der Ratschläge aus [Blo08] (Item 11) implementiert. Wegen der erwarteten Anzahl von maximal fünf derartigen Stellen in einer kompletten Portalsystembeschreibung, wird die exponentielle Laufzeit in Kauf genommen und sich gegen zusätzlichen Aufwand durch ausgefeiltere Implementierungen entschieden. Durch das Auffinden der möglichen Konfliktstellen kann dem Benutzer im Fehlerfall ziemlich genaue Rückmeldung gegeben werden, wo das Problem liegt.

```
parameters { parameter[Bool] ... required }
flavors {
  one : activatedWhen parameter = 'true'
  ...
  two : activatedWhen parameter = 'false'
  ...
}
```

Quelltext 5.5: Beispiel für Parameter endlicher Wertemenge in Bedingungen

Parameter mit endlicher Wertemenge Bei Parametern endlicher Wertemengen wird bei jedem Hinzufügen eines verbotenen Wertes bzw. einer verbotenen Schablone geprüft, ob noch annehmbare Werte existieren, und bei genau einem verbleibenden Wert dieser als fester Wert eingetragen.

Ausprägungen mit Parametern endlicher Wertemengen Wie in Abschnitt 4.2.2 erklärt, muss ggf. der gegenseitige Ausschluss gelockert werden. Quelltext 5.6 zeigt dies an einem weiteren Spezialfall, einem Parameter mit nur zwei möglichen Werten. Bei aktiver Aktivierungsbedingung „one“ muss die Verneinung der Parameteranforderungen von Ausprägung „two“ und „three“ gelten. Dabei bedeutet die Verneinung eines Parameters ohne konkreten Wert, dass er nicht gesetzt sein darf und die Verneinung eines mit konkretem Wert, dass er diesen Wert nicht annehmen darf. Im Beispiel würde das also bedeuten, dass „parameter“ obligatorisch ist, aber weder „true“ noch „false“ sein darf, was für einen Booleschen Parameter unmöglich ist.
Dies wird verhindert, indem vor dem Erstellen einer jeden negierten Regel geprüft wird, ob Konflikte mit anderen Ausprägungen derselben Gruppe auftreten. Insbesondere wird bei endlichen Wertemengen geprüft, ob noch erlaubte Werte existieren. Dabei werden nicht nur verbotene Werte sondern auch verbotene Schablonen berücksichtigt. Die Lockerungslogik befindet sich in der Ausprägungsklasse, die zum Testen der Kompatibilität auf die `Validation`-Klassen zurückgreift.

```
flavors {
  one : ...
    parameters { parameter[Bool] required }
  two : ...
    parameters { parameter =  'true' }
```

```
  three : ...
    parameters { parameter = 'false' }
}
```

Quelltext 5.6: Beispiel für die Auflockerung des gegenseitigen Ausschlusses

Konfliktprüfung von Schablonenparametern Da die Mächtigkeit der Schablonenparameter in etwa der von regulären Ausdrücken entspricht, kann nicht mit vertretbarem Aufwand entschieden werden, ob zwei Schablonen kompatibel sind oder nicht. Daher werden zwei verschiedene Schablonenparameter als inkompatibel definiert. Dies ist eine akzeptable Einschränkung, weil es keinen Anwendungsfall gibt, bei dem ein Parameter gleichzeitig zwei Schablonen entsprechen muss.
Ebenso kann nicht entschieden werden, ob eine verbotene Schablone noch Werte aus einem unendlichen Wertebereich erlaubt. Bei einem endlichen dagegen können alle möglichen Werte gegen die Schablone getestet werden. Als Konsequenz verbieten betroffene Ausprägungen keine Schablonen, sondern nur feste Werte. Bei Ausprägungen wird sowieso die Semantik soweit gelockert, dass keine Widersprüche zwischen den einzelnen Varianten entstehen (vgl. Abschnitt 4.2.2). So kann sich der Anwender sicher sein, dass der gegenseitige Ausschluss keine ungültigen Anforderungen erzeugt. Insgesamt bleibt die Beschreibung damit so streng wie möglich, ohne dass die Aussagemöglichkeiten eingeschränkt werden.
An dieser Stelle sei darauf hingewiesen, dass bei Schablonen festgestellt werden kann, ob sie einen endlichen Wertebereich haben: Eine Schablone hat genau dann einen endlichen Wertebereich, wenn alle variablen Teile einen endlichen Wertebereich haben.

Zirkuläre Referenzen Um zu verhindern, dass Referenzstrukturen unendlicher Tiefe erzeugt werden, wird beim Verlinken überprüft, ob der Zielparameter eine zirkuläre Referenz enthält. Dazu muss zuerst rekursiv der Zielparameter mit vorausgehender Prüfung verlinkt werden.[17]

5.4 Fehlerbehandlung

Für jede Art von Fehler in der Beschreibung oder Konfiguration gibt es eine von `UnsuccessfulChecked` abgeleitete Benachrichtigungsklasse, in der alle weiteren Informationen zu dem Fehler enthalten sind. Abbildung 5.7 zeigt die zugehörigen Klassen und ihre Beziehungen untereinander.
Fehler in Konfigurationsbeschreibungen werden als `Exception` geworfen, die eine Benachrichtigungsklasse mit dem Grund beinhaltet, während Fehler in Konfigurationen den Programmablauf nicht unterbrechen, sondern gesammelt und am Schluss an den Nutzer übermittelt werden. Der Unterschied ist damit begründet, dass fehlerhafte

[17] Die Umsetzung ist im Prototyp noch nicht erfolgt, da zirkuläre Referenzen praktisch nur mutwillig hervorgerufen werden können.

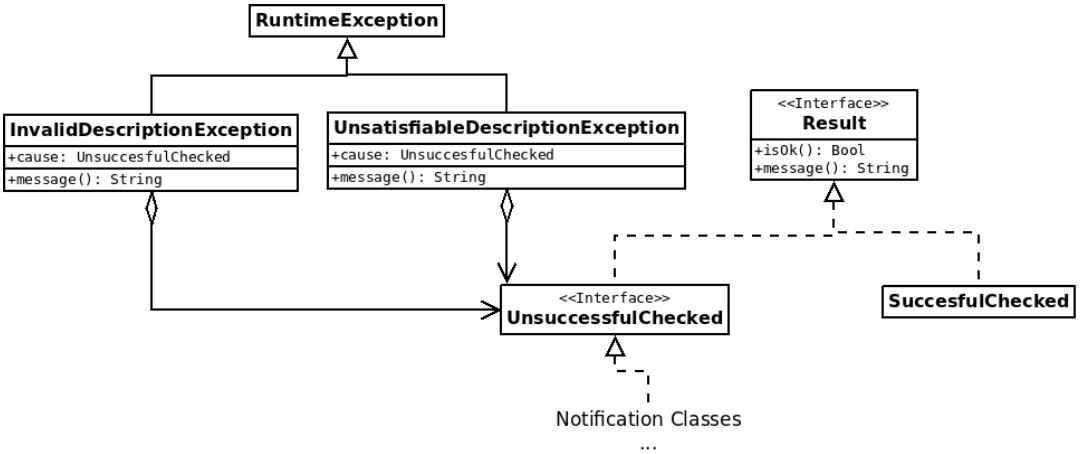

Abbildung 5.7: Klassendiagramm der Fehlerbenachrichtungen

Beschreibungen schwerwiegende Fehler darstellen, die nicht auftreten sollen und die wahrscheinlich zusätzliche Personen zur Klärung erfordern.

5.5 Bemerkungen zu Funktionstests und Releasebau

Für jede spezifizierte Funktionalität und insbesondere für die Vertreter der häufigsten Fehlerklassen (vgl. Unterabschnitt 4.2.1) wurden automatische Tests implementiert und in den Buildprozess integriert. Die einzelnen Tests umfassen größtenteils alle Schichten von der externen DSL bis hin zur Validierung der gemeldeten Resultate, manche testen aber auch gezielt einzelne Schichten, so dass hier die Notation in interner DSL-Syntax Verwendung findet. Als Technologie werden *ScalaTest*[18] in Verbindung mit einem *JUnit*[19]-Runner verwendet, der mittels des *SureFire*-Plug-ins[20] in *maven* integriert ist. Da alle 50 Tests insgesamt weniger als eine Sekunde Laufzeit haben, können sie nicht nur nach jeder Kompilierung gestartet werden, sondern lassen auch keine Anzeichen auf schlechte Performance erkennen.
Wie in der tarent üblich, wurde für den *Konfigurator* ein *Jenkins*-Auftrag[21] (ehemals *Hudson*) angelegt, der ausgelöst durch *svn*-Commits die Artefakte baut und nach Bestehen der Tests in ein tarent-internes Snapshot-Repository überträgt.

[18] http://www.scalatest.org/
[19] http://www.junit.org/
[20] http://maven.apache.org/plugins/maven-surefire-plugin/
[21] http://jenkins-ci.org/

6 Bewertung durch Benutzungstests

Um die Handhabbarkeit der Beschreibungssprache und des *Konfigurators* zu evaluieren, wurde eine kleiner Feldtest mit fünf Software-Entwicklern und Projektleitern durchgeführt, den dieser Abschnitt zusammenfasst.

6.1 Aufgabe und Methode

Nach einer 15-minütigen Präsentation meinerseits sollten die Teilnehmer Beschreibungsdateien für einen vorgegebenen Ausschnitt der realen Portalkonfiguration erstellen und anhand von verschiedenen bewusst fehlerhaften Konfigurationen die Validierungsfähigkeit und die Fehlermeldungen des *Konfigurators* testen. Zudem galt es, den Umgang mit widersprüchlichen Beschreibungen zu erproben.
Der Konfigurationsausschnitt bestand aus 15 Parametern, erstreckte sich über drei Komponenten und deckte mit Ausprägungen, Schablonenparametern, Referenzen und optionalen Funktionalitäten alle häufigen Fehlerquellen (vgl. Unterabschnitt 4.2.1) sowie den vollen Funktionsumfang der Beschreibungssprache ab. Alternativ konnte auch eine selbst gewählte Konfiguration beschrieben werden, was allerdings keiner der fünf Teilnehmer tat. Als Unterstützung diente eine im internen Wiki abgelegte Dokumentation, u. a. weil zwischen meinem Vortrag und dem eigentlichen Ausprobieren aus terminlichen Gründen mitunter mehrere Tage vergingen. Das Feedback wurde im persönlichen Gespräch bzw. zusammengefasst als E-Mail aufgenommen. Bei den gestellten Fragen wurde besonders auf die Erfahrungen mit der Sprache und den Fehlerinformationen bei falschen Konfigurationen eingegangen. Zudem habe ich mir die erstellten Dateien angesehen.

6.2 Ergebnis

Das Schreiben der Beschreibungsdateien dauerte zwischen 30 Minuten und einer Stunde. Insgesamt kamen die Beteiligten gut mit der Struktur zurecht, hätten aber eine längere persönliche Erklärung für sehr hilfreich befunden. Die Fehlermeldungen des Parsers bei Syntaxfehlern waren zwar „meist hilfreich“, aber es dauerte im Einzelfall ein bisschen, bis man sich an die Funktionsweise gewöhnt hatte. Konkret fiel der harte Abbruch bei vergessenen Kommata und dergleichen negativ auf, weil dies bei modernen IDEs so gut wie nicht mehr vorkommt. Weiterhin gab es bei jedem meist unterschiedliche Wünsche auf Anpassung der Syntax. Es kristallisierte sich aber beispielsweise heraus, dass die Verwendung von „=“ als Schlüsselwort und „#“ für Nicht-Zeilenkommentare unintuitiv sei. Semantisch war einzig der Sinn von Ausprägungen

teilweise unklar und hätte einer ausführlicheren Erklärung oder weiteren Beispielen im Wiki bedurft. Als Anmerkung: Die Fülle der Dokumentation litt leider dran, dass ich mit Hinblick auf diese Aufgabe ähnliche Beispiele weggelassen habe.
Das Zufriedenstellen des Parsers wurde von allen als größte Hürde gesehen. Sobald dies erreicht wurde, wurden die Fehlermeldungen, die sowohl falsche Beschreibungen als auch falsche Konfigurationen betrafen, durchweg als hilfreich wahrgenommen. Einer Person bereitete die Validierung „[...] sogar ein bisschen Spaß, nachdem die Syntaxfehler beseitigt waren. Denn man konnte sehen, wie die Anforderungen umgesetzt wurden und es funktionierte.“. Arbeitsbedarf zeigte sich nur bei der Typvalidierung, da in der jetzigen Implementierung erst für wenige Typen reguläre Ausdrücke implementiert sind[1]. In diesem Zusammenhang gilt es auch mit den anderen Projektbeteiligten zu klären, ob welche der ca. 30 von mir identifizierten Typen zusammengefasst werden können.
Insgesamt kann man den Feldversuch auf jeden Fall als Erfolg bewerten, gerade weil die Entwicklung bislang ohne externes Feedback ablief. Daher waren Anpassungswünsche für die Syntax erwartet worden und diese sind auch dank der selbst entwickelten Sprache möglich. Dass die Erstellung der Beschreibungsdateien so lange dauerte, war dann doch ein wenig überraschend, wird aber zu großen Teilen der kurzen Präsentation zugeschrieben. Im Nachhinein hätte die Einführung trotz terminlicher Zwänge von immerhin fünf Personen länger im Voraus angesetzt und ausführlicher gestaltet werden sollen.
Der *Konfigurator* stellt im momentanen Entwicklungsstand eine solide Grundlage dar, die entsprechend dieses Feedbacks angepasst und in einem nächsten Schritt testweise in den ersten Projekten eingeführt werden kann.

[1] Die vollständige Implementierung aller formalen Typkriterien hätte den Rahmen dieser Arbeit gesprengt.

7 Ausblick

In dieser Arbeit wurde nach gründlicher Evaluation verschiedenster Technologien eine Beschreibungssprache und ein Konfigurationsvalidierer erstellt, mit dem Beschreibungen auf Erfüllbarkeit und Konfigurationen auf Validität überprüft werden können. Die Tauglichkeit dieser Lösung wurde sowohl durch automatische Tests und als auch durch Benutzertests bestätigt.
Für die Zukunft steht nun eine Einführung in allen Portalprojekten bevor und eine Eingliederung in den Entwicklungs- und Installationsprozess, die in dieser Arbeit nur skizziert wurde. Hier lautet das mittelfristige Ziel, alle Änderungen über den *Konfigurator* durchzuführen, um sie nachvollziehbar zu protokollieren sowie auf beliebige Konfigurationsstände zurückspringen zu können.
Es ist auch denkbar, dass die Beschreibungssprache später anhand von Annotationen im Quellcode automatisch generiert wird, so dass die Entwickler mit gewohnten Werkzeugen für eine fortwährende Aktualität sorgen können.
Auf der anderen Seite stellt auch eine prozessintegrierte Überwachung der Güte und eine fortlaufende Verbesserung der Konfigurationsbeschreibungen ein Feld für weitere Arbeiten dar. So ist insbesondere die Kommunikation zwischen konfigurierenden Rollen und den Erstellern von Beschreibungen technisch zu unterstützen. Konkret sollten im Einzelfall notwendige, manuelle Konfigurationsschritte mitgeschnitten werden und anschließend in die Konfigurationsbeschreibungen einfließen.

Literaturverzeichnis

[BC90] Gilad Bracha and William Cook. Mixin-based inheritance. In *Proceedings of the European conference on object-oriented programming on object-oriented programming systems, languages, and applications*, OOPSLA/E-COOP '90, pages 303–311, New York, NY, USA, 1990. ACM.

[BHS78] Edward H. Bersoff, Vilas D. Henderson, and Stan G. Siegel. Software configuration management. In *Proceedings of the software quality assurance workshop on Functional and performance issues*, pages 9–17, New York, NY, USA, 1978. ACM.

[Blo08] Joshua Bloch. *Effective Java Second Edition.* Addison Wesley, Boston, MA, USA, 2008.

[Buc96] Fletcher J. Buckley. *Implementing configuration management - hardware, software, and firmware (2. ed.).* IEEE, 1996.

[Eva03] Eric Evans. *Domain-Driven Design: Tacking Complexity in the Heart of Software.* Addison-Wesley, Boston, MA, USA, 2003.

[FFJ00] Alexander Felfernig, Gerhard E. Friedrich, and Dietmar Jannach. UML as domain specific language for the construction of knowledge-based configuration systems. *International Journal of Software Engineering and Knowledge Engineering (IJSEKE)*, 10:449–469, 2000.

[Fow99] Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Addison Wesley, Boston, MA, USA, 1999.

[FP11] Martin Fowler and Rebecca Parsons. *Domain-Specific Languages.* Addison-Wesley, Boston, MA, USA, 2011.

[GGL+03] Patrick Goldsack, Julio Guijarro, Antonio Lain, Guillaume Mecheneau, Paul Murray, and Peter Toft. Smartfrog: Configuration and automatic ignition of distributed applications. In *HP Openview University Association Conference (HP OVUA)*, pages 1–9, 2003.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns.* Addison-Wesley, Boston, MA, 1995.

[Gho11] Debasish Ghosh. *DSLs in action.* Manning Publications Co., Greenwich, CT, USA, 2011.

[LY99] Tim Lindholm and Frank Yellin. *The JavaTM Virtual Machine Specification, Second Edition.* Prentice Hall, Upper Saddle River, NJ, USA, 1999.

[MF89] Sanjay Mittal and Felix Frayman. Towards a generic model of configuraton tasks. In *Proceedings of the 11th international joint conference on Artificial intelligence - Volume 2*, pages 1395–1401, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[OSV10] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala Second Edition.* Artima, Inc., 2010.

[Par07] Terence Parr. *The Definitive ANTLR Reference - Building Domain-Specific Languages.* The Pragmatic Bookshelf, 2007.

[Pro08] Java Community Process. JSR 286: Portlet specification 2.0. `http://www.jcp.org/en/jsr/detail?id=286`, 2008.

[SF96] Daniel Sabin and Eugene C. Freuder. Configuration as composite constraint satisfaction. In *Proceedings Artificial Intelligence and Manufacturing. Research Planning Workshop*, pages 153–161. AAAI Press, 1996.

[Stu97] Markus Stumptner. An overview of knowledge based configuration. *AI Commun.*, 10:111–125, April 1997.

[TB02] Steve Traugott and Lance Brown. Why order matters: Turing equivalence in automated systems administration. In *Proceedings of the 16th USENIX conference on System administration*, pages 99–120, Berkeley, CA, USA, 2002. USENIX Association.

[vDKV00] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: an annotated bibliography. *SIGPLAN Not.*, 35:26–36, June 2000.

[WSC+07] Jules White, Douglas C. Schmidt, Krzysztof Czarnecki, Christoph Wienands, Gunther Lenz, Egon Wuchner, and Ludger Fiege. Automated model-based configuration of enterprise Java applications. In *Proceedings of the 11th IEEE International Enterprise Distributed Object Computing Conference*, Washington, DC, USA, 2007. IEEE Computer Society.