Diplomarbeit

Erweiterung einer Simulationsumgebung um automatische Stub-Generierung

Michael Meinel
DLR Simulations- und Softwaretechnik, BA Mannheim

DLR

BERUFSAKADEMIE | MANNHEIM
university of cooperative education | staatliche studienakademie

Erweiterung einer Simulationsumgebung um automatische Stub-Generierung

Diplomarbeit

für die Prüfung zum
Diplom-Ingenieur (Berufsakademie)

des Studienganges Informationstechnik
an der Berufsakademie Mannheim

von

Ing.-Ass. (BA)
Michael Meinel

19. September 2007

Bearbeitungszeitraum:	3 Monate
Kurs:	TIT04AIN
Ausbildungsfirma:	DLR Deutsches Zentrum für Luft- und Raumfahrt e.V., Köln-Porz
Einrichtung:	Simulations- und Softwaretechnik
Gutachter der Ausbildungsfirma:	Dipl.-Math. Andreas Schreiber
Gutachter der Studienakademie:	Dipl.-Inf. Giovanni Falcone

Erklärung

Hiermit versichere ich, dass die vorliegende Arbeit von mir selbstständig angefertigt wurde und ich keine weiteren als die angegebenen Hilfsmittel benutzt habe.

Köln-Porz, den 19. September 2007

Vorwort

Diese Diplomarbeit schließt mein dreijähriges Studium der Informationstechnik an der Berufsakademie Mannheim ab. Sie entstand in der Einrichtung „Simulations- und Softwaretechnik“ des Deutschen Zentrum für Luft- und Raumfahrt e.V.

Ich hoffe, dass das Ergebnis dieser Arbeit nicht nur mich in Hinblick auf Softwareengineering und verteilte Architekturen weiter gebracht hat, sondern auch denen, die die Software in ihrem täglichen Geschäft nutzen, eine Hilfe ist.

Der Inhalt – die Erweiterung einer Simulationsumbegung um automatische Stub-Generierung – stellt aus meiner Sicht die notwendige Kombination aus bekannten Techniken mit modernen Konzepten dar. Die Arbeit mit innovativen, noch in der Entwicklung steckenden Technologien war zwar zeitweise recht nervenaufreibend, um so mehr hat der anschließende Druchbruch jedoch zum Weitermachen motiviert.

An dieser Stelle möcht ich noch meinen Dank an alle, die mich während meines Studiums und beim Erstellen dieser Arbeit unterstütz haben, hervorbringen. Besonderer Dank geht dabei an Jens, Roland, Thijs und Anastasia, die mir viele hilfreiche Tipps gegeben und mich auf Schwachstellen meiner Arbeit aufmerksam gemacht haben. Aber auch dem Rest meiner Arbteilung, die die Arbeitzeit sehr angenehm gestaltet haben, bin ich dankbar.

Außerdem danke ich meiner Ausbildungsleitung, die sich das ganze Studium hindurch hingabevoll um ihre Schützlinge gekümemrt hat, sowie dem GeZet und den vielen netten Menschen, die ich dort kennengelert habe, die einen wichtigen Teil meiner Freizeit in Mannheim vereinnahmt haban, ohne dadurch mein Studium negativ zu beeinflussen.

Zusammenfassung

Das Reconfigurable Computing Environment (RCE) ist eine Integrationsumgebung für verteilt laufende Anwendungen. Es entstand im Rahmen eines Projektes für die Entwicklung eines schiffbauliches Entwurfs- und Simulationssystems (SESIS), dessen Ziel es ist, verteilte Resourcen transparent zugreifbar zu machen. Das System ist modular aufgebaut und lässt sich durch Extensions (Plugins) erweitern.

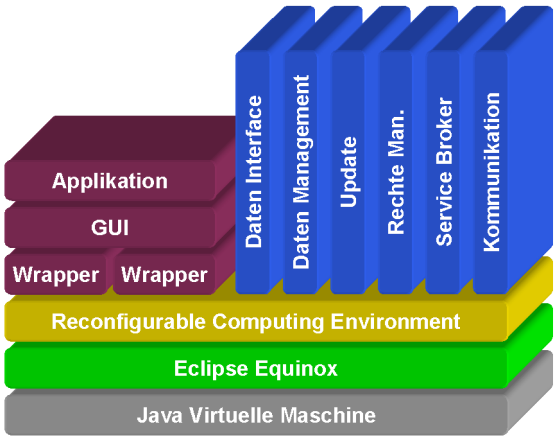

Abbildung: Modulare Architektur von RCE [17]

Um Extensions auch von entfernten RCE-Installationen zugänglich zu machen, kann eine Kommunikations-Komponente genutzt werden. Damit dies möglich ist, wird lokal eine Stub-Implementierung der gewünschten Funktionalität benötigt, die Methoden-Aufrufe an den entsprechenden entfernten Rechner weiterleitet. Diese Stub-Implementierung musste bisher vom Entwickler immer parallel zu der eigentlichen Extension entwickelt werden.

Da der Code, der hinter einem Stub steht, eine feste Struktur hat und sich einfach an die konkreten Anforderungen anpassen lässt, liegt es nahe, diesen Stub generieren zu lassen und somit den Entwickler zu entlasten. Gleichzeitig hat dieses Vorgehen auch noch weitere Vorteile. So können neu entwicklete Extensions unmittelbar nachdem sie veröffentlicht wurden, sofort auch von entfernten Instanzen genutzt werden.

Um die Möglichkeiten von Code-Generierung im Rahmen von RCE auszutesten, wurden zunächst Untersuchungen bezüglich der prinzipiellen Machbarkeit

vorgenommen. Anschließend wurde das Konzept anhand einer prototypischen Implementierung eines solchen Stub-Generators überprüft und auf seine Leistungsfähigkeit getestet.

Abstract

The simulation environment Reconfigurable Computing Environment (RCE) aims at integrating distributed computing applications. It was first developed for the SESIS project, which is situated in the domain of naval architecture. The main functionality is to make distributed resources accessible in a transparent manner. Therefore the system has a modular architecture that is extendable using extensions (plugins).

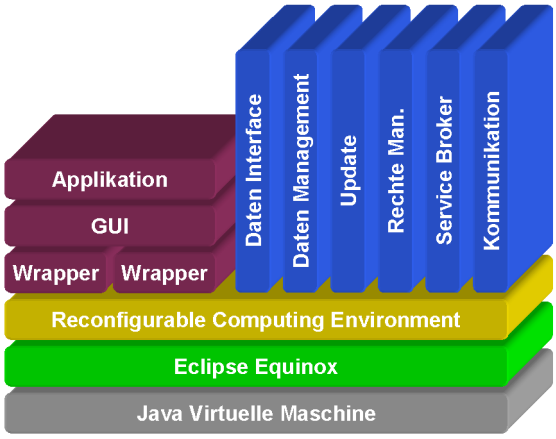

Figure: Modular architecture of RCE [17]

In order to access extensions that run in a remote domain, locally installed stubs are needed. These redirct any method call to a communication bundle that is part of RCE. As of now, those stubs have to be implemented and maintained by the developer together with the actual extensions.

Since the code of the local stubs follows some rather strict patterns it possible to have them generated automatically. Not only the developers would benefit from enforcing automatic stub generation but this would also add some extra value to the RCE framework itself. If the stubs were generated it would be possible to remotely access new extension from the moment on they were published.

To evaluate the possiblities of such a code generation mechanism first some general analyses in respect of possible approaches were made. Afterwards a prototype was implemented and tested for performance and sanity in the context of the RCE environment.

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungen und Begriffe

API Application Programming Interface.
Unter einem API werden alle Schnittstellen zusammengefasst, die ein einheitliches Aufgabengebiet zuzuordnen sind. Dies kann zum Beispiel Ein- und Ausgabe (IO-API) sein. Aber es kann sich dabei auch um einen Satz von Schnittstellen handeln, die ein Programm bereitstellt, um Erweiterungen zuzulassen.

DLR Deutsches Zentrum für Luft- und Raumfahrt e.V.

DOM Document Object Model.
Unter DOM werden Techniken zusammengefasst, die eine objektorientierte Darstellung von Dokumenten erlauben.

GPL General Public License.
Die GPL ist eine weit verbreitete Open-Source-Lizenz, die umfangreiche Anforderungen bezüglich der Veröffentlichung von „abgeleiteten Werken“ macht.

GUI Graphical User Interface.
Eine GUI ist eine grafische Benutzerschnittstelle für Computer-Programme.

IDE Integrated Development Environment.
Ein IDE ist ein Programmpaket, das verschiedene Programmier-Werkzeuge zusammenfast und zumeist mittels einer GUI zugänglich macht.

IDL Interface Description Language.
IDL ist eine Sprache, mit der man Programmschnittstellen insbesondere für verteilte Anwendugnen programmiersprachenunabhängig definieren kann. Sie wurde von Object Management Group definiert.

ITU Internatinal Telecommuniction Union.
Bei der ITU handelt es sich um eine Unterorganisation der Vereinten Nationen, die unter anderem internationale Standards festlegt und sich um die Verteilung von (Funk-)Frequenzbändern kümmert.

ITU-T ITU Telecommunication Standadization Sector.
Die ITU-T ist eine Abteilung der ITU, die sich mit der Spezifikation und Herausgabe von technischen Normen, Standards und Empfehlungen beschäftigt.

JDK Java Developers Kit.
Das JDK ist eine Sammlung an Werkzeugen, die zur Entwicklung von Java-Programmen benötigt werden. Es wird von Sun kostenlose zur Verfügung gestellt.

JFace .
Ein Java-Bibliothek die auf SWT aufsetzt und spezialisierte Widgets bereitstellt.

JVM Java Virutal Machine.

OSGi Open Service Gateway Initiative.

RCE Reconfigurable Computing Environment.

RCP Rich Client Platform.
RCP ist eine von der Eclipse Foundation bereitgestellte Sammlung von OSGi-Bundles.

RPC Remote Procedure Call.
RPC bezeichnet ein Verfahren, bei dem Methoden oder Prozeduren auf einem entfernt laufenden Rechner aufgerufen werden

SC Simulations- und Softwaretechnik.

SESIS schiffbauliches Entwurfs- und Simulationssystem.

SWT Standart Widget Toolkit.
Das SWT ist eine Java-Bibliothek, die Klassen zur Erstellung von GUIs bereitstellt.

UML Unified Modeling Language.
UML ist ein Standard, mit hilfe dessen die Struktur von Programmen grafisch dargestellt werden kann. Der Standart wurde von der Object Management Group herausgegeben.

WSDL Webservice Definition Language.
WSDL ist eine auf XML basierende Beschreibungssprache für Dienste, die über über RPC zugänglich sind.

XML Extensible Markup Language.
XML ist ein erweiterbares Format für die sturkturierte Speicherung computerlesbarer Daten.

1. Einleitung

Die vorliegende Diplomarbeit entstand am Deutschen Zentrum für Luft- und Raumfahrt e.V. (DLR) in der Abteilung Verteilte Systeme und Komponentensoftware der Einrichtung Simulations- und Softwaretechnik. Inhalt ist die Erweiterung der Integrationsumgebung Reconfigurable Computing Environment (RCE) um automatische Code-Generierung.

1.1. Die Einrichtung Simulations- und Softwaretechnik

Das DLR ist das Forschungszentrum der Bundesrepublik Deutschland für Luft- und Raumfahrt. Seine umfangreichen Forschungs- und Entwicklungsarbeiten in Luftfahrt, Raumfahrt, Energie und Verkehr sind in nationale und internationale Kooperationen eingebunden. Über die eigene Forschung hinaus ist das DLR als Raumfahrtagentur im Auftrag der Bundesregierung für die Planung und Umsetzung der deutschen Raumfahrtaktivitäten zuständig. Das DLR beschäftigt ca. 5.300 Mitarbeiterinnen und Mitarbeiter, es unterhält 28 Institute bzw. Test- und Betriebseinrichtungen und ist an neun Standorten vertreten. [1]

Die Querschnittseinrichtung Simulations- und Softwaretechnik hat ihre Kompetenz im Bereich Software Engineering. Die derzeitigen Themenschwerpunkte sind die komponentenbasierte Softwareentwicklung für verteilte Systeme, Software für eingebettete Systeme und die Software-Qualitätssicherung. [3]

Zentrales Aufgabengebiet der Abteilung Verteilte Systeme und Komponentensoftware ist die komponentenbasierte Softwareentwicklung für verteilte Systeme. Die derzeitigen Schwerpunkte der Abteilung sind [5]:

- Komponentenbasierte Softwareentwicklung
- Management wissenschaftlicher Daten
- GUI-Entwicklung
- Verteiltes Rechnen
- Grid Computing

1.2. Die Integrationsumgebung RCE

Im Verbundprojekt SESIS entwickelt Simulations- und Softwaretechnik zusammen mit Partnern aus Schiffbauindustrie und Informationstechnik ein integriertes schiffbauliches Entwurfs- und Simulationssystem. Der Fokus liegt dabei auf der frühen Entwurfsphase, in der die wesentlichen Parameter eines Schiffneubaus festgelegt werden. SESIS unterstützt insbesondere die Zusammenarbeit von Werft und Zulieferunternehmen. [4]

Als Basis für die im SESIS-Projekt entwickelte Software dient das RCE, welches als Kernkomponente für das Projekt ebenfalls neu erstellt wird. RCE ist ein Komponenten-basiertes System, dessen Stärken die hohe Flexibilität sowie die Integration von vernetzten Teilsystemen ist. [17]

Besonders wichtig ist dabei, dass die Programme, die von verschiedenen Anbietern mit verschiedenen Spezialkenntnissen kommen (Steuerelektronik, Schiffbauingenieure, Strömungstechniker etc.), verteilt laufen können und trotzdem für die Projektpartner transparent ansprechbar sind. Dafür stellt RCE einerseits eine durch Plugin-Techniken erweiterbare Basis als auch eine einheitliche Benutzeroberfläche bereit.

2. Aufgabenstellung

Um eine Extension entfernt zugreifbar zu machen, muss es für diese eine Stub-Implementierung geben. Dabei handelt es sich um eine Klasse, die die gleiche Schnittstelle implementiert, wie die eigentliche Klasse, jedoch sämtliche Aufrufe an eine entfernt laufende RCE-Instanz weiterleitet. Abbildung 1 stellt diesen Ablauf schematisch dar. Dargestellt ist die Klasse `SomeExt`, die die Schnittstelle `ISomeExt` implementiert. Zusätzlich ist eine lokale Stub-Implementierung dieser Schnittstelle (`SomeExtStub`) zu sehen, die zwar die gleichen Methoden zur Verfügung stellt, Aufrufe jedoch über die Kommunikations-Komponente und ein Netzwerk weiterleitet.

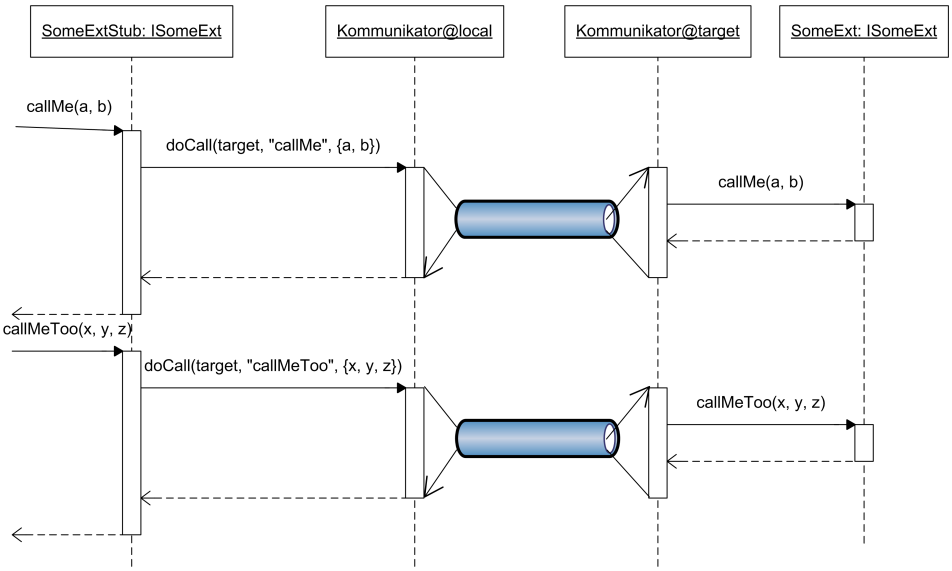

Abbildung 1: Übertragen eines Aufrufes durch einen Kommunikator [eigen]

Die Stub-Implementierungen der verschiedenen RCE-Extensions werden bisher von den Extension-Entwicklern von Hand programmiert und bereitgestellt. Außerdem muss sie vom Entwickler parallel zum eigentlichen Programm gepflegt und konsistent gehalten werden. Neue Versionen des Stubs müssen mit jeder neuen Extension-Version verteilt werden, vor allem, wenn die entwickelte Extension nur als entfernt zugreifbarer Dienst angeboten werden soll.

Da der Stub-Code immer eine sehr ähnliche Struktur aufweist, ist er gut für automatische Generierung geeignet (siehe Abschnitt 5.2). Dadurch wird nicht nur dem Entwickler Arbeit abgenommen sonder auch die anderen Probleme werden dadurch gelöst.

In den folgenden Kapitel wird zunächst ein Grundkonzept zur Generierung von Stubs im Rahmen des RCE entwicklet. Dabei ist vor allem der Kontext so festzulegen, dass der Stub-Generator – neben dem Problem der häufigen Code-Dopplung – auch möglichst viele von den anderen Schwachstellen der manuellen Stub-Entwicklung schließt, ohne dabei neue Probleme zu erzeugen. Weiterhin sind die Rahmenbedingungen für den erfolgreichen Einsatz des Generators zu spezifizieren.

Sind der Kontext und die Bedingungen für den Ablauf des Generierungsprozesses festgelegt, so wird aufbauend auf bisherigen Code-Generierungs-Konzepten eine an die Anforderungen im RCE angepasste Variante entworfen. Dabei werden auch die zur Verfügung stehenden Hilfsmittel – insbesondere Java-Bibliotheken – herausgesucht und bewertet.

Die Bewertung der einzusetzenden Hilfsmittel geschieht anhand von zwei Gesichtspunkten. Zunächst ist von Interesse, wie gut sich die Bibliothek in den vom RCE vorgegebenen Rahmen integriert (vergleiche Abschnitt 7.1.1). Anschließend müssen die für tauglich befundenen Softwarelösungen auf Ihre Leistungsfähigkeit überprüft werden. Dazu kommen Benchmark-artige Performancetests zum Einsatz (siehe Abschnitte 7.2 und 7.3).

In einer abschließenden Prototyp-Implementierung des Stub-Generators, welche in Abschnitt 8 beschrieben ist, wird dann die Tauglichkeit des erstellten Konzeptes und die Leistungfähigkeit der Implementierung in der Praxis getestet.

3. Architektur von RCE

RCE wurde auf Grundlage der OSGi-Framework-Implementation des Eclipse-Projektes (Equinox) erstellt. Diese auf der Abbildung 2 dargestellte Architektur ermöglicht die Erweiterung mittels Plugins. Weiterhin bietet Eclipse Unterstützung für grafische Benutzeroberflächen mittels SWT und JFace sowie Möglichkeiten zur Projektverwaltung aus der IDE-Komponente.

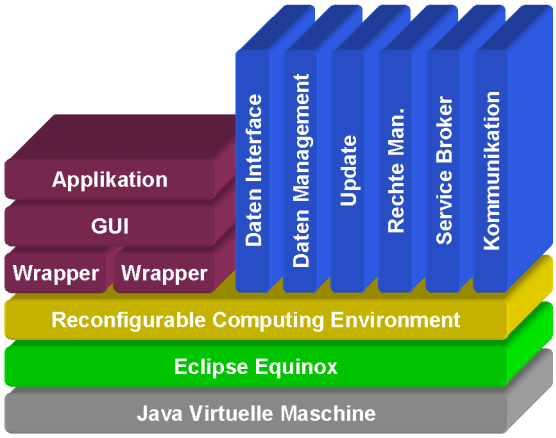

Abbildung 2: Architektur der Integrationsumgebung RCE [17]

3.1. Das OSGi-Framework

Das OSGi-Framework ist ein von der Open Service Gateway Initiative (OSGi) herausgegebener Standard, der derzeit in der Version R4 vorliegt. Die hohe Modularität, welche die Mikrokernel-Archtektur ausmacht, wird durch *Bundles* und *Services* erreicht.

Ein *Bundle* ist dabei eine Software-Komponente, die in Form eines Jar-Archivs oder eines Ordners im Dateisystem vorliegt. In diesem finden sich neben Java-Klassen auch Dateien mit Metadaten sowie weitere Resourcen wie Icons, Hilfetexte und ähnliches.

Durch die Metadaten können Abhängigkeiten zwischen verschiedenen Bundles definiert werden. Dabei wird das Prinzip der Imports und Exports genutzt: Ein Bundle kann mehrere (Java-)Packages exportieren. Diese können dann wiederum

durch ein anderes Bundle importiert werden, was zu einer Abhängigkeit zwischen diesen Bundles führt. Beim Laden des OSGi-Kerns werden alle Bundles zunächst eingelesen und anschließend wird versucht, die Abhängigkeiten der Bundles aufzulösen.

Ein Bundle kann die in Abbildung 3 dargestellten Zustände haben. Nach der Installation ist es zunächst `INSTALLED`. Sobald alle Abhängigkeiten aufgelöst werden konnten, wechselt der Zustand nach `RESOLVED`. Nun kann das Bundle gestartet werden, um über den Zustand `STARTING` in den Zustand `ACTIVE` zu gelangen. Ist ein Bundle `ACTIVE`, so kann es uneingeschränkt genutzt werden oder über den `STOPPING`-Zustand wieder zurück nach `RESOLVED` fallen. Wird ein Bundle wieder deinstalliert, so gelangt es in den Zustand `UNINSTALLED`. Aus diesem Zustand heraus kann das Bundle nicht mehr verwendet werden und darf sogar physikalisch von der Festplatte entfernt werden.

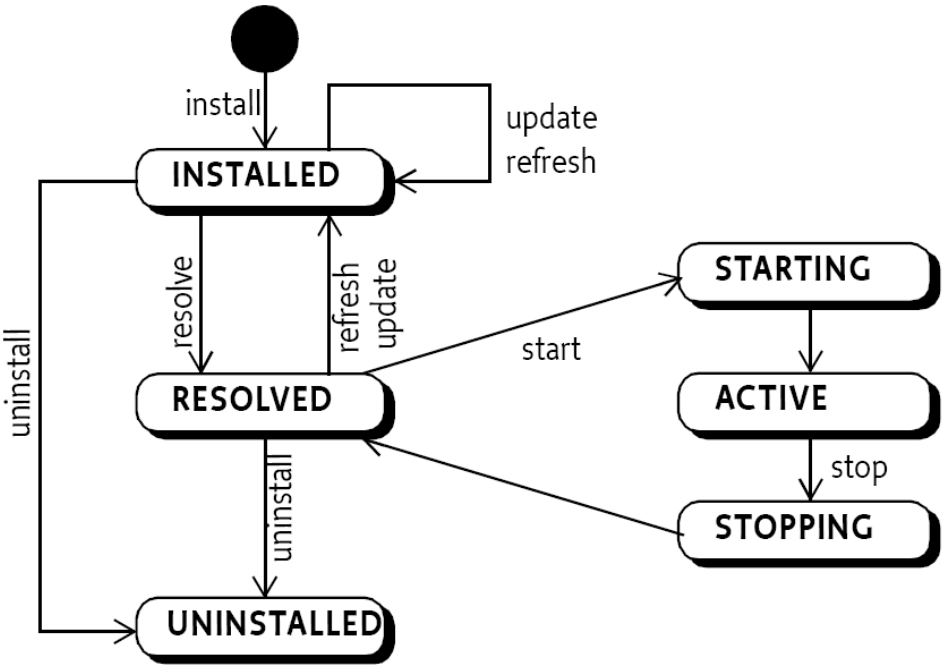

Abbildung 3: Mögliche Zustände von OSGi-Bundles [8]

Neben den Bundles nehmen die *Services* eine zentrale Rolle im OSGi-Standard ein. Jedes Bundle kann mehrere Services definieren, die anschließend beim OSGi-Kern (Registry) registriert und abgefragt werden können. Somit werden die einzelnen Bundles weitgehend entkoppelt. Die Registrierung von Services muss von den Bundles allerdings aktiv vorgenommen werden, das heißt ein Bundle muss beim Starten eine Methode der OSGi-Registry aufrufen. [8]

3.2. Die Eclipse Rich Client Platform

Die Integrated Development Environment (IDE) Eclipse basiert auf der OSGi-Spezifikation und bringt eine eigene Implementierung dieses Standards mit.

Darauf aufbauend bringt Eclipse eine Reihe von Bundles mit, die die Rich Client Platform (RCP) bilden. Ein Merkmal der RCP sind Extension-Points und Extensions. Diese stellen eine alternative zu den Services des OSGi-Frameworks dar. Den Extensions liegt immer eine XML-Datei zu Grunde. In dieser Datei können für jedes Bundle *Extension-Points* und *Extensions* definiert sein. Ein Extension-Point ist dabei mit einer Steckdose vergleichbar. In diese Steckdose kann dann eine passende Extension eingesteckt werden. Über dieses System sind flexible Architekturen, wie die auf Abbildung 4 dargestellte, aufbaubar.

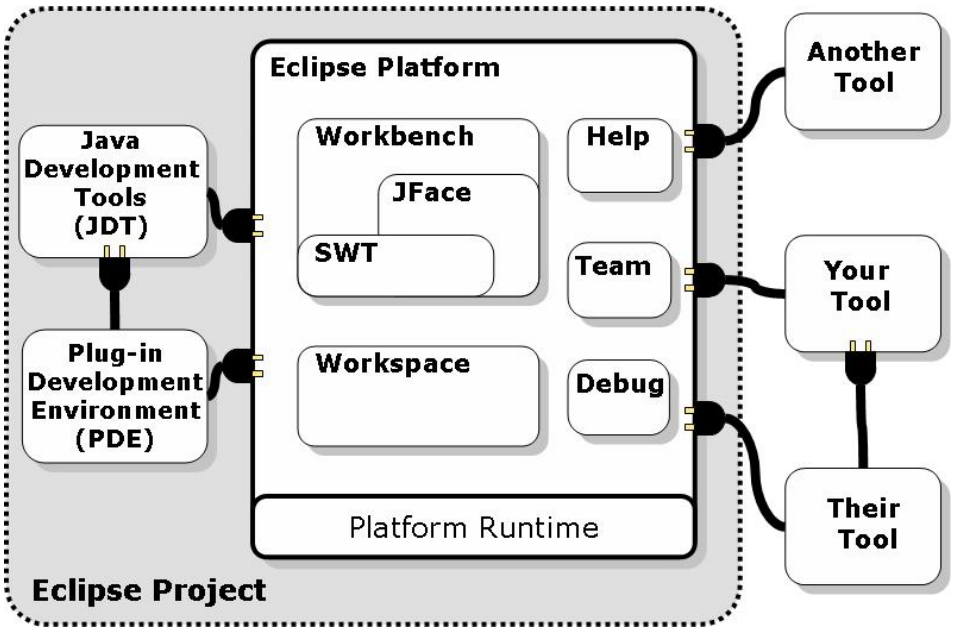

Abbildung 4: Aufbau einer Eclipse-Anwendung mit Extensions [14]

Durch den Einsatz von Extension-Points und Extensions ist es möglich, Bundles über einen Lazy-Loading-Mechanismus in das Gesamtsystem zu laden. Da in der XML-Datei alle notwendigen Informationen stehen, ist es möglich, die Bundles erst zu dem Zeitpunkt in den Speicher zu laden, wenn sie wirklich benötigt werden. Dies ist ein wesentlicher Vorteil gegenüber OSGi-Bundles. Da letztere aktiv die von ihnen bereitgestellten Services bei der Registry anmelden müssen, ist es notwendig, dass sie beim Start der Platform bereits geladen und gestartet werden. [14]

3.3. Systemarchitektur von RCE

Auf der Basis aus dem OSGi-Framework Equinox und der RCP setzt die RCE-Kernkomponente auf. Diese ist bereits selbst als OSGi-Bundle realisiert und bietet RCE-spezifische Extension-Points an. Außerdem gehört zum Basissystem ein fester Satz von Erweiterungen, die für alle weiteren Bundles zugänglich sind: [17]

- Kommunikations-Komponente
- Service-Broker
- Privilegien-Komponente
- Update-Komponente
- Daten-Management

Für die Generierung von Stubs sind sowohl der Service-Broker als auch die Kommunikations-Komponenente von besonderer Bedeutung.

3.3.1. Der Service-Broker

Ein Ziel bei der Entwicklung von RCE ist es, dem Anwender den Zugriff auf verteilte Ressourcen einfach und transparent zu gestatten. Zu diesem Zweck gibt es einen Service-Broker, der ermitteln kann, welche RCE-spezifischen Extensions in den über Netzwerk angeschlossenen RCE-Instanzen laufen.

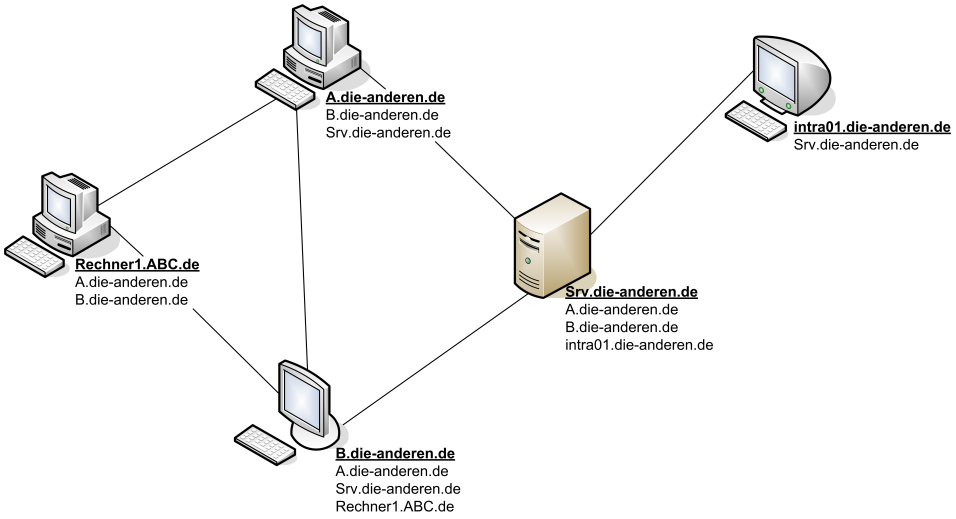

Abbildung 5: Netzwerktopologie mehrerer vernetzter RCE-Instanzen [eigen]

Die Suche nach Erweiterungen läuft dabei mehrstufig und dezentral ab. Es gibt keine zentrale Stelle, bei der alle laufenden RCE-Instanzen registriert sind. Stattdessen führt jede Instanz eine Liste der bekannten und erreichbaren „Nachbar-Instanzen“, die es direkt ansprechen kann. Diese Nachbarn besitzen wiederum eine solche Liste und so weiter.

Die somit entstehende Netzwerktopologie, die auf Abbildung 5 dargestellt ist, bietet unter anderem den Vorteil, dass mit einer kleinen Anzahl von bekannten Nachbarn ein großes Netz abgedeckt werden kann.

3.3.2. Die Kommunikations-Komponente

Zur Kommunikation zwischen den verschiedenen RCE-Instanzen wird die Kommunikations-Komponente genutzt. Diese stellt eine schmale, einfach nutzbare Schnittstelle bereit, um einen Nachrichtenaustausch mittels RPC-Techniken zu ermöglichen.

Die Kommunikations-Komponente verpackt einen Methodenaufruf in eine Anfrage. Diese Anfrage beinhaltet dabei den eindeutigen Bezeichner der Ziel-RCE-Instanz. Außerdem enthält sie die aufzurufende Java-Klasse, den Namen der Methode sowie die Übergabeparameter.

Diese Anfrage wird anschließend einem Kommunikator übergeben, welcher sich darum kümmert, dass sie serialisiert und an die Ziel-Instanz verschickt wird. Nachdem der Methodenaufruf am Zielobjekt durchgeführt wurde, wird auch das Ergebnis ebenfalls serialisiert und an den Aufrufer zurückgeschickt. Dabei können verschiedene, über einen Extension-Point erweiterbare Kommunikations-Protokolle zum Einsatz kommen.

Der lokale Aufruf einer beliebigen (Stub-)Methode wird also stets auf einen Aufruf einer speziellen Methode umgelenkt. Die wesentlichen Informationen über die ursprünglich aufgerufenen Methode (Klasse, Name, Parameter) werden dabei als Parameter übergeben.

3.3.3. Zugriffskontrolle im RCE-Netzwerk

Über die dargestellte Architektur wird Partnern aus verschiedenen Unternehmen, die an einem Projekt arbeite, die Möglichkeit geboten, auf verteilte Resourcen

zuzugreifen. Um schützenswerte Daten zu sichern ist deshalb eine Rechteverwaltung notwendig, die im RCE-Basispaket in Form der Privilegien-Komponente vorhanden ist. Diese arbeitet auf Grundlage von X.509-Zertifikaten.[1]

Jeder RCE-Nutzer bekommt ein eigenes, von einer zentralen Stelle signiertes, Zertifikat. Dieses ist passwortgeschützt und somit nur für den Eigentümer zugänglich. Beim Start einer RCE-Instanz muss der Nutzer sein Passwort eingeben, damit dieses Zertifikat entschlüsselt werden kann. Gleichzeitig wird der Benutzer somit authentifiziert. Anschließend nutzt RCE das nun entschlüsselte Zertifikat, um damit ein neu angelegtes (Proxy-)Zertifikat mit begrenzter Gültigkeitsdauer zu signieren. Dieses Proxy-Zertifikat dient zur Identifikation des Signierers als auch als gültiges RCE-Zertifikat, da die Signatur der ausgebenden Stelle vererbt wird (Chain of Trust). Die Abbildung 6 stellt die Beziehungen der einzelnen Zertifikate dar. [7], [22]

Die Vergabe der Rechte erfolgt stets durch den Besitzer der angesprochenen RCE-Instanz. Dieser darf festlegen, welcher Anwender welche Extensions auf seiner Instanz nutzen darf. Dazu wird ein Rollen-Konzept verwendet, bei dem jedem Anwender verschiedene Rollen wie `Administrator` oder `Benutzer` zugeordnet werden. Je nach zugeordneter Rolle und Extension werden dann Rechte zum Starten, Anhalten, Lesen oder Schreiben vergeben. Außerden können mehrere Anwender in einer Gruppe zusammengefasst und darüber mit Rollen assoziiert werden. Welcher Anwender sich hinter einem Zertifikat verbirgt, wird durch das Subject des jeweiligen Zertifikats festgelegt. [17], [12]

[1] X.509 ist ein Standard von der ITU-T, der eine Public-Key-Infrastruktur definiert und weite Verbreitung gefunden hat.

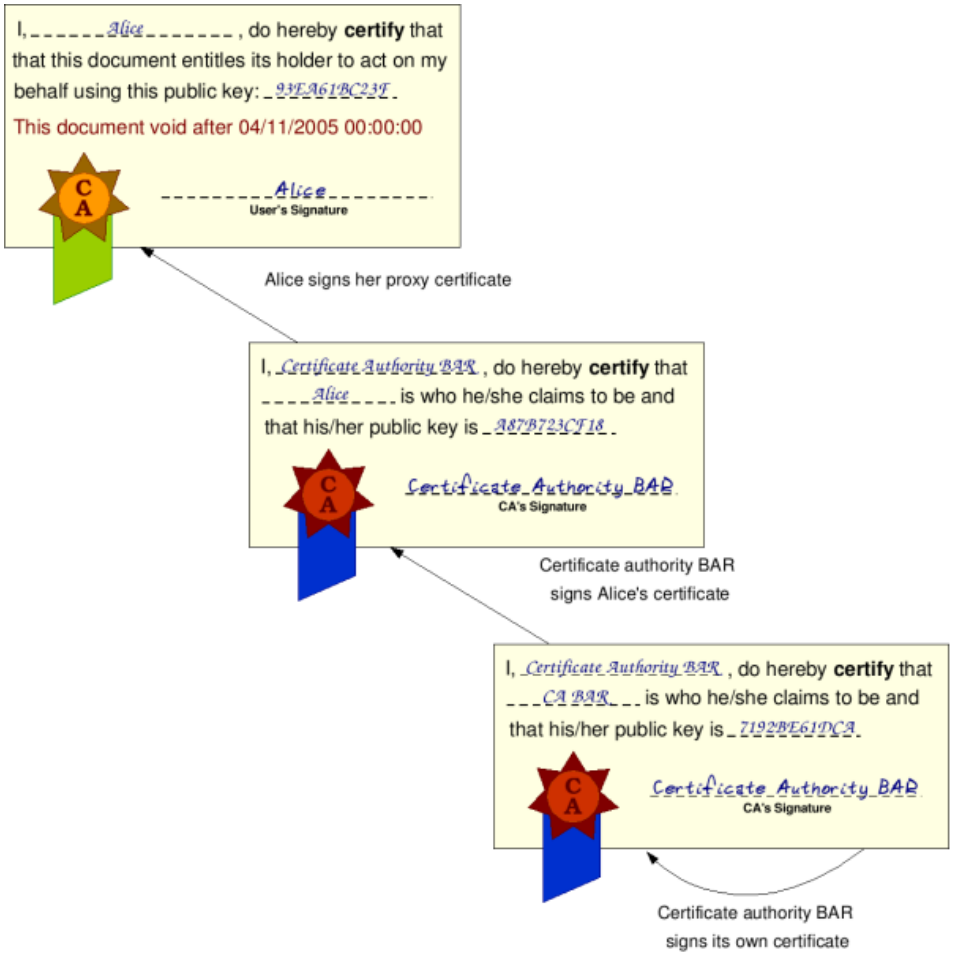

Abbildung 6: Die Chain of Trust gemäß RFC3820 [21]

4. Mögliche Lösungsansätze

Für die Implementierung eines Stub-Generators sind prinzipiell zwei verschiedene Lösungsansätze denkbar, die im Folgenden genauer betrachtet werden:

- Stub-Generierung während der Übersetzung einer RCE-Methode oder
- Stub-Generierung zur Laufzeit von RCE.

4.1. Generierung während der Übersetzung

Der erste Ansatz ist, die Stub-Generierung beim Übersetzen der Java-Quellen in Bytecode anzustoßen. Dies würde bedeuten, dass ein weiteres Werkzeug in die Build-Tool-Chain aufgenommen würde. Der Build-Prozess setzt derzeit auf dem Tool Maven auf. Dieses Werkzeug kann ausgehend von einer Beschreibung der Eingangsdaten verschiedene Ziele (*goals*) erreichen. Der Weg dorthin – also welche Werkzeuge wie aufzurufen sind – wird von Maven selbstständig unter Zuhilfenahme von Plugins ermittelt und ausgeführt. [19] Für das RCE wurde Maven um ein Plugin erweitert, das aus einer XML-Datei alle Bestandteile eines OSGi-Bundles ermittelt und diese zusammenpackt.

Um während dieses Vorgangs die jeweiligen Stubs für die RCE-Extensions gleichzeitig mitzugenerieren, ist es notwendig, Maven dieses Ziel beizubringen. Dazu kann beispielsweise das existierende Plugin erweitert werden. Außerdem muss es eine Möglichkeit geben, jene Java-Klassen zu spezifizieren, für die ein Stub erstellt werden soll. Um dies zu gewährleisten kann beispielsweise eine zentrale Liste geführt werden, die alle Klassen enthält, für die ein Stub generiert werden soll. Eine andere Möglichkeit ist, die Klassen mittels Java-Annotations zu markieren.

Vorteilhaft bei dieser Generierungs-Variante ist, dass jeder Stub nur einmal generiert werden muss. Außerdem kann so den nativen Java-Compiler genutzt und von dessen Optimierungen profitiert werden. Ändert sich eine Schnittstelle oder Klasse, so ist Maven in der Lage dies mit seinen internen Mitteln zu erkennen und eine erneute Stub-Generierung anzustoßen. Der auf diese Weise generierte Stub liegt anschließend als fertig gepacktes OSGi-Bundle vor und kann einfach installiert werden. Eine Verzögerung in der Ausführung von RCE tritt nicht auf.

Hauptproblem bei diesem Ansatz ist das Verteilen der Stubs, da neue Versionen auch allen Nutzern zugänglich gemacht und von diesen installiert werden müssen. Die Entwickler müssen dafür Sorge tragen, dass die Markierung der Klassen, für

die ein Stub generiert werden soll, immer aktuell sind. Der Generierungs-Prozess an sich ist auf diese Weise zwar recht langsam, dies fällt jedoch kaum ins Gewicht, da er nur einmal beim Übersetzen von RCE notwendig ist.

4.2. Generierung zur Laufzeit

Die Rahmenbedingungen für die Laufzeit-Generierung unterscheiden sich stark von den für die Generierung während der Übersetzung. Hier liegt das komplette System ausschließlich in Form ausführbarer Dateien vor. Dank der im Abschnitt 5.2.1 beschriebenen Java-Reflection-API ist es möglich, auf effiziente Weise die relevanten Klassen-Informationen aus diesen Dateien zu extrahieren. Außerdem wird der Stub-Generator nur auf Anfrage angestoßen und auch nur jene Stubs generieren, die gerade benötigt werden.

Nutzt man diese Variante, so entsteht kein Mehraufwand für die Entwickler von Extensions und der Build-Prozess kann unangetastet bleiben. Die Stubs, die von den Entwicklern trotz des Stub-Generators von Hand entwicklet wurden, haben automatisch Vorrang, da der Generierungsprozess im Fall eines existierenden Stubs gar nicht erst gestartet wird. Zusätzlich ist es so möglich, auch Stubs für Extensions zu generieren, die ursprünglich gar nicht für den entfernten Zugriff gedacht waren. Die Versionierung kann mit den OSGi-eigenen Bundle-Versions-Mechanismus sowie über die bereits existierende Versionierung von RCE-Extensions behandelt werden.

Gegen den Einsatz eines Laufzeit-Stub-Generators spricht, dass bei der Ausführung von RCE Verzögerungen auftreten, sobald der Generator in Aktion tritt. Diese müssen minimiert werden, indem man zum Beispiel einen Cache einführt, in dem einmal generierte Stubs gespeichert werden. Somit wird erreicht, dass jeder Stub nur einmal erstellt werden muss. Weiterhin ist es für einen erfolgreichen Einsatz eines Laufzeit-Stub-Generators unbedingt notwendig, dass die Beziehungen zwischen OSGi-Bundles vollständig und konsistent sind. Dies wird zwar durch das OSGi-Framework auch schon vorausgesetzt, jedoch sind hier noch Lücken möglich, die für den Stub-Generator einen Fehlschlag zur Folge haben. Allerdings ist eine vollständige und konsistente Pflege der Beziehungen zwischen den einzelnen Bundles aus Entwicklersicht sowieso wünschenswert.

Insgesamt ist festzustellen, dass die Stub-Generierung zur Laufzeit klare Vorteile im Gegensatz zu der Stub-Generierung während der Übersetzung bietet. Der Entwickler hat keinen Mehraufwand, der Nutzer muss sich nicht um die Konsistenz

seiner Stubs kümmern sondern bekommt die Stubs, die er gerade benötigt, ohne etwas davon zu bemerken.

5. Theoretische Grundlagen für die Umsetzung

Um mögliche Lösungsvarianten zu der Problemstellung erarbeiten zu können, muss zunächst die Umgebung des Problems genauer betrachtet werden. Dazu zählen insbesondere die Java-Laufzeitumgebung sowie die Theorie der Codegenerierung.

5.1. Die Programmiersprache Java

Java ist eine 1995 von Sun Microsystems veröffentlichte Programmiersprache. Sie zeichnet sich durch strikte Objektorientierung aus. Außerdem bringt sie eine umfangreiche Klassenbibliothek mit und ist durch das Konzept der virtuellen Maschine plattformunabhängig.

5.1.1. Java Virutal Machine

Eine wesentliche Eigenschaft von Java ist, dass kompilierte Java-Programme nicht direkt auf der Hardware laufen, sondern in einer Java Virutal Machine (JVM) ausgeführt werden. Dabei handelt es sich um einen Kellerautomaten, der Befehle in einer eigenen „Maschinensprache“ entgegen nimmt. Diese Maschinensprache heißt Bytecode.

Der Einsatz einer virtuellen Maschine als Schicht zwischen den ausführbaren Programmen und dem Betriebssystem beziehungsweise der Hardware hat den Vorteil, dass Programme nicht für jede Plattform neu übersetzt werden müssen. Es ist also möglich, verschiedene Betriebssysteme und Rechnerarchitekturen mit Hilfe einer einzigen Binärversion eines Programms zu bedienen. Man ist weder gezwungen, auf verschiedenen Plattformen kompilierte Varianten bereitzustellen, noch muss man den Quelltext ausliefern. Die einzige Voraussetung ist eine lauffähige JVM auf der Zielplattform.[2]

Ein weiterer Vorteil, der durch den Einsatz einer virtuellen Maschine entsteht, ist die erhöhte Sicherheit. Da die VM jeden auszuführenden Befehl verarbeiten muss, kann auch eine sehr feingranulare Sicherheitspolitik eingeführt werden. Die JVM stellt für die auszuführende Applikation eine Sandbox dar. Das heißt, dass die Applikation nicht das System selbst sieht, auf dem sie läuft. Stattdessen entscheidet

[2]Eine Ausnahme bilden Programme, die auf nativen Code zurückgreifen. Ist dies der Fall, müssen die entsprechenden Bibliotheken für die Zielplattform zusätzlich vorliegen.

die JVM, was sie der Applikation alles mitteilt. Dies kann soweit eingeschränkt werden, dass das laufende Programm nicht einmal in der Lage ist, die tatsächliche Plattform zu bestimmen. [23]

5.1.2. Java Classloader

Jede Klasse, die innerhalb der JVM genutzt wird, liegt als eigenständige Bytecode-Datei vor. Diese Dateien müssen zur Laufzeit in die virtuelle Maschine geladen werden. Darum kümmert sich der Classloader.

Während der von der Standardbibliothek bereitgestellte Classloader für die meisten Programme ausreicht, gibt es auch die Möglichkeit, einen selbst entwickelten Classloader zum Einsatz zu bringen. Dies ermöglicht unter anderem die dynamische Erzeugung von Klassen zur Laufzeit.

Ein Classloader hat die Aufgabe, die zu ladende Java-Klasse als Byte-Array in den Arbeitsspeicher zu laden und diese Daten der JVM mitzuteilen, damit diese anschließend Instanzen der Klasse erstellen kann. [23], [20]

Eine besondere Rolle spielen die Classloader im OSGi-Framework. Hier besitzt jedes Bundle einen eigenen Classloader, der dafür zuständig ist, die im Bundle enthaltenen Klassen der JVM bereitzustellen. Da eine Java-Klasse immer nur einmal in die JVM geladen werden darf, ist es also notwendig den richtigen Classloader zu erkennen und die entsprechende Klasse von diesem zu bekommen. Bundles, die Klassen aus anderen Bundles nutzen wollen, müssen darüber Bescheid wissen, in welchem Bundle die zu nutzende Klasse definiert ist. [8]

5.2. Code-Generierung

Unter einem Code-Generator versteht man ein Programm, dass aus einer Vorlage und semantischen Regeln in der Lage ist, Programme oder Programmteile selbstständig zu erzeugen. Damit soll dem Programmierer Arbeit abgenommen werden.

Prinzipiell arbeitet ein Code-Generator nach dem Schema, das auf Abbildung 7 dargestellt ist. Er bekommt also im weitersten Sinne eine Vorlage (*Template*) und einen Satz an Regeln (*Rules*), wie diese Vorlage auszufüllen ist. Diese beiden Eingaben werden dann kombiniert und der Generator gibt das Ergebnis (*Output*) aus. [15]

Beispiele für Code-Generatoren finden sich in vielen UML-Tools. Diese sind in der Lage aus den grafisch erstellten Klassen- oder Kooperationsdiagrammen (*Rules*) die Rümpfe für die tatsächliche Implementierung (*Output*) zu generieren. Der Programmierer muss nicht mehr langwierig die einzelnen Definitionen in die Quelltext-Dateien eintippen, die er zuvor schon im UML-Modell eingegeben hat. Außerdem kann er so sein Modell gleich in mehreren Programmiersprachen (*Template*) generieren lassen. Für das Umsetzen der eigentlichen Implementierung ist er jedoch weiterhin zuständig, da hierfür keine Informationen im UML-Modell vorhanden sind. [16]

Ein anderes Beispiel sind sogenannte Wrapper-Generatoren. Diese haben als Grundlage ein bereits vorhandenes Programm (*Rules*) und sollen dies von einer anderen Programmiersprache aus zugänglich machen (*Output*). Ein bekannter Vertreter dieser Gattung ist SWIG (Simplified Interface and Wrapper Generator), der in der Lage ist, C/C++-Code mittels verschiedener, erweiterbarer *Templates*, die hier *Exits* genannt werden, an unterschiedliche andere Spachsysteme wie Python, Java oder PHP anzubinden. [9]

Ein wichtiges Einsatzgebiet von Code-Generatoren ist auch die Generierung von Stub-Klassen zum Zugriff auf verteilt angebotene Services. Dabei kommen Protokolle wie SOAP zum Einsatz, die einen Funktionsaufruf samt Parametern in eine Nachricht verpacken, die an einen Server geschickt werden kann. Dort wird der Funktionsaufruf wieder ausgepackt und ausgeführt. Das Ergebnis wird anschließend wieder als Nachricht an den Client zurückgeschickt.

Es ist natürlich möglich, den Client die Anfrage direkt per vorher definiertem Protokoll verschicken zu lassen, schöner ist es jedoch, wenn er einfach nur einen normalen Funktionsaufruf machen muss und sich ein anderer Teil der Applikation um den Rest kümmert. Dazu werden sogenannte Stubs eingesetzt. Diese bieten nach außen hin die gleiche Schnittstelle an, wie auch der Service, der auf dem entfernten Server zugänglich ist. Wird so ein Stub aufgerufen, sorgt er selbstständig dafür, dass der Funktionsaufruf in das vorher definierte Protokoll verpackt und an den entsprechenden Server verschickt wird. [10], [11]

In diesem Fall ist also einerseits das Interface (*Rules*), dass vom Service angeboten wird, als auch da zu verwendende Protokoll (*Template*) bekannt. Es liegt also nahe, diese Stubs (*Output*) zu generieren, da der notwendige Programmcode ja bekannt ist und immer sehr ähnlich aussieht.

5.2.1. Codegenerierung in Java

Durch den Einsatz von Bytecode und Classloadern hat die Java-Laufzeitumgebung das Potential, sehr dynamisch zu agrieren. So ist es unter anderem möglich, zur Laufzeit eines Programms generierten Code in die JVM zu laden und zu nutzen. Diese Fähigkeit wird als Code-Injection bezeichnet. Damit kann das Verhalten von Java-Programmen zur Laufzeit dynamisch angepasst werdn und vorher erstellte Klassen sogar erst bei der Programmausführung zu manipuliert werden. Dazu muss der genutzte Classloader soweit ergänzt werden, dass er Klassen nicht einfach nur einliesen, sondern deren Bytecode auch interpretieren und verändern kann.

Ein weiteres Hilfsmittel ist die Java-Reflection-API. Mit Hilfe dieser Java-Standard-Schnittstelle ist es möglich, Klassen zur Laufzeit zu untersuchen. Sie bietet Methoden an, um von einem Objekt zunächst die Klasse zu bestimmen. Von dieser Klasse aus können dann die definierten Methoden und Eigenschaften sowie eventuell definierte Unterklassen ausgelesen werden. Außerdem bietet das API Zugriff auf die Basisklasse sowie die implementierten Interfaces. Diese Informationen reichen aus, um die gesamte Schnittstelle einer Klasse zu bestimmen.

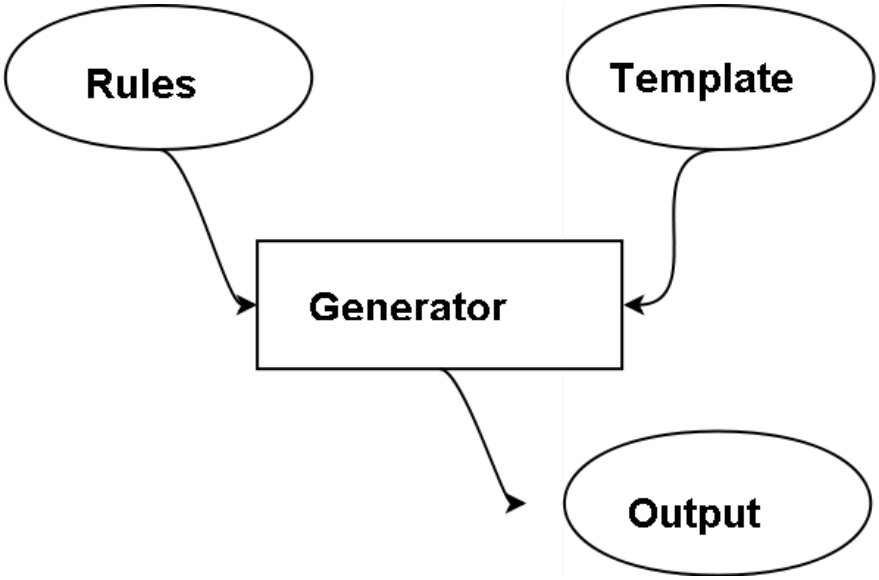

Abbildung 7: Prinzipieller Aufbau eines Code-Generators [eigen]

6. Anforderungen an die Generations-Komponente

Im Folgenden werden die Anforderungen an den Stub-Generator genauer betrachtet.

6.1. Start des RCE

Beschreibung: Der Bundle-Generator soll in der Lage sein, beim Start von RCE alle installierten Bundles so zu ergänzen, dass sie nach dem Initialisieren des OSGi-Kerns keine unaufgelösten Abhängigkeiten mehr haben.

Vorbedingungen: Der OSGi-Kern ist geladen. Alle Bundles wurden eingelesen. Alle Bundle-Abhängigkeiten wurden, soweit es geht, aufgelöst. Das Generator-Bundle wurde geladen und seine Abhängigkeiten erfolgreich aufgelöst. Der Bundle-Aktivator des Generator-Bundles wurde aufgerufen.

Nachbedingungen: Die Abhängigkeiten aller Bundles sind aufgelöst. Für noch offene Imports wurden leere Bundles mit entsprechenden Exports angelegt und installiert. Das generierte leere Bundle liegt persistent vor und kann wiederverwendet werden.

Fehlerbehandlung: Tritt ein Fehler auf, so soll der Generierungsprozess unmittelbar mit einer `GenerationException` abgebrochen werden. Alle Exceptions, die durch aufgerufene Klassen geworfen werden können, sollen aufgefangen werden und in eine `GenerationException` gekapselt werden.

6.2. Instanziierung eines Stubs

Beschreibung: Sobald der RCE-Kern die Anfrage bekommt, eine entfernt laufende Extension zu verwenden, startet er lokal einen entsprechenden Extension-Stub. Dieser kann allerdings nur gestartet werden, wenn er lokal vorliegt. Existiert kein Stub in der lokalen Installation, so wird dieser durch den Stub-Generator generiert.

Vorbedingungen: Es existiert eine entfernte Extension, die genutzt werden soll. Es existiert eine Referenz auf diese entfernte Extension. Diese Referenz enthält

- die Adresse der RCE-Instanz, in der die Extension installiert ist,

- einen Identifizierer, der die lokale RCE-Instanz eindeutig beschreibt, sowie
- den Namen und die Version der angeforderten Extension.

Weiterhin exitiert bereits ein generiertes Bundle, das entsprechend der Anforderung aus Abschnitt 6.1 generiert wurde und das Paket exportiert, in dem die aufgerufene Extension definiert wurde.

Nachbedingungen: Das Bundle, das entsprechend der Anforderung in Abschnitt 6.1 generiert wurde, wurde durch ein neues Bundle ersetzt, das neben den Metadaten des alten Bundles auch noch die Beschreibung des angeforderten Extension-Stubs sowie dessen Implementierung enthält. Das neu generierte Bundle ist installiert und seine Abhängigkeiten wurden erfolgreich aufgelöst. Die Stub-Implementierung der Extension wurde erfolgreich instanziiert.

Fehlerbehandlung: Es soll eine `GenerationException` geworfen werden, sobald ein Fehler auftritt. Der Generierungsprozess soll dabei abgebrochen werden. Wird kein generiertes Bundle gefunden, das die Vorbedingungen erfüllt, so ist dies als Fehler zu betrachten und eine `GenerationException` wird geworfen.

6.3. Konfiguration des Bundle-Generators

Beschreibung: Der Bundle-Generator soll nach den Wünschen des Benutzers konfigurierbar sein. Dabei soll der Nutzer angeben können, in welchem Verzeichnis der Bundle-Generator die generierten Bundles ablegen soll. Außerdem soll der Nutzer ein Verzeichnis angeben können, in dem temporäre Dateien abzulegen sind.

Vorbedingungen: Der RCE-Kern ist soweit hochgefahren, dass auf die intern verwalteten Konfigurationsdaten zugegriffen werden kann. Es liegt eine Konfigurationsdatei für die Stub-Generator-Komponente vor.

Nachbedingungen: Der Stub-Generator hat aus der Konfigurationsdatei die Werte eingelesen, die das Zielverzeichnis für generierte Bundles sowie das Verzeichnis für temporäre Dateien angeben.

Fehlerbehandlung: Sollte ein Wert nicht einlesbar sein, weil er nicht vorhanden ist, das Format nicht stimmt oder die Konfigurationsdatei aus anderen

Gründen nicht auslesbar ist, so soll der Generator sinnvolle Standard-Werte annehmen.

6.4. Auftreten einer `GenerationException`

Beschreibung: Alle Fehler, die im Stub-Generator auftreten, werden abgefangen und in eine `GenerationException` gekapselt. Dies ist somit die einzige Ausnahme, die den Stub-Generator verlassen kann und deutet auf ein Problem hin, das während der Code-Generierung aufgetreten ist. Der Generator soll darauf so reagieren, als wär er nicht aufgerufen worden.

Vorbedingungen: Es ist eine `GenerationException` aufgetreten, die außerhalb des Stub-Generator-Aufrufes aufgefangen wurde. Der Zustand vor dem Aufrufen des Stub-Generators ist gesichert worden und kann wieder abgerufen werden.

Nachbedingungen: Der Grund für die entstandene `GenerationException`wurde aufgezeichnet. Die Ergebnisse des Stub-Generator-Aufrufs wurden verworfen und der Zustand, der vor dem Aufruf des Generators gesichert wurde, wurde wieder hergestellt.

Fehlerbehandlung: Es gibt keine Fehler.

7. Bewertung des Einsatzes von Code-Generierung

Bevor ein Stub-Generator-Prototyp implementiert wird, muss objektiv bewertet werden, ob sich der Einsatz eines Code-Generators überhaupt lohnt. Dazu muss einerseits die technische Seite betrachtet werden um herauszufinden, welche Bibliotheken zum Einsatz kommen könnten. Auf der der anderen Seite muss aber auch überprüft werden, inwiefern generierter Code von der Leistung her mit handgeschriebenem Code mithalten kann und ob der Einsatz eines Code-Generators das Gesamt-Framework beeinflusst.

7.1. Vergleich von Laufzeit-Code-Generatoren

Für Java gibt es ein paar Bibliotheken, die in der Lage sind, Bytecode zu verarbeiten. Aus diesen soll eine geeignete ausgewählt werden. Dazu sind zunächst die relevanten Kritetrien festzulegen.

7.1.1. Anforderungen an die Bibliothek

Wie in Abschnitt 2 dargestellt, ist es Ziel, einen Stub zu generieren, der ein definiertes Interface lokal zur Verfügung stellt und automatisch die Aufrufe, die auf dieses Interface zugreifen, zu einer entfernten RCE-Instanz umzulenken. Als Hilfsmittel steht eine abstrakte Basisklasse bereit, die die Grundfunktionalität bereits enthält.

Der Generierungs-Prozess soll automatisch ablaufen, sobald ein entsprechender Stub benötigt wird, der Nutzer soll davon nichts mitbekommen. Das macht eine erweiterte Infrastruktur notwendig, damit Schnittstellenbeschreibungen automatisch generiert und dem Stub-Generator zur Verfügung gestellt werden können. Weiterhin sollten bereits generierte Stubs vorgehalten werden, um beim nächsten Aufruf des gleichen Interfaces wieder genutzt werden zu können.

Neben der technischen Seite ist auch die Lizensierung der Bibliothek von Bedeutung, da das RCE im Gegensatz zu dem Großteil der Code-Generierungs-Bibliotheken ganz klar Closed-Source ist.

Für Java gibt es einige Frameworks, die es ermöglichen, Code zu generieren. Unter [6] ist eine recht umfangreiche Sammlung zu finden. Aus diesen Bibliotheken muss die für das RCE am besten geeignete ausgewählt werden. Dazu wurden die einzelnen Bibliotheken nach den folgenden Gesichtpunkten untersucht:

- **Umfang der Bibliothek**

  Wichtig ist, dass die Bibliothek die notwendigen Features unterstützt. Dies ist im Wesentlichen, das Generieren von Proxy-Klassen für gegebene Java-Interfaces. Allerdings ist auch eine Erweiterung der Aufgabe auf Generierung von komplexeren Code zum Beispiel auf Basis von WSDL- oder IDL-Interfaces denkbar.

- **Caching-Möglichkeit**

  Da davon auszugehen ist, dass veröffentlichte Interfaces stabil sind, wäre es schön, wenn man einmal generierte Klassen auf der lokalen Festplatte ablegen kann, um sie später wieder zu verwenden.

- **Performance**

  Die Codegenerierung solle den Programmablauf nicht unnötig lange verzögern sondern schnell von statten gehen. Da Komponenten erst dann generiert werden sollen, wenn sie tatsächlich gebraucht werden, erfolgt der Aufruf des Code-Generators prinzipbedingt synchron, was eine Unterbrechung des restlichen Programm-Ablaufs nach sich zieht. Diese Unterbrechung soll nicht zu lange dauern, da Wartezeiten als störend empfunden werden.

- **Lizenz**

  Es gibt Open-Source-Lizenzen wie die General Public License (GPL), die von „abgeleiteten Werken“ verlangen, dass sie unter die gleiche oder eine kompatible Lizenz gestellt werden. Der Einsatz von Bibliotheken, die an eine derartige Lizenz gebunden sind, ist im RCE nicht möglich, da der Quelltext nicht freigegeben werden soll. Eine Übersicht über die wesentlichen Eigenschaften der verschiedenen Lizenzen ist in Anhang A zu finden.

- **Unterstützte Java-Version/-Features**

  Als Standard für die Entwicklung von RCE wird Java 1.5 eingesetzt. Die Bibliothek sollte also die Eigenschaften dieser Version unterstützen. Dabei gilt, dass es für die neuen Features in der Version 1.5 von Java zwar Erweiterungen der Sprache zum Beispiel um Generics gab, diese haben jedoch keine Änderungen an Struktur oder Inhalt der Bytecode-Dateien zur Folge. Einzig die neu eingeführten Annotations spiegeln sich direkt in Erweiterungen des Ausgabeformat wieder.

- **Anwendungsfreundlichkeit**

  Die Bibliothek soll sich so einfach wie möglich nutzen lassen, da es wenig Sinn macht, eine Bibliothek einzusetzten, für die der Entwickler erst eine neue Programmiersprache (wie etwa Java Bytecode) erlernen muss.

Im folgenden sind die Ergebnisse der Betrachtung von den sechs am meisten versprechenden Lösungen detailliert dargestellt. Ein Übersicht in Form einer Tabelle findet sich in Abschnitt 7.1.8.

7.1.2. ObjectWeb ASM

Anbieter:	ObjectWeb
Lizenz:	BSD
Homepage:	`http://asm.objectweb.org`

ObjectWeb ASM ist eine weit verbreitete Bibliothek wenn es um die Generierung von Bytecode geht. Ihr Spezialgebiet liegt in der dynamischen Erzeugung von Klassen mittels einer Objekt-Hierarchie, die die Struktur von Bytecode-Dateien abstrahiert und so Zugriff auf Bytecode-Befehle bietet. Dabei bewegt sie sich insgesamt auf einer niedrigen Ebene, die zwar hohe Performance des generierten Codes garantiert, aber auch mit hohem Aufwand beim Generieren verbunden ist.

Da es Zugriff auf Bytecode-Ebene bietet, ist es mit ASM kein Problem, auf die generierten Daten zuzugreifen. Der generierte Code kann also einfach ausgelesen und in Dateien gespeichert werden. Ein Caching-Mechanismus wäre somit einfach zu implementieren.

Die Bibliothek kommt mit einer ausführlichen Dokumentation in Form eines Handbuches und vereinfach somit den Einstieg. Allerdings sind zusätzlich zu dem vom Handbuch vermittelten Wissen auch fundierte Kenntnisse der JVM und dem von der VM verarbeiteten Bytecode notwendig. Somit sind dem Einsatz einige Hindernisse entgegengesetzt. ObjectWeb ASM wird allerdings auch von anderen Bibliotheken wie cgib oder Javassist eingesetzt, sodass man von der guten Performance profitieren und trotzdem auf Funktionen eines höheren Levels zurückgreifen kann.

7.1.3. cglib

Anbieter:	cglib Community
Lizenz:	Apache
Homepage:	`http://cglib.sourceforge.net`

cglib wird bereits in vielen Projekten erflogreich eingesetzt. So nutzt beispielsweise Hybernate cglib zur Generierung von Proxy-Objekten die Persistenzfunktionen implementieren. [2]

Der Schwerpunkt von cglib liegt im Erweitern von bereits bestehenden Objekten. So bietet es zum einen eine Proxy-Klasse, die der aus dem Java-Reflections-API ähnlich ist (vergleiche Abschnitt 7.1.5). Sie setzt jedoch auf ObjectWeb ASM (siehe Abschnitt 7.1.2) auf und erziehlt damit eine höhere Performance als die Java-interne Version, die komplett mit Reflections arbeitet.

Außer der Proxy-Klasse bietet cglib auch noch eine Enhancer-Klasse, die der Proxy-Klasse sehr ähnlich ist, jedoch neben den zu implementierenden Interfaces auch eine Basisklasse erlaubt. Somit kann ein größerer Teil der vom Kommunikations-Bundle bereitgestellten Funktionalität verwendet werden, da man auf die bereits vorhandene abstrakte Stub-Implementierung zurückgreifen kann.

Negativ fällt auf, dass von cglib generierte Klassen nicht persistiert werden können. Außerdem wird die entscheidende Funktionalität in eine Handler-Klasse ausgelagert, sodass für jeden Methodenaufruf mindestens zwei weitere, instanzübergreifende Aufrufe notwendig werden. Dies könnte Performance-Einbußen zur Folge haben.

7.1.4. Javassist

Anbieter:	Javassist Community
Lizenz:	MPL oder LGPL
Homepage:	`http://labs.jboss.com/javassist/`

Auch Javassist nutzt ObjectWeb ASM (7.1.2) als Grundlage, verfolgt allerdings einen anderen Ansatz als cglib (7.1.3). Javassist bringt einen eigenen, abgespeckten Java-Compiler mit, der es ermöglicht, neue Objekte zur Laufzeit zu erstellen und zu kompilieren. Die Syntax des Javassist-Compilers unterstützt allerdings

nicht den kompletten Java 1.5 Befehlssatz, die generierten Klassen sind aber hinterher binärkompatibel, fehlende Features müssen dementsprechend ersetzt werden.

Der Overhead beim Aufruf generierte Klassen hält sich sehr gering, da die Funktionalität direkt in einer neuen Klasse implementiert wird und auch Basis-Klassen genutzt werden können. Die vom Kommunikations-Bundle bereitgestellte Funktionalität kann also voll ausgenutzt werden.

Auf die von Javasssist generierten Klassen kann man auch direkt auf Bytecode-Ebene zugreifen, sodass ein cachen möglich ist. Man muss allderings darauf achten, dass die Klassen immer mit dem selben Classloader geladen werden, damit es nicht zwei Klassen mit dem gleichen Namen aber unterschiedlicher Implemtierung in der JVM gibt.

Javassist hat zwar Defizite in Bezug auf Java 1.5, bietet jedoch alle notwendigen Funktionalitäten an und ist auf guten Durchsatz ausgelegt [13].

7.1.5. java.reflect.Proxy

Mit `java.reflect.Proxy` stellt die Standard-Java-API eine Klasse bereit, die es ermöglicht, Interfaces zur Laufzeit zu implementieren. Dabei wird eine neue Proxy-Instanz erstellt, der anschließend ein Satz von Interfaces zugewiesen wird. Der Proxy stellt anschließend diese Schnittstelle bereit. Wird eine Methode eines dieser Interfaces aufgerufen, so wird der Aufruf an einen `InvocationHandler` weitergereicht, der dann die entsprechenden Operationen ausführen muss.

Die Implementierung basiert komplett auf der Java-Reflection-API, generierte Klassen gibt es in dem Sinn nicht, weshalb auch ein Speichern nicht möglich ist. Die Performance leidet ein wenig unter dem Overhead, der bei den Aufrufen über die Instanzgrenzen hinweg entsteht.

7.1.6. javax.tools.JavaCompiler

Der Java-Compiler ist für Java-Programme über die Schnittstelle `javax.tools.JavaCompiler` zugänglich. Dies bietet die Möglichkeit zur Laufzeit Programme zu übersetzen. Diese Funktionalität kann ähnlich wie Javassist (siehe Abschnitt 7.1.4) eingesetzt werden. Die entsprechende Bibliothek liegt jedem Java Developers Kit (JDK) bei.

Jedoch ist das JDK keine Grundvoraussetzung für RCE und somit müsste ein knapp 7 MB großes Archiv mit ausgeliefert werden. Auch die Lizensierung ist nicht ganz klar, da das JDK zwar kostenlos heruntergeladen werden kann, es jedoch nicht unter einer freien Lizenz veröffentlicht wurde. Aus Performance-Sicht ist zwar mit kompakten, effizienten generierten Klassen zu rechnen, jedoch ist der Aufwand des Generierens voraussichtlich ziemlich hoch.

7.1.7. Jython

Anbieter:	Jython Community
Lizenz:	Jython License
Homepage:	`http://www.jython.org`

Jython ist eine reine Java-Implementierung der Scriptsprache Python. Da Jython-Code wie eine normale Scriptsprache erst zur Laufzeit interpretiert wird, ist es problemlos mögich, Jython-Code auch erst zur Laufzeit zu generieren. Diese Scripte könnten auch entsprechend gespeichert werden um somit die generierten Klassen zu cachen.

Der Ansatz, Jython zu benutzen, klingt zunächst verlockend, da Python-Quelltext sehr einfach lesbar und leicht generierbar ist. Jedoch hat sich herausgestellt, dass Jython-Klassen aus Java heraus nur sehr umständlich aufrufbar sind. Außerdem ist die Performance im Verglich zu den anderen Lösungen sehr schlecht, da die Scripte nur interpretiert werden.

7.1.8. Ergebnisse des Vergleichs

Die Abbildung 8 stellt die vorangegangenen Ergebnisse noch einmal tabellarisch dar. Aus Platzgründen sind die Überschriften abgekürzt.

Auf den Einsatz von ObjectWeb ASM wurde verzichtet, da es mit cglib und Javassist zwei Alternativen gibt, die die gute Performance dieser Bibliothek zugänglich machen, jedoch eine wesentlich einfachere Schnittstelle bereitstellen.

Gegen die `Proxy`-Klasse aus der Java-Reflection-API spricht, dass es mit cglib einen vollwärtigen Ersatz mit besserer Leistung und erweitertem Funktionsumfang gibt. Deshalb wurde diese Möglichkeit nicht weiter betrachtet. Die andere native Variante mit der `JavaCompiler`-Klasse ist auf Grund ihrer unklaren Lizenz

Bibliothek	Umf.	Cache	Lizenz	Java5	Handh.
ObjectWeb ASM	20 kByte	Ja	LGPL	Bytecode	umständlich
cglib	25 kByte	Nein	MPL	unvollst.	einfach
Javassist	60 kByte	Ja	LGPL	Bytecode	einfach
java.reflect.Proxy	Built-in	Nein	+	unvollst.	einfach
javax.tools.JavaCompiler	7000 kByte	Ja	+	voll	umständlich
Jython	230 kByte	teilw.	+	unvollst.	einfach

Abbildung 8: Vergleich verschiedener Code-Generierungs-Frameworks [eigen]

und der großen Bibliothek nicht interessant. Außerdem ist auch damit zu rechnen, dass die Performance den anderen Lösungen hinterher hinkt.

Jython wurde nicht weiter betrachtet, da es keinen einfach Weg gibt, in Jython definierte Objekte aus Java-Klassen aus aufzurufen. Dies ist jedoch notwendig, da die Stubs die native Implementierung auf der Client-Seite ersetzen sollen und sich somit nahtlos in die Java-Umgebung integrieren müssen.

Für die Performance-Test bleiben also noch cglib sowie Javassist, die beide auf der ObjectWeb ASM Bibliothek basieren.

7.2. Test der Leistungsfähigkeit von cglib und Javassist

Die Bewertung ergab, dass Javassist unter der Voraussetzung von effizientem generierten Code und akzeptablen Laufzeiten die Bibliothek erster Wahl sein sollte. Jedoch wird auch cglib weiter betrachtet, da dieser Ansatz bessere Ergebnisse bei der Generierungs-Zeit verspricht und somit als Alternative in Betracht kommt, falls die Performance von Javassist widererwartend Probleme mit sich bringen sollte.

Für die Bewertung der Performance wurde eine Test-Anwendung entworfen, die dem Anwendungsszenario möglichst nahe kommt. Im Mittelpunkt steht das in Abbildung 9 beschriebene Interface. Dieses wird durch einen private innere Klasse des `InterfaceContainers` implementiert. Der `InterfaceContainer` stellt dabei eine Methode `doCall(...)` bereit, die einen Methodennamen und Parameter erwartet und die innere Klasse aufruft.

Um auf diese Klasse zugreifen zu können, muss ein Stub generiert werden, der die Methoden des Interfaces auf einen Aufruf der oben beschriebenen Methode `doCall(...)` umlenkt. Dies kommt der RPC-Technik mittels Kommunikations-Komponente, wie sie unter 3.3.2 beschrieben wurde, sehr nahe.

```
/* This is the interface used for performance testing. */
interface TestInterface {
    /* Gets the current time and returns the difference in
     * milliseconds expired since started.
     */
    Long stopTime(Date started);

    /* This method calculates the factorial of n by
     * recursively calling itself.
     */
    Long factorial(Long n);

    /* This method calculates the factorial of n by
     * recursively calling the stubFactorial method of
     * the stub passed.
     */
    Long stubFactorial(Long n, TestInterface stub);
}
```

Abbildung 9: Interface für Leistungstest

Gemessen wurden zwei Werte: Zunächst die Dauer, die eine Bibliothek braucht, um ein entsprechendes Stub-Objekt zu generieren und anschließend die Zeit die zwischen dem Aufruf der Stub-Methode und dem Aufruf der eigentlichen Methode im `InterfaceContainer` vergeht.

Die Quellen des gesamten Test-Programms sind in Anhang B zu finden. Zum Messen der Zeiten kamen Mini-Benchmarks zum Einsatz. Als Messgröße dient die Zeit, welche mit der Java-Funktion `System.currentTimeMilli()` ermittelt wird. [18] Die Tabelle 10 stellt die ermittelten Ergebnisse dar. Da die Werte für beide Bibliotheken fast identisch sind, kann davon ausgegangen werden, dass der durch die generierten Methoden entstandene Overhead vernachlässigbar ist. Auch die Zeit, die das Generieren der Stubs in Anspruch nimmt, ist vernachlässigbar klein.

Durch die Performance-Test-Ergebnisse wurde die Auswahl von Javassist bestätigt, da keine bedeutenden Leistungsverluste zu erwarten sind und der Komfort höher ist. Außerdem können so einmal erstellte Stubs über Neustarts des System hinweg wiederverwendet werden. Dies ist vorteilhaft, da so langwierige Netzwerkkommunikation vermieden werden kann.

Methode	Messung	cglib	Javassist
Geriereung	nativ	<1 ms	<1 ms
`stopTime(...)`	nativ	<1 ms	<1 ms
`factorial(...)`	nativ	6 ms	6 ms
`stubFactorial(...)`	nativ	175 ms	174 ms

Abbildung 10: Aufrufzeiten des Generators und des generierten Codes [eigen]

7.3. Leistungsmessungen am Prototypen

Auf Basis der Javassist-Bibliothek ist der im Abschnitt 8 beschriebenen Prototyp entstanden. Um zu prüfen, ob der Einsatz dieses Stub-Generators in der Praxis sinnvoll ist, wurden auch an ihm Leistungstest durchgeführt. Dazu sind drei Testfälle erstellt worden, die Extremfälle darstellen. Anschließend wurde die Zeit an verschiedenen Stellen im Code gemessen um so anschließend die Laufzeit verschiedener Teile des Programms zu ermitteln.

7.3.1. Die Klasse `StopWatch` zur Zeitmessung

Die Generierung von Stubs findet nicht nur Methoden- und Objektübergreifend statt, die beteiligten Klassen kommen teilweise auch aus verschiedenen Bundles. Deshalb ist es notwendig, sich eine geeignete Methode zur Zeitmessung zu überlegen. Da sämtliche Aufrufe, die an der Generierung beteiligt sind, synchron ablaufen, kann der ausführende Thread als Basis zur Zeitmessung genommen werden.

Dieser Ansatz wurde in der Klasse `StopWatch` umgesetzt. Diese bietet die beiden Klassen-Methoden `stopTime(Integer label_)` und `getDelta(Integer label1, Integer label2)` an. Dabei ist die erste Methode dazu da, den aktuellen Zeitpunkt unter dem angegebenen Bezeichner `label_` abzuspeichern. Dazu wird die Java-Methode `System.currTimeMillis()` benutzt, die die aktuelle Zeit in Millisekunden zurück gibt. Die zweite Methode liefert die absolute Differenz, die zwischen den beiden Zeitpunkten `label1` und `label2` verstrichen ist, in Millisekunden zurück. Ist einer der beiden Zeitpunkte nicht vorher über `stopTime(...)` festgelegt worden, so gibt sie 0 zurück.

Intern hält die Klasse `StopWatch` für jeden Thread, in dem sie aufgerufen wurde, eine Instanz von sich selbst. Somit ist es möglich, auch über Objektgrenzen hinweg die selbe `StopWatch` zu verwenden, solange die Aufrufe synchron sind.

7.3.2. Szenarien für den Prototyp-Test

Um die Laufzeiten beim Generieren, beim Aufruf und bei der Serialisierung zu ermitteln, wurden drei Testfälle in Form von RCE-Extension angelegt. Der erste Testfall bietet ein Interface, welches 36 Methoden definiert. Die Parameter und Rückgabetypen sind alle einfache Java-Typen. Dies stellt ein Extremfall dar, bei dem viel generiert werden muss. Der zweite Testfall bietet ein Interface an, das nur eine Methode bereitstellt, die nur einen Parameter erwartet. Dieser Parameter ist allerdings ein komplexes Objekt, das serialisiert werden muss. Das Interface des dritten Testfalls bietet eine Methode an, die nur einfach Java-Typen erwartet und nichts zurück gibt. Diese Methode ist achtfach überladen.

Für das Testen wurden zunächst alle Testfälle einmal als lokale Extensions installiert und einmal über eine entfernte RCE-Instanz zugänglich gemacht. Der Test für den Zugriff auf die entfernten Extensions wurde zweimal durchgeführt, da beim ersten mal die Generierung angestoßen wird, beim zweiten mal der Stub jedoch schon vorliegt. Außerdem wurde ein Stub-Bundle von Hand erstellt und zur Kommunikation genutzt.

Die Zeit wurde an den in Abbilung 11 dargestllten Punkten genommen. Daraus wurden anschließend die folgenden Zeiten gemessen:

- **Stub-Generierung** Dies entspricht der Zeit, die zwischen den Punkten 2 und 6 verstrichen ist. Diese Zeit konnte nur für den Fall gemessen werden, in dem weder der lokale Stub noch die Extension lokal vorlagen.
- **Instanziierung** Dies ist die Zeit, die der Programmfluss brauchte, um von Punkt 1 nach Punkt 7 zu gelangen.
- **Klasesn generieren** Dies ist die reine Generierungs-Zeit für den Stub an gemessen zwischen Punkt 3 und 4, jedoch ohne die Zeit für das langsame Persistieren des Bundles. Da dies ein Teil der Generierung ist, ist auch diese Zeit nur für den oben genannten Fall messbar.
- **Bundle persistieren** Dies ist die Zeit zwischen Punkt 4 und 5, die das Programm braucht, um alle generierten Komponenten auf die Festplatte zu schreiben. Für die Messbarkeit gilt das gleiche wie für die Stub-Generierung insgesamt.
- **Extension aufrufen** Dieser Test, der die Zeit zwischen den Punkten 7 und 8 misst, gibt Auskunft über die Leistungsfähigkeit der generierten

Stubs im Vergleich zu handgeschriebenen. Da bei dieser Messung Nachrichten über das Netzwerk verschickt werden, kann es zu umgebungsbedingten Messunschärfen kommen. Diese wurden weitgehend dadurch abgeschaltet, dass beide beteiligten RCE-Instanzen auf der selben Maschine liefen und der Messwert von zehn ausführungen gemittelt wurde.

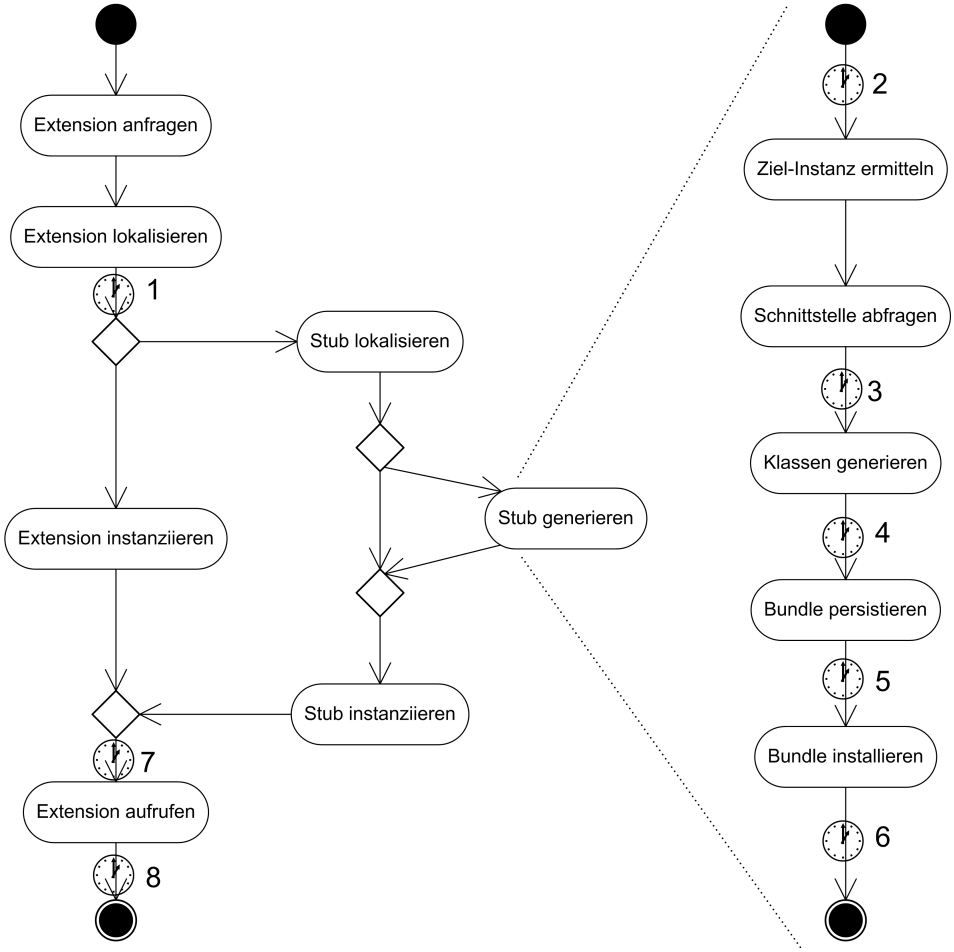

Abbildung 11: Zeitmesspunkte beim Aufruf von Extensions [eigen]

Die Tabellen in den Abbildungen 12, 13 sowie 14 stellen die Ergebnisse dar. Deutlich zu erkennen ist, dass die Stub-Instanziierung wesentlich länger dauert, wenn das Bundle erstellt werden muss. Sobald das Stub-Bundle jedoch einmal erstellt wurde, sinkt die Zeit für die Instanziierung erheblich.

Der Großteil der verstrichenen Zeit ist den Aufrufen über das Netzwerk zuzurechenen, von denen allein bei der Instanziierung des Stubs drei notwendig sind. Hier liegen ganz klar die Reserven, die allerdings auch durch die Kommunikations-Komponente genutzt werden müssen. In Hinblick auf die Aufrufszeiten ist der Unterschied zwischen generierten Stubs und denen, die vom Entwickler bereitge-

stellten werden, vernachlässigbar klein. Dieses Ergebnis war auch so zu erwarten, da der interne Ablauf der generierten Stubs dem Ablauf von ausprogrammierten entspricht.

	lokal	**entf. (1.)**	**entf. (2.)**	**lok. Stub**
Stub-Generierung		453 ms		
Instanziierung	<1 ms	531 ms	43 ms	41 ms
Klassen gen.		235 ms		
Bundle pers.		31 ms		
Ext. aufrufen	<1 ms	17 ms	16 ms	17 ms

Abbildung 12: Gemessene Zeiten für Testfall 1 [eigen]

	lokal	**entf. (1.)**	**entf. (2.)**	**lok. Stub**
Stub-Generierung		126 ms		
Instanziierung	<1 ms	157 ms	42 ms	42 ms
Klassen gen.		31 ms		
Bundle pers.		47 ms		
Ext. aufrufen	<1 ms	21 ms	21 ms	20 ms

Abbildung 13: Gemessene Zeiten für Testfall 2 [eigen]

	lokal	**entf. (1.)**	**entf. (2.)**	**lok. Stub**
Stub-Generierung		125 ms		
Instanziierung	<1 ms	175 ms	42 ms	41 ms
Klassen gen.		47 ms		
Bundle pers.		31 ms		
Ext. aufrufen	<1 ms	16 ms	16 ms	17 ms

Abbildung 14: Gemessene Zeiten für Testfall 3 [eigen]

8. Implementierung eines Prototypen

Nachdem im Abschnitt 5 die Entscheidung für Javassist gefallen ist, wird im folgenden die Umsetzung des Stub-Generator-Prototypen vorgestellt.

Die Generierung von Stub-Bundles läuft in zwei Stufen ab. Zunächst wird ein Dummy-Bundle generiert, das zwar alle wesentlichen OSGi-Bundle-Informationen enthält, jedoch keine Implementierungen mitbringt. Dieses wird in der zweiten Stufe durch ein erweitertes Bundle ersetzt, welches die eigentlichen Stubs bereitstellt. Die zweite Stufe der Generierung wird allerdings nur angestoßen, wenn das Bundle auch wirklich benötigt wird.

8.1. Klassenmodell des Stub-Generators

Das Klassendiagramm in Abbildung 15 zeigt die zentralen Klassen des Stub-Generators.

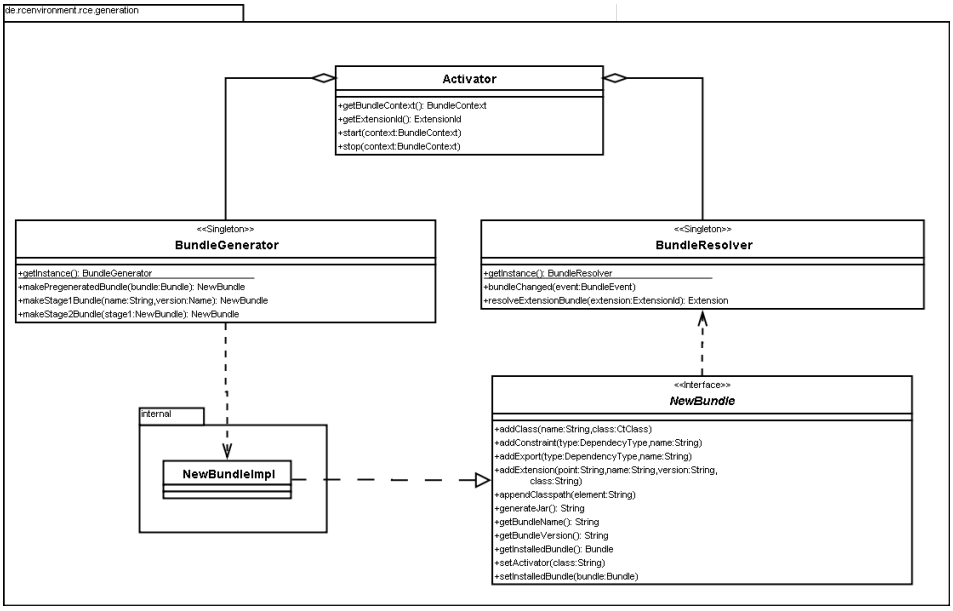

Abbildung 15: Das Klassenmodell des Stub-Generators [eigen]

8.1.1. Die Klasse `Activator`

Der `Activator` implementiert den OSGi-Bundle Activator. Dieser wird von dem OSGi-Framework beim ersten Bundle-Zugriff instanziiert und die `start()`-Methode wird ausgeführt. Dabei erstellt der `Activator` jeweils eine Instanz des

`BundleResolver`s und des `BundleGenerator`s, die beide als Singleton vorliegen. Die Klassen werden als Singleton und nicht als Untility-Klasse realisiert, da sie beim Erstellen einmalig ihre Eigenschaften entsprechend der Laufzeitumgebung und der vom Anwender bereitgestellten Konfigurationsparameter initialisieren werden müssen. Zu beiden Singletons hält der `Activator` einen Referenz, sodass diese erst wieder gelöscht werden, wenn das Bundle durch den Aufruf der `stop()`-Methode beendet wurde.

8.1.2. Die Klasse `BundleResolver`

Der `BundleResolver` ist die zentrale Klasse des Stub-Generators. Sie steuert den Generierungsprozess und hält alle dazu notwendigen Informationen. Wesentlich sind die beiden Methoden `resolveBundle()` und `resolveExtensionBundle()`. Um `NewBundle`-Instanzen zu generieren greift sie auf den `BundleGenerator` zurück.

Die erste der beiden Methoden wird beim Instanziieren der Klasse implizit für alle Bundles mit nicht aufgelösten Abhängigkeiten, die der OSGi-Kern geladen hat, aufgerufen. Dabei erledigt sie die Arbeitsschritte der ersten Stufe der Generierung (vergleiche 8.2). Am Ende steht ein neues OSGi-Bundle das bereits installiert ist und dessen Abhängigkeiten aufgelöst sind. Außerdem sind die Abhängigkeiten des Bundles, das den Aufruf dieser Funktion eingeleitet hat, im Normalfall anschließend auch auflösbar.

Die zweite Methode erstellt auf Basis eines generierten Bundles der ersten Stufe ein erweitertes Bundle, dass neben den Informationen der ersten Stufe auch noch die notwendigen Klassen enthält. Dazu benutzt es den `InterfaceHelper`, welcher in der Lage ist Schnittstellenbeschreibungen für Klassen, Interfaces und Aufzählungen aus einer entfernten RCE-Instanz zu extrahieren, zu serialisieren und lokal bereitzustellen.

Um aufbauend auf den Interfaces Stub-Klassen zu generieren, nutzt der `BundleResolver` die Utility-Klasse `StubGenerator`.

Die Bundles der zweiten Generation enthalten mindestens eine Extension. Diese wird ebenfalls generiert und von `GenericPlatformExtension` abgeleitet. Diese Klasse erbt von der Basis-Klasse für alle RCE-Extensions und enthält insbesondere die Methode `getDefault()`, welche den Stub der RCE-Applikation zurückgibt.

Am Ende der Methode wurde das alte Bundle aus der ersten Stufe durch ein neues ersetzt.

8.1.3. Die Klasse `BundleGenerator`

Die Singleton-Klasse `BundleGenerator` implementiert einige Factory-Methoden, die neue Klassen erzeugen. Diese implementieren das `NewBundle`-Interface. Dazu greift der `BundleGenerator` auf die interne Implementierung `NewBundleImpl` zurück. Bei der Erstellung unterscheidet er ob das generierte Bundle für die erste oder zweite Stufe der Generierung benötigt wird.

Durch eine Konfigurationsdatei kann dem `BundleGenerator` mitgeteilt werden, in welchem Verzeichnis die Bundles ihre temporären Dateien ablegen sollen. Außerdem ist es möglich das Verzeichnis festzulegen, in denen die generierten Bundles abgelegt werden sollen. Sind diese Attribute in der Konfigurationsdatei nicht gesetzt, so stellt die Klasse automatisch sinnvolle Vorgaben bereit.

8.1.4. Das Interface `NewBundle`

Der Stub-Generator das Interface `NewBundle` bereit, das Methoden enthält, um ein OSGi-Bundle zur Laufzeit zu erstellen.

Die Methoden erlauben es, Abhängigkeiten, exportierte Pakete, Eclipse-Erweiterungen sowie neue Klassen (in Form von `CtClass` aus der Javassist-Bibliothek) hinzuzufügen. Weiterhin bietet es die Möglichkeit, das Bundle zu einer Jar-Datei zusammenzupacken. Dabei werden auch alle Informationen über das Bundle, die bisher im Arbeitsspeicher gehalten wurden, auf die Festplatte geschrieben. Sobald dies geschehen ist, wird das Bundle als *frozen* markiert, das heißt es sind keine Änderungen an den Daten mehr möglich.

Der `BundleGenerator` benutzt, wie unter 8.1.3 beschrieben, eine interne Implementierung dieses Interfaces mit dem Namen `NewBundleImpl`. Diese Klasse greift auf die Utility-Klasse `JarPacker` zurück, um ein (temporäres) Verzeichnis zu einer Jar-Datei zusammenzupacken.

8.1.5. Die Utility-Klasse `InterfaceHelper`

Dem `InterfaceHelper` kommt eine weitere zentrale Rolle zu. Er ist so implementiert, dass er sowohl als lokale Klasse als auch als Stub für eine entfernte Klasse genutzt werden kann. Dazu stellt er lokal die Methode `getRemoteInterfaces(...)` bereit. Diese ruft auf der (im Methodenaufruf angegebenen) Gegenseite die Methode `getInterfaces(...)` auf.

Letztere ist dafür verantwortlich, dass die Applikation hinter der angegebenen RCE-Methode ermittelt und deren Schnittstellen in serieller Form zum Aufrufer übertragen werden. Dazu nutzt der Helper die Klasse `InterfaceDescription`. Diese stellt Factory-Methoden bereit, um entweder anhand einer Java `Class`-Instanz oder eines XML-DOM-Knoten eine neue Instanz zu erstellen. Weiterhin verfügen die Instanzen über Methoden, die die geladene Beschreibung als XML (serialisiert) zurückgiben und anhand der geladenen Beschreibung eine neue (deserialisierte) `CtClass` erstellen können.

Diese Technik macht es sehr einfach, unbekannte benötigte Typen in der lokalen RCE-Instanz bereitzustellen. Da diese generierten Typen jedoch ausschließlich als Interface verwendet werden sollen, auch wenn es sich streng genommen um Klassen handelt, werfen alle generierten Methodenrümpfe Ausnahmen.

8.1.6. Die Utility-Klasse `StubGenerator`

Der `StubGenerator` ist für die Zusammenfassung von Interfaces und gegebenenfalls einer Basis-Klasse zu einem Stub für die Kommunikations-Komponente zuständig. Je nachdem ob eine Basis-Klasse existiert, erstellt er einen Stub, der direkt auf vom API der Kommunikations-Komponenten bereitgestellten Klasse `AbstractStub` basiert, oder einen Stub, der eine Unterklasse von dieser als privates Feld enthält. Auf jeden Fall muss er das Interface `CommunicationStub` implementieren, um reibungsfrei zu funktionieren.

Für die von den Interfaces und der Basisklasse definierten Methoden werden Rümpfe generiert, die Aufrufe durch die `doDefaultCall(...)`-Methode aus dem `AbstractStub` zu einer entfernten Instanz umleiten. Dabei prüft bereits der Generator, ob der Rückgabetyp und die Parametertypen serialisierbar sind. Findet er dabei einen Fehler, so wird die Methode so implementiert, dass sie bei einem späteren Aufruf eine Ausnahme auslöst.

8.2. Erste Stufe der Stub-Generierung

Beim Start von RCE wird, falls vorhanden, auch das Stub-Generator-Bundle gestartet und der `BundleResolver` wird initialisiert. Dieser registriert sich als `BundleListener` beim OSGi-Framework, damit er über alle Änderungen, die Bundles betreffen, informiert wird.

Anschließend werden alle Bundles nach zwei Kriterien untersucht. Zunächst wird überprüft, ob das gerade betrachtete Bundle bereits ein generiertes ist. Ist dies der Fall, so wird intern eine Referenz darauf gespeichert, da es sich um ein Bundle aus der ersten Stufe handelt und es somit später noch gebraucht werden könnte. Bundles aus der zweiten Stufe werden nicht mehr explizit als generiert markiert.

Danach wird untersucht, ob sich das Bundle in einem Zustand befindet, in dem alle Abhängigkeiten aufgelöst sind. Dies sind gemäß Abschnitt 3.1 `RESOLVED`, `STARTING`, `ACTIVE` und `STOPPING`. Ist dies nicht der Fall, so wird die unter 8.1.2 angesprochene `resolveBundle()`-Methode aufgerufen.

8.2.1. Ablauf der Methode `resolveBundle()`

Die Methode bekommt als Parameter nur das Bundle mit, dessen Abhängigkeiten aufzulösen sind. Bei diesen werden zunächst die zu importierenden Pakete (Manifest-Header `Import-Package`) sowie die benötigten Bundles (Manifest-Header `Require-Bundle`) ermittelt und mit den gefundenen Paketen und Bundles abgeglichen. Somit wird festgestellt, welche Bundles generiert werden müssen.

Danach wird zunächst die Liste der notwendigen, nicht gefundenen Bundles abgearbeitet. Für jedes dieser Bundles wir mit Hilfe des `BundleGenerator`s ein Bundle der ersten Stufe angelegt. In dieses werden anschließend die auflösbaren, importierten Pakete und die auflösbaren Bundle-Abhängigkeiten aus dem übergeordneten Bundle übernommen.

Anschließend wird die Liste der nicht-auflösbaren Paketimporte nach den Paketen durchsucht, die mit dem Namen des neu generierten Bundles beginnen. Diese Importe werden dann in die Liste der Exporte des neuen Bundles übernommen. Zum Schluss wird das Bundle zusammengepackt und installiert.

Sind alle unbekannten Bundles auf diese Weise aufgelöst worden, so wird nun für jeden übrig gebliebenen Paket-Import auch ein Bundle erstellt, dessen Name dem Namen des zu importierenen Paketes entspricht. Der Rest läuft analog zum Vorgehen bei der Auflösung der Bundle-Abhängigkeiten.

Nach dem Generieren der Paket-Bundles wird nochmals versucht, die Abhängigkeiten des übergeordneten Bundles aufzulösen, was zu diesem Zeitpunkt funktionieren sollte.

Sind alle bereits geladenen Bundles auf diese Weise aufgelöst worden, so ist der `BundleResolver` erfolgreich initialisiert und RCE kann vollständig gestartet wer-

den. Der Resolver tritt erst wieder in Aktion, wenn zum Beispiel ein (GUI-)Plugin auf eine entfernte RCE-Extension zugreifen will, der passende Stub jedoch nicht vorliegt.

8.3. Zweite Stufe der Stub-Generierung

Alle RCE-Extensions werden über eine zentrale Methode im RCE-Kernmodul gestartet und abgefragt. Dabei wird eine Beschreibung der gewünschten Extension übergeben und der Kern entscheidet, ob es sich um eine lokale oder entfernt laufende Extension handelt. Wird dabei festgestellt, dass die RCE-Extension entfernt vorliegt, wird automatisch versucht, den entsprechenden Stub zu instanzieren. Erst wenn dies fehlschlägt, wird `resolveExtensionBundle()`-Methode des `BundleResolver`s aufgerufen und der zweite Schritt der Code-Generierung wird durchlaufen. Somit wird die Anzahl der Aufrufe der Generierungs-Routinen minimiert.

8.3.1. Ablauf der Methode `resolveExtensionBundle()`

Die Methode erwartet als Parameter lediglich die Beschreibung der RCE-Extension, für die der Stub generiert werden soll. Der Rückgabewert ist die Instanz einer von `GenericPlatformExtension` abgeleiteten Klasse, die den generierten Stub als Default-Objekt (über die Methode `getDefault()`) zurückgibt. Diese Erweiterung stammt aus einem neu generierten Bundle, das ein entsprechendes Bundle aus der ersten Stufe ersetzt.

Zunächst versucht die Methode das Bundle zu bestimmen, in dem der Stub liegen soll. Dabei wird nach einem Bundle aus der ersten Stufe gesucht, dessen Name den Prefix des Extension-Names bildet. Wird kein derartiges Bundle gefunden, so gilt die Code-Generierung an diesem Punkt als gescheitert und es wird eine `GenerationException` geworfen.

Ist ein entsprechendes Stufe-1-Bundle gefunden worden, so wird daraus nun mittels `BundleGenerator` ein Bundle der zweiten Stufe generiert. Dieses enthält die selben Import- und Export-Informationen wie des Basis-Bundle.

Anschließend wird die `getRemoteInterfaces()`-Methode des `InterfaceHelper` genutzt, um die Schnittstellen der entfernt laufenden RCE-Methode zu ermitteln. Als Rückgabewert liefert die Methode eine Liste von `InterfaceDescription`s. Die Einträge dieser Liste werden anschließend überprüft. Dabei wird getestet, ob die

beschriebene Schnittstelle bereits als Java-Klasse oder -Interface lokal vorliegt. Nur wenn dies nicht der Fall ist, wird die `InterfaceDescription` genutzt, um mit Hilfe der Javassist-Bibliothek den entsprechenden Java-Typ zu generieren (siehe Abschnitt 8.1.5). Anschließend sollten alle für den Stub benötigten Typen als Java-Bytecode vorliegen.

Aus der Liste der `InterfaceDescription`s werden nun jene ausgewählt, die von dem Stub der Applikation implementiert werden müssen. Je nachdem, ob unter den Beschreibungen eine (Basis-)Klasse enthalten ist, wird die entsprechende Methode des `StubGenerator`s aufgerufen und eine Stub-Klasse wird erstellt. Diese hat nach Konvention den selben Namen wie die originale Klasse der entfernten Instanz, liegt allerdings in dem Unterpaket `stub` unterhalb des Paketes in dem die originale Klasse liegt.

Nachdem nun der lauffähige Stub generiert wurde, muss nur noch die passende Unterklasse der `GenericPlatformExtension` erstellt werden. Dieser Klasse wird ein Konstruktor hinzugefügt, der den Klassennamen für den generierten Stub entsprechend setzt, sodass die generierte Erweiterung sich wie unter 8.1.2 beschrieben verhalten kann. Diese Extension wird außerdem für die Registrierung an einem Eclipse-Extension-Point vorbereitet, indem eine passende `plugin.xml`-Datei erzeugt wird.

Alle generierten Klassen werden anschließend zusammen mit der `plugin.xml`-Datei zu einem neuen Bundle zusammengepackt. Dieses Bundle wird anschließend in das OSGi-Framework installiert, wobei es das Bundle aus der ersten Stufe ersetzt. Die enthaltene Extension wird abschließend instanziiert und zurückgegeben.

8.3.2. Entfernte Interfaces durch `InterfaceHelper` herunterladen

Der `InterfaceHelper` ist eine Utility-Klasse, hat also nur statische Methoden. Er besteht aus den zwei in Abschnitt 8.1.5 beschriebenen Teilen: Zum einen die lokal ausgeführte Methode `getInterfaces()` und zum anderen den Methoden-Stub `getRemoteInterfaces()`.

Die Methode `getInterfaces()` erwartet als Eingabe den Namen und die Version einer RCE-Extension. Diese wird anschließend lokal geladen und die Applikation wird mittels `getDefault()` instanziiert. Von der geladenen Applikation können durch die Java-Reflection-API die implementierten Interfaces ermittelt

werden. Diese werden als `InterfaceDescription` samt aller inneren Klassen serialisiert. Enthält das Applikations-Objekt noch weitere Methoden, die durch keines der Interfaces abgedeckt werden, wird außerdem die Klasse selbst in eine `InterfaceDescription` umgewandelt.

Alle erzeugten `InterfaceDescription`s werden anschließend in ein XML-Dokument kombiniert, das als Zeichenkette zurückgegeben wird. Die Eingabeparameter sowie der Rückgabewert sind als Zeichenketten konzipiert, da so die Möglichkeit einer Serialisierung gesichert ist.

Die zweite Methode – `getRemoteInterfaces()` – erwartet eine vollständige Extension-Beschreibung (samt Host und Instanzkennzeichner) als Parameter. Anhand dieser Identifikation kann eine Anfrage an den `InterfaceHelper`, der in der entfernten Instanz läuft, gestartet werden. Der Rückgabewert dort der durch die Kommunikations-Komponente aufgerufenen Methode `getInterfaces()` wird anschließend wieder zu einem XML-DOM-Dokument geparst, aus dem wiederum alle Knoten extrahiert werden, die eine `InterfaceDescription` darstellen. Wenn diese Knoten noch entsprechend in `InterfaceDescription`-Instanzen verpackt wurden, werden diese als Liste von der (Stub-)Methode zurückgeliefert.

8.4. Behandlung von auftretenden Fehlern

Tritt während der Generierung der ersten Stufe ein Fehler auf, so wird das gerade betrachtete Bundle verworfen und es wird mit dem Auflösen der Abhängigkeiten des nächsten, nicht aufgelösten Bundles fortgefahren. Dies hat zur Folge, dass das verworfenen Bundle nicht in den Status `RESOVLED` übergehen und somit auch nicht verwendet werden kann (vergleiche Abschnitt 3.1).

Fehler, die während der zweiten Stufe der Generierung auftreten, resultieren prinzipiell in einer `GenerationException`. Alle anderen Exceptions werden an den Stellen gefangen, wo sie auftreten. Sollten sie nicht behandelbar sein, so werden sie auch in `GenerationException`s verpackt und neu geworfen. Die `GenerationException`s werden erst an dem Punkt im RCE-Kern gefangen, an dem die zweite Stufe Stub-Generierung ausgelöst wurde. Dort wird die aufgetretene Exception protokolliert und anschließend wird das Programm so weiter ausgeführt, als ob der Bundle-Generator nicht in Aktion getreten wäre.

8.5. Versionierung von generierten Stubs

Sobald eine neue Version einer RCE-Extension erstellt wird, müssen auch die Stubs angepasst werden. Das funktioniert dank der RCE-Architektur automatisch, da eine RCE-Extension nicht nur über ihren Namen sondern auch durch ihre Version eindeutig bestimmt ist. Dies hat zur Folge, dass die alten Stubs nicht mehr erkannt werden, sobald sich die Version der Extension ändert. Dadurch wird ein erneuter Generierungs-Prozess notwendig, der wiederum automatisch ausgelöst wird.

8.6. Einschränkungen und offene Probleme der Implementierung

Die vorgestellte Implementierung stellt einen Prototypen dar, der zwar weitgehend funktionsfähig ist, jedoch auch noch Raum für Verbesserungen lässt. Im Folgenden werden die Probleme beschrieben, die vor dem ständigen Einsatz in einem Produktivsystem verbessert werden sollten.

8.6.1. Setzen des Classpath für Javassist

Beim Generieren der Klassen wird auf die Bibliothek Javassist zurückgegriffen. Innerhalb dieser Bibliothek nimmt die Klasse `ClassPool` eine zentrale Rolle ein, da sie innerhalb des Frameworks und vor allem für den internen Compiler die Rolle eines Classloader einnimmt (siehe Abschnitt 5.1.2).

Damit der Compiler erfolgreich die Stub-Klassen generieren kann, müssen ihm alle verwendeten Klassen bekannt sein. Dabei stellen die generierten Intefaces kein Problem dar, da diese durch den geleichen `ClassPool` generiert wurden, wie die Stub-Klasse selber. Allerdings müssen auch die Typen, die durch die Imports von Dritt-Bundles genutzt werden, gefunden werden. Dazu werden eben jede Bundles in den Klassenpfad des genutzten `ClassPool`s aufgenommen.

Dieser Vorgang geschieht derzeit während der ersten Stufe der Stub-Generierung. Der ermittelte Pfad wird in den Bundle-Metadaten gespeichert und später für den zweiten Schritt wieder ausgelesen. Dieser Vorgang ist fehleranfällig, da sich das System zwischen der ersten und der zweiten Stufe der Stub-Generierung verändern kann. Außerdem ist die Ermittlung der Position der Bundles auf dem Datenträger noch sehr rudimentär.

Da der `ClassPool` auch nur in der zweiten Generations-Stufe notwendig ist, reicht es völlig aus, ihn während des Aufrufes von `resolveExtensionBunlde()` dynamisch zu initialisieren.

8.6.2. Rekursives Auflösen von Java-Typen

Die Ermittlung und Generierung von Basistypen durch die `InterfaceHelper`-Klasse funktioniert bisher ohne Rekursion. Das heißt, dass der `InterfaceHelper` nur die Typen zurückliefert, die von dem Stub direkt implementiert werden. Tritt jedoch ein Fehler beim Deserialisieren dieser `InterfaceDescription`s auf, die auf einen weiteren, unbekannten Typen zurückzuführen sind, so bricht die Stub-Generierung gemäß der Beschreibung unter 8.4 komplett ab, weil eine `GenerationException` ausgelöst wird.

Besser wäre es hier, dass der `IntefaceHelper` erneut aufgerufen wird, um die noch fehlenden Schnittstellen und Typen ebenfalls als `InterfaceDescription` entfernt zu serialisieren und der lokalen Instanz bereitzustellen. Dies würde die Erfolgswahrscheinlichkeit des Stub-Generators erheblich erhöhen, birgt aber auch die Gefahr, dass früher oder später zwei Klassen mit dem selben Namen auftauchen.

8.6.3. Ermitteln von Schnittstellen ohne Instanziierung

In der bisherigen Implementierung ist es notwendig, eine RCE-Extension zu instanziieren, um die notwendigen Interfaces zu ermitteln. Dies ist zwar eine einfache Lösung, jedoch nicht besonders elegant, da somit Bundles geladen und gestartet werden müssen, auch wenn sie später nicht mehr benötigt werden sollten.

Da die Schnittstellen ohnehin in ein wohldefiniertes XML-Format serialisiert werden, bietet es sich an, diese Serialisierung zu speichern. Dazu könnte zum Beispiel die `plugin.xml`-Datei verwendet werden, die jeder Extension beiliegt und auch beliebig erweiterbar ist.

9. Fazit

Das Generieren von Stubs zur Kommunikation auf Grundlage von Schnittstellenbeschreibungen ist seit einiger Zeit übliche Praxis. Dies war einer der Gründe, warum diese Technik auch in die Integrationsumgebung RCE Einzug halten sollte. Die durchgeführten Tests haben das Konzept als durchführbar bestätigt.

Dadurch, dass die Stub-Generierung automatisch abläuft, werden die Möglichhkeiten, die das RCE-System bietet, erweitert. So kann es in Verbindung mit dem Service-Broker und der Kommunikations-Komponente als eine hoch dynamische, verteilte Architektur genutzt werden, deren Einsatzmöglichkeiten über die schiffbauliche Anwendung hinaus gehen.

Das RCE ist allerdings noch in der Entwicklungsphase. Außerdem ist es das erste System, das auf diese Architektur aufsetzt, das von den Projektpartnern vor allem aus dem softwaretechnischen Bereich entwickelt wurde. Die Komponenten sind somit in gewisser Hinsicht als Prototypen anzusehen, die allerdings stabil genug laufen, um auch in Produktiv-Systemen eingesetzt zu werden.

Der im Rahmen dieser Arbeit entwickelte Prototyp sollte primär dazu dienen, die erarbeiteten Konzepte auf Tauglichkeit zu überprüfen. Er ist deshalb auch nicht als vollständig anzusehen, er kann allerdings als solide Ausgangsbasis für weitere Entwicklungen genommen werden. Insbesondere die Probleme, die mit dem Eclipse-Framework (zum Beispiel in Bezug auf die Classloader) auftreten, wurden robust und praxistauglich gelöst.

Denkbare Erweiterungen sind beispielsweise die Erweiterung des Stub-Generators, sodass auch andere Kommunikationswege als die Kommunikations-Komponente eingesetzt werden können. So kann etwa SOAP-basierte Web-Services integriert werden.

Ein klare Schwäche des Prototypen ist hingegen die zeitraubende Netzwerk-Kommunikation. Hier besteht klar Verbesserungsbedarf, wie die Leistungstests gezeigt haben. Ein Ansatz ist beispielsweise zu versuchen, die Kommunikation asynchron ablaufen zu lassen und parallel andere Aufgaben zu erledigen.

A. Lizenzen der Code-Generierung-Bibliotheken

Die Bibliotheken, die im Abschnitt **??** untersucht wurden, fallen alle unter mehr oder weniger liberale Open-Source-Lizenzen. Im folgenden werden die angewandten Lizenzen in ihren Besonderheiten kurz betrachtet, ohne jedoch den kompletten Lizenztext wiederzugeben.

Wichtig bei den Lizenzen ist, ob es sie ein *Copyleft* enthalten. Darunter versteht man Open-Source-Lizenzen, die eine Klausel enthalten, dass abgeleitete Werke auch unter die selbe Lizenz fallen müssen oder zumindest auch im Quelltext zugänglich gemacht werden müssen.

Apache-License Die von der Apache-Foundation herausgegebene Lizenz ist eine Open-Source-Lizenz ohne Copyleft. Sie erlaubt Nutzung und Änderung am Quelltext, solange explizit die Teile genannt werden, die unter der Apache-Lizenz stehen. Außerdem muss die Software einen Hinweis auf den Urheber dieser Softwareteile enthalten.

Wortlaut: `http://www.apache.org/licenses/`

BSD-Lizenz Die BSD-Lizenz (Berkley Software Distribution License) enthält keine Copyleft-Klausel. Sie erlaubt die Nutzung, Änderung und Weitergabe sämtlicher Quelltexte, solange die originale Copyright-Notiz erhalten bleibt.

Wortlaut: `http://www.opensource.org/licenses/bsd-license.php`

LGPL Die *Lesser General Public License* ist eine Copyleft-Lizenz, die es jedoch erlaubt, die lizensierte Software als *externen Teil* zu verwenden. Dies ist zum Beispiel der Fall, wenn ein Bibliothek zwar verwendet wird, jedoch selber nicht verändert wurde.

Wortlaut: `http://www.gnu.org/licenses/lgpl.html`

MPL Bei der *Mozilla Public License* handelt es sich um eine schwache Copyleft-Lizenz. Geänderter Quelltext muss zwar weiterhin unter der MPL veröffentlicht werden, darf jedoch auch in Closed-Source-Programmen verwendet werden.

Wortlaut: `http://www.mozilla.org/MPL/MPL-1.1.html`

B. Quelltext zur Testumgebung

Im Folgendne sind die kommentierten Originalquelltexte der Umgebung wiedergegeben, die für die Performance-Tests unter 7.2 eingesetzt wurden.

B.1. TestInterface.java

```
1  package eu.led_inc.generator.szenario;

   import java.util.Date;

5  /**
    *  This is the interface used for performance testing.
    *
    * @author Michael Meinel
    */
10 public interface TestInterface {
       /**
        * Gets the current time and returns the difference in
        * milliseconds expired since started.
        */
15     Long stopTime(Date started);

       /**
        * This method calculates the factorial of n by
        * recursively calling itself.
20      */
       Long factorial(Long n);

       /**
        * This method calculates the factorial of n by
25      * recursively calling the stubFactorial method of
        * the stub passed.
        */
       Long stubFactorial(Long n, TestInterface stub);
   }
```

B.2. Container.java

```
1  package eu.led_inc.generator.szenario;
2
3  import java.util.Date;
4
5  /**
6   * @author Michael Meinel
7   */
8  public class Container {
9          /**
10          * This holds an instance of the InnerImplementation class
11          * implementing the TestInterface.
12          */
13         private TestInterface innerImplementation = null;
14
15         /**
16          * Singleton instance.
17          */
18         private static Container instance = null;
19
20         /**
21          * Constructor.
22          */
23         public Container() {
24                 innerImplementation = new InnerImplementation();
25                 instance = this;
26         }
27
28         public static Container getInstance() {
29                 return instance;
30         }
31
32         /**
33          * This method delegates the call to the InnerImplementation.
34          *
35          * @param method Name of the method to call.
36          * @param parameters The parameters to pass to the method.
37          * @return The result of the called method.
38          */
39         public Long call(String method, Object... parameters) {
40                 Class<?>[] types = new Class<?>[parameters.length];
41                 for (int i = 0; i < types.length; i++) {
42                         if (i == 1) {
43                                 types[i] = TestInterface.class;
44                         } else {
45                                 types[i] = parameters[i].getClass();
46                         }
47                 }
48                 try {
49                         return (Long) innerImplementation
50                                 .getClass()
51                                 .getMethod(method, types)
52                                 .invoke(innerImplementation, parameters);
53                 } catch (Exception e) {
54                         return null;
55                 }
56         }
57
58         /**
59          * This is an implementation of the TestInterface that performs
60          * the requested task but is due to it's protection level only
61          * accessible through the call() method of Container.
62          * @author Michael Meinel
63          */
64         private class InnerImplementation implements TestInterface {
65                 /**
66                  * @inheritDoc
67                  *
68                  * {@see TestInterface#factorial(Long)}
69                  */
70                 public Long factorial(Long n) {
71                         if (n <= 0) {
72                                 return 1L;
73                         } else {
74                                 return n * factorial(n - 1);
75                         }
```

```
76                  }
                    /**
                     * @inheritDoc
                     *
81                   * {@see TestInterface#stopTime(Date)}
                     */
                    public Long stopTime(Date started) {
                            Date now = new Date();
                            return now.getTime() - started.getTime();
86                  }
                    /**
                     * @inheritDoc
                     *
91                   * {@see TestInterface#stubFactorial(Long, TestInterface)}
                     */
                    public Long stubFactorial(Long n, TestInterface stub) {
                            if (n <= 0) {
                                    return 1L;
96                          } else {
                                    return n * stub.stubFactorial(n - 1, stub);
                            }
                    }
101         }
}
```

B.3. Generator.java

```
1   package eu.led_inc.generator.test;

3   import eu.led_inc.generator.szenario.TestInterface;

    /**
     * @author Michael Meinel
     */
8   public interface Generator {
            /**
             * This should return a new instance of a generated adaptor that
             * exposes the TestInterface using the Container to invoke it's
             * code.
13           *
             * @return The new instance of the class implementing TestInterface.
             */
            TestInterface makeStubInstance();
    }
```

B.4. Main.java

```
1   package eu.led_inc.generator.test;

3   import java.util.Date;

    import eu.led_inc.generator.szenario.Container;
    import eu.led_inc.generator.szenario.TestInterface;

8   /**
     * @author meinel
     *
     */
    public class Main {
13
            private static final int RUNS = 1000;
            private static final long FACTORIAL = 50L;
            private static final int ROUNDS = 400;
```

```
18          /**
             * @param args
             */
            public static void main(String[] args) {
                    new Container();
23
                    Long start;

                    Long timeCGCreate = 0L;
                    Long timeCGStopTime = 0L;
28                  Long timeCGFactorial = 0L;
                    Long timeCGStubFactorial = 0L;

                    Long timeJACreate = 0L;
                    Long timeJAStopTime = 0L;
33                  Long timeJAFactorial = 0L;
                    Long timeJAStubFactorial = 0L;

                    Long startAll = System.currentTimeMillis();
                    for (int run = 0; run < ROUNDS; run++) {
38                          /* * * cglib * * */
                            start = System.currentTimeMillis();
                            TestInterface testClass = new eu.led_inc.generator.cglib.Generator()
                                    .makeStubInstance();
                            timeCGCreate += System.currentTimeMillis() - start;
43
                            timeCGStopTime += testClass.stopTime(new Date());

                            start = System.currentTimeMillis();
                            for (int i = 0; i < RUNS; i++) {
48                                  testClass.factorial(FACTORIAL);
                            }
                            timeCGFactorial += System.currentTimeMillis() - start;

                            start = System.currentTimeMillis();
53                          for (int i = 0; i < RUNS; i++) {
                                    testClass.stubFactorial(FACTORIAL, testClass);
                            }
                            timeCGStubFactorial += System.currentTimeMillis() - start;

58                          /* * * Javassist * * */
                            start = System.currentTimeMillis();
                            testClass = new eu.led_inc.generator.javassist.Generator().makeStubInstance();
                            timeJACreate += System.currentTimeMillis() - start;

63                          timeJAStopTime += testClass.stopTime(new Date());

                            start = System.currentTimeMillis();
                            for (int i = 0; i < RUNS; i++) {
                                    testClass.factorial(FACTORIAL);
68                          }
                            timeJAFactorial += System.currentTimeMillis() - start;

                            start = System.currentTimeMillis();
                            for (int i = 0; i < RUNS; i++) {
73                                  testClass.stubFactorial(FACTORIAL, testClass);
                            }
                            timeJAStubFactorial += System.currentTimeMillis() - start;
                    }
                    Long overallTime = System.currentTimeMillis() - startAll;
78
                    System.out.println("␣␣␣␣␣␣␣␣␣␣␣␣␣␣test␣runs:␣" + ROUNDS);
                    System.out.println("␣␣␣␣␣␣factorial␣cylcles:␣" + RUNS);
                    System.out.println("␣␣␣␣␣␣␣␣␣␣␣factorial␣of:␣" + FACTORIAL);
                    System.out.println("␣␣␣␣␣␣␣␣␣␣␣overall␣time:␣" + overallTime + "␣ms");
83                  System.out.println();
                    System.out.println("cglib\n-----");
                    System.out.println("␣␣␣␣␣␣␣␣␣␣␣␣create␣stub:␣" + timeCGCreate / ROUNDS + "␣ms");
                    System.out.println("␣␣␣␣␣␣␣␣␣␣stopTime(...):␣" + timeCGStopTime / ROUNDS + "␣ms");
                    System.out.println("␣␣␣␣␣␣␣␣␣␣factorial(10):␣" + timeCGFactorial / ROUNDS + "␣ms␣");
88                  System.out.println("stubFactorial(10,␣...):␣" + timeCGStubFactorial / ROUNDS + "␣ms");
                    System.out.println();
                    System.out.println("Javassist\n----------");
                    System.out.println("␣␣␣␣␣␣␣␣␣␣␣␣create␣stub:␣" + timeJACreate / ROUNDS + "␣ms");
                    System.out.println("␣␣␣␣␣␣␣␣␣␣stopTime(...):␣" + timeJAStopTime / ROUNDS + "␣ms");
93                  System.out.println("␣␣␣␣␣␣␣␣␣␣factorial(10):␣" + timeJAFactorial / ROUNDS + "␣ms");
                    System.out.println("stubFactorial(10,␣...):␣" + timeJAStubFactorial / ROUNDS + "␣ms");
            }
```

```
        private static Long adjustTiming() {
98              Long offset = 0L;
                for (int i = 0; i < ROUNDS; i++) {
                        Date start = new Date();
                        Date stop = new Date();
                        offset += stop.getTime() - start.getTime();
103             }
                offset /= ROUNDS;
                return offset;
        }
}
```

B.5. Generator.java (cglib)

```
1   package eu.led_inc.generator.cglib;

3   import java.lang.reflect.Method;

    import net.sf.cglib.proxy.InvocationHandler;
    import net.sf.cglib.proxy.Proxy;
    import eu.led_inc.generator.szenario.Container;
8   import eu.led_inc.generator.szenario.TestInterface;

    public class Generator implements eu.led_inc.generator.test.Generator {

        public TestInterface makeStubInstance() {
13              return (TestInterface)
                        Proxy.newProxyInstance(ClassLoader.getSystemClassLoader(),
                                               new Class<?>[] { TestInterface.class },
                                               new ProxyInvocationHandler());
        }
18
        private class ProxyInvocationHandler implements InvocationHandler {
                public Object invoke(Object instance, Method method, Object[] parameters)
                                throws Throwable {
                        return Container.getInstance().call(method.getName(), parameters);
23              }
        }
}
```

B.6. Generator.java (Javassist)

```
1   package eu.led_inc.generator.javassist;

    import javassist.ClassPool;
    import javassist.CtClass;
5   import javassist.CtMethod;
    import javassist.CtNewMethod;
    import javassist.Modifier;
    import eu.led_inc.generator.szenario.Container;
    import eu.led_inc.generator.szenario.TestInterface;
10
    /**
     * @author Michael Meinel
     */
    public class Generator implements eu.led_inc.generator.test.Generator {
15
        private static int count = 0;
        /**
         * @see eu.led_inc.generator.test.Generator#makeStubInstance(Container)
         */
20      public TestInterface makeStubInstance() {
            try {
                ClassPool.getDefault().importPackage("eu.led_inc.generator.szenario");
                CtClass testInterface = ClassPool.getDefault()
                        .get(TestInterface.class.getCanonicalName());
```

```
                        CtClass newClass = ClassPool.getDefault().makeClass("StubClass" + count++);
                        newClass.addInterface(testInterface);

                        for (CtMethod method: testInterface.getDeclaredMethods()) {
                                String parameters = method.getParameterTypes()[0].getName()
                                                + " arg0";
                                String arguments = "arg0";
                                if (method.getParameterTypes().length > 1) {
                                        parameters += ", " + method.getParameterTypes()[1].getName()
                                                        + " arg1";
                                        arguments += ", arg1";
                                }

                                String methodBody = "public Long " + method.getName() + "("
                                                + parameters + ") {\n"
                                  + "    return Container.getInstance().call(\""
                                  + method.getName() + "\", new Object[] {"
                                  + arguments + "});\n"
                                  + "}\n";

                                CtMethod newMethod = CtNewMethod.make(methodBody, newClass);
                                newMethod.setModifiers(newMethod.getModifiers() & ~Modifier.ABSTRACT);
                                newClass.addMethod(newMethod);
                        }

                        return (TestInterface) newClass.toClass().newInstance();
                } catch (Exception e) {
                        e.printStackTrace();
                        return null;
                }
        }

}
```

Literatur

[1] *Das DLR im Überblick.* `http://www.dlr.de/desktopdefault.aspx/tabid-636/`. – Stand: 18. September 2007

[2] *Homepage der Code Generation Library.* `http://cglib.sourceforge.net`. – Stand: 17. September 2007

[3] *Homepage der Einrichtung Simulations- und Softwaretechnik.* `http://www.dlr.de/sc/`. – Stand: 18. September 2007

[4] *Homepage des SESIS-Projektes.* `http://www.sesis.de`. – Stand: 9. September 2007

[5] *Homepage „Verteilte Systeme und Komponentensoftware“.* `http://www.dlr.de/sc/verteiltesysteme/`. – Stand: 18. September 2007

[6] *Open Source ByteCode Libraries in Java.* `http://java-source.net/open-source/bytecode-libraries`. – Stand: 11. September 2007

[7] *Public-key and attribute certificate frameworks.* X.509 (ITU-T Recommendation). `http://www.itu.int/rec/T-REC-X.509-200003-I`. Version: März 2000 (X — Data Networks and Open System Communications)

[8] *OSGi Service Platform Core Specification, Release 4.* `http://www2.osgi.org/Release4/Download`. Version: Jul. 2006

[9] Beazley, D. M.: SWIG : An Easy to Use Tool for Integrating Scripting Languages with C and C++. In: *4th Annual Tcl/Tk Workshop*, 1996

[10] Bloomer, J. : *Power Programming with RPC.* O'Reilly, 1992

[11] Boger, M. : *Java in verteilten Systemen. Nebenläufigkeit, Verteilung, Persistenz.* Dpunkt Verlag, 1999

[12] Brandes, T. ; Christiansen, K. ; Esins, E. ; Klein, J. ; Krämer-Fuhrmann, O. ; Metsch, T. ; Rossow, D. ; Schreiber, A. ; Schrödter, S. : *System Design für ein schiffbauliches Entwurfs- und Simulationssystem.* Mai 2007

[13] Chiba, S. : Javassist — A Reflection-based Programming Wizard for Java. In: *Proceedings of OOPSLA'98 Workshp on Reflective Programming in C++ and Java*, 1998

[14] Daum, B. : *Java-Entwicklung mit Eclipse 3. Anwendungen, Plugins und Rich Clients*. Dpunkt Verlag, 2004

[15] Herrington, J. : *Code Generation in Action*. Manning Publications, 2003

[16] Kecher, C. : *UML 2.0 – Das umfassende Handbuch*. Galileo Press, 2006

[17] Klein, J. ; Krämer-Fuhrmann, O. ; Metsch, T. ; Nurzenski, A. ; Schreiber, A. : *RCE System Design – Reconfigurable Computing Environment*. Febr. 2007

[18] Langer, A. ; Kreft, K. : Java Performance: Micro-Benchmarking. In: *JavaSPEKTRUM* (2005), Jul.

[19] Massol, V. ; O'Brien, T. M.: *Maven. Das Entwicklerheft*. O'Reilly, 2005

[20] Neward, T. : *Server-Based Java Programming*. Manning Publications, 2000

[21] Sotomayor, B. : *The Globus Toolkit 4 Programmer's Tutorial*. `http://gdp.globus.org/gt4-tutorial/`. Version: 2007

[22] Tuecke, S. ; Welch, V. ; Engert, D. ; Pearlman, L. ; Thompson, M. : *Internet X.509 Public Key Infrastructure (PKI) Proxy Certificate Profile*. RFC 3820 (Proposed Standard). `http://www.ietf.org/rfc/rfc3820.txt`. Version: Jun. 2004 (Request for Comments)

[23] Ullenboom, C. : *Java ist auch eine Insel. Programmieren mit der Java Standard Edition Version 5 / 6*. Galileo Press, 2006