INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Fortgeschrittenenpraktikum

Entwicklung von Adaptern für Datenquellen auf Linux-Systemen

Michael Dürr

Aufgabensteller: Prof. Dr. Heinz-Gerd Hegering

Betreuer: Vitalian Danciu
Martin Sailer
Nils gentschen Felde

Abgabetermin: 15. Dezember 2006

INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Fortgeschrittenenpraktikum

Entwicklung von Adaptern für Datenquellen auf Linux-Systemen

Michael Dürr

Aufgabensteller: Prof. Dr. Heinz-Gerd Hegering

Betreuer: Vitalian Danciu
Martin Sailer
Nils gentschen Felde

Abgabetermin: 15. Dezember 2006

Hiermit versichere ich, dass ich das vorliegende Fortgeschrittenenpraktikum selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 15. Dezember 2006

. . . . . .

(Unterschrift des Kandidaten)

Schon seit einiger Zeit zeichnet sich eine klare Tendenz eines Wandels vom reinen Netz- und Systemmanagement hin zum Service-Management ab. Um Service-Management-Werkzeuge zu realisieren, ist es wünschenswert, diesen eine geeigneten Service-Management-Informationsbasis zur Verfügung zu stellen ähnlich der Management-Informationsbasis beim Internet Management, die einen direkten Zugriff auf die einen Dienst beschreibenden Service-Attribute zur Verfügung stellt. Um eine derartige Informationsbasis mit den notwendigen Dienstinformationen anzureichern, bedarf es unter anderem einer passenden Service-Monitoring-Architektur, die es ermöglicht, aus der Vielzahl von Management-Ressourcen Management-Informationen abzuleiten, zu korrelieren, zu aggregieren und in geeigneter Form zu verfeinern. Für diesen Zweck wurde am Lehrstuhl für Kommunikationssysteme und Systemprogrammierung der Ludwig-Maximilians-Universität München die Service-Management-Architektur SMONA entwickelt. Eine Schicht dieser Architektur beherbergt Adapter, deren Aufgabe darin besteht, einen einheitlichen Zugriff auf die von den zahlreichen auf dem Markt etablierten Management-Werkzeugen oder selbst angefertigten Skripten gelieferten Management-Informationen zu bewerkstelligen. Außerdem sollen die Management-Informationen in einem normalisierten Format zur Verfügung gestellt werden, indem die Diversität der vorhandenen Informationsformate verschiedener Management-Werkzeuge auf dasselbe normalisierte Format abgebildet werden.

Im Rahmen dieses Fortgeschrittenenpraktikums wurden Adapter für Datenquellen auf Linux-Systemen entwickelt. Diese zeichnen sich besonders durch ihre leichte Erweiterbarkeit um neue Datenquellen aus. Dabei wurde speziell auf die Einbindung von Datenquellen abgezielt, die sich durch externe Programmaufrufe auslesen lassen. Die Einbindung von Werkzeugen als neue Datenquellen erfolgt dabei durch die Erweiterung von Konfigurationsdateien. In solchen Dateien können datenquellenspezifische Constraints vorgegeben werden sowie das Kommando, welches die zugewiesene Datenquelle ausliest. Damit ist es auf einfache Art und Weise möglich, einem Adapter neue Datenquellen schnell hinzuzufügen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

Schon lange werden im Bereich des Netz- und System-Managements Werkzeuge eingesetzt, die es ermöglichen, Management-Informationen über die vorhandenen Systemelemente auszulesen bzw. Management-Aktionen an diesen Elementen auszuführen. Vor einigen Jahren hat sich der Schwerpunkt vom reinen System- und Netz-Management hin zum Dienst-Management verlagert. Dienste können als Ergebnis von Operationen eines Verbundes von Ressourcen verstanden werden, wobei Ressourcen vorhandene Geräte und Anwendungen beschreiben [DHHS 06].

Netze und darin enthaltene Systeme dienen dazu, eine Vielzahl von Diensten bereitzustellen. Der Betrieb eines Dienstes kann dabei mit einem hohen Verwaltungsaufwand verbunden sein. Solche Dienste entstehen durch die Zusammenarbeit verschiedener Ressourcen, die sich sowohl aus Hardware-Elementen (Berechnungs-, Speicher- und Netzwerkelemente) als auch Software-Komponenten zusammensetzen. Eine wesentlichen Funktion, um Dienste ihren entsprechenden Anforderungen gerecht betreiben zu können, bietet das Fehler-Management. Fehler-Management beschäftigt sich mit dem Entdecken, Eingrenzen und Beheben von abnormalem Systemverhalten [HAN 99]. Die Aufgabe besteht darin, die Verfügbarkeit eines Netzes, seiner Systemelemente und der damit erbrachten Dienste hoch zu halten, indem eine schnelle Entdeckung und Behebung von Fehlern gewährleistet wird. Typische Teilaufgaben sind hierbei die Überwachung der Netz- und Systemelemente, das Entgegennehmen von Alarmen, die Diagnose von Fehlerursachen, das Feststellen von Fehlerfortpflanzungen, das Einleiten und Überprüfen von Fehlerbehebungsmaßnahmen, das Führen eines *Trouble-Ticket-Systems* sowie die Hilfestellung für den Benutzer zum Beispiel in Form eines *User Help Desk*.

Einen wichtigen Teil zur Beschreibung der Qualität eines Dienstes stellt auch die Einarbeitung von Rückmeldungen der Kunden in das Fehler-Management solcher Dienste dar. Beispielsweise muss ein Dienst nicht nur dahingehend verwaltet werden, Fehlfunktionen aufzudecken oder Dienstausfälle zu beheben. Es ist genauso wichtig, die Rückmeldungen über unzureichende Verfügbarkeit (Beispiel Webserver) oder verminderte Leistung (Beispiel Grid-Computing) in die Analyse und das Fehler-Management einfließen zu lassen, um bestehenden *Service Level Agreements (SLAs)* gerecht zu werden [DHHS 06]. Eine Aufgabe des Dienst-Management besteht darin, Abhängigkeiten verschiedener Ressourcen beurteilen zu können und ausgehend davon Ressourcen auf die entsprechenden Dienste abzubilden. Grundsätzlich wird das Dienst-Management durch die Aggregation und Korrelation der Daten einzelner Ressourcen realisiert. Um eine Dienstsicht zu erzielen, ist es notwendig, eine transparente Verfeinerung dieser Daten bereitzustellen.

Ein großes Problem stellt derzeit die Vielfalt der auf dem Markt etablierten Werkzeuge zum Überwachen und Konfigurieren von Netz- und Systemelementen dar. Ein Dienst-Management-System sollte die Fähigkeit besitzen, die Daten von System- und Netz-Management-Werkzeugen zu einer einzigen Dienstsicht zu kombinieren. Ziel ist es, trotz der vorherrschenden Heterogenität der auf dem Markt etablierten Werkzeuge, diese Sicht zu erwirken, indem gelieferte Daten der zugrundeliegenden Ressourcen in geeigneter Weise kombiniert werden. Ein Weg, um dies zu bewerkstelligen, wäre die Entwicklung neuer Werkzeuge, welche die Fähigkeiten besitzen, Abhängigkeiten zwischen Ressourcen oder Subdiensten zu erkennen und Aggregationen durchzuführen. Dieser Ansatz ist jedoch kaum durchführbar, da viele Netz- und Systemelemente an lang erprobten und weit verbreiteten Protokollen ausgerichtet sind und eine Neustrukturierung in den meisten Fällen zu kostspielig wäre. Daher ist es sinnvoll, die vorhandenen, lang bewährten und ausgereiften Werkzeuge beizubehalten und Wege zu finden, die ressourcenspezifischen Daten zu einer Dienstinformation zu aggregieren [DHHS 06].

Kapitel 2

Einordnung und Analyse

Am Lehrstuhl für Kommunikationssysteme und Systemprogrammierung der Ludwig-Maximilians-Universität München wurde die *Service Monitoring Architektur (SMONA)* entworfen [DaSa 05]. Diese Architektur erzielt die in der Einleitung geschilderte Dienstsicht, indem sie die Ressourcendaten der für das Dienst-Management vorhandenen Werkzeuge in geeigneter Weise abruft, filtert, aggregiert und korreliert. Im folgenden Abschnitt soll die Architektur erläutert und ihre Kernbestandteile vorgestellt werden. Danach wird die Problemstellung dieser Arbeit im Zusammenhang mit der Architektur erläutert (Abschnitt 2.2) und ein allgemeiner Anforderungskatalog für Adapter diskutiert (Abschnitt 2.3).

2.1 Konzeption von SMONA

SMONA verfolgt den Ansatz, technische Informationen, wie sie schon lange durch das Ressourcen-Monitoring etablierter Werkzeuge zur Verfügung gestellt werden, für das Dienst-Management wiederzuverwenden. Dienste werden durch ihre Attribute beschrieben, in Anlehnung an Attribute, wie sie von Ressourcen in *Management Information Bases (MIBs)* verwaltet werden [DgFS 07]. Ähnlich der Repräsentation von *Managed Objects* in einer MIB ist es wünschenswert, für das Service-Management eine *Service Management Information Base (SMIB)* zu verwalten [Sail 05]. SMONA ermöglicht die Ressourcendaten, die von Netz, System- und Anwendungs-Management-Werkzeugen aufgegriffen werden, so zu kombinieren, dass sie den zu verwaltenden Dienst möglichst vollständig reflektieren [DaSa 05]. Dabei müssen sowohl Abhängigkeiten der einzelnen Ressourcen, als auch Zusammenhänge zwischen den verschiedenen Diensten beachtet werden. Außerdem ist es notwendig, dass Mechanismen zur Verfügung stehen, direkte Anfragen an die Ressourcen durchzuführen bzw. Konfigurationen an ihnen vorzunehmen. Es muss dem Dienst-Management-System möglich sein, sich für den Empfang bestimmter Ereignisse bei den entsprechenden Ressourcen oder anderen Subdiensten anzumelden, wobei auch entsprechende Constraints definierbar sind, welche die Auslieferung der Ereignisse steuern. Vorstellbar wäre hier beispielsweise die Definition von Schwellwerten, die vorgeben, wann der Messwert einer Datenquelle dem Dienst-Management-System berichtet werden soll. Es muss festgelegt werden, wie auf solche passiv generierten Daten verschiedener Ressourcen reagiert werden soll, wann aktive Anfragen an Ressourcen gestartet werden müssen, wie die erzeugten Daten zu bewerten sind und wann schließlich Ereignisse für das Dienst-Management generiert werden sollen. Bei der Analyse zur Realisierung der Architektur wurden daher verschiedene Teilaufgaben ausgearbeitet, die mit Hilfe verschiedener Software-Komponenten in unterschiedlichen Schichte erfüllt werden. Abbildung 2.1 zeigt die wichtigsten Bestandteile und Schichten von SMONA.

Im Folgenden soll kurz auf die Kernaufgaben der einzelnen Schichten eingegangen werden.

Ressource layer and platform specific layer Die unterste Schicht enthält die Ressourcen, die überwacht bzw. konfiguriert werden sollen. Hier können die unterschiedlichsten Arten von Ressourcen angesiedelt sein. Dabei handelt es sich bei einer Log-Datei genauso um eine Ressource wie bei einem Datenbank-Management-System. Ressourcen können also ein beliebiges Objekt mit den unterschiedlichsten Attributen repräsentieren. Die in der plattformspezifischen Schicht angesiedelten Management-Werkzeuge umfassen sowohl integrierte Systeme als auch selbst angefertigte Skripten zur Lösung konfigurationsspezifischer Aufgaben.

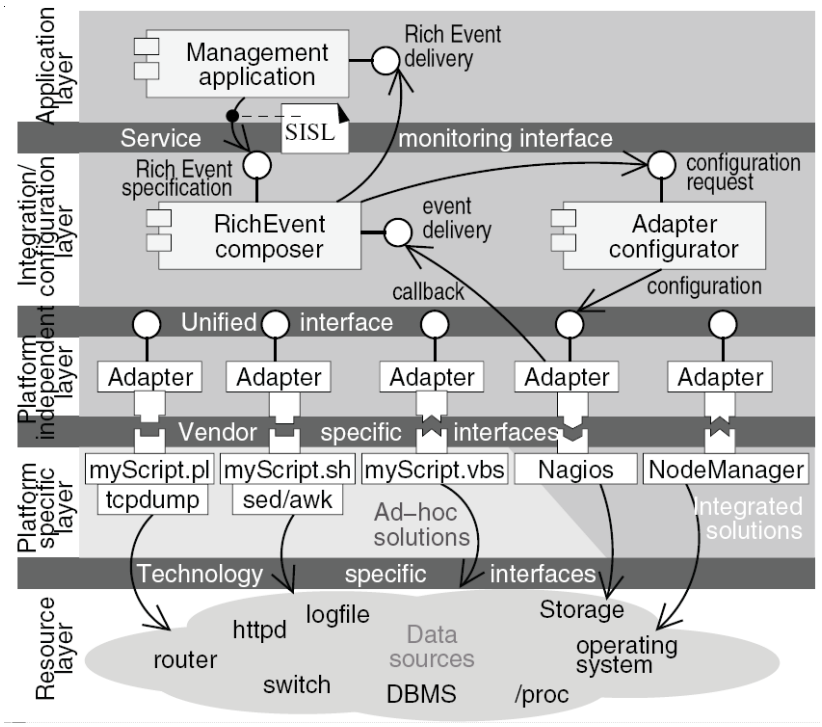

Abbildung 2.1: Bestandteile der Service-Monitoring-Architektur (Quelle: [DaSa 05])

Plattform independent layer Um eine homogene Dienstperspektive zu ermöglichen ist unerlässlich, die Vielfalt an heterogenen und plattformspezifischen Management-Werkzeugen in einer plattformunabhängig Schicht zu verbergen. Diese Schicht beherbergt die einheitlich konfigurierbaren Adapter, die sich die Werkzeuge der plattformspezifischen Schicht zunutze machen. Abhängig von der Konfiguration der Adapter, extrahieren diese die notwendigen Informationen aus den verwendeten Management-Werkzeugen, um sie in einer normalisierten Form der Integrations- und Konfigurationsschicht zur Verfügung zu stellen. Eine neue Ressource kann hinzugefügt werden, indem das verwendete Werkzeug oder Skript durch einen entsprechenden Adapter, der durch eine standardisierte Schnittstelle konfiguriert werden kann, gekapselt wird.

Integration and configuration layer Die Konfiguration der Adapter wird an zentraler Stelle von der Integrations- und Konfigurationsschicht übernommen. Hauptaugenmerk dieser Schicht liegt auf den zwei Komponenten, die sich mit der Aufgabe der Konfiguration von Adaptern sowie mit der Komposition der Daten beschäftigen. Beide greifen auf die Funktionalität der platformunabhängigen Schicht zurück. Der *Adapter Configurator* erledigt die Konfiguration der Adapter über ihre einheitliche Schnittstelle, ausgelöst durch Konfigurationsanfragen des *Rich Event Composers*. Solche Anfragen entstehen aus der Notwendigkeit heraus, bestimmte Dienstüberwachungsaufgaben durchzuführen.

Application layer Konfigurierte Adapter übermitteln ihre Daten an den *Rich Event Composer*. Dort werden die Daten ausgewertet, verfeinert, zu *Rich Events* aggregiert und schließlich der Management-Anwendung zur Verfügung gestellt. Die Architektur wurde für das Dienst-Management entworfen. Ebenso kann sie aber auch auf ein regelbasiertes Management angewendet werden, welches auf *Rich Events* basiert, sowie auf Grid-Szenarios, in welchen eine Vielzahl von heterogenen Ressourcendaten aggregiert werden müssen.

2.2 Problemstellung

Zur Realisierung von SMONA (vgl. Kapitel 2.1) werden unter anderem einheitliche Zugriffspunkte auf die vorhandenen Werkzeuge benötigt, die die Ressourceninformationen bereitstellen bzw. eine Konfiguration der

Ressourcen ermöglichen. Diese Zugriffspunkte sollen durch Adapter realisiert werden, deren Aufgabe darin besteht, die Ressourcendaten in einer normalisierten Form zur Verfügung zu stellen. Die zu definierende Schnittstelle für den Zugriff auf Adapter soll möglichst flexibel und leicht erweiterbar gestaltet werden.

Der Fokus dieser Arbeit liegt auf der Implementierung der plattformunabhängig Schicht sowie auf der Definition der Schnittstelle zur Integrations- und Konfigurationsschicht. Einen wichtigen Kerngedanken stellt dabei die Handhabung der Heterogenität der verschiedenen Datenquellen dar. Ziel ist es, die Diversität der verschiedenen Management-Werkzeuge in dieser Schicht zu verbergen und gleichzeitig eine einheitliche Schnittstelle zur Konfiguration aller Adapter zur Verfügung zu stellen. Die Daten der entsprechenden Ressource müssen normalisiert werden, da für verschiedene Plattformen auch unterschiedliche Werkzeuge und damit auch unterschiedliche Formate zum Einsatz kommen. Adapter müssen auch auf die Handhabung temporaler Aspekte ausgelegt sein. Beispielsweise muss gewährleistet sein, dass einem Adapter in seiner Konfiguration mitgeteilt werden kann, wann bzw. in welchen Intervallen er von seiner Ressource Daten abfragen, auswerten und an die Integrations- und Konfigurationsschicht liefern soll. Um flexibel auf neue Ressourcen (zum Beispiel Management-Werkzeuge) reagieren zu können, muss es möglich sein, der plattformunabhängigen Schicht auf einfache Weise neue Ressourcen hinzuzufügen, indem nur ein neuer Adapter mit der vorgegebenen Schnittstelle erzeugt wird. Die Management-Werkzeuge können sowohl zur Konfiguration von Ressourcen, als auch zur Abfrage von Informationen von Ressourcen herangezogen werden. Desweiteren können Informationen aktiv von der Integrations- und Konfigurationsschicht angefordert werden bzw. ihr in Form von Ereignissen passiv zugeführt werden. Es müssen also Kategorien von Adaptern definiert werden, die auf die jeweiligen Anforderungen ausgelegt sind.

Aufgabe dieses Fortgeschrittenenpraktikums war es, ein Adapter Framework zu entwickeln, mit welchem es möglich ist, diese Anforderung für eine möglichst große Teilmenge der zur Verfügung stehenden Ressourcen zu entwickeln.

2.3 Anforderungen

Zunächst sollen die grundlegenden Anforderungen diskutiert werden, denen Adapter genügen müssen, damit sie zur Lösung der in Abschnitt 2.2 geschilderten Aufgaben herangezogen werden können. Zusätzlich werden weitere Anforderungen diskutiert, die unabhängig von der in Abschnitt 2.1 vorgestellten Architektur, sinnvoll erscheinen.

2.3.1 Normalisierter Zugriff

Adapter sollen die Möglichkeit bieten, auf eine einheitliche Art und Weise Informationen bereitzustellen bzw. Konfigurationen vorzunehmen, die derzeit mit Hilfe der unterschiedlichsten Netz-Management-Werkzeuge erledigt werden. Ein wichtiges Kriterium bei der Entwicklung von Adaptern stellt dabei die Anforderung der normalisierten Zugriffsmöglichkeit auf die entsprechenden Werkzeuge dar. Beispielsweise müssen Adapter so angesprochen werden können, dass diese (unabhängig von dem verwendeten Management-Werkzeug) stets dieselben Informationen liefern bzw. dieselbe Auswirkung bei der Konfiguration einer Ressource erzielen. Selbst angefertigte Skripte sollen von Adapter genauso als Ressource verwendet werden können wie die Ausgabe mächtiger Management-Werkzeuge wie *IBM Tivoli NetView* [IBM] oder *Nagios* [Nag]. Wünschenswert wäre auch ein normalisierter Zugriff nicht nur über Management-Werkzeuge, sondern auch über Systemplattformen hinweg. Dabei gilt es zu beachten, dass für Systemelemente unterschiedlicher Hersteller auch verschiedene Konfigurations- und Monitoring-Werkzeuge zum Einsatz kommen können. Systeminformationen werden in der Regel auf den diversen Plattformen unterschiedlich gespeichert und an verschiedenen Stellen verwaltet. Natürlich kann dieser Wunsch nur in seltenen Fällen befriedigt werden, da der Schnitt der Attribute gleichartiger Ressourcen verschiedener Hersteller oft nur ein unzureichendes Bild der Ressourcen im Einzelnen wiederspiegelt. So besitzen zwei Switches von verschiedenen Herstellern unter Umständen ähnliche Management-Funktionen. Darüberhinaus hat jedes Element aber seine herstellerabhängigen Zusatzfunktionen, die durch die Integrations- und Konfigurationsschicht natürlich auch als Attribute zur Verfügung gestellt werden sollen, für die eine Normalisierung aufgrund ihrer Diversität jedoch nicht möglich ist.

2.3.2 Programmiersprachenunabhängige Schnittstelle

Verschiedene Aufgaben können mit unterschiedlichen Programmiersprachen besser oder schlechter gelöst werden. Beispielsweise kann der Zugriff auf Ressourcen wie Log-Dateien um einiges leichter in Perl als in Java implementiert werden. Um die Möglichkeit zu wahren, unabhängig von der verwendeten Programmiersprache zur Implementierung der einzelnen Adapter einen einheitlichen Zugriff auf diese zu gewährleisten, muss zur Kommunikation zwischen Adaptern und der Integrations- und Konfigurationsschicht eine programmiersprachen- und plattformunabhängige Schnittstelle zur Verfügung gestellt werden.

2.3.3 Asynchronität der Kommunikation

Es liegt in der Natur einiger Netz- und Systemelemente, dass die Abfrage einer bestimmten Statusinformation eine gewisse Zeitspanne in Anspruch nehmen kann. Beispielsweise könnten Temperaturmesswerte an den Prozessoren eines Mehrprozessorsystems erfragt werden. Alleine die Durchführung dieser Messungen könnte schon mehrere Sekunden in Anspruch nehmen. Auch Programmaufrufe, welche darauf abzielen, Konfigurationen an bestimmten Komponenten vorzunehmen, könnten unter Umständen viel Zeit benötigen, bevor das aufgerufene Programm letztendlich mit einer Fehlermeldung abbricht. Ein Beispiel wäre der Aufruf eines Skripts, welches zur Durchführung seiner Aufgaben eine SSH-Verbindung einrichten muss. Diese Verbindung kann aber vielleicht nicht hergestellt werden. Bevor das SSH-Programm diesen Fehler meldet, können jedoch schon etliche Sekunden vergangen sein. Verhält sich ein Programm auf ähnliche Weise, führt das dazu, dass auch ein Adapter der die Rückgabewerte dieses Programms der Integrations- und Konfigurationsschicht zur Verfügung stellen soll, während dieser Zeit für weitere Anfragen blockiert ist. Es ist daher notwendig, Adapter so zu modellieren, dass sie nur asynchrone Aufrufe auf derartige Ressourcen bzw. Konfigurationswerkzeuge zulassen. Zusätzlich muss gewährleistet sein, dass Programmaufrufe nach einer gewissen Zeitspanne auch abgebrochen werden können und die Integrations- und Konfigurationsschicht über das Fehlverhalten informiert wird.

2.3.4 Autorisierung und Authentifizierung

Adapter müssen die Fähigkeit besitzen, die Integrations- und Konfigurationsschicht zu authentifizieren und nur autorisierten Entitäten Zugriff zu gewährleisten. Andernfalls sind Adapter *Denial of Service* Attacken und unbefugten Zugriffen ausgesetzt. Es könnte beispielsweise der Fall eintreten, dass ein Adapter von einer anderen Instanz der Integrations- und Konfigurationsschicht mit Informationsanfragen bombardiert wird, wodurch der Adapter unter Umständen lahmgelegt wird. Außerdem müssen Management-Aktionen an Ressourcen autorisiert werden, um nur befugten Instanzen die Möglichkeit zu bieten, Veränderungen an Netz- bzw. Systemelementen vorzunehmen.

2.3.5 Selbstbeschreibung der Adapterfähigkeiten

Eine wichtige Eigenschaft, um Adapter effektiv einzusetzen, beruht auf ihrer Fähigkeit, sich selbst beschreiben zu können. Für die Integrations und Konfigurationsschicht würde es zu viel Aufwand bedeuten, eine Datenbasis zu pflegen, die immer den aktuellsten Wissenstand über alle vorhandenen Adapter reflektiert. Wünschenswert wäre eine Beschreibung, die nicht nur die Aufrufssyntax für die einzelnen Funktionen eines Adapters liefert, sondern auch die Semantik ihrer Parameter wiederspiegelt. Dieses Ziel ist sehr ehrgeizig und in einer heterogenen Welt wie der des Netz- und System-Managements schwer zu realisieren.

2.3.6 Verwaltung von Historien

Viele Messwerte, die der ein oder andere Adapter liefert, unterliegen mehr oder weniger großen Schwankungen. Beispielsweise kann die gemessene Bandbreite einer TCP-Verbindung von einer auf die andere Sekunde stark abfallen und gleich darauf wieder den vorangegangenen Wert annehmen. Wird nun eine Messung zu genau dem Zeitpunkt genommen, an welchem der starke Einbruch erfolgt, führt dies natürlich zu einem falschen

Ergebnis in Bezug auf die über einen längeren Zeitraum gemessene Durchschittsbandbreite. Damit Adapter aufgrund solcher Schwankungen nicht fälschlicherweise ein Ereignis in der Integrations- und Konfigurationsschicht auslösen, sollten sie so konfigurierbar sein, dass es möglich ist, Historien mit in die Berechnung des Messwertes einfließen zu lassen. Für das TCP-Beispiel wäre denkbar, das arithmetische Mittel über die letzten X Messungen als Rückgabewert zu liefern.

2.3.7 Zusammenfassung

In den vorangegangenen Abschnitten wurden einige wichtige Anforderungen an Adapter erläutert. Diese Liste beinhaltet sicherlich nicht alle Eigenschaften, die Adapter besitzen sollten, um ihre Aufgaben vollständig und zufriedenstellend zu bewältigen. Die Geschilderten dienen jedoch als gute Richtlinie für die Konzeption von Adaptern. Vielen der Anforderungen werden die in dieser Arbeit implementierten Adapter gerecht. Leider mussten jedoch zwei Anforderungen in den Hintergrund gestellt werden. Zur Realisierung eines ersten Prototyps wurde hier weniger Wert auf Sicherheitsaspekte gelegt. Ebenso wurden weder Syntax noch Semantik für eine Selbstbeschreibung von Adaptern spezifiziert. Dies lag vor allem daran, dass es derzeit noch keine vollständige Implementierung der Integrations- und Konfigurationsschicht vorliegt und deren Anforderungen an eine Selbstbeschreibung von Adaptern noch nicht definiert wurde. In der Schnittstelle wurde eine zukünftige Erweiterung um die Fähigkeit der Selbstbeschreibung berücksichtigt.

Kapitel 3

Konzeption der Adapter

3.1 Kategorien von Adaptern

Ausgehend von der Problemstellung (vgl. Kapitel 2.2) und den Anforderungen an Adapter (vgl. Kapitel 2.3) lassen sich verschiedene Anwendungsfälle für Adapter ableiten (vgl. Abbildung 3.1).

Unabhängig von der Aufgabe, die ein Adapter erfüllen soll, ist es stets notwendig, diesen zuvor mit den entsprechenden Parametern zu konfigurieren. Beispielsweise muss ein Adapter, dessen Aufgabe es ist, eine bestimmte Ressource zu überwachen und gegebenenfalls ein Ereignis in der Integrations- und Konfigurationsschicht auszulösen, über das nötige Wissen verfügen, in welchen Intervallen er die Ressource überprüfen soll. Andere Adapter benötigen unter Umständen die Vorgabe eines Schwellwertes, anhand dessen sie entscheiden können, ob die Notwendigkeit besteht, eine Änderung an der Konfiguration einer Ressource vorzunehmen.

In den Abschnitten 2.2 und 2.3 wurde bereits auf die unterschiedlichen Einsatzgebiete von Adaptern eingegangen. Prinzipiell lassen sich daraus drei grundlegende Anwendungsfälle für Adapter ableiten.

Push Adapter

Es existiert eine Vielzahl an Netz- und Systemelementen, die permanent überwacht und kontrolliert werden müssen. Ein typisches Beispiel wäre ein Webserver, dessen gegenwärtige Auslastung durch Anfragen nach Webseiten oder anderen Inhalten überprüft werden muss. Es gibt eine Menge von Ressourcendaten, wie beispielsweise die Prozessorauslastung des Rechners, auf dem der Webserver ausgeführt wird, oder den ausgehenden Datenverkehr auf seinen Netzwerkschnittstellen, die von einem Adapter kontrolliert werden könnten. Wird dabei ein kritischer Wert beobachtet, liegt es im Ermessen des zuständigen Adapters zu entscheiden, ob die Integrations- und Konfigurationsschicht über diesen Wert informiert werden soll, oder nicht. Adapter dieser Gruppe werden im Folgenden als *Push Adapter* referenziert. Die Konfiguration dieses Adaptertyps tendiert

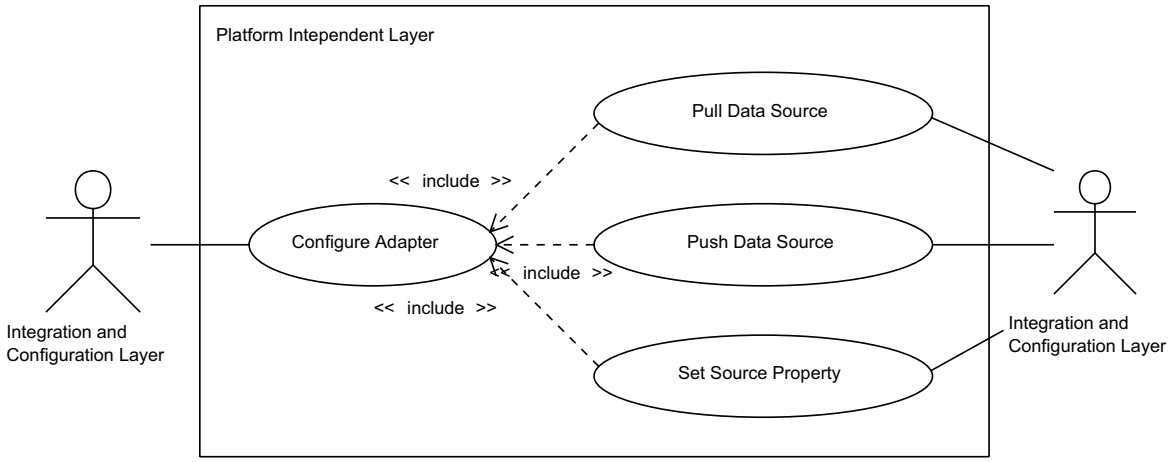

Abbildung 3.1: Anwendungsfälle zum Einsatz von Adaptern

meist zur Definition von oberen und/oder unteren Schranken für den Messwert einer Ressource. Es kann sich aber auch lediglich um die Festlegung eines Zeitintervalls handeln, der dem Adapter vorgibt, wie häufig er Statusinformationen über die entsprechende Ressource liefern soll.

Pull Adapter

Es liegt in der Natur von Netzen und den dazugehörigen Systemelementen, dass es zu Störungen oder sogar zu Unterbrechungen von Verbindungen kommen kann. Allein diese Eigenschaft fordert – zusätzlich zu der passiven Informationsvermittlung von Push Adaptern – die aktive Abfrage von Ressourcen. Es exisitert eine Vielzahl von Systemelementen, die hin und wieder angestoßen werden müssen, um ihre unbeeinträchtigte Verfügbarkeit zu ermitteln. Beispielsweise könnte der Fall eintreten, dass ein Webserver, der über einen Switch mit einem Router verbunden ist und durch einen Push Adapter überwacht wird, keine Informationen über den Zustand seiner Auslastung liefert. Um die Ursache für den Verlust dieser Daten zu ermitteln, kann es notwendig sein, alle an der Diensterbringung beteiligten Hard- und Software-Komponenten aktiv zu überprüfen. In diesem Beispiel könnte ein defekter Switch die Ursache darstellen. Handelt es sich um ein managebares Element, könnte der Defekt durch die Anfrage an einen Adapter, dem diese Ressource zugeordnet wurde, leicht festgestellt werden. Adapter die eine Schnittstelle zur aktiven Abfrage von Ressourcen zur Verfügung stellen, werden fortan als *Pull Adapter* bezeichnet.

Set Adapter

Bisher wurden nur Anwendungsfälle betrachtet, die die Überwachung von Ressourcen und das Aufdecken von Unregelmäßigkeiten im Fehler-Fanagement im Sinn hatten. Die wesentliche Aufgabe des Fehler-Management ist es aber, auf derartige Unregelmäßigkeiten zu reagieren. In der Regel müssen die fehlerhaften Netz- oder Systemelemente umkonfiguriert oder sogar ausgetauscht werden, um eine zufriedenstellende Performance des betroffenen Dienstes wiederherzustellen. An zentraler Stelle soll es möglich sein, Konfigurationen vornehmen zu können, um die gewünschte Dienstsicht zu erzielen. Zur Unterstützung dieses Zieles kommen auf der plattformunabhängigen Schicht *Set Adapter* zum Einsatz, die einen direkten Eingriff in die Konfiguration der zu einem Dienst zugehörigen Elemente vornehmen können. Beispielsweise könnten Adapter definiert werden, die Änderungen an der Netzwerkkonfiguration vornehmen oder sich um die Verwaltung von Ressourcen kümmern, die ein Dienst zur Verfügung stellt.

3.2 Definition von Adaptern und Adapter Sourcen

Es ist offensichtlich, dass es eine ungeheure Vielfalt an Ressourcen gibt, die von Adaptern verwaltet und überprüft werden müssen. Es ist undenkbar für jede dieser Ressourcen einen eigenen Adapter zur Verfügung zu stellen. Daher muss ein Design gefunden werde, welches feingranular genug ist, die vorhandenen Ressourcen der Integrations- und Konfigurationsschicht transparent zur Verfügung zu stellen. Gleichzeitig soll dieses Design skalieren.

Der Ansatz der in dieser Arbeit verfolgt wurde, um diese Transparenz zu erwirken und gleichzeitig ein Adapter Framework zu erarbeiten, das skaliert, ist die Unterscheidung zwischen *Adaptern* und *Adapter Sourcen*.

Adapter Adapter sollen als die Komponenten verstanden werden, die der Integrations- und Konfigurationsschicht als Zugriffspunkte auf die Netz- und Systemressourcen zur Verfügung gestellt werden. Die Integrations- und Konfigurationsschicht ist sich also nur über die Existenz der Schnittstelle zu Adaptern bewusst.

Adapter Sourcen Adapter Sourcen dienen dazu, intern den Zugriff auf Ressourcen zu bewerkstelligen. Ressourcen lassen sich dahingehend kategorisieren, dass ein Zugriff auf einen bestimmten Datentyp vorausgesetzt wird. Beispielsweise liefert eine CPU ihre Auslastung als Datentyp Float, wohingegen ein Netzwerkinterface die Anzahl aller über die Schnittstelle eingehender Pakete als Integerwert speichert. Es liegt also nahe, Ressourcen, die denselben Datentyp liefern, über ein datentypspezifisches Interface

anzusprechen – Adapter Sourcen. Je nach Adaptertyp werden adapterspezifische Typen von Adapter Sourcen verwaltet.

Die Zusammenhänge sind in Abbildung 3.2 veranschaulicht.

Es wurde versucht, die Gemeinsamkeiten von Ressourcen in jeweils einer Adapterinstanz zu realisieren und die vorhandenen Unterschiede in Adapter Sourcen zu unterteilen. Die Aufgabe der Adapterinstanz liegt darin, einen einheitlichen Zugangspunkt zu den von ihr verwalteten Adapter Sourcen zur Verfügung zu stellen. Dabei werden Adapter in die zuvor erwähnten Kategorien Pull, Push und Set Adapter unterteilt (vgl. Kapitel 3.1). Adapter Sourcen spezialisieren sich darauf, die Ressourcen nach ihren Typen zu unterscheiden. Dies beginnt bei der einfachen Unterteilung von Ressourcen nach denen von ihnen gelieferten Datentypen (Float, Integer, String, ...) und endet bei der Aufteilung in die Art ihrer Konfigurationseingriffe in die Netz- und Systemelemente (Änderung von Flags, Hinzuführen und Entfernen von Ressourcen, ...), die einen Dienst konstituieren.

Der Kerngedanke bei dieser Aufteilung liegt in der Wiederverwendbarkeit einer Adapterinstanz zum Zugriff auf viele verschiedene Ressourcen. Dabei soll es möglich sein, auf einfache Art und Weise die Fähigkeiten eines Adapters zu erweitern. Die Idee ist es, durch einfache Konfigurationsvorgaben neue Adapter Sourcen zu erzeugen und diese der Integrations- und Konfigurationsschicht als vollwertige Adapter zu präsentieren. Die Struktur der plattformunabhängig Schicht, die sich aus diesem Konzept ergibt, ist in Abbildung 3.3 für das Beispiel eines Pull Adapters skizziert.

Hier verwaltet ein `PullAdapter` die Adapter Sourcen vom Typ `PullIntegerSource` und `PullFloatSource`. Objekte vom Typ `PullIntegerSource` verwalten Datenquellen (hier Skripte und Programme), die Daten vom Typ Integer liefern. Objekte vom Typ `PullFloatSource` verwalten Ressourcen, die Daten vom Typ Float liefern. Auf die Ressourcen wird zum Beispiel über Aufrufe von Skripten zugegriffen. Diese Aufrufe werden als externe Programmaufrufe realisiert. Obwohl die Klasse `PullAdapter` alleine die Schnittstelle zur Integrations- und Konfigurationsschicht veröffentlicht, erscheinen alle in der plattformspezifischen Schicht vorhandenen Ressourcen der Integrations- und Konfigurationsschicht als vollwertige Adapter. Die Details zur Realisierung dieses Konzepts folgen in Abschnitt 4.2.

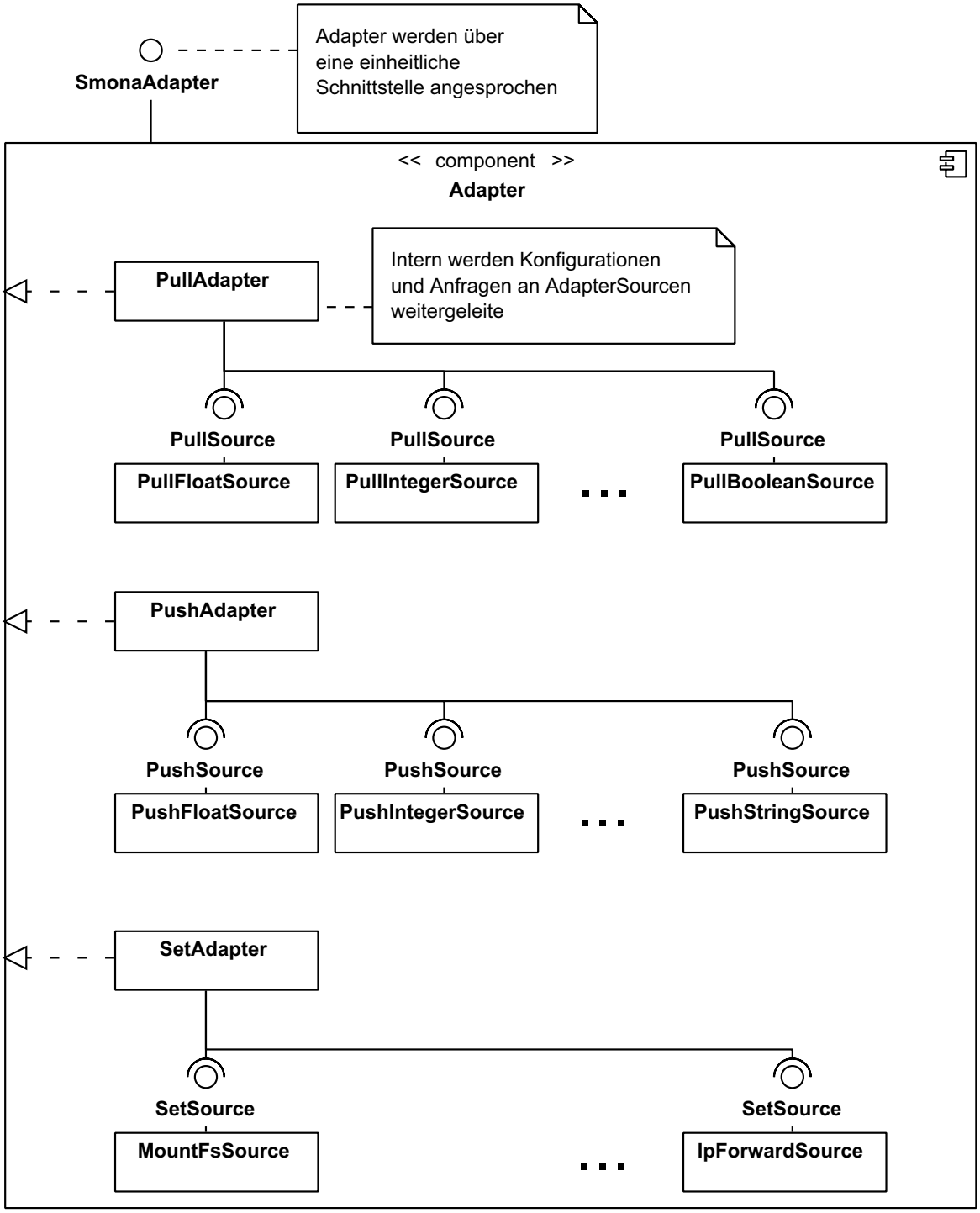

Abbildung 3.2: Adapter und Adapter Sourcen

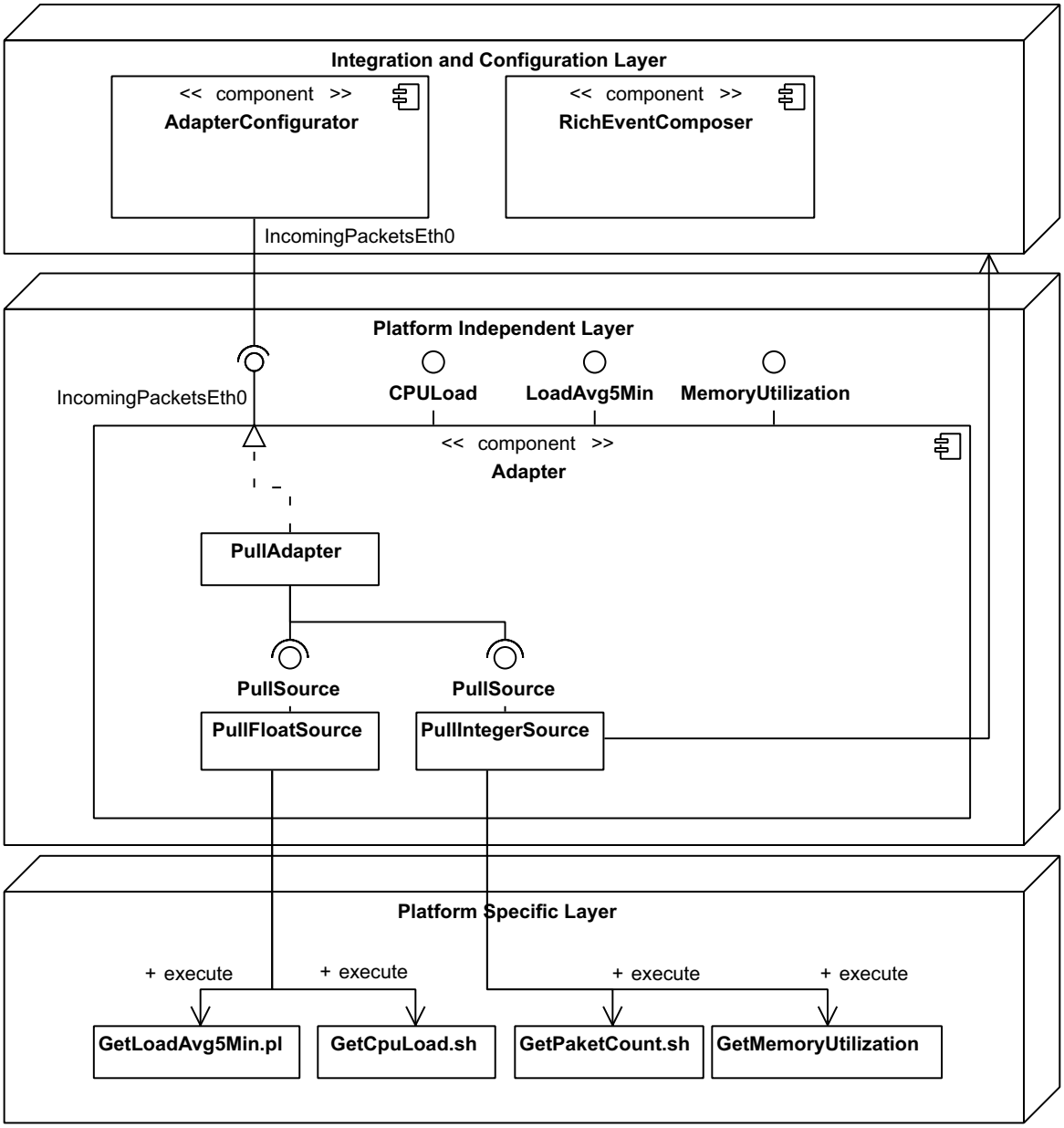

Abbildung 3.3: Visualisierung der Umsetzung der Konzeption von Adaptern

Kapitel 4

Design und Entwurf der Adapter

Nachdem das Konzept für die Realisierung von Adaptern behandelt wurde, gilt es, ein geeignetes Design zu finden, um das Konzept in einen Software-Entwurf umzusetzen. Zunächst wird in Kapitel 4.1 die Spezifikation der Schnittstelle zwischen Integrations- und Konfigurationsschicht und der plattformunabhängigen Schicht erläutert, in der die Adapter angesiedelt sind. Abschnitt 4.2 behandelt dann die Kernkomponenten, die zur Realisierung der Adapter als Software-Komponenten zu implementieren sind.

4.1 Design der Adapter Schnittstelle

Wie in Abschnitt 2.3 erläutert, soll die Schnittstelle zwischen Integrations- und Konfigurationsschicht und der plattformunabhängigen Schicht so gestaltet sein, dass sie unabhängig von verwendeten Programmiersprachen ist, die zur Implementierung der Komponenten in den Schichten verwendet werden. Sie soll einen normalisierten Zugriff auf Adapter Sourcen ermöglichen und die Möglichkeit bieten, asynchrone Aufrufe auf Adapter auszuführen. Im Folgenden soll die Wahl von CORBA als Middleware zur Definition der Schnittstelle diskutiert werden. Danach wird die Verwendung der von CORBA bereitgestellten Dienste zur Realisierung der Schnittstelle erläutert.

4.1.1 Motivation für CORBA als Middleware

In Kapitel 2.3.2 wurde bereits angesprochen, dass eine programmiersprachenunabhängige Schnittstelle gefunden werden muss, um die Flexibilität in der Auswahl der zur Problemlösung geeigneten Programmiersprache zu erhalten. Zur Implementierung der Schnittstelle zwischen der plattformunabhängig Schicht und der Integrations- und Konfigurationsschicht wurde die *Common Object Request Broker Architecture (CORBA)* gewählt. CORBA verfolgt die von Hardware, Betriebssystemen und Programmiersprachen unabhängige Zusammenarbeit von Objekten bzw. Anwendungen über Rechnergrenzen hinweg [LP 98]. Der Standard wird von der *Object Management Group (OMG)* verwaltet. Die technischen Implementierungen des Standards CORBA bezeichnet man als *Object Request Brokers (ORBs)*. Für das hier gewählte Design werden ORBs dazu verwendet, Aufrufe der Integrations- und Konfigurationsschicht an die Adapter weiterzuleiten bzw. Rückmeldungen der Adapter zurückzuführen.

Für die Verwendung von CORBA sprechen – neben der potentiellen Verwendbarkeit jeglicher Programmiersprache zur Implementierung der verschiedenen Elemente einer Schicht – eine Vielzahl anderer Argumente [HAN 99]. Die Spezifikation von den kommunizierenden Instanzen erfolgt objektorientiert. Clients und Server werden als Objekte modelliert und es können alle Vorteile der objektorientierten Entwicklung, wie zum Beispiel Vererbung und Polymorphismus, bei der Definition von CORBA-Objekten ausgenutzt werden. Von dieser Eigenschaft wird beispielsweise beim Entwurf der verschiedenen Adaptertypen Set, Push und Pull Gebrauch gemacht (vgl. Abschnitt 3.1 und 5.1). Die Einführung eines Brokers flexibilisiert das herkömmliche Client-Server-Modell. Er trennt die Schnittstelle eines Server-Objekts von seiner Implementierung. Der ORB ist in der Regel verteilt implementiert, was jedoch vor den Client- und Server-Objekten verborgen bleibt. Anders als beim traditionellen Client-Server-Modell sind hier zwischen Clients und Server nicht nur m:1 sondern auch 1:m oder sogar n:m Beziehungen möglich. Clients müssen nicht wie beim ursprünglichen *Remote Procedure Call (RPC)* explizit die Adresse eines Servers angeben, um Aufrufe auf dessen Objekte durchzuführen. Die Kommunikation zwischen Client- und Server-Objekten erfolgt vollkommen symmetrisch. Damit ist es

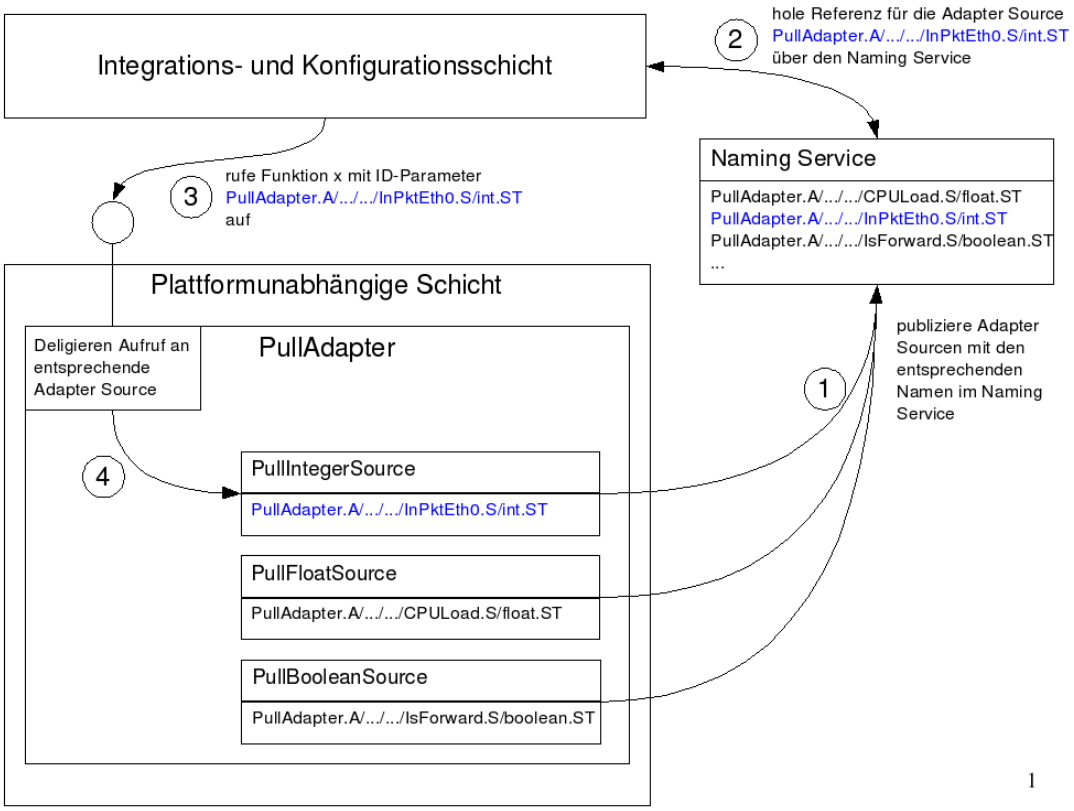

Abbildung 4.1: Schematischer Kommunikationsablauf zwischen Integrations- und Konfigurationsschicht und Plattform-unabhängiger Schicht

möglich, auch eine asynchrone Kommunikation zwischen CORBA-Objekten auf einfache Art und Weise zu realisieren. Der ORB kümmert sich um die Weiterleitung von Methodenaufrufen auf entfernte Objekte. Dabei ist es vollkommen irrelevant, wo solche Objekte lokalisiert sind. Aufrufe auf Server-Objekte geschehen vollkommen transparent, das heißt für Client-Objekte unterscheiden sich Methodenaufrufe auf Server-Objekte nicht von Aufrufen auf lokale Instanzen. CORBA wird außerdem von vielen Management-Plattformen unterstützt. Es existieren auch Schnittstellen für die Verwendung gängiger Protokolle wie SNMP und damit die Möglichkeit, auf die Implementierungen bestehender Informationsmodelle zuzugreifen (Beispiel MIBs).

4.1.2 Publizieren von Adapter Sourcen

Da Adapter den Zugriff auf Ressourcen intern als Adapter Sourcen kapseln, stellt sich die Frage, wie die Integrations- und Konfigurationsschicht Zugriff auf die einzelnen Ressourcen erlangt. Es steht ja nur eine Schnittstelle zur Verfügung, über die dennoch alle Adapter Sourcen angesprochen werden sollen. Damit die Integrations- und Konfigurationsschicht Adapter bzw. die von ihnen verwalteten Adapter Sourcen auffinden kann, müssen diese publiziert werden. Dafür wird der CORBA-Naming-Service verwendet. Abbildung 4.1 veranschaulicht schematisch den Ablauf der Kommunikation zwischen Integrations- und Konfigurationsschicht sowie der plattformunabhängigen Schicht. Für jede Adapter Source, die von einem Adapter verwaltet wird, publiziert dieser seine Schnittstelle unter einem anderem Namen, der die Fähigkeit der Adapter Source beschreibt (1). Die Integrations- und Konfigurationsschicht kann sich über den CORBA-Naming-Service Referenzen auf die Adapter Sourcen verschaffen (2). Dabei erscheinen die Adapter Sourcen der Integrations- und Konfigurationsschicht wie vollwertige Adapter. Damit ein Adapter feststellen kann, welche seiner Adapter Sourcen gerade angesprochen wird, müssen die Zugriffe auf die Schnittstelle über parametrisierte Methodenaufrufe geschehen (3). Anhand des Parameters identifiziert der Adapter dann die zu verwendende Adapter Source (4).

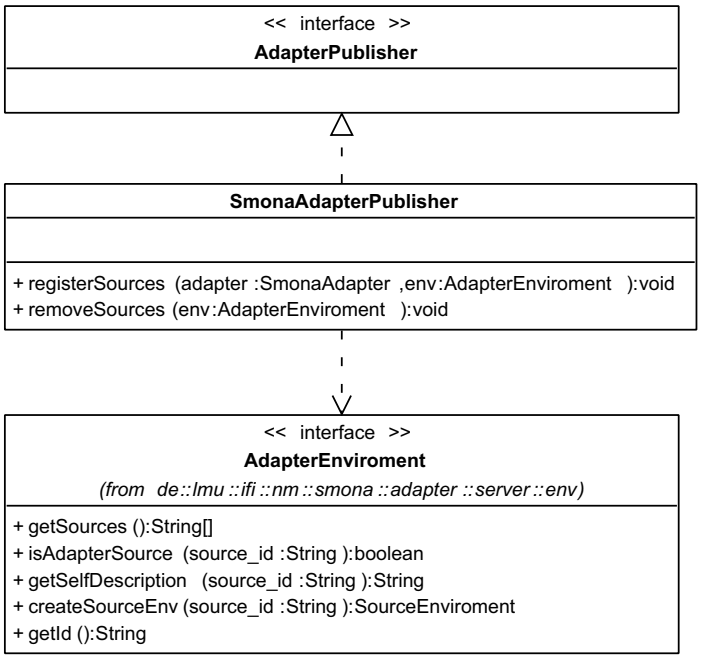

Abbildung 4.2: Involvierte Klassen beim Publizieren von Adapter Sourcen

Derzeit ist noch kein eindeutiges Naming-Schema definiert worden, das zur Publikation der Adapter Sourcen verwendet werden soll. Die verwendeten Beispielnamen sollen nur verdeutlichen, dass es durchaus Sinn macht, einem Namen eine gewisse Semantik zu verleihen, obgleich Adapter Sourcen in Zukunft die Möglichkeit zur Verfügung stellen sollen, auf Anfrage eine Selbstbeschreibung zu liefern (vgl. Kapitel 2.3.5).

Das Publizieren der Adapter Sourcen übernimmt die Klasse `SmonaAdapterPublisher`, die das Interface `AdapterPublisher` aus dem Paket `de.lmu.ifi.nm.smona.adapter.server` implementiert (vgl. Kapitel B). In Abbildung 4.2 sind die Klassen und ihre Methoden aufgelistet, die zur Publikation der einzelnen Adapter Sourcen verwendet werden (vgl. Kapitel 5.1).

Die Methode `registerSources()` der Klasse `SmonaAdapterPublisher` übernimmt das Publizieren aller Adapter Sourcen, die aus dem Parameter vom Typ `AdapterEnviroment` ausgelesen werden können. Instanzen dieser Klasse stellen eine Umgebung zur Verfügung, die alle Arten von Adapter Sourcen beinhaltet, die von einem Adapter des Type `SmonaAdapter` verwaltet werden. Diese Umgebung wird beim Starten jedes Adapters aus einer Konfigurationsdatei eingelesen. Die Methoden der Klasse `AdapterEnviroment` bieten Zugriffsmöglichkeiten auf die verschiedenen Elemente der Umgebung. Beispielsweise kann mit der Methode `getSources()` eine Liste aller verfügbaren Adapter Sourcen angefordert werden. Diese Liste beihaltet die Namen aller Adapter Sourcen, unter denen sie im CORBA Naming Service publiziert wurden. Die Methode `createSourceEnviroment()` dient dazu, für eine Adapter Source dynamisch eine Umgebung bereitzustellen, die die Konfigurationsparameter für diese Adapter Source enthält.

4.1.3 Zugriff auf Ressourcen, Adapter Sourcen und Adapter

Der Schlüssel zur Identifikation der Ressourcen liegt im Design der CORBA-Interfaces. Die Idee ist es, einem Adapter nur wenige Methoden zur Konfiguration seiner Adapter Sourcen oder zur Abfrage bestimmter Eigenschaften zu verleihen. Diese Methoden werden parametrisiert aufgerufen. Der entsprechende Parameter dient als Identifikator der Adapter Source, auf die die entsprechende Methode angewandt werden soll (vgl. Kapitel 4.1.2). Das Interface `SmonaAdapter` [1], welches alle Adapter implementieren müssen, gibt einige Methoden vor, die allen Adaptern gemein sind. Abbildung 4.3 veranschaulicht den Aufruf der `start()`-Methode eines

[1] Dieses Interface ist in der Datei `SmonaAdapter.idl` definiert (vgl. Anhang C.3).

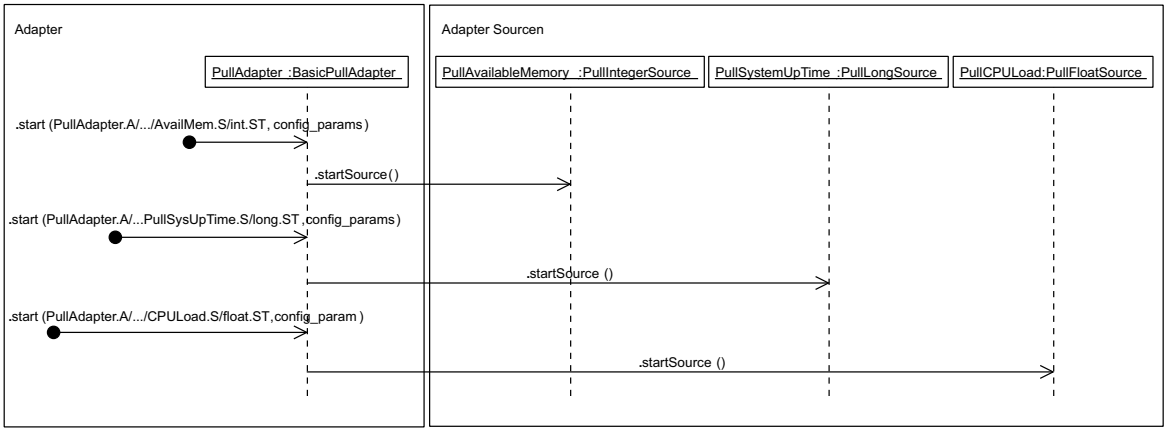

Abbildung 4.3: Beispielaufruf der `start()`-Methode eins Pull Adapters

Pull Adapters, die abhängig von ihrem Identifikationsparameter unterschiedliche Adapter Sourcen konfiguriert.

Alle Einzelheiten der Parameter und Klassen sollen hier nicht besprochen werden. Für eine detaillierte Diskussion wird auf Kapitel 4.2 verwiesen. Die Abbildung verdeutlicht, wie auf eine Instanz vom Typ `PullAdapter` die Methode `start()` aufgerufen wird. Der erste Parameter in dieser Methode gibt den eindeutigen Identifikator an, der es dem Adapter ermöglicht, den Aufruf an die entsprechende Adapter Source zu delegieren. Dieser Ansatz bietet folgende Vorteile:

Dynamische Instantiierung von Adapter Sourcen Da Ressourcen vor ihrer Benutzung stets konfiguriert werden müssen, kann der Adapter erst zur Laufzeit entscheiden, welche Adapter Sourcen instantiiert werden müssen. Es werden also nur Adapter Sourcen erzeugt, die während des Betriebs auch wirklich verwendet werden. CORBA bietet mit seinen konfigurierbaren *Portable Object Adaptern (POA)* eine ähnliche Möglichkeit, Instanzen dynamisch zu erzeugen. Beispielsweise könnte man eine eigene POA implementieren, die einen `ServantLocator` installiert, der die dynamische Erzeugung von Adaptern übernimmt (vgl. [AKS 05], 263 ff.). Bei Adapter Sourcen handelt es sich in der Regel um Objekte, die über einen langen Zeitraum existieren, jedoch nur einen Bruchteil dieser Zeit wirklich aktiv sind. Der Overhead, der sich durch die Methoden `postinvoke()` und `preinvoke()` für jeden einzelnen Aufruf einer Methode auf einen Adapter dabei ergibt, steht nicht im Verhältnis zu den Operationen, die eine Adapter Source tatsächlich auszuführen hat, weswegen der gewählte Ansatz verfolgt wurde.

Erleichterte Implementierung der asynchronen Kommunikation Auf den ersten Blick nicht ersichtlich ist zudem, dass mit den Möglichkeiten der Konfiguration einer eigenen POA und der damit verbundenen Installation eines `ServantLocator` es unter Umständen zu Problemen bei der Implementierung der geforderte Asynchronität der Kommunikation kommen kann (vgl. Kapitel 2.3.3). Die Methode `postinvoke()` würde aufgerufen, während die entsprechende Adapterinstanz noch damit beschäftigt ist, die Ressource auszulesen bzw. zu konfigurieren. Ein zu speicherndes `Cookie` könnte eventuell noch Informationen benötigen, die erst nach Beendigung der Interaktion mit der entsprechenden Ressource zur Verfügung steht.

Schnelles Hinzufügen von neuen Ressourcen Da ein Adapter den Zugriff auf mehrere Ressourcen verwaltet, die sich in ihrer Funktionalität ähnliche sind, ist es relativ schnell möglich, eine neue Ressource durch einen Adapter zur Verfügung zu stellen. Anders als bei dem Ansatz, für jede Ressource einen eigenen Adapter zu implementieren, kann hier eine neue Ressource einfach dadurch hinzugefügt werden, indem eine vorgegebene Konfiguration angepasst wird (vgl. Kapitel 5.5.1) oder eine vorgegebene Schnittstelle für Adapter Sourcen implementiert wird (vgl. Kapitel 5.1 und 5.5.2). Die dadurch zur Laufzeit reduzierte Menge an Adapterinstanzen trägt dazu bei, die verwendeten Speicherressourcen auf dem jeweiligen Systemelement klein zu halten.

4.1.4 Diskussion weiterer CORBA Dienste

Als Implementierung des des ORB des CORBA-Standards für Java wurde *JacORB* gewählt [Bros 06].

Verwendung des CORBA-Naming-Services CORBA bringt eine Vielzahl an definierten Diensten mit sich. Derzeit wird für das gewählte Design nur der CORBA-Naming-Service verwendet (vgl. Kapitel 4.1.2). Er dient auf Adapterseite dazu, eine Adapter Sourcen unter einem eindeutigen Namen zu publizieren. Auf Seite der Integrations- und Konfigurationsschicht wird er benutzt, diese Namen auf die Objektreferenzen der verschiedenen Adapter Sourcen aufzulösen.

Diskussion des CORBA-Event-Services In Kapitel 2.3.3 wurde bereits auf die Notwendigkeit der asynchronen Kommunikation mit den Adaptern hingewiesen. Obwohl CORBA mit seinem Event-Service einen Dienst bereitstellt, die asynchrone Kommunikation zu unterstützen, wurde in dieser Implementierung kein Gebrauch von dieser Möglichkeit gemacht. Der Grund für diese Entscheidung liegt darin, dass die OMG die Definition zur Abbildung der Konstrukte der *Interface Definition Language* [OMG 04] auf Konstrukte der Programmiersprache Java [OMG 02] derzeit immer noch nicht vollständig standardisiert hat (vgl. [Bros 06], Kapitel 14). Ein weiterer Grund liegt in der flexiblen Vorgabe von Intervallen, die eingehalten werden müssen, bevor die Integrations- und Konfigurationsschicht über einen misslungenen Aufruf einer Adapterfunktion benachrichtigt wird. Der Event-Service bietet die Möglichkeit, Regeln für ORBs, Threads und sogar einzelne Objekte zu definieren, wie zum Beispiel eine maximale Zeitspanne zur Ausführung asynchroner Aufrufe (vgl. [Bros 06], Kapitel 15). Diese Funktionalität ist für das hier gewählte Design jedoch nicht feingranular genug, da nur Adapter als Objekte durch den ORB erreichbar sind, ein Adapter jedoch eine Vielzahl an Adapter Sourcen verwalten soll. Diese haben in der Regel unterschiedliche Vorgaben für oben beschriebene Intervalle. Diese Unterteilung wird in Kapitel 3.2 erläutert.

Diskussion des Dynamic Invocation Interface CORBA bietet eine Vielzahl von weiteren Diensten, um die Entwicklung verteilter objektorientierter Anwendungen zu erleichtern. Unter anderem können Komponenten wie das *Dynamic Skeleton Interface (DSI)* oder das *Dynamic Invocation Interface (DII)* mit dem dazugehörigen *Interface Repository (IR)* großartige Unterstützung liefern. Im Zusammenhang mit der Forderung nach einer Selbstbeschreibung der Adapter (vgl. Kapitel 2.3.5) könnte das IR eine gute Basis liefern, um die Signaturen von Methoden und damit ihrer Aufrufsyntax zur Verfügung zu stellen. Unter Verwendung des DII könnte die Integrations- und Konfigurationsschicht mit Hilfe von *Reflection* [2] dynamische Zugriffe auf Adapter realisieren, ohne vorher die sonst notwendigen *Stubs* zur Verfügung zu haben. Es liegt am Design der Integrations- und Konfigurationsschicht, von dieser Möglichkeit Gebrauch zu machen. Am Design der Adapterschnittstelle müssen hierfür keine weiteren Veränderungen vorgenommen werden. Es gilt zu beachten, dass die Anwendung von DII dazu führt, dass sich der zusätzliche Aufwand der dynamischen Erzeugung von Methodenaufrufen negativ auf die Komplexität der Integrations- und Konfigurationsschicht auswirkt. Außerdem ist es nicht unwahrscheinlich, dass die Integrations- und Konfigurationsschicht eine Vielzahl an Adaptern gleichzeitig verwalten und ansprechen muss und die Verwendung des DII nicht mehr skaliert. Daher wurde im Design der Adapter angedacht, alle Adapter Sourcen im Naming-Service zu publizieren und der Integrations- und Konfigurationsschicht die Möglichkeit zu bieten, Adapter Sourcen nach einer individuell gestaltbaren Selbstbeschreibung zu fragen. Diese könnte unter Umständen auch semantische Aussagen über die Ausführung von Funktionsaufrufen auf Adapter Sourcen beinhalten. Um eine *Brute-Force*-Suche nach einem bestimmten Adapter zu vermeiden, sollte in jedem Fall ein sinnvolles Namensschema zur Publikation der Adapter Sourcen gefunden werden.

4.2 Kernkomponenten des Adapter Framework

Nachdem nun das Design zur Kommunikation vollständig behandelt wurde, soll auf den Entwurf der Interfaces und der abstrakten Klassen, die zur Realisierung der Adapter implementiert werden müssen, diskutiert

[2] In diesem Zusammenhang könnte auch der CORBA-Reflection-Mechanismus von Interesse sein (vgl. dazu auch [OMG 06]).

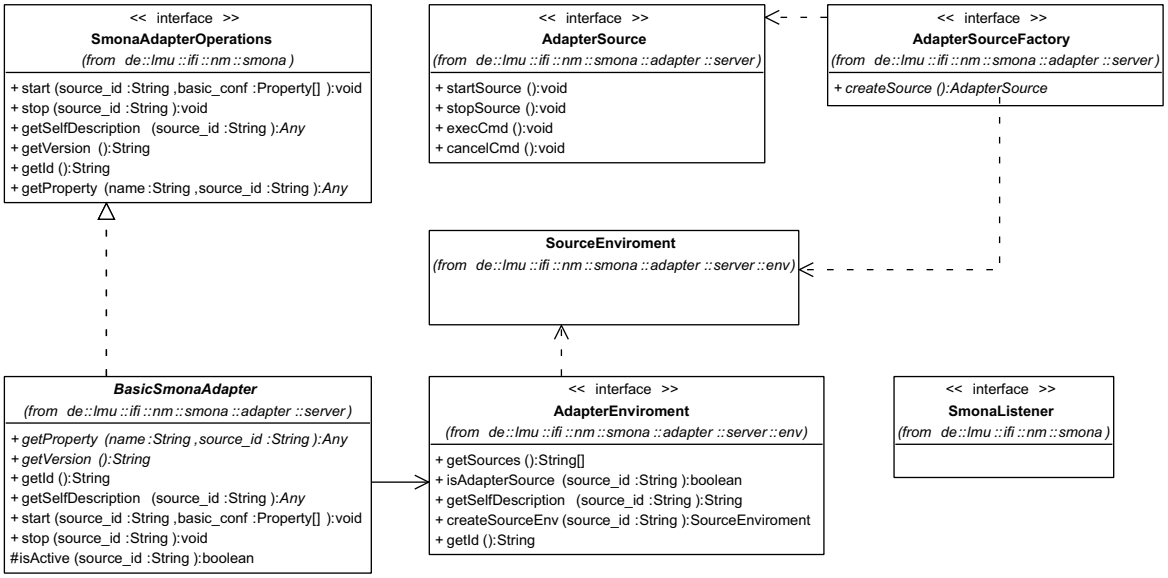

Abbildung 4.4: Interfaces und Abstrakte Klassen von Adaptern und Adapter Sourcen

werden. Dabei wurde versucht, die Klassen so zu definieren, dass sie möglichst leicht wiederverwendet werden können, um einen neuen Adapter zu implementieren. Beispiele dazu finden sich in Abschnitt 5.5. Abbildung 4.4 veranschaulicht die Basisklassen bzw. Basis-Interfaces, die definiert wurden, um die Aufteilung in Adapter und Adapter Sourcen zu bewerkstelligen.

`BasicSmonaAdapter` Die abstrakte Klasse `BasicSmonaAdapter` stellt die Oberklasse für alle Adaptertypen dar, die es zu implementieren gilt. Sie besitzt Methoden zum Ausführen von Funktionalitäten, die jeder Adapter besitzt. Diese Klasse implementiert einige Methoden des Interfaces `SmonaAdapterOperations`, welches automatisch bei der Erzeugung der Java Stubs und Skeletons aus den vorgegebenen IDL-Definitionen von Adaptern erzeugt wurde [3]. Die wichtigsten Methoden stellen `start()` und `stop()` dar, die zur Konfiguration, zum Starten sowie zum Stoppen einer Adapter Source verwendet werden. Stets wird dabei ein eindeutiger Identifikator mitgeliefert (Parameter `source_id`), um die entsprechende Adapter Source zu identifizieren (vgl. Kapitel 4.1.3).

`AdapterEnviroment` Bei der Generierung einer Adapterinstanz wird ein Objekt vom Typ `AdapterEnviroment` mitgegeben. Dieses Objekt muss vor dem Erzeugen eines Adapters mit Hilfe einer Konfigurationsdatei generiert werden (vgl. Kapitel 5.4.1). Ein Adapter verwendet es zum Beispiel dazu, dynamisch Informationen über Adapter Sourcen zu erfragen, die auch dann bereitstehen müssen, wenn von der entsprechenden Adapter Source noch keine Instanz existiert.

`AdapterSourceFactory` Jedem Adapter wird bei der Instantiierung ein Objekt vom Typ `AdapterSourceFactory` mitgegeben. Diese Instanz dient dazu, dynamisch Objekte vom Typ `AdapterSource` zu erzeugen und sie mit einer Umgebung vom Typ `SourceEnviroment` auszustatten.

`AdapterSource` Die eigentliche Aufgabe, den Zugriff auf Ressourcen durchzuführen, übernehmen Instanzen vom Typ `AdapterSource`. Ähnlich der Klasse `BasicSmonaAdapter` besitzt das Interface die Methoden `start()` und `stop()`, jedoch natürlich ohne den Parameter `source_id`. Klassen, die dieses Interface implementieren, erhalten bei ihrer Initialisierung ein Objekt vom Typ `SourceEnviroment`. Dieses stellt ihnen die Konfiguration für den Zugriff auf die zugeordnete Ressource zur Verfügung.

[3] Die Generierung von Java Stubs und Skeletons kann in [Bros 06] nachgelesen werden.

`SourceEnviroment` Instanzen dieser Klasse dienen dazu, einer dynamisch erzeugten Adapter Source Constraints zur Verfügung zu stellen, die bei der Konfiguration von Adapter Sourcen eingehalten werden müssen (vgl. Kapitel 5.4.1). Dabei kann es sich beispielsweise um maximal bzw. minimal erlaubte obere und untere Schranken für ein Intervall handeln, der einer Adapter Source vorgibt, wie häufig sie auf eine Ressource zugreifen darf bzw. soll.

`SmonaListener` Objekte vom Typ `SmonaListener` dienen zur Kommunikation mit der Integrations- und Konfigurationsschicht. Da der Zugriff auf Ressourcen asynchron verlaufen soll, ist es notwendig, dass die Integrations- und Konfigurationsschicht bei der Konfiguration von Adapter Sourcen ein Callback-Objekt mitliefert. Damit ist es möglich, Rückgabewerte an die Integrations- und Konfigurationsschicht zurück zu liefern bzw. sie über abgebrochene oder fehlgeschlagene Zugriffe zu benachrichtigen. Fehlerhafte Methodenaufrufe auf Adapter bedingt durch falsche oder fehlende Parameter werden durch das Werfen von Exceptions behandelt. Das Interface `SmonaListener` beinhaltet keine Deklarationen von Methoden. Stattdessen existiert für jeden Adaptertyp (Set, Pull und Push) ein Kind-Interface, welches die Methodensignaturen für den entsprechenden Anwendungsbereich vorgibt.

Kapitel 5

Implementierung der Adapter

In Abschnitt 4 wurde ausführlich auf das Design zur Umsetzung des Adapterkonzepts eingegangen. Dabei wurden die Basis-Interfaces und abstrakten Klassen entworfen, die ein Adapter implementieren muss, um das Konzept in einen Prototypen umzusetzen. Dieser Abschnitt befasst sich nun mit der konkreten Implementierung von Adaptern. Im ersten Teil soll die Implementierung anhand eines Beispiels erläutert werden. Der zweite Teil beschäftigt sich dann mit der Erweiterung des Adapter-Frameworks um neue Ressourcen, Adapter Sourcen oder Adapter hinzuzufügen.

5.1 Implementierung am Beispiel eines Pull Adapters

Im Folgenden soll anhand eines Beispiels gezeigt werden, wie die in Kapitel 4.2 vorgestellten Interfaces und abstrakten Klassen implementiert und verwendet werden. Dafür sollen die Klassen eines Pull Adapters (vgl. Kapitel 3.1) betrachtet werden. Zuvor müssen mit dem Aufruf

```
1 user@machine:~/> idl idl/PullAdapter.idl
```

die notwendigen Java Stubs und Skeletons erstellt werden, die zur Implementierung verwendet werden [1]. Die rechte Spalte von Abbildung 5.1 enthält die konkreten Implementierungen der Klassen eines Pull Adapters. Die folgenden Abschnitte erläutern die Initialisierung und das Zusammenspiel der beteiligten Klassen.

Object Request Brokers (ORB) Zunächst muss ein ORB initialisiert werden, um später den erzeugten Adapter zu publizieren und mit ihm zu interagieren. Hier soll nur der dazu notwendige Code vorgestellt werden. Gute Referenzen für die Implementierung von CORBA-Applikationen in Java sind [AKS 05] und [BVD 01]. Der folgende Programm-Code erzeugt eine Instanz vom Typ `org.omg.CORBA.ORB`. Von diesem Objekt wird der Portable Object Adapter `org.omg.PortableServer.POA` bezogen, der später zum Aktivieren des Pull Adapters benötigt wird. Außerdem wird eine Referenz zum CORBA-Naming-Service erzeugt, um den neuen Adapter zu publizieren.

```
1 Properties props = System.getProperties();
2 ORB orb = ORB.init(new String[0], props);
3 //get root POA
4 POA rootPOA = POAHelper.narrow(
5     orb.resolve_initial_references("RootPOA"));
6 // get Name Service
7 NamingContextExt rootNC = NamingContextExtHelper.narrow(
8     orb.resolve_initial_references("NameService"));
```

`XmlPullEnviroment` Der nächste Schritt ist die Erzeugung der Umgebung, die später benötigt wird, um dynamisch Adapter Sourcen vom Typ `PullSource` zu generieren. Als Konfigurationsdatei dient eine XML-Datei, in der alle Ressourcen vermerkt sind, für die der Pull Adapter Objekte vom Typ `PullSourcen` erzeugen können muss. Die Beschreibung der Konfiguration wird in Kapitel 5.5.1 erläutert. Es wird ein Objekt der

[1] Das IDL Interface `PullAdapter` ist in Anhang C.3 aufgeführt.

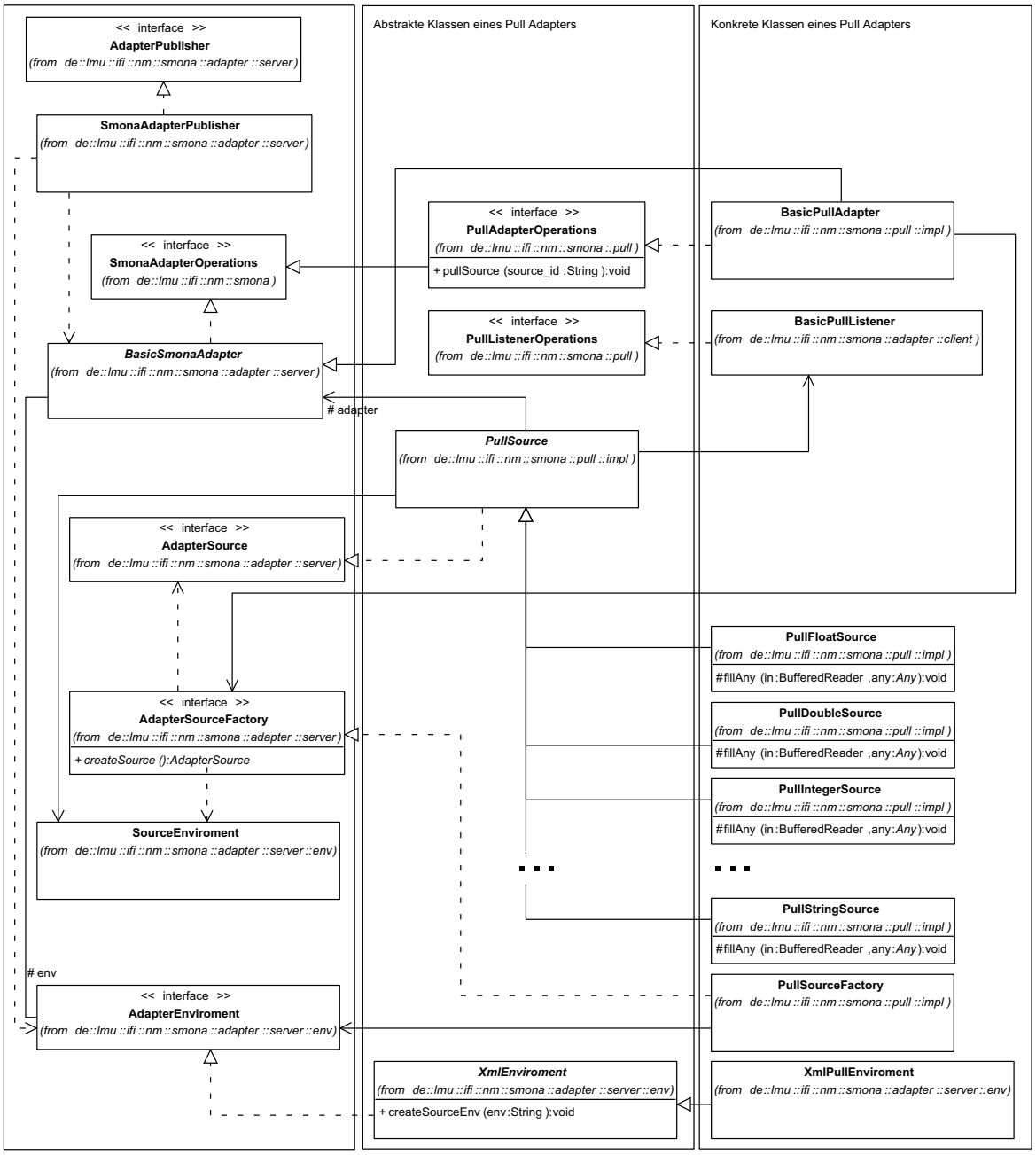

Abbildung 5.1: Abhängigkeiten der Klassen für das Beispiel eines Pull Adapters.

Klasse `XmlPullEnviroment` initialisiert welches die Klasse `XmlEnviroment`. implementiert. Die Methode `createSourceEnv()` dient später dazu, dynamisch ein Objekt der Klasse `SourceEnviroment` zu erzeugen (vgl. Kapitel 4.2).

```
1 String cfg_file = "etc/pull_adapter.xml";
2 // create a Pull Enviroment
3 AdapterEnviroment pull_env = null;
4 if(cfg_file.endsWith(Constants.XML_FILE)) {
5 pull_env = new XmlPullEnviroment(Constants.PULL_ADAPTER, cfg_file);
6 }
7 else{
8     System.exit(Constants.INVALID_CONFIGURATION);
9 }
```

`PullSourceFactory` Als nächstes wir ein Objekt der Klasse `PullSourceFactory` erzeugt. Diese Klasse implementiert das Interface `AdapterSourceFactory`. Die Methode `createSourceEnv()` wird so implementiert, das sie dynamisch Objekte der Klasse `SourceEnviroment` generiert. (vgl. Kapitel 4.2). Jede Instanz einer Klasse, die die abstrakte Klasse `PullSource` erweitert, erhält später eine Referenz auf ein Objekt der Klasse `SourceEnviroment`. Die gerade erzeugte Instanz vom Typ `XmlPullEnviroment` wird im Konstruktor der Klasse `PullSourceFactory` zusammen mit dem erzeugten `ORB` übergeben.

```
1 // create PullSourceFactory instance
2 AdapterSourceFactory factory = new PullSourceFactory(orb, set_env);
```

`BasicPullAdapter` Die Klasse `BasicPullAdapter` erweitert die Klasse `BasicSmonaAdapter`. Sie erbt unter anderem die Methoden `start()`, die sie zum Konfigurieren und Starten benötigt sowie die Methode `stop()` zum Stoppen ihrer Adapter Sourcen. Zusätzlich muss die Klasse `BasicPullAdapter` das Interface `PullAdapterOperations` implementieren, welches automatisch beim Kompilieren des IDL-Interfaces `PullAdapter` generiert wurde. Die einzige Methode, die die Klasse `BasicPullAdapter` zusätzlich implementiert ist die Methode `pullSource()`. Sie dient dazu, Daten von Objekten vom Typ `PullSource` abzufragen, die zuvor mit dem Aufruf der Methode `start()` konfiguriert wurden. Da die Klasse `BasicPullAdapter` von der Klasse `BasicSmonaAdapter` erbt, muss sie in der Klasse `PullAdapterPOATie` verpackt werden. In Java ist das Erben von mehreren Klassen nicht erlaubt. Daher müssen Adapter vom Typ `X` das Interface `XOperations` implementieren. Der Adapter `X` wird dann in einem Objekt der Klasse `XPOATIE` verpackt, welches Aufrufe von Methoden an die Klasse `X` delegiert (vgl. auch [OMG 02]). Im Konstruktor wird der `ORB`, die eben erzeugte Instanz vom Typ `PullSourceFactory` und die Umgebung vom Typ `XmlPullEnviroment` übergeben.

```
1 // create the BasicPullAdapter
2 PullAdapterPOATie a_adapter = new PullAdapterPOATie(
3     new BasicPullAdapter(orb, factory, set_env));
```

Danach muss der neue Adapter noch im POA aktiviert und beim ORB registriert werden.

```
1 // register PullAdapter with orb
2 rootPOA.activate_object(a_adapter);
3 PullAdapter adapter = PullAdapterHelper.narrow(
4     rootPOA.servant_to_reference(a_adapter));
```

`SmonaAdapterPublisher` Zum Schluss benötigt man noch ein Objekt der Klasse `SmonaAdapterPublisher`, um alle `PullSourcen` des neuen Adapters im CORBA-Naming-Service zu publizieren (vgl. Kapitel 4.1.2). Dazu wird die Methode `registerSources()` aufgerufen. Aus der Instanz `pull_env` vom Typ `XmlPullEnviroment` werden die zu publizierenden Adapter Sourcen ausgelesen.

```
1 //create a SmonaAdapterPublisher instance
2 AdapterPublisher publisher = new SmonaAdapterPublisher(rootNC);
3 // puplish sources through naming service
4 publisher.registerSources(adapter, pull_env);
```

Schließlich muss der POA noch aktiviert und der ORB gestartet werden.

```
1 // activate POA manager
2 rootPOA.the_POAManager().activate();
3 orb.run();
```

Kindklassen der Klasse `PullSource` Es existieren verschiedene Arten von Ressourcen. Derzeit sind die Basistypen implementiert. Die abstrakte Klasse `PullSource` besitzt alle Fähigkeiten, um mit Ressourcen zu interagieren (vgl. Kapitel 5.2). Die Kindklassen implementieren lediglich die Methode `fillAny()`.Der Parameter vom Type `BufferedReader` enthält den Rückgabewert des Prozesses, der auf die entsprechende Ressource aufgerufen wurde. Aufgabe der Methode ist es, den Wert korrekt auszulesen und in ein Objekt vom Type `org.omg.CORBA.Any` einzufügen. Dieses Objekt wird später verwendet, den Datenwert an die Integrations- und Konfigurationsschicht zurück zu liefern. Es gibt derzeit folgende Erweiterungen der Klasse `PullSource`:

- `PullFloatSource`
- `PullIntegerSource`
- `PullLongSource`
- `PullDoubleSource`
- `PullShortSource`
- `PullStringSource`
- `PullBooleanSource`

Ein Beispiel für einen komplexen Datentyp wird in Kapitel 5.5.2 gegeben.

`BasicPullListener` Diese Klasse hat keinen Einfluss auf die Instantiierung eines Pull Adapters. Sie dient als Callback-Objekt für alle Pull Adapter-Ressourcen (vgl. Kapitel 4.2). Die Integrations- und Konfigurationsschicht muss bei der Konfiguration bzw. beim Start einer Adapter Source ein Objekt dieser Klasse mitliefern. Die Klasse implementiert zwei Methoden des Interface `PullAdapterOperations`, die eine Pull Source aufrufen kann, um Rückgabewerte bzw. Fehlermeldungen an die Integrations- und Konfigurationsschicht weiterzuleiten. Ein Beispiel zum Aufruf dieser Methoden liefert Kapitel 5.2.5.

5.2 Beispiel Szenario für einen Pull Adapter

Da die wichtigsten Klassen von Pull Adaptern und ihre Aufgaben behandelt wurden, sollen nun an einem Beispiel die verschiedenen Interaktionen zwischen den Instanzen der Klassen aufgezeigt werden. Dazu wird eine Adapter Source vom Typ `PullFloatSource` verwendet. Die Aufgabe dieser Instanz soll das Auslesen der gegenwärtigen CPU-Auslastung sein.

5.2.1 Die Basiskonfiguration von Adaptern

Adapter Sourcen müssen vor ihrer Verwendung konfiguriert werden. Dies geschieht während des Aufrufens der `start()`-Methode auf den zuständigen Adapter (vgl. Kapitel 5.2.2). Dazu wird dem Adapter eine Sequenz von Instanzen der Klasse `Property` übermittelt (vgl. Abbildung 5.2).

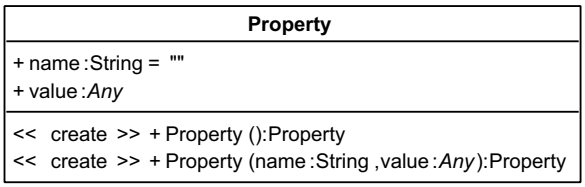

Abbildung 5.2: Die Klasse Property

Dabei handelt es sich um die Java Implementierung der Struktur `Property`, die im IDL Interface `SmonaAdapter` definiert wird (vgl. Anhang C.3). Properties bilden Name-Wert-Paare, wobei der Wert durch ein Objekt vom Typ `org.omg.CORBA.Any` gekapselt wird. Damit können alle Arten von primitiven Datentypen und alle durch CORBA-Interfaces definierten Objekte verpackt werden.

Alle Adapter Sourcen benötigen verschiedene `Property`-Objekte, um ihre Funktion zu erfüllen. Im Folgenden werden die `Property`-Objekte erläutert, die alle Adapter Sourcen benötigen. Die verwendeten Namen entsprechen den gültigen `name`-Attributen der Klasse `Property`:

`source_id` Wie in Kapitel 3.2 beschrieben wird dem Methodenaufruf auf einen Adapter ein Identifikationsparameter mitgegeben. Dieser Parameter `source_id` dient dazu, die zuständige Adapter Sourcen zu identifizieren, die die entsprechende Aktion ausführen soll.

`listener` Damit eine Adapter Source das Ergebnis des Methodenaufrufs an die Integrations- und Konfigurationsschicht übermitteln kann, muss bei der Konfiguration ein Callback-Objekt vom Type `SmonaListener` übermittelt werden. Im Falle einer Pull Source handelt es sich dabei um ein Objekt vom Typ `PullListener`.

`interval` Der Interval gibt vor, wie lange eine Adapter Source auf die Beendigung eines externen Programmaufrufs warten soll, bevor sie diesen abbricht und eine entsprechende Fehlermeldung an die Integrations- und Konfigurationsschicht sendet.

Wird in einer Konfiguration eine `source_id` verwendet, für die keine Adapter Source publiziert wurde, reagiert der Adapter mit dem Werfen einer `NoSuchSourceException`. Sollte eines der erwähnten `Property`-Objekte bei der Konfiguration fehlen, wird eine `MissingPropertyException` geworfen. Trägt ein `Property`-Objekte einen ungültigen Wert (vgl. Kapitel 5.4.1), wird eine `InvalidPropertyValueException` generiert. Sollte für den Namen einer `Property` kein gültiger Typ existieren, endet dies im Wurf einer `NoSuchPropertyException`.

Dies sind nur die obligatorischen `Property`-Objekte, die jede Adapter Source zur Konfiguration benötigt. Push Sourcen können z.B noch ein `Property`-Objekt benötigen, um obere und/oder untere Schranken zu definieren, welche die Auslösung eines Ereignisses in der Integrations- und Konfigurationsschicht steuern (vgl. Kapitel 5.4.1).

5.2.2 Konfiguration und Start der Adapter Source

Die erste Methode, die die Integrations- und Konfigurationsschicht auf einen Adapter aufrufen muss ist die `start()`-Methode. In Abbildung 5.3 sind die Interaktionen der Konfiguration dargestellt.

Für Pull Adapter Sourcen bewirkt die `start()`-Methode die Konfiguration der entsprecheden Adapter Source. Nachdem die `start()`-Methode aufgerufen wurde, wird als erstes überprüft, ob die Adapter Source mit dem Namen `source_id` zuvor schon gestartet wurde. Intern wird hier die Instanz `source_map` vom Typ `HashMap` angelegt, die Paare von Adapter Sourcen und ihren Namen verwaltet, die gerade aktiv sind. Sollte dies der Fall sein, wird eine Exception vom Typ `AdapterAlreadyActiveException` geworfen, die der Integrations- und Konfigurationsschicht anzeigt, dass sie die Adapter Source erst stoppen muss, um sie danach neu zu starten. Ist die gewünschte Adapter Source inaktiv, wird durch das `factory`-Objekt eine neue Instanz vom Typ `PullFloatSource` erzeugt. Dieses wird der `source_map` hinzugefügt und schließlich die `startSource()`-Methode auf die neue Adapter Source aufgerufen. Intern bewirkt dieser Aufruf die Konfiguration von Attributen, die dazu verwendet werden, die asynchrone Kommunikation zu realisieren.

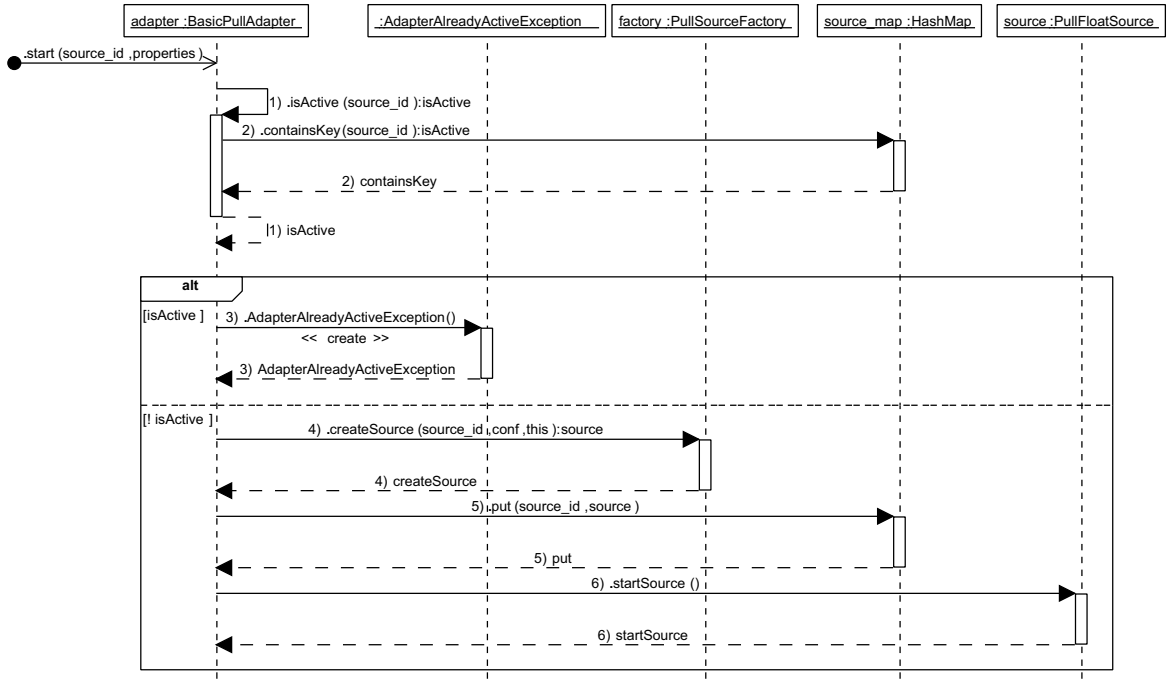

Abbildung 5.3: Interaktionen beim Aufruf der `start()`-Methode

5.2.3 Stoppen der Adapter Source

Um die im folgenden Kapitel 5.2.4 beschriebenen Vorgänge besser zu verstehen, soll hier die Behandlung der `stop()`-Methode von Adaptern vorgezogen werden. Abbildung 5.4 veranschaulicht die Interaktionen der beteiligten Klassen.

Die `stop()`-Methode kann jederzeit auf einen Adapter aufgerufen werden. Erfolgt der Aufruf während der Abarbeitung einer zuvor initialisierten Anfrage an die zu stoppende Instanz der Klasse `PullFloatSource`, kann es vorkommen, das das Ergebnis dieser Anfrage erst nach Beendigung der `stop()`-Methode zurückgeliefert wird. Diese Eigenschaft wird in Kauf genommen, da es die Implementierung der asynchronen Kommunikation unnötig verkomplizieren würde, ein anderes Verhalten zu erwirken. Als erstes wird von der Instanz der Klasse `BasicPullAdapter` überprüft, ob die mitgelieferten `source_id` eine gültige Pull Source beschreibt. Dazu wird das Objekt `env` vom Typ `XMLPullEnviroment` abgefragt. Kann keine Pull Source für die entsprechende `source_id` gefunden werden, wird eine Exception vom Typ `NoSuchSourceException` generiert. Handelt es sich um eine gültige ID, wird die zugehörige Pull Source aus dem `source_map` Objekt entfernt. Danach wird die Methode `stopSource()` der Instanz der Klasse `PullFloatSource` aufgerufen, um die Pull Source zu stoppen.

5.2.4 Veranschaulichung der asynchronen Kommunikation

Im Folgenden soll die Implementierung der asynchronen Kommunikation erläutert werden, die in Kapitel 2.3.3 diskutiert wurde. Dabei wird als Beispiel der Aufruf der Methode `pullSource()` untersucht und auf die von ihr ausgelösten Interaktionen eingegangen. Das Design wurde so gewählt, dass auch beim Hinzufügen neuer Klassen, die die Klasse `PullSource` erweitern, keine Änderungen an den folgenden Bausteinen vorzunehmen sind. Das Diagramm in Abbildung 5.5 veranschaulicht die Interaktionen der an der Durchführung eines `pullSource()`-Aufrufes beteiligten Klassen.

Die Klassen `AbortTimer` und `CmdProcessor` implementieren die `run()`-Methode, die sie von der Klasse `java.lang.Thread` erben. In der `startSource()`-Methode der Klasse `PullSource` wird jeweils eine Instanz der Klasse `CmdProcessor` und der Klasse `AbortTimer` instantiiert und gestartet

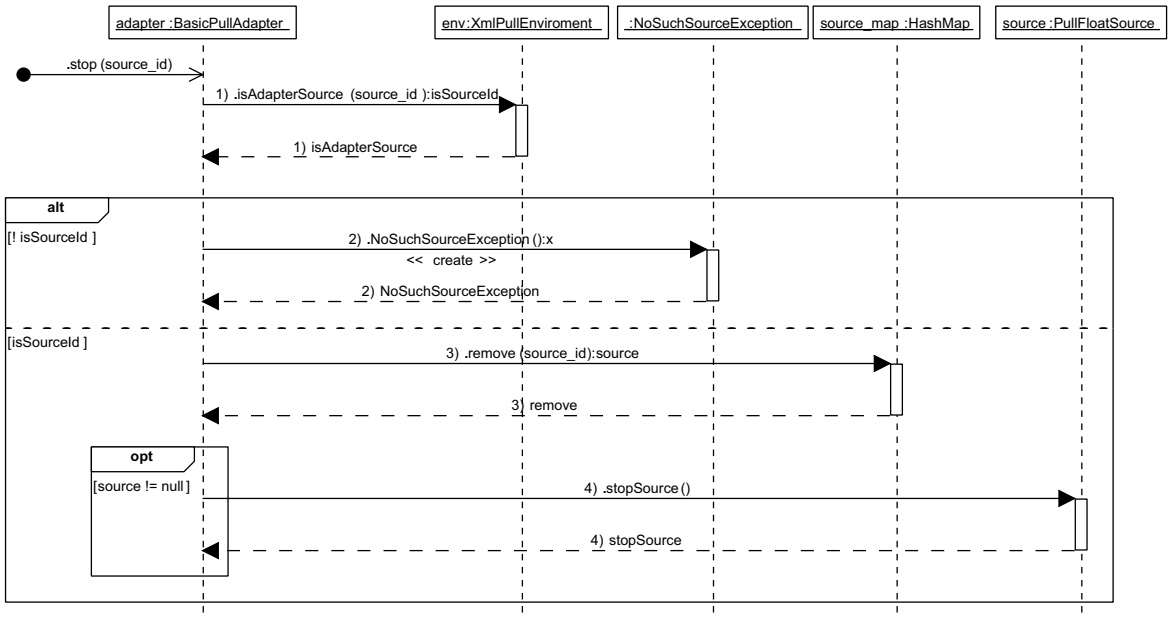

Abbildung 5.4: Interaktionen beim Aufrufen der `stop()`-Methode

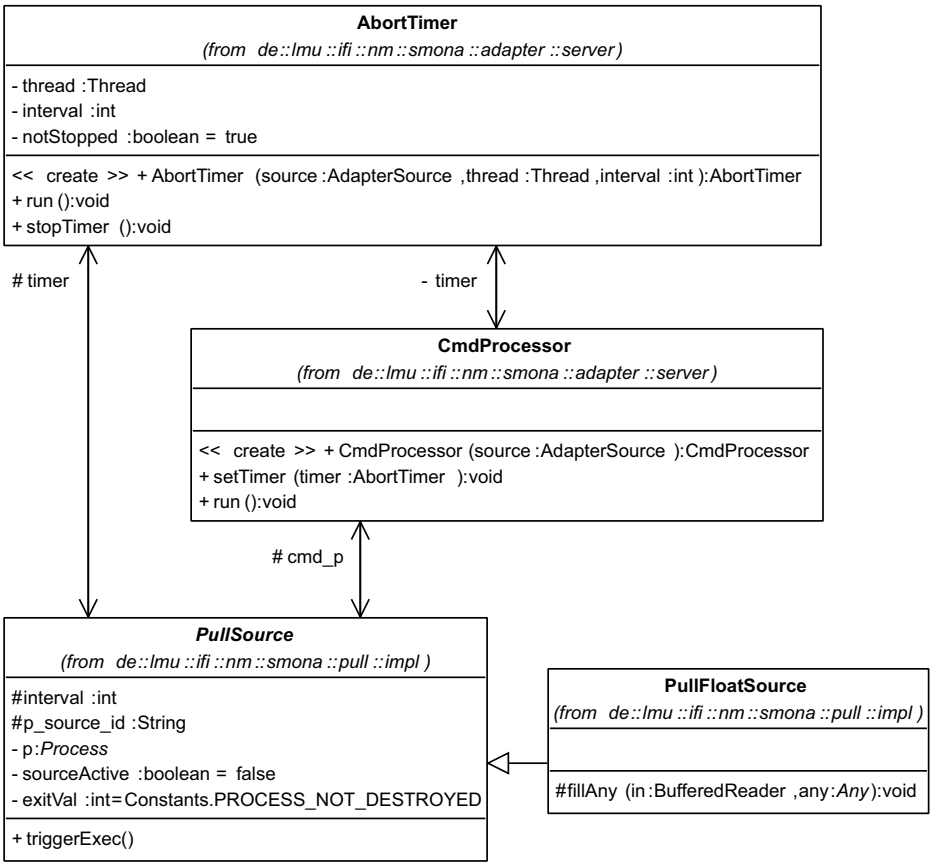

Abbildung 5.5: Beteiligte Klassen zur Realisierung der asynchronen Kommunikation

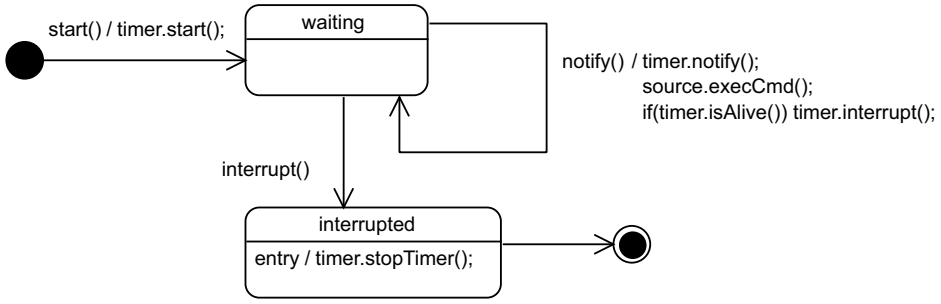

Abbildung 5.6: Zustände der Instanz vom Typ CmdProcessor

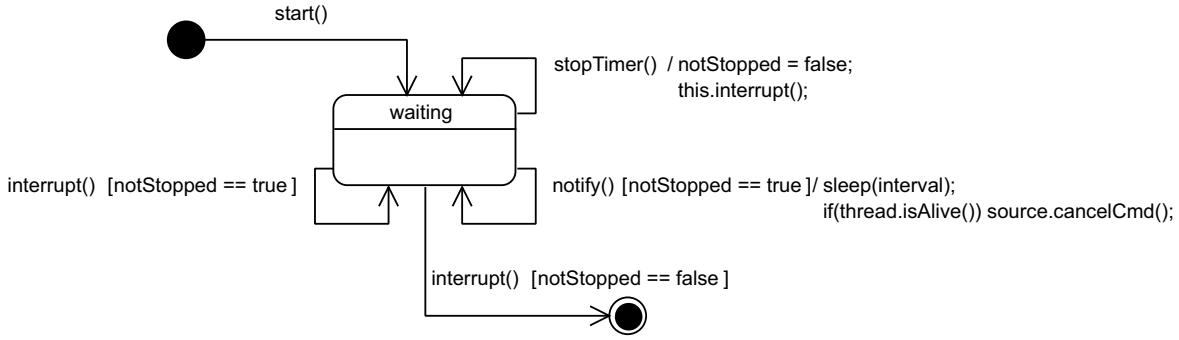

Abbildung 5.7: Zustände der Instanz vom Typ AbortTimer

(`Thread.start()`). Die Instanz der Klasse `CmdProcessor` übernimmt die Aufgabe mit Hilfe eines externen Programmaufrufs auf diejenige Source zuzugreifen, die von der Instanz der Klasse `PullFloatSource` verwaltet wird. Die Instanz der Klasse `AbortTimer` dient dazu, einen externen Programmaufruf abzubrechen, wenn dieser nicht innerhalb des vorgegebenen Intervalls `interval` zurückgekehrt ist. Die Zustände, die die Objekte der beiden Klassen durchlaufen, sind in den Abbildung 5.6 und 5.7 dargestellt. Das Objekt `source` ist vom Typ `PullSource`, das Objekt `timer` vom Typ `AbortTimer` und das Objekt `thread` vom Typ `CmdProcessor`.

Der Ablauf wird hier stichpunktartig aufgelistet. Abbildung 5.6 wird im Folgenden mit T_P, Abbildung 5.7 mit T_A referenziert.

Schritt 1 (T_P): Auf `thread` wird die `start()`-Methode aufgerufen.

Schritt 2 (T_P): `thread` ruft auf `timer` die `start()`-Methode auf und wartet auf den Aufruf von `notify()` [2].

Schritt 3 (T_A): Auf `timer` wurde von `thread` die Methode `start()` aufgerufen. `timer` wartet auf ein `notify()` von `thread`.

Schritt 4 (T_P): Durch den Aufruf von `pullSource()` auf `source` wird innerhalb dieser Methode die Methode `notify()` auf `thread` aufgerufen. `thread` ruft `notify()` auf `timer` auf und startet mit dem Aufruf `execCmd()` auf `source` die Ausführung eines externen Programmaufrufs, der die entsprechende Source ausliest und das Ergebnis an die Integrations- und Konfigurationsschicht zurückliefert.

Schritt 5 (T_A): Auf `timer` wurde von `thread` die Methode `notify()` aufgerufen. Die Variable `notStopped` beinhaltet den Wert `true`. Der Thread `timer` versucht `interval` Sekunden zu schlafen.

Verzweigung Für den Fall, dass das mit der Methode `execCmd()` aufgerufene Kommando innerhalb des

[2] Die `run()`-Methode der Klassen `CmdProcessor` und `AbortTimer` sind synchronisiert, so dass keine Aufrufe von `notify()` verloren gehen können.

vorgegebenen Intervalls zurückkehrt, geht es bei Schritt 6, ansonsten bei Schritt 8 weiter.

Schritt 6 (T_P): `thread` überprüft, ob `timer` noch existiert. Sollte dies der Fall sein, unterbricht `thread` `timer` mit dem Aufruf `interrupt()` und kehrt in den Zustand *waiting* zurück.

Schritt 7 (T_A): Auf `timer` wurde von `thread` `interrupt()` aufgerufen. `timer` begibt sich wieder in den Zustand *waiting*.

Sprung zu Schritt 4 Der Aufruf von `pullSource()` wurde erfolgreich abgeschlossen und ein neuer Aufruf wird erwartet.

Schritt 8 (T_A): `timer` wacht wieder auf, d.h. dass der Aufruf `execCmd()` noch nicht zurückgekehrt ist. `timer` testet, ob `thread` noch existiert, und ruft in diesem Fall `cancelCmd()` auf `source` auf. Dieser Aufruf bewirkt, dass der Prozess, der das externe Kommando ausführt, abgebrochen wird. Dadurch wird der Aufruf von `execCmd()` deblockiert. `timer` geht wieder in den Zustand *waiting* über.

Schritt 9 (T_P): Der blockierte Aufruf von `execCmd()` auf `source` wurde gelöst. In der Methode `execCmd()` wird ein Aufruf erzeugt, um die Integrations- und Konfigurationschicht über den misslungenen externen Programmaufruf zu informieren. `thread` geht – ohne den Zustand von `timer` zu testen – wieder in den Zustand *waiting* über. Der Aufruf von `pullSource()` wurde ohne Erfolg beendet und ein neuer Aufruf wird erwartet.

Es fehlen noch die Transitionen, die durch den Aufruf der `stopSource()`-Methode bedingt werden. Die Methode sorgt dafür, dass die Threads `thread` und `timer` beendet werden. Der Aufruf kann jederzeit erfolgen.

Schritt 1 (T_P): Innerhalb der Methode `stopSource()` der Klasse `PullFloatSource` wird die Methode `interrupt()` auf `thread` aufgerufen (`thread` befindet sich im Zustand *waiting*). `thread` ruft die Methode `stopTimer()` auf `timer` auf und beendet sich.

Schritt 2 (T_A): Auf `timer` wurde im Zustand *waiting* die Methode `stopTimer()` aufgerufen. Diese Methode setzt das Flag `notStopped` auf `false` und unterbricht den `timer` per Aufruf der Methode `interrupt`.

Schritt 3 (T_A): `timer` wurde durch die Methode `interrupt` unterbrochen und überprüft das Flag `notStopped`. Da es auf `false` gesetzt ist, beendet sich auch `timer`. Beide Threads haben die `run()`-Methode verlassen.

5.2.5 Ausführung des externen Programmaufrufs

Der Schlüssel zum Ausführen des externen Programmaufrufs liegt in den Methoden `execCmd()` und `cancelCmd()` der abstrakten Klasse `PullSource`. Wie in Kapitel 5.2.4 beschrieben, dient die Methode `execCmd()` dazu, den externen Programmaufruf auf der gewünschten Source auszuführen. Sollte ein solcher Aufruf blockiert sein, kann dieser mit Hilfe der Methode `cancelCmd()` abgebrochen werden. Hier sollen nur die wichtigsten Methodenaufrufe betrachtet werden. Das Abfangen von Exceptions wurde ausgespart. Außerdem wird nur auf den Fall eingegangen, dass die Pull Source ohne Historien arbeitet (vgl. dazu Kapitel 5.2.6).

Listing 5.1: Code Fragment der Methode `execCmd()`

```
1  1  public synchronized void execCmd() {
2  2      ...
3  3      BufferedReader in;
4  4      Any any = orb.create_any();
5  5      // get exec() output
6  6      p = Runtime.getRuntime().exec(src_env.getCmd() + "␣"
7  7              + src_env.getCmdParamString());
8  8      in = new BufferedReader(new InputStreamReader(p.getInputStream()));
9  9      try{
10 10         p.waitFor();
11 11     }
```

```
12 12      catch(InterruptedException e) {
13 13          e.printStackTrace();
14 14      }
15 15      // Check Process state and decide what to do
16 16      if(exitVal == Constants.PROCESS_NOT_DESTROYED) {
17 17          // check exit status of cmd
18 18          if(p.exitValue() == Constants.EXEC_SUCCESS) {
19 19              // cast return val of func call and fill any with correct type
20 20              try {
21 21                  fillAny(in, any);
22 22                  // call listener
23 23                  listener.send_Value(p_source_id, any);
24 24              }
25 25              catch(UselessValueException uv_ex) {}
26 26              // close BufferedReader
27 27              in.close();
28 28          }
29 29          else {
30 30              this.send_SourceError(Constants.UNFORCED_EXIT, p.exitValue());
31 31          }
32 32      }
33 33      else {// Constants.TIME_EXCEEDED
34 34          this.send_SourceError(Constants.FORCED_EXIT, Constants.TIME_EXCEEDED);
35 35      }
36 36      // allow new pullSource() calls
37 37      sourceActive = false;
38 38      // reset exit status
39 39      exitVal = Constants.PROCESS_NOT_DESTROYED;
40 40      ...
41 41  }
42 42
43 43  public void cancelCmd() {
44 44      exitVal = Constants.TIME_EXCEEDED;
45 45      p.destroy();
46 46  }
```

Der Aufruf des externen Kommandos geschieht in Zeilen 6 und 7. Die Kommandozeile wird von dem Objekt `src_env` vom Typ `SourceEnviroment` ausgelesen. `src_env` wird im Rumpf der `start()`-Methode der Klasse `BasicPullAdapter` durch die zuständige `factory` vom Typ `PullSourceFactory` für die Pull Source generiert (vgl. Kapitel 5.2.2). Das erzeugte Objekt `p` vom Typ `java.lang.Process` wird als Attribut verwaltet. Damit erhält man die Möglichkeit, einen blockierten Prozess mit dem Methodenaufruf `cancelCmd()` zu beenden (Zeile 43ff). Diese Methode wird unter Umständen von einer Instanz der Klasse `AbortTimer` aufgerufen, um einen externen Programmaufruf zu beenden, sollte dieser nicht innerhalb des vordefinierten Intervalls zurückgekehrt sein (vgl. Kapitel 5.2.4). Außerdem wird noch ein Exit-Status vermerkt, um im Rumpf der Methode `execCmd()` festzustellen, ob sich `p` regulär beendet hat oder nicht. Über eine Instanz der Klasse `java.io.BufferedReader` erhält man die Ausgabe von `p` (Zeile 8). Es muss auf die Beendigung des Prozesses gewartet werden (Zeile 9 - 14). Andernfalls könnte ein blockierter Programmaufruf nicht beendet werden. Es wird überprüft, ob der Programmaufruf erfolgreich ausgeführt wurde. Drei Fälle können hier eintreten:

1. `p` wurde nicht durch einen Aufruf der Methode `cancelCmd()` zerstört und der Exit-Status von `p` liefert den Integerwert `0` (vgl. Zeile 16 - 28).

   In diesem Fall wird die abstrakte Methode `fillAny()` aufgerufen, die die Ausgabe in ein Objekt vom Typ `org.omg.CORBA.Any` verpackt (Zeile 21). Die Konstante `Constants.EXEC_SUCCESS` trägt den Wert `0` da dieser unter Linux die erfolgreiche Ausführung eines Kommandos indiziert. Für andere Plattformen muss dieser Wert gegebenenfalls angepasst werden. Die Implementierung dieser Methode erfolgt in den typspezifischen Kindklassen von `PullSource` (vgl. Kapitel 5.1). Anschließend wird über das Objekt `listener` vom Typ `PullListener` der Integrations- und Konfigurationschicht der

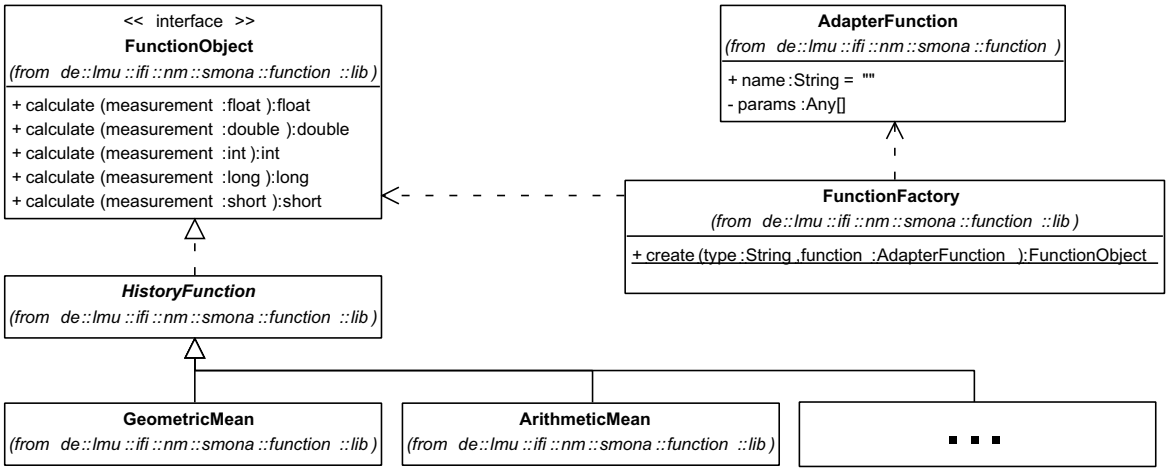

Abbildung 5.8: Klassen zur Verwaltung von Historien

Wert zugesendet (vgl. Kapitel 5.1).

2. `p` wurde nicht durch einen Aufruf der Methode `cancelCmd()` zerstört, jedoch der Exit-Status von `p` lieferte einen Integerwert ungleich `0` (vgl. Zeile 29 - 31).

   Die Integrations- und Konfigurationschicht muss über den misslungenen Programmaufruf informiert werden. Dazu wird auf `listener` die Methode `send_SourceError()` aufgerufen, die als Fehlercode den Rückgabewert des Aufrufs erhält. Zusätzlich wird eine Konstante angegeben, die der Integrations- und Konfigurationschicht anzeigt, dass der Programmaufruf nicht aufgrund seines Blockierens abgebrochen wurde.

3. `p` wurde durch einen Aufruf der Methode `cancelCmd()` zerstört (vgl. Zeile 33 und 34).

   Auch hier muss die Integrations- und Konfigurationschicht über den misslungenen Programmaufruf informiert werden. Die Parameter der Methode `send_SourceError()` werden dazu jedoch so gewählt, dass sie anzeigen, dass der Programmaufruf aufgrund seins Blockierens abgebrochen wurde.

5.2.6 Verwaltung von Historien

In Kapitel 2.3.6 wurde bereits die Notwendigkeit für die Verwaltung von Historien erläutert. Die Umsetzung wird durch die Definition von Funktionen erwirkt. Diese können bei der Konfiguration einer Adapter Source angegeben werden. Für verschiedene Ressourcen sind auch unterschiedliche Funktionen sinnvoll. Funktionen, die auf Zahlentypen angewandt werden, können in der Regel nicht auf den Typ Boolean übertragen werden. Beispielsweise kann kein arithmetisches Mittel von einer Menge von Boolean Werten berechnet werden. Für das vorliegende Adapter-Framework wurden die zwei Beispielfunktionen arithmetisches und geometrisches Mittel implementiert. Um die Anwendung einer Funktion auf eine Source zu erwirken, kann ein `Property`-Objekt vom Typ `function` in der Startkonfiguration verwendet werden. Funktionen sind sowohl für Pull als auch für Push Sourcen sinnvoll, da diese über die notwendigen Historien ihrer zugeordneten Source verfügen. Die Klassen, die bei der Erzeugung von Funktionen beteiligt sind, werden in Abbildung 5.8 veranschaulicht.

Wurde ein `Property`-Objekt beim Aufruf der `start()`-Methode auf den Adapter mitgeliefert, trägt die angesprochene Adapter Source dafür Sorge, dass auf die Daten der zugrundeliegenden Ressource die gewünschte Funktion ausgeführt wird. In der IDL `AdapterFunction` wurde zur Spezifizierung der Funktion eine Struktur `AdapterFunction` definiert (vgl. Anhang C.4). Auf eine Instanz vom Typ `FunctionFactory` wird die Methode `create()` aufgerufen. Als Parameter erhält sie den Typ der Funktion, die ausgeführt werden soll, sowie ein Objekt vom Typ `AdapterFunction`, welches den Namen der Funktion und die dazugehörigen Parameter beinhaltet. Für das arithmetische Mittel könnte ein Parameter beispielsweise die Anzahl der letzten `X` Werte angeben, die ein Adapter in die Berechnung einfließen lassen soll. Die `create()`-Methode generiert aus diesen Parametern ein Objekt vom Typ `FunctionObject`. Ein Objekt dieses Typs speichert die

Historie intern als Array. Wird ein neuer Wert der zugrundeliegenden Ressource generiert, kann auf das Funktionsobjekt die Methode `calculate()` mit dem neuen Wert als Parameter aufgerufen werden. Dieser ersetzt den ältesten im Array gespeicherten Wert und berechnet entsprechend der vorgegebenen Funktion den potentiellen Rückgabewert für die Integrations- und Konfigurationsschicht. Das Interface `FunctionObject` wird von der abstrakten Klasse `HistoryFunction` implementiert, die die Basisfunktionalität von Funktionen besitzt. Derzeit lassen sich Instanzen von den konkreten Klassen `ArithmeticMean` und `GeometricMean` erzeugen.

5.3 Schnittstelle für Set Adapter

Set Adapter fallen etwas aus dem Rahmen der einheitlich definierten Schnittstelle. Dies liegt daran, dass der Wunsch geäußert wurde, dass die Signaturen für Set Sourcen spezifisch auf die entsprechende Aktion abgestimmt sein sollen. In der Regel bedeutet dies, dass für jede Set Source ein eigener Adapter geschrieben werden muss. Als Beispiel wurde im Rahmen dieser Arbeit ein Mount-Adapter implementiert. Dieser wird in einer eigenen IDL definiert, wo auch die zwei adapterspezifischen Methoden `mount()` und `unmount()` beschrieben sind.

Listing 5.2: IDL Definition für einen Mount Adapter

```
1
2  #include "SetAdapter.idl"
3
4  module de {
5  module lmu {
6  module ifi {
7  module nm {
8  module smona {
9  module set {
10 module mount {
11     interface MountAdapter : SetAdapter {
12
13         /** Mount filesystem
14         */
15         void mountFS(
16             in string source_id, in string mnt_point, in string mnt_device)
17             raises (NoSuchSourceException, AdapterNotConfiguredException);
18
19         /** Unmount filesystem
20         */
21         void umountFS(in string source_id, in string mnt_point)
22             raises (NoSuchSourceException, AdapterNotConfiguredException);
23     };
24 };
25 };
26 };
27 };
28 };
29 };
30 };
```

Die Methoden haben festgelegte Parameter wie zum Beispiel einen Parameter vom Typ `string`, der dazu verwendet wird, das Dateisystem zu spezifizieren, welches eingehängt werden soll. In vielen Fällen können Set Adapter so spezifiziert werden, dass die Signatur für alle Plattformen Sinn macht. Ein Beispiel wäre ein Adapter, der das IP-Forwarding auf einem Internet-Host aktiviert bzw. deaktiviert. Ein Beispiel zur Implementierung dieses Set Adapters liefert Kapitel 5.5.3.

5.4 Funktionsweise der Adapter

Im Folgenden soll die Konfiguration und Anwendung der verschiedenen Adapter erläutert werden. Zunächst wird das Konfigurationsschema der drei Adaptertypen erläutert und im Anschluss die Aufruftechnik für den jeweiligen Adaptertyp.

5.4.1 Konfiguration

Jeder Adapter wird mit einer zuvor erstellten, auf XML basierenden Konfigurationsdatei gestartet, die es ihm erlaubt festzustellen, welche Arten von Ressourcen der Adapter zur Verfügung stellen soll. Beim Start eines Adapters liest dieser die Konfigurationsdatei ein und erstellt eine Liste von Adapter Sourcen, die er über den CORBA Naming Service publiziert. Erfolgt ein Aufruf auf eine Adapter Source des Adapters, kann dieser (anhand der zugrundeliegenden Konfiguration) dynamisch eine Instanz der Adapter Source erzeugen und den Aufruf abarbeiten. Die XML-Datei wird nur einmal eingelesen. Intern wird eine Hash-Tabelle zur Speicherung der Konfigurationen angelegt. Konfigurationen werden anhand eines XML-Schemas validiert um Konfigurationsfehlern von vornherein so weit wie möglich vorzubeugen (vgl. Anhang C.1).

Konfiguration eines Pull Adapters

Da derzeit für Pull und Set Adapter die gleichen Konfigurationsparameter verwendet werden können, wird hier nur auf die Konfigurationsparameter von Pull Adaptern eingegangen. Besonderheiten für Push Adapter werden in Kapitel 5.4.1 behandelt.

Listing 5.3: Konfigurationsdatei für einen Pull Adapter

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <adapter
3      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4      xsi:noNamespaceSchemaLocation="file:etc/Adapter.xsd"
5      id="pull_adapter_10A4F678433C00D2A002AF678433C784F330067800">
6      <pull_adapter>
7          <source id="PullAdapter.A/.../.../.../CPULoad.S/float.ST">
8              <src_type>float</src_type>
9              <description>Source to pull CPU Load</description>
10             <interval>
11                 <type>milliseconds</type>
12                 <max_allowed>5000</max_allowed>
13                 <min_allowed>1000</min_allowed>
14             </interval>
15             <process>
16                 <cmd>bin/system_info/cpu_load</cmd>
17                 <cmd_params>
18                     <param>1</param>
19                 </cmd_params>
20             </process>
21         </source>
22         <source id="PullAdapter.A/.../.../.../OneMinuteAvgLoad.S/float.ST">
23             <src_type>float</src_type>
24             <description>Source to pull CPUAvgLoad of last minute</description>
25             <interval>
26                 <type>milliseconds</type>
27                 <max_allowed>5000</max_allowed>
28                 <min_allowed>1000</min_allowed>
29             </interval>
30             <process>
31                 <cmd>bin/system_info/avg_1_load</cmd>
32                 <cmd_params>
```

```
33                      <param>1</param>
34                  </cmd_params>
35              </process>
36          </source>
37          </source>
38          <source id="PullAdapter.A/.../.../.../OneMinuteAvgLoad.S/float.ST">
39          ...
40          </source>
41      </pull_adapter>
42  </adapter>
```

Unabhängig vom Adaptertyp umschließt jede Konfigurationsdatei ihren Inhalt mit dem *Document Root Element* `<adapter>...</adapter>`. Danach folgt das Element, das den Adapter Typ spezifiziert, für den die Konfiguration ausgelegt ist – im Beispiel `<pull_adapter>...</pull_adapter>`. Innerhalb dieses Elements folgen beliebig viele Blöcke von `<source>...</source>` Elementen, die die Konfigurationen für die Adapter Sourcen im einzelnen definieren. Jedes `<source>...</source>` Element hat ein `id`-Attribut, das den Namen beinhaltet, unter welchem diese Adapter Source im CORBA-Naming-Service publiziert werden soll. Die Kind-Elemente des `<source>...</source>` Elements sind bei Pull Adaptern folgende:

`<src_type>...</src_type>` Dieses Element spezifiziert den Ressourcentyp. Im Allgemeinen sind hier alle einfachen Datentypen wie `float, integer, double, long, short` sowie `string` möglich. Für komplexe Datentypen müssen eigene Erweiterungen geschrieben werden (siehe Abschnitt 5.5.2).

`<description>...</description>` Adapter Sourcen werden im Naming-Service bekannt gegeben. Damit ist es möglich, für Instanzen der Integrations- und Konfigurationsschicht die passenden Adapter Sourcen zu finden und zu benutzen. Damit die Integrations- und Konfigurationsschicht Informationen über eine Adapter Source erfragen kann, wird jeder eine Selbstbeschreibung beigefügt, die die möglichen Konfigurationsparameter sowie ihre Bedeutung wiederspiegelt. Diese Information trägt das `<description>...</description>` Element. Derzeit liefert es nur eine textuelle Beschreibung der Adapter Source.

`<interval>...</interval>` Wie in Kapitel 5.2.4 beschrieben, verläuft die Kommunikation mit Adaptern asynchron. Ausnahmen sind Funktionen zum Starten und Stoppen eines Adapters sowie zur Abfrage von Versionsnummer, Adapter-ID und der Beschreibung der Adapter Sourcen. Unabhängig vom Adaptertyp kann die Integrations- und Konfigurationsschicht ein Limit für die maximale Ausführungszeit eines Aufrufs auf eine Adapter Source vorgeben. Wird dieses Limit überschritten, beendet der Adapter den Aufruf und informiert die Integrations- und Konfigurationsschicht über den fehlgeschlagenen Zugriff. Um willkürlichen Konfigurationen vorzubeugen, kann über die Kindelemente `<max_allowed>...</max_allowed>` und `<min_allowed>...</min_allowed>` ein Bereich festgelegt werden, in dem sich ein Konfigurationsintervall bewegen muss, damit eine Adapter Source diesen akzeptiert. Es darf beispielsweise nicht möglich sein, einen Push Adapter so zu konfigurieren, dass die zuständige Adapter Source im Intervall von wenigen Millisekunden ihre Quelle ausliest, um die Integrations- und Konfigurationsschicht mit diesen Daten dann zu bombardieren. Das Kindelement `<type>...</type>` dient der Angabe der Zeiteinheit (Millisekunden, Sekunden, Stunden, ...), die zur Konfiguration verwendet werden soll. Derzeit sind nur Angaben in Millisekunden möglich.

`<process>...</process>` Die derzeit implementierten Adapter Sourcen erzeugen ihre Daten, indem sie auf externe Programme und Skripte zurückgreifen. In Zukunft sollten hier auch Klassen dynamisch geladen werden können (vgl. Kapitel 5.5.4). Das `<process>...</process>`-Element spezifiziert dazu die entsprechenden Kommandos und deren Parameter. Das Kindelement `<cmd>...</cmd>` beinhaltet das auszuführende Kommando, das Kindelement `<params>...</params>` eine Liste von Kommandoparametern. Die jeweils zuständige Adapter Source muss wissen, wie die Parameterliste zu interpretieren ist. Parameterlisten machen auch für Set Adapter Sinn, um eine plattformunabhängige Aufrufsyntax der CORBA Schnittstelle zu gewährleisten (vgl. Kapitel 5.3).

Dieselben Elemente sind auch für Set Adapter spezifiziert. Push Adapter dürfen zusätzlich das Element `<thresholds>...</thresholds>` beinhalten (vgl. Kapitel 5.4.1 und Anhang C.1).

Besonderheiten bei Push Adaptern

Wie in 5.4.1 erwähnt, besitzen Push Adapter zusätzlich das Konfigurationselement `<thresholds>...</thresholds>`. Die Integrations- und Konfigurationsschicht hat die Möglichkeit eine Push Adapter Source so zu konfigurieren, dass diese einen Messwert nur dann weiterleitet, wenn er sich innerhalb vorgegebener Schranken befindet. Dabei kann sowohl eine obere als auch eine untere Schranke für den entsprechenden Messwert definiert werden. Die Angabe einer Schranke ist optional. Um willkürlich gewählten Schranken einer Integrations- und Konfigurationsschicht vorzubeugen, muss das `<thresholds>...</thresholds>`-Element verwendet werden. Innerhalb des `<source>...</source>`-Elements der Konfigurationsdateien von Push Sourcen ist dieses Element verpflichtend. Es stellt sicher, dass stets Grenzwerte definiert sind, so dass die Integrations- und Konfigurationsschicht nicht die Möglichkeit erhält, unsinnige Push-Ereignisse auszulösen und eine Push Source unnötig zu belasten. Ein Beispiel wäre einer Adapter Source, die die gegenwärtige CPU-Auslastung überwacht, die Grenzwerte 0% bzw. 100% vorzugeben. Dieses Constraint ist für einige Push Sourcen natürlich nicht sinnvoll. Es besteht wohl kaum ein Anwendungsfall dafür, die Anzahl der ausgehenden Pakete einer Netzschnittstelle zu beschränken. Die Generalisierung dieses Constraints trägt aber dazu bei, unbeabsichtigte Fehlkonfiguration zu vermeiden und leichter zu entdecken. Für Push Sourcen, die keine Grenzwerte benötigen, wird das Element `<thresholds>...</thresholds>` ignoriert. Das `<thresholds>...</thresholds>`-Element beinhaltet die Kindelemente `<max_allowed>...</max_allowed>` und
`<min_allowed>...</min_allowed>`, um eine maximal erlaubte untere bzw. obere Schranke zu definieren. Für die Schranken wird derselbe Datentyp angenommen, wie der der im `<src_type>...</src_type>`-Element angegeben wurde.

5.4.2 Ausführung von Adaptern

Wie in Kapitel 3.1 erwähnt, existieren drei verschiedene Arten von Adaptern. Hier soll kurz gezeigt werden, wie die drei Beispielimplementierungen ausgeführt werden können. Für die drei Adaptertypen existiert derzeit ein einziges Server-Testprogramm, das je nach Aufrufparameter den entsprechenden Adapter startet. Soll dieses Programm für neue Adaptertypen verwendet werden, muss es unter Umständen erweitert werden. Für die Implementierung sourcenspezifischer Adapter sollten eigene Programme geschrieben werden (vgl. Kapitel 5.1). Als Grundlage kann hier der Programmcode des Server-Testprogramms verwendet werden.

Um das Programm testen zu können wurde für jeden Adaptertyp ein Client-Testprogramm implementiert. Da die Integrations- und Konfigurationschicht derzeit noch nicht implementiert ist, simuliert es den Zugriff auf die drei verschiedenen Adaptertypen. Den Client-Testprogrammen wird als Parameter eine XML-basierte Java-Properties-Datei übergeben, die je nach Adaptertyp und den von ihm verwalteten Adapter Sourcen mit geeigneten Properties zu füllen ist.

Die allgemeine Aufrufsyntax des Server-Test Programms lautet

```
1 user@machine:~/> ./start-server adapter_type config_file
```

Dabei gibt `adapter_type` den zu startenden Adaptertyp an. Möglich Typen sind

- `pull_adapter`
- `push_adapter` und
- `<set>_adapter`

Da es sich bei Set Adaptern um Adapter handelt, die für jeden Ressourcentyp eine eigene Methodensignatur anbieten, muss `<set>` durch den entsprechenden Typ ersetzt werden. Derzeit ist nur das Beispiel eines Mount Adapters implementiert, für welchen der Adaptertyp dann `mount_adapter` lautet. Sollen weitere Adaptertypen mit dem Programm `start-server` gestartet werden, muss dazu die Klasse `AdapterServer` aus dem Paket `de.lmu.ifi.nm.smona.adapter.server` erweitert werden.

Der Parameter `config_file` referenziert eine XML-basierte Konfigurationsdatei, wie sie in Kapitel 5.4.1 behandelt wurde.

Die in den folgenden Beispielen verwendete Verzeichnisstruktur wird in Anhang B erläutert.

Pull Adapter Nachdem die Konfiguration für einen Pull Adapter vorgenommen wurde (vgl. Kapitel 5.4.1) kann dieser mit dem Kommando `start-server` gestartet werden.

```
1 user@machine:~/> ./start-server pull_adapter etc/pull_adapter.xml
```

Die Datei `pull_adapter.xml` enthält derzeit Definitionen für zwei Beispiel Adapter Sourcen. Es kann die derzeitige CPU-Auslastung sowie der auf dem Interface `eth0` eingehende IP-Datenverkehr abgefragt werden. Zur Abfrage der CPU-Auslastung kann das Client-Testprogramm mit der entsprechenden Java-Properties-Datei `test/pull_cpu_load.xml` aufgerufen werden.

```
1 user@machine:~/> ./start-client test/pull_cpu_load.xml
```

Die Properties-Datei enthält Definitionen für `Property`-Objekte, die dann im Client-Testprogramm erzeugt werden (vgl. Kapitel 5.2.1).

Listing 5.4: Konfiguration für einen Client zum Testen der Adapter Source CPULoad eines Pull Adapters

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
3  <properties version="1.0">
4      <comment>
5          This is to test the cpu pull adapter capabilities.
6          The following keys may be specified:
7          adapter_type: the type of the adapter to generate a Test Client for
8          source_id: The name of the source which it was published with
9                     Modify this value to check the exception handling
10         interval:  The maximum Amount of time an adapter should wait
11                    for the source to return a result before sending an
12                    error message. Modify this value to check the
13                    threshold boundaries defined in the adapter configuration
14                    files.
15     </comment>
16     <entry key="adapter_type">pull_adapter</entry>
17     <entry key="source_id">PullAdapter.A/.../.../CPULoad.S/float.ST</en try>
18     <entry key="measurement_type">float</entry>
19     <entry key="interval">2000</entry>
20 </properties>
```

Das Client-Testprogramm führt einige Aufrufe auf die Adapter Source aus. Dabei werden auch unsinnige Aufrufe durchgeführt, um die korrekte Behandlung von Exceptions zu demonstrieren. Die Konsolenausgabe sollte selbsterklärend sein. Sollen Aufrufe geändert werden muss derzeit das Client-Testprogramm [3] editiert werden.

Push Adapter Push Adapter werden mit dem Kommando

```
1 user@machine:~/> ./start-server push_adapter etc/push_adapter.xml
```

gestartet.

Die Datei `push_adapter.xml` enthält dieselben Definitionen von Adapter Sourcen wie das vorangegangene Beispiel von Pull Adaptern. Zur Abfrage der CPU Auslastung kann das Client-Test Programm mit der Java Properties Datei `test/push_cpu_load.xml` aufgerufen werden.

```
1 user@machine:~/> ./start-client test/push_cpu_load.xml
```

[3] Das Testprogramm für Pull Adapter Sourcen ist in `de.lmu.ifi.nm.smona.adapter.client.PullTestClient` definiert.

Die entsprechende Beispiel-Properties-Datei definiert zusätzliche `Property`-Objekte um eine Funktion zu definieren, die auf die Quelldaten angewandt werden soll, sowie eine obere Schranke, die vorgibt, wann die Integrations- und Konfigurationsschicht über die gegenwärtige CPU-Auslastung informiert werden soll.

Listing 5.5: Konfiguration für einen Test-Client zum Testen der Adapter Source CPULoad eines Push Adapters

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties version="1.0">
    <comment>
        This is to test the cpu push adapter capabilities.
        The sepearation of the function entry is function specific.
        E.g. "geometric_mean:10" means to excecute function
        "geometric_mean:10" with a history of 10 values
    </comment>
    <entry key="adapter_type">push_adapter</entry>
    <entry key="source_id">PushAdapter.A/.../.../CPULoad.S/float.ST</entry>
    <entry key="measurement_type">float</entry>
    <entry key="interval">2000</entry>
    <entry key="threshold_max">90.8</entry>
    <entry key="function">arithmetic_mean:10</entry>
</properties>
```

Set Adapter Als Beispiel für einen Set Adapter wird hier der Mount Adapter angeführt. Um diesen zu starten, kann das Server-Testprogramm folgendermaßen aufgerufen werden:

```
user@machine:~/> ./start-server mount_adapter etc/set_adapter.xml
```

Die Datei `set_adapter.xml` enthält derzeit alle Definitionen für Set Adapter Sourcen. Um einen Mount-Vorgang zu simulieren, kann das Client-Testprogramm mit der Java-Properties-Datei `test/set_mount_fs.xml` aufgerufen werden.

```
user@machine:~/> ./start-client test/set_mount_fs.xml
```

Die entsprechende Beispiel-Properties-Datei definiert keine zusätzliche `Property`-Objekte. Da Set Adapter in ihrer Schnittstelle die zu Grunde liegende Plattform möglichst nicht wiederspiegeln sollen, werden die einzuhängenden Dateisysteme in der Konfiguration der zuständigen Adapter Source gekapselt. Damit muss natürlich jede Dateisystemquelle eines Systems im Naming-Service bekannt geben werden, was die Definition eines geeigneten Naming-Schemas unumgänglich macht (vgl. Kapitel 4.1.4).

Listing 5.6: Konfiguration für einen Test-Client zum Testen der Adapter Source Mount eines Set Adapters

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties version="1.0">
    <comment>This is to test the set_mount_fs adapter capabilities</comment>
    <entry key="adapter_type">mount_adapter</entry>
    <entry key="source_id">MountAdapter.A/.../.../StringArray.ST</entry>
    <entry key="interval">1000</entry>
</properties>
```

5.5 Erweiterungen für Adapter

Das Adapter-Framework ist so konzipiert, dass es auf verschiedenen Wegen erweitert werden kann. Generell stehen dazu drei Möglichkeiten zur Verfügung:

- Hinzufügen von Datenquellen in der Konfigurationsdatei
- Hinzufügen einer neuen Adapter Source
- Hinzufügen eines neuen Adaptertyps

Diese Möglichkeiten werden in den folgenden Abschnitten erläutert.

5.5.1 Erweitern der Adapterkonfiguration

Die einfachste Methode, die Funktionalität eines Adapters zu erweitern, besteht darin, in der entsprechenden Konfigurationsdatei zusätzliche Adapter Sourcen zu definieren. Will man beispielsweise ein existierendes Programm oder Skript als Ressource bzw. Management-Werkzeug einbinden, lässt sich dies oft schon durch einen zusätzliche `<section>...</section>`-Eintrag bewerkstelligen (vgl. Kapitel. 5.4.1). Ein einfaches Beispiel wäre das Hinzufügen einer Adapter Source, die die eingehenden IP-Pakete auf dem Interface `eth0` eines Routers $node1$ überwachen soll. Dazu könnte folgendes XML-Fragment der entsprechenden Adapter Source hinzugefügt werden:

```
...
<source id="PullAdapter.A/../node1.HN/ifi\.lmu\.de.DN/InPktEth0.S/int.ST">
    <src_type>float</src_type>
    <description>Adapter Source to pull Packet Count Type on eth0</description>
    <interval>
        <type>milliseconds</type>
        <max_allowed>3000</max_allowed>
        <min_allowed>0</min_allowed>
    </interval>
    <process>
        <cmd>bin/system_info/pkt_count</cmd>
        <cmd_params>
            <param>-o</param>
            <param>eth0</param>
        </cmd_params>
    </process>
</source>
...
```

Damit diese Konfiguration verwendet werden kann, wird natürlich vorausgesetzt, dass ein entsprechendes Skript oder Programm `pkt_count` existiert und der Pfad dorthin korrekt spezifiziert wurde. Müssen Programmparameter in der korrekten Reihenfolge angegeben werden, muss diese Reihenfolge auch für die `<param>...</param>`-Elemente eingehalten werden.

Listing 5.7: Beispiel Skript zum Überwachen der ein- und ausgehenden IP Pakete

```
#!/bin/bash
# This example script reports an integer value containing the
# number of pakets received or transmitted  by an interface
#
# parameter:
# -i or -o for incomming/outgoing packets
# ethX (or sth. similar) for interface X

if [ ! \( $1 = "-o" -o $1 = "-i" \) -o x$2 = x ]
then
   echo "usage: $0 {-i|-o} interface" >&2
   exit 1
fi

line=$(cat /proc/net/dev | grep $2)
if [ x = "x$line" ]
```

```
17 then
18    echo "Error: Interface does not exist" >&2
19    exit 2
20 fi
21 if [ $1 = "-o" ]
22 then
23    # compare /proc/net/dev to understand output
24    echo $line | awk '{print $11}'
25 else
26    # compare /proc/net/dev to understand output
27    echo $line | awk '{print $3}'
28 fi
```

Natürlich kann es vorkommen, dass eine fehlerhafte Konfiguration vorliegt. Dabei müssen zwei Fälle unterschieden werden:

Programm nicht vorhanden oder nicht ausführbar Sollte ein Aufruf auf ein Kommando erfolgen, das nicht gefunden oder ausgeführt werden kann, wird eine `NoSuchSourceException` geworfen, die auf die entsprechende Fehlkonfiguration des Adapters hinweist. Dieses Verhalten sollte in einer zukünftigen Version abgeändert werden (vgl. Kapitel 6).

Falsche Parameter angegeben In diesem Fall liefert die entsprechende Adapter Source eine Fehlermeldung mit dem von dem aufgerufenen Programm generierten Fehlercode zurück. Die Rückgabe verläuft über einen asynchronen Aufruf auf das Callback-Objekt, das während der Adapter Sourcen-Konfiguration übergeben wurde (vgl. Kapitel 5.2.1).

5.5.2 Hinzufügen einer neuen Adapter Source

Eine etwas komplexere Variante, einem Adapter zusätzliche Funktionalität hinzuzufügen, besteht darin, die entsprechende Adapter Sourcen-Klasse zu erweitern. Diese Beschreibung bezieht sich auf Pull und Push Sourcen, Set Source bedürfen einer gesonderten Erweiterung (vgl. Kapitel 5.5.3). Ein Beispiel wäre die Einführung eines weiteren Datentyps `exampleType` für einen Pull Adapter, der eine Sequenz bestehend aus einer Integer-Zahl und einem String liefert.

Schreiben einer IDL Definition und Erzeugen der Java Stubs und Skeletons

Da es sich in diesem Beispiel um keinen einfachen Datentyp handelt, muss zuerst eine Struktur definiert werden, die von den Adaptern zur Integrations- und Konfigurationsschicht gesendet werden kann.

Listing 5.8: IDL-Definition des Datentyps ExampleType

```
1  module de {
2  module lmu {
3  module ifi {
4  module nm {
5  module smona {
6  module complexType {
7
8      /**
9      This struct is a holder for the ExampleType
10     */
11     struct ExampleType {
12         long value;
13         string str;
14     };
15 };
16 };
17 };
```

```
18 };
19 };
20 };
```

Diese Definition wird benutzt, um mit dem JacoORB Kommando `idl` die entsprechenden Java Stubs und Skeletons zu erzeugen, die wir später zur Implementierung der `fillAny()`-Methode benötigen (vgl. Kapitel 5.5.2):

```
1 user@machine:~/> idl idl/ExampleType.idl
```

Erweitern der Klasse `PullSource`

Nun muss die Klasse `PullExampleTypeSource` erzeugt werden, welche die abstrakten Klasse `PullSource` aus dem Paket `de.lmu.ifi.nm.smona.pull.impl` erweitert.

Listing 5.9: Code-Skelett der Klasse`PullExampleTypeSource`

```
1  package de.lmu.ifi.nm.smona.pull.impl;
2
3  import java.io.*;
4  import org.omg.CORBA.*;
5  import de.lmu.ifi.nm.smona.Property;
6  import de.lmu.ifi.nm.smona.SmonaAdapterPackage.*;
7  import de.lmu.ifi.nm.smona.adapter.server.*;
8  import de.lmu.ifi.nm.smona.adapter.server.env.*;
9  import de.lmu.ifi.nm.smona.complexType.*;
10
11 public class PullExampleTypeSource extends PullSource {
12     protected void fillAny(BufferedReader in, Any any) throws IOException,
13         UselessValueException {
14     }
15 }
```

Hier sind nur zwei Erweiterungen durchzuführen:

1. Zunächst muss ein Konstruktor geschrieben werden, der lediglich den Konstruktor der Elternklasse mit den entsprechenden Parametern aufruft.

```
1 public PullExampleTypeSource(ORB orb, String source_id, Property[] properties,
2         BasicSmonaAdapter adapter, SourceEnviroment src_env)
3         throws InvalidPropertyValueException, NoSuchPropertyException,
4         MissingPropertyException {
5     super(orb, source_id, properties,adapter, src_env);
6 }
```

2. Desweiteren muss noch die abstrakte Methode `fillAny()`, die von der Klasse PullSource vererbt wird, implementiert werden. Die Methode erledigt die Auswertung der Ausgabe des aufgerufenen Kommandos und das Verpacken des Ergebnisses in den CORBA-Typ `Any`. Es sei darauf hingewiesen, dass die Integrations- und Konfigurationsschicht, die das `Any`-Objekt wieder entpacken soll, wissen muss, wie die Daten dieses Typs verpackt wurden. Um zu vermeiden, dass das Vorwissen über die Art dieses Typs der Integrations- und Konfigurationsschicht bekannt sein muss, soll in Zukunft der Aufruf der `getSelfDescription()`-Methode des CORBA-Interface `SmonaAdapter.idl` die entsprechenden Informationen liefern (vgl. Kapitel 4.1.3 und 5.4.1). Angenommen das aufgerufenen Kommando liefert eine Ausgabe der Form

```
1 123  eins_zwei_drei
```

   dann könnte der Code, der diese Ausgabe für den Typ `exampleType` formatiert, folgendermaßen lauten:

```
protected void fillAny(BufferedReader in, Any any) throws IOException {
    String str = in.readLine();

    String[] result = str.trim().replaceAll("␣␣*", ":").split(":");
    int value = Integer.parseInt(result[0]);
    // check whether a function has to be applied
    if(func_obj != null) {
        value = func_obj.calculate(value);
    }
    String s = result[1];
    //fill Any
    ExampleType type = new ExampleType(value, s);
    ExampleTypeHelper.insert(any, type);
}
```

Erweitern der Klasse `PullSourceFactory`

Da ein neuer Datentyp eingeführt werden soll, muss natürlich auch die entsprechende Fabrik (Factory) für diesen Typ erweitert werden. Dafür reicht es, folgendes Code-Segment der Methode `createSource()` der Klasse `PullSourceFactory` (aus dem Paket `de.lmu.ifi.nm.smona.pull.impl`) hinzuzufügen:

```
...
else if(src_env.getType().equals(Constants.EXAMPLE_TYPE_SOURCE)) {
    // create PullStringSource instance
    return new PullExampleTypeSource(
        orb, source_id, properties, adapter, src_env);
}
...
```

Die Konstante `EXAMPLE_TYPE_SOURCE` in der Klasse `Constants`, wird im Paket `de.lmu.ifi.nm.smona.adapter` definiert, zum Beispiel in der Form:

```
...
public static final String EXAMPLE_TYPE_SOURCE = "exampleType";
...
```

Erweiterung des XML-Schemas

Damit beim Parsen einer Konfigurationsdatei, die den neuen Datentyp enthält, keine Fehler erzeugt werden, muss dem XML-Schema noch der entsprechende Datentyp bekannt gegeben werden:

Listing 5.10: Auszug aus dem XML-Schema zur Validierung von Konfigurationsdateien (vgl. Anhang C.1)

```
...
<!-- allowed source types for adapter -->
    <xsd:simpleType name="sourceTypes">
        <xsd:restriction base="xsd:normalizedString">
            <xsd:enumeration value="float"></xsd:enumeration>
            ...
            <!-- new type -->
            <xsd:enumeration value="exampleType"></xsd:enumeration>
            ...
        </xsd:restriction>
    </xsd:simpleType>
...
```

Erweitern der Konfigurationsdatei

Den letzten Schritt stellt die Erweiterung der Konfigurationsdatei des Adapters dar. Für dieses Beispiel könnte folgendes XML Fragment dem Dokument hinzugefügt werden:

```
<source id="PullAdapter.A/../../PullExample.S/pullExample.ST">
    <src_type>exampleType</src_type>
    <description>Adapter Source to pull exampleType type</description>
    <interval>
        <type>milliseconds</type>
        <max_allowed>3000</max_allowed>
        <min_allowed>0</min_allowed>
    </interval>
    <process>
        <cmd>bin/system_info/example_output</cmd>
        <cmd_params>
            <param>123</param>
            <param>eins_zwei_drei</param>
        </cmd_params>
    </process>
</source>
```

Besonderheiten beim Erweitern von Push Source

Für Push Sourcen gelten die meisten Erweiterungsregeln wie für Pull Sourcen. Wie bei der Erweiterung einer Pull Source müssen auch bei einer Push Source die entsprechenden Konfigurationsdateien, das XML-Schema sowie die zur `PullSourceFactory` äquivalenten Klasse `PushSourceFactory` angepasst werden. Die Erweiterung der zur `PullSource` äquivalenten Klasse `PushSource` gestaltet sich jedoch etwas aufwändiger. Grund dafür ist, dass bei Push Sourcen auch die Definition von unteren und oberen Schranken erlaubt ist, die im Konstruktor ausgewertet werden müssen bzw. in der `fillAny()`-Methode besonderer Behandlung bedürfen. Im Folgenden soll die Verwendung des in Kapitel 5.5.2 besprochenen IDL-Datentyps `ExampleType` als Beispiel für einen komplexen Push-Datentyp behandelt werden. Dabei soll der Integer-Wert einer oberen und einer unteren Schranke genügen. Der String-Wert wird nur geparst und zusammen mit dem Integer Wert in ein Objekt vom Typ `ExampleType` verpackt.

Listing 5.11: Code-Skelett für Konstruktor und Attribute der Klasse `PushExampleTypeSource`

```
public class PushExampleTypeSource extends PushSource {
    ...
    private int max_threshold;
    private int min_threshold;
    private boolean check_max_threshold = false;
    private boolean check_min_threshold = false;
    private FunctionObject func_obj = null;

    public PushExampleTypeSource(
            ORB orb, String source_id, Property[] properties,
            BasicSmonaAdapter adapter, SourceEnviroment src_env)
            throws InvalidPropertyValueException, NoSuchPropertyException,
            MissingPropertyException{

        // create a boolean array to remember checked an unchedcked properties
        boolean[] isUnchecked = new boolean[3];
        Arrays.fill(isUnchecked, true);

        // validate properties
        for(Property a_prop: properties) {
            // source_id
            ...
```

```
23              // interval
24              ...
25              // listener
26              ...
27              // threshold_max
28              ...
29              // threshold_min
30              ...
31              // function
32              ...
33          }
34          // has every property been checked
35          ...
36          // check whether max_threshold > min_threshold if given
37          ...
38          super.orb = orb;
39          super.adapter = adapter;
40          super.src_env = src_env;
41          super.setName(getClass().getName() + "-Thread");
42      }
43      ...
44  }
```

Der Konstruktor überprüft zunächst, ob alle obligatorischen Properties vorhanden sind. Dazu gehören:

- die `source_id`, unter der diese Adapter Source im CORBA-Naming-Service publiziert wird,
- ein `interval`, das vorgibt, wie lange maximal auf die Rückkehr eines externen Programmaufrufs gewartet werden soll, ehe ein Fehler an die Integrations- und Konfigurationsschicht gesendet wird, sowie
- ein `listener`-Objekt, welches als Callback-Objekt dient, um der Integrations- und Konfigurationsschicht asynchrone Nachrichten zu vermitteln.

Um zuletzt überprüfen zu können, ob alle obligatorischen Werte in den Properties mitgeliefert wurden bzw. Duplikate von `Property`-Objekten zu bemerken, wird in dem `boolean`-Array `isUnchecked[]` vermerkt, welche Properties schon gelesen wurden. Das Code-Fragment, welches diese drei Properties ausliest könnte folgendermaßen aussehen:

```
1   if(a_prop.name.equals(SOURCE_ID.value)  && isUnchecked[0]) {
2       super.p_source_id = a_prop.value.extract_string();
3       if(! super.p_source_id.equals(source_id)) {
4           throw new InvalidPropertyValueException(
5                   "ambiguous␣sources␣specified", Constants.ERROR_CODE);
6       }
7       isUnchecked[0] = ! isUnchecked[0];
8   }
9   else if(a_prop.name.equals(INTERVAL.value)  && isUnchecked[1]) {
10      super.interval = a_prop.value.extract_long();
11      if(interval < Integer.parseInt(src_env.getMinAllowedInterval()) ||
12              interval > Integer.parseInt(src_env.getMaxAllowedInterval())) {
13          throw new InvalidPropertyValueException(
14                  "Interval␣out␣of␣bounds", Constants.ERROR_CODE);
15      }
16      isUnchecked[1] = ! isUnchecked[1];
17  }
18  else if(a_prop.name.equals(LISTENER.value)  && isUnchecked[2]) {
19      super.listener = PushListenerHelper.extract(a_prop.value);
20      if(listener == null) {
21          throw new InvalidPropertyValueException(
22          "No␣listener␣given", Constants.ERROR_CODE);
23      }
```

```
24          isUnchecked[2] = ! isUnchecked[2];
25  }
```

Bei dem Attribut `src_env` handelt es sich um eine Instanz der Klasse `SourceEnviroment`, die die Vorgaben für die unteren und oberen Schranken von Intervallen und anderen datenquellenspezifischen Grenzwerten verwaltet (vgl. Kapitel 5.1 und Anhang C.2). In diesem Beispiel soll für das eingelesene Intervall außerdem getestet werden, ob es sich innerhalb der in der Konfigurationsdatei (vgl. Abschnitt 5.4.1) vorgegebenen Schranken befindet. Hierfür werden die Attribute

- `max_threshold`,
- `min_threshold`,
- `check_max_threshold` und
- `check_min_threshold`

verwendet. `min_threshold` und `max_threshold` dienen dazu, die untere und die obere Schranke zu speichern. Sind die Schranken jeweils angegeben, wird dies in den Attributen vom Typ boolean `check_min_threshold` bzw. `check_max_threshold` vermerkt, um später innerhalb der `fillAny()`-Methode entscheiden zu können, welche Werte an die Integrations- und Konfigurationsschicht gesendet werden sollen und welche nicht.

```
1  else if(a_prop.name.equals(PushAdapter.THRESHOLD_MAX)) {
2      this.max_threshold = a_prop.value.extract_long();
3      if(max_threshold > Integer.parseInt(src_env.getMaxAllowedThreshold()) ||
4              max_threshold < Integer.parseInt(src_env.getMinAllowedThreshold())) {
5          throw new InvalidPropertyValueException(
6                  "Invalid max_threshold value: "
7                  + max_threshold, Constants.ERROR_CODE);
8      }
9      else {
10         this.check_max_threshold = true;
11     }
12 }
13 else if(a_prop.name.equals(PushAdapter.THRESHOLD_MIN)) {
14     this.min_threshold = a_prop.value.extract_long();
15     if(min_threshold > Integer.parseInt(src_env.getMaxAllowedThreshold()) ||
16             min_threshold < Integer.parseInt(src_env.getMinAllowedThreshold())) {
17         throw new InvalidPropertyValueException
18                 "Invalid min_threshold value: "
19                 + min_threshold, Constants.ERROR_CODE);
20     }
21     else {
22         this.check_min_threshold = true;
23         log.debug("Minimum threshold will be checked");
24     }
25 }
```

Die letzte Property, die in diesem Beispiel ausgewertet werden soll, ist die `function`-Property. Ist diese vorhanden, wird entsprechend dem enthaltenen Wert eine Instanz der Klasse `AdapterFunction` generiert, die dazu dient, Historien in die Ergebnisse eines Push Adapters einzubeziehen. Derzeit sind Funktionen zur Berechnung des geometrischen sowie des arithmetischen Mittels implementiert (vgl. dazu auch Kapitel 6).

```
1  else if(a_prop.name.equals(FUNCTION_TYPE.value)) {
2      // get Function
3      AdapterFunction function = AdapterFunctionHelper.extract(a_prop.value);
4      // create the Function object
5      func_obj = FunctionFactory.create(Constants.INTEGER_SOURCE, function);
6  }
```

Es wird noch überprüft, ob alle für obligatorisch erklärten Properties eingelesen wurden und die angegebenen Schranken sich nicht gegenseitig ausschließen.

```
if(isUnchecked[0] || isUnchecked[1] || isUnchecked[2]) {
    log.warn("Properties missing - adapter source not started");
    throw new MissingPropertyException(
            "Property missing", Constants.ERROR_CODE);
}
if(check_max_threshold && check_min_threshold && min_threshold > max_threshold) {
    throw new InvalidPropertyValueException(
            "Invalid thresholds: min_threshold > max_threshold",
            Constants.ERROR_CODE);
}
```

Alle weiteren im Konstruktor übergebenen Parameter werden zur Initialisierung der Attribute der Elternklasse verwendet.

Es fehlt noch die Implementierung der `fillAny()`-Methode der Klasse `PushExampleTypeSource`:

Listing 5.12: Implementierung der `fillAny()`-Methode

```
protected void fillAny(BufferedReader in, Any any) throws IOException,
        UselessValueException {
    String str = in.readLine();
    String[] result = str.trim().replaceAll("  *", ":").split(":");
    int value = Integer.parseInt(result[0]);
    String s = result[1];
    if(func_obj != null) {
        value = func_obj.calculate(value);
    }
    // check whether this value is in bounds
    boolean sendVal = false;
    if(check_max_threshold) {
        if(value > max_threshold) {
            sendVal = true;
        }
    }
    if(check_min_threshold) {
        if(value < min_threshold) {
            sendVal = true;
        }
    }
    if(sendVal || ! (check_max_threshold || check_min_threshold)) {
        // fill Any
        ExampleType type = new ExampleType(value, s);
        ExampleTypeHelper.insert(any, type);
    }
    else {
        throw new UselessValueException("Value in bounds: " + value);
    }
}
```

Zunächst wird in der Methode `fillAny()` die Ausgabe des aufgerufenen Prozesses ausgewertet (`in.readLine()`) und den entsprechenden Variablen zugewiesen. Daraufhin wird überprüft, ob sich die gemessenen Werte innerhalb der zuvor festgelegten Schranken befinden, falls diese festgelegt wurden. Wurde die Property `function` eingelesen, wird nun noch die entsprechende Funktion auf das Ergebnis aufgerufen, bevor zuletzt der Datentyp `ElementType` erzeugt und in ein CORBA-`Any`-Objekt verpackt wird.

5.5.3 Hinzufügen eines neuen Adaptertyps

Die wohl aufwendigste Methode, einen Adapter um neue Adapter Sourcen zu erweitern, ist die Definition eines neuen Adaptertyps. Für Push und Pull Adapter ist derzeit kein weiterer Anwendungsfall ersichtlich, für den eine derartige Erweiterung sinnvoll erscheint. Denkbar wäre ein hybrider Adaptertyp der sowohl Pull- als auch Push-Funktionalitäten besitzt, wofür derzeit aber auch kein Anwendungsfall existiert. Für das Hinzufügen von Set Adapter Sourcen muss stets dieser Erweiterungsansatz verfolgt werden. Anhand eines einfachen Beispiels soll kurz auf die wesentlichen Schritte bei diesem Vorgehen eingegangen werden. Dennoch ist es unumgänglich, sich für diesen Fall intensiv mit dem vorhandenen Programm-Code auseinander zu setzen. In den folgenden Abschnitten wird erläutert, wie der einfache Set Adapter `IPForwardAdapter` implementiert wird, der auf einem Internet-Host IP-Forwarding ein- bzw. ausschaltet.

Definition einer neuen Adapter-IDL

Set Adapter sollen für jede Adapter Source, die damit angesprochen werden soll, eine eigene datenquellenspezifische CORBA-Schnittstelle zur Verfügung stellen. Dazu ist es notwendig, die IDL-Schnittstelle `SmonaAdapter.idl` zu erweitern (vgl. Anhang C.3).

Listing 5.13: IDL-Definition des Set Adapters `IPForwardAdapter`

```
1  #include "SetAdapter.idl"
2
3  module de {
4  module lmu {
5  module ifi {
6  module nm {
7  module smona {
8  module set {
9  module ipForward {
10
11     interface IPForwardAdapter : SetAdapter {
12
13         /** enable ip forwarding
14         */
15         void enable(in string source_id)
16             raises (NoSuchSourceException, AdapterNotConfiguredException);
17
18         /** disable ip forwarding
19         */
20         void disable(in string source_id)
21             raises (NoSuchSourceException, AdapterNotConfiguredException);
22     };
23 };
24 };
25 };
26 };
27 };
28 };
29 };
```

Damit das `IPForwardAdapter`-Interface auch die Methoden und Konstanten, die jeder Adapter mit sich bringen muss, zur Verfügung stellt, erweitert das `IPForwardAdapter`-Interface das `SetAdapter`-Interface aus der IDL `SetAdapter.idl`. `SetAdapter` seinerseits erweitert das `SmonaAdapter`-Interface aus der IDL `SmonaAdapter.idl`. Das `IPForwardAdapter`-Interface stellt lediglich die Methode `enable()` zum Einschalten und die Methode `disable()` zum Ausschalten von IP-Forwarding zur Verfügung. Die Methoden können für den Fall, dass eine Ressource mit dem Namen `source_id` nicht existiert, die Exception `NoSuchSourceException` werfen. Sollte eine der Methoden auf eine Adapter Source aufgeru-

fen werden, die zuvor nicht gestartet, d.h. konfiguriert wurde, kann die Exception `AdapterNotConfigured-Exception` geworfen werden. Mit dem Aufruf

```
user@machine:~/> idl idl/IPForwardAdapter.idl
```

werden die entsprechenden Stub-, -Helper- und Holderklassen erzeugt.

Implementierung der Klasse `IPForwardSource`

Wie alle anderen Adapter greifen auch Set Adapter auf die Kapselung ihrer Funktionalität in Adapter Sourcen zurück (vgl. Kapitel 3.2). Die eigentliche Ausführung des Ein- bzw Ausschaltens von IP-Forwarding wird in der Klasse `IPForwardSource` implementiert. Diese Klasse erweitert die Klasse `SetSource`.

Listing 5.14: Skelett der Klasse `IPForwardSource`

```
package de.lmu.ifi.nm.smona.set.ipForward.impl;

import java.io.BufferedReader;
import java.io.IOException;

import org.omg.CORBA.Any;
import org.omg.CORBA.ORB;

import de.lmu.ifi.nm.smona.Property;
import de.lmu.ifi.nm.smona.SmonaAdapterPackage.InvalidPropertyValueException;
import de.lmu.ifi.nm.smona.SmonaAdapterPackage.MissingPropertyException;
import de.lmu.ifi.nm.smona.SmonaAdapterPackage.NoSuchPropertyException;
import de.lmu.ifi.nm.smona.adapter.server.BasicSmonaAdapter;
import de.lmu.ifi.nm.smona.adapter.server.UselessValueException;
import de.lmu.ifi.nm.smona.adapter.server.env.SourceEnviroment;
import de.lmu.ifi.nm.smona.set.impl.SetSource;

public class IPForwardSource extends SetSource {

    public void enable() {
        // code to implement
    }

    public void disable() {
        // code to implement
    }

    protected void fillAny(BufferedReader in, Any any) throws IOException,
            UselessValueException {
        // code to implement
    }
}
```

Da diese Klasse keine besonderen Properties erwartet, muss kein eigener Konstruktor geschrieben werden. Ein gesonderter Konstruktor wäre notwendig, wenn Properties erwartet würden, die zur Initialisierung von bestimmten Attributen verwendet werden müssten. Diese könnten zum Beispiel in den Methoden `enable()` oder `disable()` benötigt werden. Hier kann jedoch direkt der `super()`-Konstruktor aufgerufen werden.

Listing 5.15: Konstruktor der Klasse `IPForwardSource`

```
public IPForwardSource(ORB orb, String source_id, Property[] properties,
        BasicSmonaAdapter adapter, SourceEnviroment src_env)
        throws InvalidPropertyValueException, NoSuchPropertyException,
        MissingPropertyException {
    // no special properties -> call super() constructor
```

```
6       super(orb, source_id, properties, adapter, src_env);
7   }
```

Anders als man vermuten möchte, dienen die Methoden `enable()` und `disable()` nur dazu, das entsprechende Kommando zu initialisieren, welches später von der Klasse `BasicIPForwardAdapter` ausgeführt wird (vgl. Kapitel 4.2 und 5.5.3).

Listing 5.16: Methoden `enable()` und `disable()` der Klasse `IPForwardSource`

```
1   public void enable() {
2       String cmd_str = super.src_env.getCmd();
3       // the first parameter of the <cmd_params>...</cmd_params>
4       // section of the corresponding <source>...</source> section of the
5       // set_adapter.xml configuration file contains the value
6       // to enable ip forwarding -> index 0 of the String array args provides
7       // the argument to enable ip forwarding
8       String[] args = super.src_env.getCmdParams();
9       String enableStr = args[0];
10      this.exec_cmd = cmd_str + "␣" + enableStr;
11  }
12
13  public void disable() {
14      String cmd_str = super.src_env.getCmd();
15      // the second parameter of the <cmd_params>...</cmd_params>
16      // section of the corresponding <source>...</source> section of the
17      // set_adapter.xml configuration file contains the value
18      // to disable ip forwarding -> index 1 of the String array args provides
19      // the argument to disable ip forwarding
20      String[] args = super.src_env.getCmdParams();
21      String disableStr = args[1];
22      this.exec_cmd = cmd_str  + "␣" + disableStr;
23  }
```

Das Kommando `cmd_str`, welches ausgeführt werden soll, sowie die dazugehörigen Parameter (`enableStr` bzw. `disableStr`) werden beim Start des Adapters aus der entsprechenden XML-Konfigurationsdatei `set_adapter.xml` eingelesen und in einer Instanz der Klasse `SourceEnviroment` abgelegt (vgl. Kapitel. 4.2 und 5.5.3). Um das Kommando zu initialisieren, wird von dieser Instanz mit dem Methodenaufruf `super.src_env.getCmdParams()` ein `String[]`-Array geliefert, das die zuvor eingelesenen Parameter enthält. Die Parameter der Konfigurationsdatei müssen in der Reihenfolge definiert sein, wie sie die entsprechende Adapter Source erwartet. Das Einlesen der Parameter mag etwas umständlich anmuten. Der Grund für dieses Vorgehen liegt darin, dass Adapter natürlich auf verschiedenen Plattformen angewendet werden sollen, was voraussetzt, dass auch plattformspezifischen Kommandos mit kommandospezifischen Parameterlisten zum Einsatz kommen können. In diesem Fall müsste also der Parameter, der in die Variable `enableStr` eingelesen wird, dem Parameter vorausgehen, der in die Variable `disableStr` eingelesen wird. Ein Auszug aus der entsprechenden Konfigurationsdatei könnte folgendermaßen aussehen (vgl. Kapitel 5.5.3):

```
1   <set_adapter>
2       ....
3      <source id="...">
4           ...
5           <process>
6               <cmd>bin/system_info/ip_forward</cmd>
7               <cmd_params>
8                   <param>enable</param>
9                   <param>disable</param>
10              </cmd_params>
11          </process>
12      </source>
13      ....
14  </set_adapter>
```

Zuletzt muss noch die obligatorische Methode `fillAny()` implementiert werden, deren Aufgabe es ist, die Ausgabe des Kommandos an die Integrations- und Konfigurationsschicht zu übermitteln. Die Methode `fillAny()` wird nur aufgerufen, wenn die Ausführung des Kommandos erfolgreich war. Für Set Sourcen spielt diese Ausgabe eine weniger wichtige Rolle, da sie nur Informationen auswerten kann, die das aufgerufenen Kommando auf der Standardausgabe liefert. Für dieses Beispiel soll genau diese Ausgabe an die Integrations- und Konfigurationsschicht zurückgeliefert werden.

Listing 5.17: Implementierung der `fillAny()`-Methode der Klasse `IPForwardSource`

```
1  protected void fillAny(BufferedReader in, Any any) throws IOException {
2      StringBuffer buffer = new StringBuffer();
3      String line = null;
4      while((line = in.readLine()) != null) {
5          buffer.append(line);
6          buffer.append("\n");
7      }
8      any.insert_string(buffer.toString());
9  }
```

Implementierung der `BasicIPForwardAdapter` Klasse

Als nächstes muss der Adapter selbst implementiert werden. Um dies zu bewerkstelligen wird die Klasse `BasicIPForwardAdapter` erzeugt, welche die Klasse `BasicSetAdapter` erweitert und das Interface `IPForwardAdapterOperations` implementiert.

Listing 5.18: Code-Skelett der Klasse `BasicIPForwardAdapter`

```
1  package de.lmu.ifi.nm.smona.set.ipForward.impl;
2
3  import org.omg.CORBA.Any;
4  import de.lmu.ifi.nm.smona.SmonaAdapterPackage.AdapterNotConfiguredException;
5  import de.lmu.ifi.nm.smona.SmonaAdapterPackage.NoSuchPropertyException;
6  import de.lmu.ifi.nm.smona.SmonaAdapterPackage.NoSuchSourceException;
7  import de.lmu.ifi.nm.smona.set.impl.BasicSetAdapter;
8  import de.lmu.ifi.nm.smona.set.ipForward.IPForwardAdapterOperations;
9
10 public class BasicIPForwardAdapter extends BasicSetAdapter
11         implements IPForwardAdapterOperations {
12
13     public String getVersion() {
14         //code to implement
15     }
16
17     public void disable(String source_id) throws NoSuchSourceException,
18             AdapterNotConfiguredException {
19         //code to implement
20     }
21
22     public void enable(String source_id) throws NoSuchSourceException,
23             AdapterNotConfiguredException {
24         //code to implement
25     }
26 }
```

Der Konstruktor muss für die neue Klasse nicht erweitert werden. Es reicht aus den `super()` Konstruktor der Klasse `SetAdapter` aufzurufen. Dieser übernimmt die korrekte Initialisierung der notwendigen Attribute.

```
1  public BasicIPForwardAdapter(
2          ORB orb, AdapterSourceFactory factory, AdapterEnviroment env) {
```

```
3       super(orb, factory, env);
4   }
```

Die Methode `getVersion()` sollte einen String zurückliefern, der die aktuelle Versionsnummer des Adapters enthält. Anhand der Version können Kompatibilitätsüberprüfungen mit zukünftigen Versionen durchgeführt werden. Es könnte beispielsweise der Fall eintreten, dass die Integrations- und Konfigurationsschicht versucht, eine Methode auf einen Adapter aufzurufen, die er noch nicht bzw. nicht mehr besitzt.

```
1   public String getVersion() {
2       return "1.0";
3   }
```

Etwas mehr Aufmerksamkeit verdienen die beiden verbleibenden Methoden `enable()` und `disable()`. Beide besitzen einen ähnlichen Programm-Code, so dass die folgende Beschreibung auf beide Methoden zutrifft.

Listing 5.19: Methode `enable()` der Klasse `BasicIPForwardAdapter`

```
1   public void enable(String source_id) throws NoSuchSourceException,
2           AdapterNotConfiguredException {
3       // check, whether such an adapter source was published
4       if(! env.isAdapterSource(source_id)) {
5           throw new NoSuchSourceException(
6               "Source␣<" + source_id + ">␣does␣not␣exist", Constants.ERROR_CODE);
7       }
8       IPForwardSource source;
9       synchronized(source_map) {
10          source = (IPForwardSource)super.source_map.get(source_id);
11      }
12      if(source == null) {
13          throw new AdapterNotConfiguredException("Adapter␣<"
14                  + source_id + ">␣not␣configured", Constants.ERROR_CODE);
15      }
16      // set the cmd to execute
17      source.enable();
18      // trigger set of source
19      // this call returns immediately
20      source.triggerExec();
21
22  }
```

Zunächst wird überprüft, ob die angegebene `source_id` den Namen enthält, unter welchem die Adapter Source (`IPForwardSource`) im CORBA-Naming-Service publiziert wurde. Anschließend wird versucht, die entsprechende Instanz der Klasse `IPForwardSource` in der Hash-Tabelle `src_map` aufzufinden (vgl. Kapitel 4.2). Schließlich wird die Methode `enable()` auf die Instanz der Klasse `IPForwardSource` aufgerufen, was dazu führt, dass das Kommando zum Einschalten des IP-Forwardings initialisiert wird (vgl. Kapitel 5.5.3). Die Ausführung des Kommandos erledigt die Methode `triggerExec()`, die die Klasse `IPForwardSource` von der Klasse `SetSource` erbt.

Die Methode `disable()` unterscheidet sich von der Methode `enable()` nur an der Stelle, die die Initialisierung des auszuführenden Kommandos bewirkt.

Listing 5.20: Methode `disable()` der Klasse `BasicIPForwardAdapter`

```
1   public void disable(String source_id)
2           throws NoSuchSourceException, AdapterNotConfiguredException {
3       // see method enable()
4       ...
5       // set the cmd to execute
6       source.disable();
7       // trigger set of source
```

```
8       // this call returns immediately
9       source.triggerExec();
10  }
```

Erweitern der Klasse `SetSourceFactory`

Der Klasse `SetSourceFactory` muss noch der neue Ressourcentyp bekannt gemacht werden. Dazu muss die Methode `createSource()` erweitert werden:

```
1   public AdapterSource createSource(String source_id, Property[] properties,
2           BasicSmonaAdapter adapter) throws InvalidPropertyValueException,
3           NoSuchPropertyException, NoSuchSourceException, MissingPropertyException {
4
5       ...
6       // new Source
7       else if(src_env.getType().equals(Constants.IP_FORWARD_SOURCE)) {
8
9           // create IPForwardSource instance
10          return new IPForwardSource(orb, source_id, properties, adapter, src_env);
11      }
12      ...
13  }
```

Beispielskript zur Konfiguration des IP-Forwarding

Auf einem Linux-System wir IP-Forwarding über das `/proc`-Verzeichnis aktiviert. Je nachdem, ob der Kernel in der Datei `/proc/sys/net/ipv4/ip_forward` eine `1` bzw. eine `0` vorfindet, wird IP-Forwarding aktiviert bzw. deaktiviert. Ein Skript, welches das Ein- bzw. Ausschalten des IP-Forwarding bewirkt, muss lediglich diesen Wert verändern. Es sei darauf hingewiesen, dass ein solches Skript mit `root`-Rechten ausgeführt werden muss.

Listing 5.21: Beispielskript zum Ein- bzw. Ausschalten von IP-Forwarding auf einem Linux-System

```
1   #!/bin/bash
2   # This example script en/disables ip forwarding
3   #
4   # parameter:
5   #    enable
6   #    disable
7   if [ x$1 = "xenable" ]
8   then
9      /bin/echo "1" > /proc/sys/net/ipv4/ip_forward
10  elif [ x$1 = "xdisable" ]
11  then
12     /bin/echo "0" > /proc/sys/net/ipv4/ip_forward
13  else
14     echo "usage: $0 {enable|disable}" >&2
15     exit 1
16  fi
```

Erweiterung der Konfigurationsdatei `set_adapter.xml`

Natürlich muss auch für diese Art der Erweiterung die zugehörige Konfigurationsdatei angepasst werden. Dafür kann folgendes XML Fragment, als zusätzliche Adapter Source der Datei `set_adapter.xml` hinzugefügt werden.

```
...
<source id="IPForwardAdapter.A/.../.../IPForward.S/EMPTY.ST">
    <src_type>ip_forward</src_type>
    <description>Adapter Source to enable/disable ip forwarding</description>
    <interval>
        <type>milliseconds</type>
        <max_allowed>5000</max_allowed>
        <min_allowed>0000</min_allowed>
    </interval>
    <process>
        <cmd>bin/system_info/ip_forward</cmd>
        <cmd_params>
            <param>enable</param>
            <param>disable</param>
        </cmd_params>
    </process>
</source>
...
```

Damit keine Probleme während der Validierung der Konfigurationsdatei entstehen, gilt es, den neuen Ressourcentyp auch im XML-Schema bekannt zu geben. Dazu genügt ein Eintrag der Form

```
<xsd:enumeration\texttt{value="ip_forward"></xsd:enumeration>}
```

in der Datei `Adapter.xsd` (vgl. Kapitel 5.5.2).

Erzeugen eines IP-Forward-Adapters

Um den neuen Adaptertyp zum Leben zu erwecken muss dieser korrekt initialisiert und publiziert werden. Das Schema der Instantiierung der verschiedenen Objekte kann in Kapitel 4.2 nachgelesen werden.

Listing 5.22: Programm-Code zum Erzeugen eines IP-Forward-Adapters

```
package de.lmu.ifi.nm.smona.adapter.server;

import org.apache.log4j.Logger;
import org.omg.CORBA.*;
import org.omg.PortableServer.*;
import org.omg.CosNaming.*;
import java.util.*;

import de.lmu.ifi.nm.smona.adapter.*;
import de.lmu.ifi.nm.smona.adapter.server.env.*;
import de.lmu.ifi.nm.smona.set.ipForward.*;
import de.lmu.ifi.nm.smona.set.ipForward.impl.*;
import de.lmu.ifi.nm.smona.set.impl.*;

public class IPForwardAdapterServer {
    // perform logging
    static Logger log = Logger.getLogger(IPForwardAdapterServer.class);

    public IPForwardAdapterServer() {
        try {
            String cfg_file = "etc/set_adapter.xml";
            Properties props = System.getProperties();
            ORB orb = ORB.init(new String[0], props);
            //get root POA
            POA rootPOA = POAHelper.narrow(
                    orb.resolve_initial_references("RootPOA"));
            // get Name Service
```

```
28              NamingContextExt rootNC = NamingContextExtHelper.narrow(
29                      orb.resolve_initial_references("NameService"));
30              //create a SmonaAdapterPublisher instance
31              AdapterPublisher publisher = new SmonaAdapterPublisher(rootNC);
32              // create a SetEnviroment
33              AdapterEnviroment set_env = null;
34              if(cfg_file.endsWith(Constants.XML_FILE)) {
35                  // we compare with Constants.SET_ADAPTER, because all
36                  // specific adapters inherit from set...
37                  set_env = new XmlSetEnviroment(Constants.SET_ADAPTER, cfg_file);
38              }
39              else{
40                  System.exit(Constants.INVALID_CONFIGURATION);
41              }
42              // create SetSourceFactory instance
43              AdapterSourceFactory factory = new SetSourceFactory(orb, set_env);
44              // create the example SetAdapter instance IPForwardAdapter
45              IPForwardAdapterPOATie a_adapter = new IPForwardAdapterPOATie(
46                      new BasicIPForwardAdapter(orb, factory, set_env));
47              // register SetAdapter with orb
48              rootPOA.activate_object(a_adapter);
49              IPForwardAdapter adapter = IPForwardAdapterHelper.narrow(
50                      rootPOA.servant_to_reference(a_adapter));
51              // puplish sources through naming service
52              publisher.registerSources(adapter, set_env);
53              // activate POA manager
54              rootPOA.the_POAManager().activate();
55              orb.run();
56          } catch (Exception e) {
57              e.printStackTrace();
58          }
59      }
60
61      public static void main(String[] args) {
62          new IPForwardAdapterServer();
63      }
64  }
```

Den so erzeugten Adapter startet man dann z.B mit folgendem Kommando:

```
1  user@machine:~/> jaco -cp lib/xercesImpl.jar:lib/log4j-1.2.14.jar:$CLASSPATH \
2      -DORBInitRef.NameService=corbaloc::localhost:5000/NameService \
3      de.lmu.ifi.nm.smona.adapter.server.IPForwardAdapterServer
```

5.5.4 Hinweise zum Einbinden von eigenen Klassen

Bestimmte Adapter Sourcen sollten nicht darauf basieren nur ein externes Kommando aufzurufen um mit Ressourcen zu interagieren. Vorstellbar wäre es anstelle des Aufrufes einer `exec()`-Methode der Klasse `java.lang.Runtime` einen Classloader zu beauftragen, die Klassen zu laden, die die entsprechenden Ressourcen auslesen bzw. konfigurieren [4]. Derzeit ist kein solches Beispiel implementiert, da es nicht trivial ist, eine Abstraktionsebene einzuführen, die einen möglichst generischen Zugriff auf solche Klassen ermöglicht. Für den Aufruf einer `exec()`-Methode der Klasse `java.lang.Runtime` wird immer ein String übergeben, der das auszuführende Kommando beinhaltet. Der Rückgabewert ist stets ein Objekt vom Typ `InputStream`. Will man eine derartige Erweiterung in das Adapter-Framework einbinden, muss auf jeden Fall der gleiche Ansatz verfolgt werden wie bei der Implementierung der asynchronen Kommunikation für den Aufruf von externen Kommandos (vgl. Kapitel 5.2.4). Es existieren keine “sauberen” Möglichkeiten

[4] Hier könnte auch das JavaBeans-Framework [Jav] von Nutzen sein.

Java-Threads mit einem Kommando zu stoppen [OaWo 99]. Um also einen Thread zu beenden, der aufgrund eines blockierenden Aufruf keine Invariante überprüfen kann, muss man sicherstellen, dass der blockierende Aufruf abgebrochen werden kann. Beispiele wären, neben dem Zerstören von externen Prozessen (vgl. Kapitel 5.2.4) das Schließen von `InputStreams` von Dateien oder Sockets mittels der Methode `destroy()`.

Kapitel 6

Zusammenfassung und Ausblick

Im Rahmen dieser Arbeit sollten Adapter für Datenquellen auf Linux-Systemen entwickelt werden. Dabei entstanden nicht einzelne Adapter, die für die Verwaltung einer spezifischen Datenquelle herangezogen werden können, vielmehr wurde ein Adapter-Framework geschaffen, welches auf einfache Art und Weise um neue Datenquellen erweitert werden kann (vgl. Abschnitt 4). Es wurden verschiedene Anwendungsfälle herausgearbeitet und in Folge dessen die drei Kategorien Pull, Push und Set Adapter definiert (vgl. Kapitel 3.1). Adapter, die mit diesem Framework realisiert werden können, basieren auf dem Aufruf externer Programme. Dabei kann es sich zum Beispiel um den Aufruf eines selbst angefertigten Skriptes handeln. Genauso könnten aber auch Management-Werkzeuge mit eingebunden werden, solange diese eine Schnittstelle zur Verfügung stellen, um die verwalteten Datenquellen über Kommandozeilenaufrufe anzusprechen. Da als Programmiersprache zur Implementierung des Frameworks Java verwendet wurde, kann das Framework prinzipiell ohne größere Anpassungen am Quellcode auch auf andere Plattformen portiert werden.

In Abschnitt 2.3 wurden einige Anforderungen formuliert, die Adapter generell erfüllen sollten. Das in dieser Arbeit entstandene Framework besitzt die meisten der geforderten Eigenschaften. Es wurde eine Schnittstelle definiert (vgl. Kapitel 4.1), die einen einheitlichen Zugriff auf alle Arten von Adaptern beschreibt. Alle durch das vorliegende Framework eingliederbaren Datenquellen können über diese Schnittstelle angesprochen werden und liefern ihre Ergebnisse in einer normalisierten Form. Zur Definition der Schnittstelle wurde CORBA-IDL verwendet. Damit ist es möglich weitere Adapter in allen anderen Programmiersprachen zu implementieren, für die eine Abbildung der IDL auf die entsprechende Sprache standardisiert wurde. Die Forderung nach asynchroner Kommunikation wird von allen Adaptern (Push, Pull und Set Adapter) erfüllt. Push und Pull Adapter besitzen die Fähigkeiten, Historien über die von ihnen verwalteten Datenquellen anzulegen. Damit ist es möglich, bei der Erstellung eines Messwertes Funktionen anzuwenden, wobei sowohl der aktuelle Messwert als auch die Werte der Historie in das Ergebnis mit einfließen können. Denkbare Funktionen wären das arithmetische oder das geometrische Mittel, die im Rahmen dieser Arbeit auch als Beispielfunktionen implementiert wurden. Die Integrations- und Konfigurationsschicht benötigt eine Möglichkeit, Informationen über die Fähigkeiten der zur Verfügung stehenden Adapter abzufragen. Obgleich derzeit noch kein Format für eine Selbstbeschreibung von Adaptern festgelegt ist, wurde die Schnittstelle so definiert, dass die Anfragemöglichkeit nach den Fähigkeiten von Adaptern auf einfache Art und Weise nachträglich integriert werden kann.

Im Verlauf dieser Arbeit wurde das Konzept eingeführt, Datenquellen auf Adapter Sourcen abzubilden (vgl. Kapitel 3.2). Dieses Konzept ermöglicht die einfache Integration neuer Datenquellen. Adapter können beliebig viele Adapter Sourcen verwalten, solange diese über einen externen Programmaufruf die zu verwaltenden Datenquellen ansprechen können. Solche Datenquellen müssen nicht durch die Implementierung eines neuen Adapters angebunden werden, sondern können einfach der Konfigurationsdatei des entsprechenden Adapters hinzugefügt werden und stehen somit sofort zur Verfügung. Alle Adapter Sourcen werden im CORBA-Naming-Service publiziert. Eine Adapter Source wird dynamisch erzeugt, wenn die Integrations- und Konfigurationsschicht eine Konfiguration an der entsprechenden Adapter Source vornehmen will. Damit sind also nur Adapter Sourcen aktiv, die auch wirklich gerade benötigt werden.

Eine Anforderung, auf die im Rahmen dieser Arbeit nicht eingegangen werden konnte, stellt die Authentifizierung der Integrations- und Konfigurationsschicht sowie die Autorisierung dieser gegenüber Adaptern dar. Derzeit kann prinzipiell jede Applikation, die Kenntnis von der Beschaffenheit der Schnittstelle besitzt, Adapter konfigurieren, starten, anfragen und stoppen. Theoretisch können Adapter auf einfache Art und Weise lahmgelegt werden, indem man sie einfach stoppt oder mit sinnlosen Anfragen überschüttet. In zukünftigen

Arbeiten sollten daher geeigneten Sicherheitsmechanismen integriert werden.

In Abschnitt 5.4.1 wurde darauf hingewiesen, dass natürlich nicht alle Management-Werkzeuge bzw. die von ihnen verwalteten Ressourcen über externe Programmaufrufe angesprochen werden können, um sie durch Adapter der Integrations- und Konfigurationsschicht zur Verfügung zu stellen. In zukünftigen Versionen sollte daher besonders Wert auf die Integration von Classloadern zum dynamischen Laden von Klassen gelegt werden, die solche Werkzeuge anbinden. Denkbar wären auch Erweiterungen basierend auf dem JavaBeans-Framework (vgl. Kapitel 5.5.4), um eine einfache Integration von zusätzlichen Komponenten zu bewerkstelligen. Schwierig erweist sich dabei die Definition einer einheitlichen Schnittstelle zur Anbindung solcher Komponenten, da die Attribute der unterschiedlichen Ressourcen stark variieren können und eine vollständige Übereinstimmung der Ressourcenattribute nur selten vorliegt.

Die derzeit implementierten Adapter besitzen die Fähigkeit, einfache Datentypen als Rückgabewert der angeforderten Information einer Datenquelle zu liefern. Dennoch wurde auch an Erweiterungen um komplexere Datenstrukturen bei der Konzeption und dem Entwurf des Adapter-Frameworks gedacht (vgl. Kapitel 5.5). Auch auf die Anwendung von Funktionen auf Messwerte und die zur Verfügung stehenden Historien wurde Rücksicht genommen. Es sind aber auch Szenarien denkbar, in denen Adapter nicht nur komplexe Datentypen oder gemittelte Werte zur Verfügung stellen, sondern unter Umständen auch Aggregationen von Informationen verschiedener Datenquellen vornehmen. Um beispielsweise die Rechenleistung eines Grid-Verbundes zu ermitteln, könnte die auf den einzelnen Rechnern zur Verfügung stehende Rechenleistung (Komplement der gegenwärtigen CPU-Auslastung) mit der Erreichbarkeit der jeweiligen Rechner (erfolgreiches Anpingen) korreliert werden. Das Sammeln dieser Informationen durch eine einzige Instanz würde wohl kaum skalieren. Um dennoch solche Szenarien mit Adaptern zu realisieren, wäre die Konzeption einer Adapterhierarchie sinnvoll. Vorstellbar wären eine Art Proxy-Adapter mit Aggregationsfunktionalitäten, denen ein Verbund von “primitiven” Subadaptoren zugeordnet werden kann. Die Berechnung der Rechenleistung könnte auf solche Proxy-Adapter abgewälzt werden, die ihrerseits die Einzelschritte der Berechnung auf die ihnen untergebenen Subadaptoren abbilden.

Einen letzten Punkt, dem in nachfolgenden Arbeiten verstärkte Aufmerksamkeit gewidmet werden sollte, stellt das zu verwendende Namensschema zur Adressierung von Adaptern dar. Es wurde im Verlauf dieser Arbeit an der Definition eines passenden Schemas gearbeitet. Leider konnte dieses nicht festgeschrieben werden, da derzeit noch kein konkreter Entwurf der Integrations- und Konfigurationschicht zur Verfügung steht und daher die vollständigen Anforderungen an ein Schema nicht gegeben waren. Ein Schema sollte so gewählt sein, dass es ein einfaches Auffinden von Adapter Sourcen ermöglicht und die Semantik der zugrundeliegenden Datenquelle wiederspiegelt.

In der Konzeption, dem Design sowie der Implementierung der vorliegenden Arbeit wurde stets versucht, eine klare Aufteilung der Aufgaben von Adaptern in Komponenten, Klassen und deren Methoden zu gewährleisten. Damit soll es auf einfache Art und Weise möglich sein, neue Datenquellen mit Hilfe von Adaptern schnell zur Verfügung zu stellen. Es bleibt zu hoffen, dass sich das gewählte Konzept und Design auch für nachfolgende Anwendungsfälle bewährt.

Anhang A

Hinweise zur Installation

Java Runtime Enviroment Das Adapter-Framework wurde mit JRE 1.5.0 getestet. Es sollte darauf geachtet werden, dass auch die Variablen `JRE_HOME` sowie `JAVA_HOME` richtig gesetzt sind, da JacORB diese verwendet um nach den Java Binaries zu suchen. Sollten diese Variablen auf eine ältere JRE verweisen, bricht JacORB mit dem Hinweis ab, dass die Version der `class`-Dateien nicht kompatibel mit der JRE ist.

Listing A.1: Ausgabe des Kommandos `java`

```
1 machine@user:~/> java -version
2 java version "1.5.0_09"
3 Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0_09-b03)
4 Java HotSpot(TM) Client VM (build 1.5.0_09-b03, mixed mode)
```

Ant Build Tool Es wird empfohlen, dass JacORB aus den Sourcen kompiliert wird, da dann alle Pfade und Kommandos plattformabhängig erzeugt werden und keine weiteren Konfigurationen an JacORB vorgenommen werden müssen. Dazu wird Apache Ant verwendet. Zum kompilieren wurde Version 1.6.5 Verwendet.

Listing A.2: Ausgabe des Kommandos `ant`

```
1 machine@user:~/> ant -version
2 Apache Ant version 1.6.5 compiled on October 26 2005
```

JacORB JacORB ist eine freie Implementierung des OMG CORBA Standards. Hier wurde die Beta-Version 2.3.0 verwendet (`JacORB_2.3.0_beta2`). Es ist darauf zu achten, das Verzeichnis `$JACORB_HOME/bin` in den Programm Pfad aufzunehmen.

Listing A.3: Ausgabe des Kommandos `jaco`

```
1 machine@user:~/> jaco -version
2 java version "1.5.0_09"
3 Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0_09-b03)
4 Java HotSpot(TM) Client VM (build 1.5.0_09-b03, mixed mode)
```

Die Bibliotheken der JacORB Implementierung 2.3.0 Beta wurden auch in das Verzeichnis `lib` aufgenommen (vgl. Anhang B). Soll mit dieser Version gearbeitet werden, muss JacORB nicht zusätzlich installiert werden. Wird eine andere Version bevorzugt, müssen die Skripten zum Ausführen der Adapter geändert werden.

Adapter Um die Binaries der Adapter zu generieren muss im Wurzelverzeichnis lediglich ein `make`-Aufruf ausgeführt werden. Sollen vorhandene `class`-Dateien zuvor entfernt werden. sollte dem `make`-Aufruf ein `make`
`class-clean` bzw. ein `make dist-clean` vorausgehen.

Anhang B

Bestandteile der Adapter

Folgende Dateien und Verzeichnisse sind im Quellcode des Adapter Pakets enthalten

```
1  +- bin/
2  +- clear-ns
3  +- de/
4  +- etc/
5  +- idl/
6  +- lib/
7  +- list-ns
8  +- Makefile
9  +- ns_db/
10 +- start-client
11 +- start-ns
12 +- start-server
13 +- test/
```

`bin` Dieses Verzeichnis und die darin enthaltenen Unterverzeichnisse beherbergen alle Skripten, die für die Implementierungszwecke als Beispielwerkzeuge verwendet wurden.

`clear-ns` Mit dem Aufruf dieses Skripts werden die CORBA Naming Kontexte gelöscht, die die JacORB Implementierung des CORBA Naming Service nach seiner Beendigung angelegt hat.

`de` Dieses Verzeichnis und die darin enthaltenen Unterverzeichnisse beherbergen den Quellcode der Adapter Implementierung

`etc` Alle Konfigurationsbeispiele die zum Testen der Adapter verwendet wurden befinden sich in diesem Verzeichnis (siehe auch `start-server`).

`idl` In diesem Ordner sind alle Schnittstellendefinitionen abgelegt, die zur Kommunikation mit den Adapter dienen.

`lib` Alle verwendeten Bibliotheken mit Ausnahme des Java Development Kits sind hier abgelegt

`list-ns` Falls der CORBA Naming Service aktiv ist, ermöglicht dieses Skript die formatierte Anzeige der im CORBA Naming Service publizierten Objekte.

`Makefile` Das Makefile zum Löschen bzw. Erzeugen der Java Stubs und Skeletons aus den IDL Definitionen sowie aller Java class-Dateien.

`ns_db` Dieses Verzeichnis wird bei der Beendigung der JacORB Implementierung des CORBA Naming Service verwendet, die gegenwärtigen Objektreferenzen und ihre Namen zu speichern

`start-client` Dieses Skript kann parametrisiert aufgerufen werden um Testbeispiele für die Implementierung der Adapter zu starten (siehe auch `test`).

`start-ns` Der Aufruf dieses Skriptes bewirkt das Starten der JacORB Implementierung des CORBA Naming Service

`start-server` Um möglichst schnell, viele Adapter zu testen, kann dieses Skript parametrisiert aufgerufen werden. Es ermöglicht das Ausführen von Pull, Push und Set Adapter mit unterschiedlichen Konfigurationen (siehe auch `etc`).

test Dieses Verzeichnis beherbergt alle Konfigurationsdateien für das Ausführen der Testbeispiele (siehe auch `start-client`).

Der Quellcode für die Adapter ist im Verzeichnis `de` zu finden. Es werden nur die Pakete erläutert, die nicht automatisch durch den JacORB IDL Compiler generiert werden:

de.lmu.ifi.nm.smona.adapter Dieses Paket dient nur zur Aufspannung eines neuen Namenraumes für Adapter. Die einzige Klasse innerhalb des Paketes definiert alle Konstanten, auf die die anderen Klassen zurückgreifen.

de.lmu.ifi.nm.smona.adapter.client Da noch keine Implementierung der Integrations- und Konfigurationsschicht vorliegt, wurden einige Testklassen geschrieben, um die Komponenten *Rich Event Composer* und *Adapter Configurator* (vgl. Kapitel 2.1) zu simulieren. Diese können leicht wiederverwendet werden um neue Adaptertypen anzubinden und zu Testen. Die Testklassen werden über Java-Properties-Dateien konfiguriert.

de.lmu.ifi.nm.smona.adapter.server Dieses Paket enthält die Kernkomponenten des Adapter Frameworks. Dazu gehören Schnittstellendefinitionen von Fabrikklassen, die dazu dienen, Instanzen von `AdapterSourcen` zu generieren, Klassen die dazu dienen `AdapterSourcen` beim CORBA Naming Service zu publizieren, Klassen die dazu dienen, die Logik der asynchronen Kommunikation zu realisieren sowie Klassen, die dazu dienen, Server Instanzen zum Testen von Adaptern zu erzeugen.

de.lmu.ifi.nm.smona.adapter.server.env Um die Konfiguration eines Adapters einzulesen und zu verwalten, stellt dieses Paket die entsprechenden Klassen zur Verfügung. Außerdem sind hier Klassen enthalten, die als Umgebungsobjekte für die dynamisch erzeugten Instanzen vom Typ `AdapterSource` verwendet werden können.

de.lmu.ifi.nm.smona.function.lib Sollen die Instanzen der dynamisch erzeugten `AdapterSourcen` mit der Verwaltung von Historien umgehen, können Instanzen der dazu gehörigen Funktionen von den Klassen dieses Pakets generiert werden. Dazu gehören Funktionen wie das arithmetische oder das geometrische Mittel.

de.lmu.ifi.nm.smona.pull.impl Dieses Paket beinhaltet sowohl die Implementierungen der durch den JacORB IDL Compiler bereitgestellten Java Skeletons, also auch die Implementierungen aller Typen von Adapter Sourcen, die für Pull Adapter benötigt werden. Zusätzlich beherbergt das Paket die Implementierung der Fabrikklasse zur dynamischen Erzeugung von Pull Adapter Sourcen.

de.lmu.ifi.nm.smona.push.impl Dieses Paket beinhaltet sowohl die Implementierungen der durch den JacORB IDL Compiler bereitgestellten Java Skeletons, also auch die Implementierungen aller Typen von Adapter Sourcen, die für Push Adapter benötigt werden. Zusätzlich beherbergt das Paket die Implementierung der Fabrikklasse zur dynamischen Erzeugung von Push Adapter Sourcen.

de.lmu.ifi.nm.smona.set.impl Anders als die Pakete `de.lmu.ifi.nm.smona.pull.impl` und `de.lmu.ifi.nm.smona.push.impl` enthält dieses Paket nur abstrakte Implementierung der Basisfunktionalitäten von Set Adapter Sourcen. Alle konkreten Implementierungen werden in separaten Paketen untergebracht, da Set Adapter nur geringfügige Gemeinsamkeiten in den Signaturen der anwendbaren Methoden besitzen. Wie die beiden zuvor besprochenen Pakete besitzt dieses auch eine konkrete Implementierung der Fabrikklasse zur dynamischen Erzeugung von Set Adapter Sourcen.

`de.lmu.ifi.nm.smona.set.ipForward.impl` Dieses Paket beinhaltet die konkrete Implementierung eines Set Adapters um das IP-Forwarding auf einem Internet Host zu aktivieren. Dazu werden die abstrakten Klassen des Pakets `de.lmu.ifi.nm.smona.set.impl` sowie die durch den JacORB IDL Compiler bereitgestellten Java Skeletons erweitert.

`de.lmu.ifi.nm.smona.set.mount.impl` Dieses Paket beinhaltet die konkrete Implementierung eines Set Adapters um auf einem Rechner Dateisysteme ein- oder auszuhängen. Dazu werden die abstrakten Klassen des Pakets `de.lmu.ifi.nm.smona.set.impl` sowie die durch den JacORB IDL Compiler bereitgestellten Java Skeletons erweitert.

Anhang C

Auszüge aus dem Programm-Code

C.1 XML-Schema zur Validierung von Konfigurationsdateien

Listing C.1: Konfigurationsdatei für einen Pull Adapter

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:annotation>
        <xsd:documentation xml:lang="de">
            A Pretty Simple Adapter Configuration Schema
        </xsd:documentation>
    </xsd:annotation>

    <!-- adapter and childs pull_adapter, push_adapter and set_adapter -->
    <xsd:element name="adapter">
        <xsd:complexType>
            <xsd:complexContent>
                <xsd:restriction base="xsd:anyType">
                    <xsd:choice>
                        <xsd:element
                            name="pull_adapter"
                            type="pullAndSetAdapterType"
                            minOccurs="1"
                            maxOccurs="1">
                        </xsd:element>
                        <xsd:element
                            name="push_adapter"
                            type="pushAdapterType"
                            minOccurs="1"
                            maxOccurs="1">
                        </xsd:element>
                        <xsd:element
                            name="set_adapter"
                            type="pullAndSetAdapterType"
                            minOccurs="1"
                            maxOccurs="1">
                        </xsd:element>
                    </xsd:choice>
                    <xsd:attribute
                        name="id"
                        type="xsd:string"
                        use="required">
                    </xsd:attribute>
                </xsd:restriction>
            </xsd:complexContent>
        </xsd:complexType>
    </xsd:element>

```

```
44      <!-- complex type for pull and set adapter types -->
45      <xsd:complexType name="pullAndSetAdapterType">
46          <xsd:complexContent>
47              <xsd:restriction base="xsd:anyType">
48                  <xsd:sequence>
49                      <xsd:element name="source" maxOccurs="unbounded">
50                          <xsd:complexType>
51                              <xsd:sequence>
52                                  <xsd:element
53                                      name="src_type"
54                                      type="sourceTypes">
55                                  </xsd:element>
56                                  <xsd:element
57                                      name="description"
58                                      type="xsd:string"></xsd:element>
59                                  <xsd:element
60                                      name="interval"
61                                      type="intervalType">
62                                  </xsd:element>
63                                  <xsd:choice>
64                                      <xsd:element
65                                          name="process"
66                                          type="processType">
67
68                                      </xsd:element>
69                                  </xsd:choice>
70                              </xsd:sequence>
71                              <xsd:attribute
72                                  name="id"
73                                  type="xsd:string"
74                                  use="required">
75                              </xsd:attribute>
76                          </xsd:complexType>
77                      </xsd:element>
78                  </xsd:sequence>
79              </xsd:restriction>
80          </xsd:complexContent>
81      </xsd:complexType>
82
83      <!-- complex push adapter type -->
84      <xsd:complexType name="pushAdapterType">
85          <xsd:complexContent>
86              <xsd:restriction base="xsd:anyType">
87                  <xsd:sequence>
88                      <xsd:element name="source" maxOccurs="unbounded">
89                          <xsd:complexType>
90                              <xsd:sequence>
91                                  <xsd:element
92                                      name="src_type"
93                                      type="sourceTypes">
94                                  </xsd:element>
95                                  <xsd:element
96                                      name="description"
97                                      type="xsd:string">
98                                  </xsd:element>
99                                  <xsd:element
100                                     name="interval"
101                                     type="intervalType">
102                                 </xsd:element>
103                                 <xsd:element
104                                     name="thresholds"
```

```
105                                 type="thresholdsType">
106                             </xsd:element>
107                             <xsd:choice>
108                                 <xsd:element
109                                     name="process"
110                                     type="processType">
111 
112                                 </xsd:element>
113                             </xsd:choice>
114                         </xsd:sequence>
115                         <xsd:attribute
116                             name="id"
117                             type="xsd:string"
118                             use="required">
119                         </xsd:attribute>
120                     </xsd:complexType>
121                 </xsd:element>
122             </xsd:sequence>
123         </xsd:restriction>
124     </xsd:complexContent>
125 </xsd:complexType>
126 
127 <!-- allowed source types for adapter -->
128 <xsd:simpleType name="sourceTypes">
129     <xsd:restriction base="xsd:normalizedString">
130         <xsd:enumeration value="float"></xsd:enumeration>
131         <xsd:enumeration value="int"></xsd:enumeration>
132         <xsd:enumeration value="string"></xsd:enumeration>
133         <xsd:enumeration value="long"></xsd:enumeration>
134         <xsd:enumeration value="short"></xsd:enumeration>
135         <xsd:enumeration value="double"></xsd:enumeration>
136         <xsd:enumeration value="boolean"></xsd:enumeration>
137         <!-- example for complex data type  -->
138         <xsd:enumeration value="exampleType"></xsd:enumeration>
139 
140         <!-- special types for set adapters
141         ADD FURTHER SET ADAPTER TYPES HERE -->
142         <xsd:enumeration value="mount"></xsd:enumeration>
143         <xsd:enumeration value="ip_forward"></xsd:enumeration>
144     </xsd:restriction>
145 </xsd:simpleType>
146 
147 <!-- childs of interval element -->
148 <xsd:complexType name="intervalType">
149     <xsd:sequence>
150         <xsd:element
151             name="type"
152             type="intervalUnit">
153         </xsd:element>
154         <xsd:element
155             name="max_allowed"
156             type="xsd:positiveInteger">
157         </xsd:element>
158         <xsd:element
159             name="min_allowed"
160             type="xsd:nonNegativeInteger">
161         </xsd:element>
162     </xsd:sequence>
163 </xsd:complexType>
164 
165 <!-- restrictions for child type of interval element -->
```

```
166     <xsd:simpleType name="intervalUnit">
167         <xsd:restriction base="xsd:normalizedString">
168             <xsd:enumeration value="milliseconds"></xsd:enumeration>
169             <!-- not implemented yet
170             <xsd:enumeration value="seconds"></xsd:enumeration>
171             -->
172         </xsd:restriction>
173     </xsd:simpleType>
174 
175     <!-- childs of process element -->
176     <xsd:complexType name="processType">
177         <xsd:sequence>
178             <xsd:element
179                 name="cmd"
180                 type="xsd:normalizedString">
181             </xsd:element>
182             <xsd:element name="cmd_params" minOccurs="0" maxOccurs="1">
183                 <xsd:complexType>
184                     <xsd:sequence>
185                         <xsd:element maxOccurs="unbounded"
186                             name="param"
187                             type="xsd:normalizedString">
188                         </xsd:element>
189                     </xsd:sequence>
190                 </xsd:complexType>
191             </xsd:element>
192         </xsd:sequence>
193     </xsd:complexType>
194 
195     <!-- childs of threshold element -->
196     <xsd:complexType name="thresholdsType">
197         <xsd:sequence>
198             <xsd:element
199                 name="max_allowed"
200                 type="xsd:decimal">
201             </xsd:element>
202             <xsd:element
203                 name="min_allowed"
204                 type="xsd:decimal">
205             </xsd:element>
206         </xsd:sequence>
207         <!-- for push adapters with boolean threshold implement this
208         <xsd:choice>
209             <xsd:sequence>
210                 <xsd:element
211                     name="max_allowed"
212                     type="xsd:decimal">
213                 </xsd:element>
214                 <xsd:element
215                     name="min_allowed"
216                     type="xsd:decimal">
217                 </xsd:element>
218             </xsd:sequence>
219             <xsd:element
220                 name="trigger_bool"
221                 type="xsd:boolean">
222             </xsd:element>
223         </xsd:choice>
224         -->
225     </xsd:complexType>
226 </xsd:schema>
```

C.2 Die Klasse SourceEnviroment

Listing C.2: Diese Klasse speichert die Umgebung für eine Adapter Source

```
1  package de.lmu.ifi.nm.smona.adapter.server.env;
2
3  public class SourceEnviroment {
4
5      private String source_id;
6      private String type;
7      private String max_allowed_interval;
8      private String min_allowed_interval;
9      private String interval_type;
10     private String max_allowed_threshold;
11     private String min_allowed_threshold;
12     private String cmd;
13     private String cmd_params;
14     private String exec_type;
15
16     // pull/set source
17     public SourceEnviroment(String a_source_id, String type,
18             String max_allowed_interval, String min_allowed_interval,
19             String interval_type, String cmd,
20             String[] cmd_params, String exec_type) {
21         this.source_id = a_source_id;
22         this.type = type;
23         this.max_allowed_interval = max_allowed_interval;
24         this.min_allowed_interval = min_allowed_interval;
25         this.interval_type = interval_type;
26         this.cmd = cmd;
27         this.cmd_params = paramToString(cmd_params);
28         this.exec_type = exec_type;
29     }
30
31     // push source
32     public SourceEnviroment(String a_source_id, String type,
33             String max_allowed_interval, String min_allowed_interval,
34             String interval_type, String max_allowed_threshold,
35             String min_allowed_threshold, String cmd, String[] cmd_params,
36             String exec_type) {
37         this.source_id = a_source_id;
38         this.type = type;
39         this.max_allowed_interval = max_allowed_interval;
40         this.min_allowed_interval = min_allowed_interval;
41         this.interval_type = interval_type;
42         this.max_allowed_threshold = max_allowed_threshold;
43         this.min_allowed_threshold =  min_allowed_threshold;
44         this.cmd = cmd;
45         this.cmd_params = paramToString(cmd_params);
46         this.exec_type = exec_type;
47     }
48
49     public String getType() {return type;}
50     public String getCmd() {return cmd;}
51     public String getCmdParam() {return cmd_params;}
52     public String getMaxAllowedInterval() {return max_allowed_interval;}
```

```
53      public String getMinAllowedInterval() {return min_allowed_interval;}
54      public String getIntervalType() {return interval_type;}
55      public String getSourceId() {return source_id;}
56      public String getMaxAllowedThreshold() {return max_allowed_threshold;}
57      public String getMinAllowedThreshold() {return min_allowed_threshold;}
58      public String getExecType() {return exec_type;}
59      private String paramToString(String[] cmd_params) {
60          StringBuffer buf = new StringBuffer();
61          for(int i = 0; i < cmd_params.length; i++) {
62              buf.append(cmd_params[i] + "␣");
63          }
64          return buf.toString();
65      }
66  }
```

C.3 IDLs für die Adapter Typen Pull, Push und Set

Listing C.3: Dieses Modul enthält alle Konstanten, Exceptions und Interfaces, die allen Adaptern gemein sind

```
1   /** Package for SMONA Adapters.
2   This package includes all idl type definitions that describe the interfaces
3   shared between the Platform Independent Layer (Adapter Layer) and the
4   Integration/Configuration Layer. All Adapter specific interfaces should be in
5   or below this package.
6   */
7   module de {
8   module lmu {
9   module ifi {
10  module nm {
11  module smona {
12
13      /** constants that may be used as key for Property structures (see the
14      definition of Property structs below). The corresponding values of the
15      Properties defined by these constants are described for each constant:
16      */
17
18      /** source_id:
19      An identifier for the source attribute that the adapter uses as a data
20      source (e.g the name string this adapter was published within the naming
21      service). An adapter uses this identifier to non-ambiguously determine the
22      source attribute to interact with.
23      */
24      const string SOURCE_ID = "source_id";
25
26      /** listener:
27      A callback object to notify the Integration/Configuration Layer. This
28      callback object may serves different purposes:
29      - deliver push events:
30          if an implementing adapter is intended to push values, this object will
31          be called to relay such events
32      - non-blocking method calls:
33          if an implementing adapter is intended to serve non-blocking calls, the
34          Integration/Configuration Layer receives callbacks through this
35          listener. Even methods without return values will generate a callback
36          to notify the Integration/Configuration Layer about an (un-)successful
37          method excecution (i.e. raise exceptions).
38      */
```

```
39      const string LISTENER = "listener";
40
41      /** interval:
42      The interval may serves two different purposes:
43      - adapter sampling interval:
44          if an implementing adapter is intended to push values periodically,
45          this value provides the interval in milliseconds a sample is sent to
46          the Integration/Configuration Layer.
47      - max interval to excecute method call:
48          if an implementing adapter is intended to serve non-blocking calls, the
49          Integration/Configuration Layer must specify the maximum amount of time
50          to wait for a notification through the callback listener, before
51          considering this call as unsuccessful.
52      */
53      const string INTERVAL = "interval";
54
55      /** Property structs hold configuaration parameters in the form of name
56      value pairs for the different adapters. Names and values depend on the
57      adapter type.
58      */
59      struct Property {
60          string name;
61          any value;
62      };
63
64      /** A configuration set for the adapter. This sequence provides all
65      Properties an adapter needs to configure an adapter source and to give
66      feedback to the Integration/Configuration Layer.
67      */
68      typedef sequence<Property> BasicConfiguration;
69
70      /** The basic interface to configure and manage an adapter. This interface
71      should serve as parent interface to all adapter types
72      */
73      interface SmonaAdapter {
74
75          /** Raised by the adapter if invalid Properties are given
76          Examples:
77              - incompatible Property name given in the sequence of the start()
78              method
79          */
80          exception NoSuchPropertyException{
81              string name;
82              long code;
83          };
84
85          /** Raised by adapter if invalid Property value is given
86          Example:
87              - interval < 0
88          */
89          exception InvalidPropertyValueException{
90              string name;
91              long code;
92          };
93
94          /** Raised by the adapter if invalid source id is given
95          Example:
96              source name that this adapter never registered with the Naming
97              Service
98          */
99          exception NoSuchSourceException{
```

```
100             string name;
101             long code;
102         };
103
104         /** Maybe raised by a method call if start configuration has not been
105         set up yet
106         */
107         exception AdapterNotConfiguredException{
108             string name;
109             long code;
110         };
111
112         /** Maybe raised by a method call if an adapter was already
113         started/configured for a source. If one wants to restart/reconfigure an
114         adapter one first has to call the stop() method.
115         */
116         exception AdapterAlreadyActiveException{
117             string name;
118             long code;
119         };
120
121         /** Maybe raised by a call to start(), if the given BasicConfiguration
122         does not contain a property an adapter needs for its configuration
123         */
124         exception MissingPropertyException {
125             string name;
126             long code;
127         };
128
129         /** Start/Configure adapter with the given BasicConfiguration sequence.
130         */
131         void start(in string source_id, in BasicConfiguration basic_conf)
132             raises (NoSuchPropertyException, InvalidPropertyValueException,
133                     AdapterAlreadyActiveException, NoSuchSourceException,
134                     MissingPropertyException);
135
136         /** Stop adadapter source. A call to this method will cause this
137         adapter to mark the corresponding source as inactive and perform the
138         necessary finalization. Only an inactive source can be started by the
139         Integration/Configuration Layer. If this source is not active, nothing
140         will happen
141         */
142         void stop(in string source_id)
143             raises (NoSuchSourceException);
144
145         /** FOR FUTURE USE:
146         An adapter source must provide self descriptive data to identify its
147         features. The CORBA Interface Repository may provides insufficient
148         information (with special regard to semantics...). The description also
149         includes ranges for values of configuration Properties.
150         Example:
151             If an adapter excpects an interval betweem 1 and 100, those bounds
152             have to be accessible and visible through the self-description
153         TODO:
154         - define a description schema (XML, Interface, struct...?)
155         */
156         any getSelfDescription(in string source_id)
157             raises (NoSuchSourceException);
158
159         /** get adapter build version
160         */
```

```
161         string getVersion();
162
163         /** Get global unique identifier of this adapter. A unique id which can
164         be used by the Integration/Configuration Layer to reference or identify
165         an adapter source. E.g. one can check, whether two sources are hosted
166         by the same adapter (good choice was the name a source was published
167         within the Naming Service)
168         */
169         string getId();
170
171         /** FOR FUTURE USE: Retrieve an adapter Property by name. At the moment
172         there exists no use case for such a call
173         */
174         any getProperty(in string name, in string source_id)
175             raises (NoSuchPropertyException, NoSuchSourceException);
176     };
177
178     /** A listener object implementing this interface will be set by the
179     start() method. The listener gets called to communicate callbacks to the
180     associated Integration/Configuration Layer.
181     That can be:
182     - the delivery of a push event
183     - the delivery of the return value/exception of a non-blocking call
184     Inheriting Interfaces must define the method signatures of this interface
185     */
186     interface SmonaListener {
187     };
188 };
189 };
190 };
191 };
192 };
```

Listing C.4: Dieses Modul enthält Konstanten, Exceptions und Interfaces, die Pull Adapter implementieren

```
1  /** Package for SMONA Pull Adapters.
2  This package includes all idl type definitions for Pull Adapters that describe
3  the interfaces shared between the Platform Independent Layer (Adapter Layer)
4  and the Integration/Configuration Layer All Pull Adapter specific interfaces
5  should be in or below this package.
6  */
7
8  #include "SmonaAdapter.idl""
9  #include "AdapterFunction.idl"
10
11 module de {
12 module lmu {
13 module ifi {
14 module nm {
15 module smona {
16 module pull {
17
18     /**
19     Description of the implementation for the inherited start() method:
20     A call to start() triggers a new monitoring process.
21     Example for Properties of the parameter BasicConfiguration:
22
23     Property    |   Name        |   Value
24     ---------------------------------------------------------------
25     1           |   source_id   |   Adapter.A/../../src.S/type.ST
26     2           |   listener    |   a_listener      # Callback reference
```

```
27      3           |   interval    |   100             # in ms
28
29      Description of the implementation for the inherited stop() method:
30      A call to this method will cause this adapter to stop monitoring the given
31      source. If this source is not active, nothing will happen
32      */
33      interface PullAdapter : SmonaAdapter {
34
35          /** Explicitly pull adapter source. This gives an upper layer the
36          possibility to pull an adapter at any given time.
37          */
38          void pullSource(in string source_id)
39              raises (NoSuchSourceException, AdapterNotConfiguredException);
40      };
41
42      /** A listener object implementing this interface will be set by the
43      start() method. The listener gets called to notify a registered
44      Integration/Configuration Layer about the return value of a non-blocking
45      call to the pullSource() method
46      */
47      interface PullListener : SmonaListener {
48
49          /** This method gets called to relay the return value of the
50          non-blocking pullSource() method.
51          - source_id:
52              The id of the source the adapter monitors. This id corresponds to
53              the value of the source_id Property given in a BasicConfiguration
54              set. This is useful if the Integration/Configuration Layer provides
55              the same PullListener reference for the monitoring of several
56              attributes to several adapters.
57          - value:
58              includes the measured value that triggered an event. The
59              Integration/Configuration Layer must have knowledge about how to
60              parse this Any value. In the future this may be determined by a
61              call to the getSelfDescription() method of the super interface
62          */
63          void send_Value(in string source_id, in any value);
64
65          /** This method gets called to relay an error generated by the source
66          itself due to an unsuccessful non-blocking call of the pullSource()
67          method.
68          - source_id:
69              The id of the source the adapter tries to pull. This id corresponds
70              to the value of the source_id Property given in a
71              BasicConfiguration set. This is useful if the
72              Integration/Configuration Layer provides the same SetListener
73              reference
74          - src_err:
75              the (optional) error string returned by the attempt to pull
76              something
77              Example:
78                  - IO error caused by the attempt to open a file
79          - code:
80              the exit value of the excecution attempt
81          */
82          void send_SourceError(
83              in string source_id, in string src_err, in long code);
84      };
85  };
86  };
87  };
```

```
88 };
89 };
90 };
```

Listing C.5: Dieses Modul enthält Konstanten, Exceptions und Interfaces, die Push Adapter implementieren

```
1  /** Package for SMONA Push Adapters. This package includes all idl type
2  definitions for Push Adapters that describe the interfaces shared between the
3  Platform Independent Layer (Adapter Layer) and the Integration/Configuration
4  Layer. All Push Adapter specific interfaces should be in or below this package.
5  */
6
7  #include "SmonaAdapter.idl"
8  #include "AdapterFunction.idl"
9
10 module de {
11 module lmu {
12 module ifi {
13 module nm {
14 module smona {
15 module push {
16
17     /**
18     Description of the implementation for the inherited start() method:
19     A call to start() triggers a new monitoring process.
20     Example for Properties of the parameter BasicConfiguration:
21
22     Property    |   Name        |   Value
23     ------------------------------------------------------------------
24     1           |   source_id   |   Adapter.A/../../src.S/type.ST
25     2           |   listener    |   a_listener      # Callback reference
26     3           |   interval    |   100             # in ms
27     4           |   max         |   80.8            # in %
28     5           |   min         |   0.0             # in %
29
30     Description of the implementation for the inherited stop() method:
31     A call to this method will cause this adapter to stop monitoring the given
32     source. If this source is not active, nothing will happen
33     */
34     interface PushAdapter : SmonaAdapter {
35         /** constants that may be used as key for Property structures.
36         The corresponding values of the Properties defined by these constants
37         are described for each constant:
38         */
39
40         /** Thresholds are optional.
41         */
42
43         /** threshold_max:
44         The upper bound for data values from the source. The adapter should
45         relay data if data values exceed this threshold
46         */
47         const string THRESHOLD_MAX = "threshold_max";
48
49         /** threshold_min:
50         The lower bound for data values from the source. The adapter should
51         relay data if data values fall below this threshold
52         */
53         const string THRESHOLD_MIN = "threshold_min";
54     };
55
```

```
56      /** A listener object implementing this interface will be set by the
57      start() method. The listener gets called to notify a registered
58      Integration/Configuration Layer about the occurrence of an event (e.g.
59      exceeding a threshold)
60      */
61      interface PushListener : SmonaListener {
62
63          /** This method gets called to relay an adapter event
64          - source_id:
65              The id of the source the adapter monitors. This id corresponds to
66              the value of the source_id Property given in a BasicConfiguration
67              set. This is useful if the Integration/Configuration Layer provides
68              the same PushListener reference for the monitoring of several
69              sources to several adapters.
70          - value:
71              includes the measured value that triggered an event. The
72              Integration/Configuration Layer must have knowledge about how to
73              parse this Any value. In the future this may be determined by a
74              call to the getSelfDescription() method of the parent interface
75          */
76          void pushEvent(in string source_id, in any value);
77
78          /** This method gets called to relay an error generated by the source
79          itself due to errors during the monitoring
80      - source_id:
81              The id of the source the adapter tries to monitor. This id
82              corresponds to the value of the source_id Property given in a
83              BasicConfiguration set. This is useful if the
84              Integration/Configuration Layer provides the same SetListener
85              reference
86          - src_err:
87              the (optional) error string returned by the monitored source
88              Example:
89                  - IO error caused by the attempt to open a file
90          - code:
91              the exit value of the excecution attempt
92          */
93          void send_SourceError(
94              in string source_id, in string src_err, in long code);
95      };
96  };
97  };
98  };
99  };
100 };
101 };
```

Listing C.6: Dieses Modul enthält Konstanten, Exceptions und Interfaces, die Set Adapter implementieren

```
1   /** Package for SMONA Set Adapters. This package includes all idl type
2   definitions for Set Adapters that describe the interfaces shared between the
3   Platform Independent Layer (Adapter Layer) and the Integration/Configuration
4   Layer All Set Adapter specific interfaces should be in or below this package.
5   */
6
7   #include "SmonaAdapter.idl"
8
9   module de {
10  module lmu {
11  module ifi {
12  module nm {
```

```
13 module smona {
14 module set {
15
16     /** forward declaration */
17     interface SetListener;
18
19     interface SetAdapter : SmonaAdapter {
20         /** This method is useful to retrieve the listener that was registered
21         with a source of a SetAdapter with the id source_id. If no listener was
22         registered with a source that was published with the id source_id, the
23         call will return null
24         */
25         SetListener getListener(in string source_id)
26             raises (NoSuchSourceException);
27     };
28
29     /** A listener object implementing this interface will be set by the
30     start() method. The listener gets called to notify a registered
31     Integration/Configuration Layer about the return value of a non-blocking
32     call to the setSource() method
33     */
34     interface SetListener : SmonaListener {
35
36         /** This method gets called to relay the return value of the
37         non-blocking setSource() method.
38         - source_id:
39             The id of the source the adapter monitors. This id corresponds to
40             the value of the source_id Property given in a BasicConfiguration
41             set. This is useful if the Integration/Configuration Layer provides
42             the same PullListener reference for the monitoring of several
43             attributes to several adapters.
44         - msg:
45             includes the (optional) message of a successful method call.
46         - code:
47             an error code
48         */
49         void send_Value(in string source_id, in string msg);
50
51         /** This method gets called to relay an error generated by the source
52         itself due to an unsuccessful non-blocking call of the setSource()
53         method.
54         - source_id:
55             The id of the source the adapter tries to modify. This id
56             corresponds to the value of the source_id Property given in a
57             BasicConfiguration set. This is useful if the
58             Integration/Configuration Layer provides the same SetListener
59             reference
60         - src_err:
61             the (optional) error string returned by the attempt to change
62             something
63             Example:
64                 - A call to mount may fails and returns an error message. This
65                 message is delivered via the src_err parameter
66         - code:
67             the exit value of the excecution attempt
68             Example:
69                 - A call to mount may fails and exits with a value != 0. This
70                 value is delivered through code parameter
71         */
72         void send_SourceError(
73             in string source_id, in string src_err, in long code);
```

```
74      };
75  };
76  };
77  };
78  };
79  };
80  };
```

C.4 IDL für das Interface `Adapter Function`

Listing C.7: Dieses Modul beinhaltet die Definition der Struktur AdapterFunction

```
1   /** Package for SMONA Adapters.
2   This package includes all idl type definitions for functions applicable
3   on SmonaAdapters.
4   */
5
6   module de {
7   module lmu {
8   module ifi {
9   module nm {
10  module smona {
11  module function {
12      /** constants that may be used as key for Property structures.
13          The corresponding values of the Properties defined by these constants
14          are described for each constant:
15      */
16
17      /**
18      This value is an optional Property name (See SmonaAdapter.idl).
19      If given, it describes the function an adapter
20      has to apply to the result of its return value(s). E.g. an adapter may
21      calculate the arithmetic mean of the last ten measurements of the
22      CPU load. The value for this key is of type Function (see below)
23      */
24      const string FUNCTION_TYPE= "function";
25
26      /**
27      This are the names of existing functions
28      */
29      const string GEOMETRIC_MEAN = "geometric_mean";
30      const string ARITHMETIC_MEAN = "arithmetic_mean";
31      // add further functions here
32
33      /**
34      This type is a holder for function parameters
35      */
36      typedef sequence<any> Parameter;
37
38      /**
39      This struct is a holder for a function and its parameters
40      */
41      struct AdapterFunction {
42          string name;
43          Parameter params;
44      };
45  };
46  };
```

```
47 };
48 };
49 };
50 };
```

Literaturverzeichnis

[AKS 05] ALEKSY, M., A. KORTHAUS und M. SCHADER: *Implementing Distributed Systems with Java and CORBA*. Springer Verlag, Heidelberg, Deutschland, 2005.

[Bros 06] BROSE G. ET AL: *JacORB 2.3 Programming Guide*. JacORB, Oktober 2006, http://www.jacorb.org .

[BVD 01] BROSE, G., A. VOGEL und K. DUDDY: *Java Programming with CORBA: Advanced Techniques for Building Distributed Applications*. John Wiley & Sons, 3 Auflage, 2001.

[DaSa 05] DANCIU, V. und M. SAILER: *A monitoring architecture supporting service management data composition*. In: *Proceedings of the 12th Annual Workshop of HP OpenView University Association*, Seiten 393–396, HP, Porto, Portugal, Juli 2005. .

[DgFS 07] DANCIU, V., N. GENTSCHEN FELDE und M. SAILER: *Declarative Specification of Service Management Attributes*. In: *Moving From Bits to Business Value: Proceedings of the 10th International IFIP/IEEE Symposium on Integrated Management (IM 2007)*, Munich, Germany, Mai 2007. .

[DHHS 06] DANCIU, V., A. HANEMANN, H.-G. HEGERING und M. SAILER: *IT Service Management: Getting the View*. In: *Kern E. M., Brügge B., Hegering H.226G. (Hrsg.): Managing Development and Application of Digital Technologies*. Springer, Juni 2006.

[GHJV 04] GAMMA, ERICH, RICHARD HELM, RALPH JOHNSON und JOHN VLISSIDES: *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2004.

[Gül 02] GÜLCÜ, CEKI: *The complete log4j Manual*. Quality Open Software, Lausanne, Schweiz, November 2002.

[HAN 99] HEGERING, H.-G., S. ABECK und B. NEUMAIR: *Integriertes Management vernetzter Systeme - Konzepte, Architekturen und deren betrieblicher Einsatz*. dpunkt-Verlag, ISBN 3-932588-16-9, 1999, http://www.dpunkt.de/produkte/management.html .

[Hero 99] HEROLD, HELMUT: *Linux-Unix-Profitools*. Addison-Wesley, Bonn, 1999.

[HoCo 02] HORSTMANN, CAY S. und GARY CORNELL: *Core Java 2, Volume II, Advanced Features*. P T R Prentice-Hall, 5 Auflage, 2002.

[IBM] *IBM Tivoli NetView*, http://www-306.ibm.com/software/tivoli/products/netview/ .

[Jav] *JavaBeans*, http://java.sun.com/products/javabeans/ .

[Lang 02] LANGNER, TORSTEN: *Verteilte Anwendungen mit Java*. Markt-+-Technik-Verlag, Martin-Kollar-Straße 10 - 12, 81829, Deutschland, 2002.

[LP 98] LINNHOFF-POPIEN, C.: *CORBA - Kommunikation und Management*. Springer, 1998.

[McLa 00] MCLAUGHLIN, B.: *Java and XML*. O'Reilly, Juni 2000.

[Nag] *Nagios*, http://www-306.ibm.com/software/tivoli/products/netview/ .

[OaWo 99] OAKS, S. und H. WONG: *Java Threads*. O'Reilly & Associates, Inc., 101 Morris Street, Sebastopol, CA 95472, April 1999.

[OMG 02] OMG THE OBJECT MANAGEMENT GROUP, 140 Kendrick Street, Building A Suite 300 Needham, MA 02494, U.S.A.: *IDL to Java Language Mapping Specification*, August 2002, http://www.omg.org/cgi-bin/doc?formal/02-08-05.pdf .

[OMG 03] OMG THE OBJECT MANAGEMENT GROUP, 140 Kendrick Street, Building A Suite 300 Needham, MA 02494, U.S.A.: *Java to IDL Language Mapping Specification*, September 2003, http://www.omg.org/cgi-bin/doc?formal/03-09-04.pdf .

[OMG 04] OMG THE OBJECT MANAGEMENT GROUP, 140 Kendrick Street, Building A Suite 300 Needham, MA 02494, U.S.A.: *Common Object Request Broker Architecture: Core Specification*, März 2004, http://www.omg.org/cgi-bin/doc?formal/04-03-01.pdf .

[OMG 06] OMG THE OBJECT MANAGEMENT GROUP, 140 Kendrick Street, Building A Suite 300 Needham, MA 02494, U.S.A.: *CORBA Reflection*, Mai 2006, http://www.omg.org/cgi-bin/doc?formal/06-05-03.pdf .

[Sail 05] SAILER, M.: *Towards a Service Management Information Base*. In: *IBM PhD Student Symposium at ICSOC05*, Amsterdam, Netherlands, December 2005. .