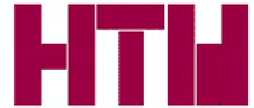
DILLINGER HÜTTE

Hochschule für
Technik und Wirtschaft
des Saarlandes
University of Applied Sciences



Hochschule für Technik und Wirtschaft des Saarlandes

Fakultät für Ingenieurswissenschaften

Master Thesis

zur Erlangung des akademischen Grades

'Master of Science Kommunikationsinformatik (M.Sc.)'

an der Hochschule für Technik und Wirtschaft des Saarlandes

im Studiengang Kommunikationsinformatik

der Fakultät für Ingenieurswissenschaften

Migration von statischer zu dynamischer Bereitstellung von Metadaten aus Drittsystemen am Beispiel RHQ und Nagios

vorgelegt von

Alexander Kiefer, B.Sc.
Matrikelnummer: 3427641

betreut durch

Prof.Dr. Reinhard Brocks

Zweitkorrektur durch

Dipl.Ing. Michael Sauer

Saarbrücken, den 30.09.2010

Eidestattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe, die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

$\overline{Ort, Datum}$ $\overline{Unterschrift\ (Vor-und\ \ Nachname)}$

Vorwort

An dieser Stelle möchte ich mich bei allen Personen bedanken, die mich bei der Entstehung dieser Arbeit unterstützt haben. Ganz besonderer Dank gebührt dabei Heiko W. Rupp von der Firma 'RedHat', der mich als technischer Berater bei der Umsetzung der Arbeit unterstützte.Er stand mir bei Rückfragen jederzeit mit Rat und Tat zur Seite, ohne seine Hilfe wäre die erfolgreiche Realisierung dieser Arbeit nicht möglich gewesen. Ebenfalls großer Dank gebührt Frank Fontaine aus der Abteilung 'TI/STW' der Dillinger Hütte, der mich während meiner Praxisphase fachlich betreute und mich bei allen technischen Fragen immer hervorragend unterstützt hat. Darüber hinaus stand er mir auch im Verlauf dieser Arbeit bei allen erdenklichen Problemen zur Seite und trug somit ebenfalls maßgeblich zum Gelingen der Arbeit bei. Desweiteren möchte ich mich bedanken bei Jörg Audörsch, der als Betreuer der Masterthesis seitens des Unternehmens Dillinger Hütte weder Zeit noch Mühe gescheut hat, zum Gelingen dieser Arbeit beizutragen, und mir mit zahlreichen Ratschlägen und Anregungen zur Seite stand. An dieser Stelle möchte ich mich außerdem bei allen anderen Mitarbeitern der Abteilung 'TI/STW' für ihre Unterstützung während meiner Praxisphase und die schöne und lehrreiche Zeit danken. Seitens der Hochschule gilt mein Dank Prof.Dr. Reinhard Brocks für die Betreuung der Arbeit und die fachlichen Anregungen, die im Rahmen mehrerer Gespräche im Verlauf dieser Arbeit entstanden sind. Diese Anregungen haben mir immer wieder geholfen, die Arbeit in die richtigen Bahnen zu lenken, und trugen stark zum Gelingen der Arbeit bei. Außerdem möchte ich mich bedanken bei Dipl.Ing. Michael Sauer für die Zweitkorrektur dieser Arbeit. In meinem privaten Umfeld gebührt ein besonderer Dank meiner Lebensgefährtin Jasmin Heyer für ihre Unterstützung in den letzten Monaten während der Entstehung dieser Arbeit und während meines Studiums, außerdem für die Durchführung der Erstkorrektur dieser Arbeit. Abschließend möchte ich mich bedanken bei meinen Eltern für jegliche Unterstützung während meiner gesamten Studienzeit und bei meiner gesamten Familie.

Abstract

Heute führen immer komplexere IT Umgebungen innerhalb von Unternehmen, mit einer stetig steigenden Anzahl an Rechnern und Anwendungen, dazu, dass die Überwachung der Systeme und ihrer Laufzeitumgebung unverzichtbar geworden ist geworden ist. Zahlreiche Anwendungen bieten Lösungen für die Systemüberwachung an, im Rahmen dieser Arbeit wurde die RHQ Systemmanagement-Suite der Firma Red-Hat und das Nagios Monitoring-System näher betrachet. Dabei wurde für das verteilte, Plug-In-basierte RHQ System ein Plug-In entwickelt, dass die Überwachung eines Nagios System durchführen kann. Die RHQ Plug-Ins benötigen Metadaten, die statisch in Form einer XML Datei geliefert werden. Da die statische Variante einige Nachteile mit sich bringt, wurde eine Analyse erstellt, die die technischen Anforderungen definiert, die erfüllt sein müssen, um ein solches System zur dynamischen Metadatenbereitstellung zu migrieren. Entsprechende Lösungsvorschläge zur Erfüllung der Anforderungen wurden ebenfalls mitgeliefert. Abschließend wurde an einem Prototypen demonstriert, wie die, für die Migration notwendigen, Anpassugen am System realisiert werden können.

Inhaltsverzeichnis

1 Einleitung 3
1.1 Einführung in die Thematik 4
1.2 Motivation 10
1.3 Aufgabenstellung 13
1.3.1 Entwicklung eines Plug-Ins für die RHQ Systemüberwachungssuite zur Anbindung des Nagios Systems 13
1.3.2 Anforderungsanalyse und Beschreibung eines Lösungskonzepts 14
1.3.3 Prototypenrealisierung der Migration von statischer zu dynamischer Metadatenbereitstellung im RHQ System 14

2 Ausgangslage 16
2.1 Projektumfeld 16
2.1.1 Das Unternehmen Dillinger Hütte 16
2.1.2 Der IT Bereich der Dillinger Hütte 18
2.1.3 Technische Informatik(TI) 18
2.1.4 Kaufmännische Informatik(KI) 18
2.1.5 Informationstechnik(IT) 19
2.2 Begriffsmodel 20
2.2.1 RHQ 21
2.2.2 Nagios 24

3 Anforderungsanalyse und Lösungskonzept 25
3.1 Schematischer Aufbau einer Systemüberwachungssuite 25
3.1.1 Server 26
3.1.2 Agent 27
3.1.3 Plugin 28
3.2 Realisierung unter Verwendung statischer Metadaten 29
3.2.1 Datenmodel zur Abbildung des Eigensystems 29
3.2.2 Beschreibung der zu überwachenden Ressourcentypen 30
3.2.3 Datenmodel zur Abbildung des Drittsystems 32
3.2.4 Mapping der Daten des Drittsystems auf das Datenmodel des Eigensystems 33
3.2.5 Persistierung der Systemdaten 34
3.3 Realisierung der Migration von statischer zu dynamischer Metadatenverwendung 36
3.3.1 Schaffung einer Schnittstelle zum dynamischen Anlegen neuer Resourcetypen 36

3.3.2 Dynamische Synchronisation der Datenhaltung zwischen Server, Agent und Plug-In . . . 37
3.4 Anwendungsfälle(Use Cases) . . . 38
3.4.1 Hinzufügen/Entfernen von Ressourcentypen bei statischer Bereitstellung der Metadaten . . . 39
3.4.2 Hinzufügen/Entfernen von Ressourcetypen bei dynamischer Bereitstellung der Metadaten . . . 41

4 Umsetzung 43
4.1 Projektplanung . . . 43
4.1.1 Entwicklung eines RHQ Plug-Ins für die Nagios Integration unter Verwendung statischer Metadaten . . . 43
4.1.2 Entwicklung eines Prototypen für die Migration zur Bereitstellung dynamischer Metadaten . . . 44
4.2 Technische Rahmenbedingungen bei der Umsetzung . . . 46
4.2.1 Betriebssystemumgebung . . . 46
4.2.2 Datenbanksystem . . . 46
4.2.3 Entwicklungsumgebung . . . 46
4.2.4 Versionskontrolle . . . 47
4.2.5 Build-Tools . . . 47
4.2.6 Advanced Management Plugin System (AMPS) . . . 47
4.3 Umsetzung der Nagios Anbindung . . . 48
4.3.1 Validierung der vorhandenen Schnittstellen zu Nagios . . . 48
4.3.2 Entwurf eines Softwaremodels zur Abbildung des Nagios Systems . . . 52
4.3.3 Entwicklung eines RHQ Plugins für die Nagios Integration unter Verwendung statischer Metadaten . . . 54
4.4 Umsetzung der Migration zur Bereitstellung dynamischer Metadaten . . 64
4.4.1 Dynamische serverseitige Generierung neuer Resourcetypen . . . 65
4.4.2 Dynamische agentenseitige Generierung neuer Resourcetypen und Synchronisation mit dem Server . . . 68
4.4.3 Dynamische Generierung neuer Resourcetypen im Plug-In . . . 72

5 Resumee 78
5.1 Zusammenfassung . . . 78
5.2 Ausblick . . . 80

6 Anhang 82
6.1 Quellenverzeichnis . . . 82

Abbildungsverzeichnis 83

1 Einleitung

In diesem Abschnitt erfolgt zunächst eine Übersicht über die folgenden Kapitel und ihre jeweiligen Inhalte. In Kapitel 1 dieser Arbeit erhält der Leser eine umfassende Einleitung in das Projekt. In Abschnitt 1.1 erhält er eine Einführung in die Thematik, anschließend wird in Abschnitt 1.2 ausführlich die Motivation für dieses Projekt beschrieben. In Abschnitt 1.3 wird zum Abschluß dieses Kapitels detailliert die Aufgabenstellung mit den daraus resultierenden Teilaufgaben erläutert. Kapitel 2 beschreibt die Ausgangslage zu Beginn dieser Arbeit in verschiedenen Kontexten. Zunächst wird in Abschnitt 2.1 das Unternehmensumfeld beschrieben, im Anschluß daran wird in Abschnitt 2.2 ein ausführliches Begriffsmodel mit allen, für das weitere Verständnis dieser Arbeit, notwendigen Begriffen und deren Erläuterungen eingeführt. Kapitel 3 enthält eine umfassende Anforderungsanalyse und definiert Lösungsansätze für die gestellten Anforderungen. Dabei wird in Abschnitt 3.1 zunächst der allgemeine Aufbau einer Systemüberwachungssuite erklärt, bevor in Abschnitt 3.4 die wichtigsten Anwendungsfälle für ein solches System, im Kontext der Verwendung von statischer und dynamischer Metadatenverwendung, beleuchtet werden. In Abschnitt 3.2 werden dann die allgemeinen technischen Anforderungen an ein solches System beschrieben, in Abschnitt 3.3 folgt die Definition der Anforderungen an eine Systemüberwachungs-Suite, die für die Migration zur dynamischen Bereiststellung von Metadaten erfüllt sein müssen. Kapitel 4 beschreibt die Umsetzung des erarbeiteten Lösungskonzeptes am Beispiel der konkreten Aufgabenstellung. Hier wird in Abschnitt 4.1 zunächst die Planung der einzelnen Projektschritte erläutert, in Abschnitt 4.2 folgt eine Beschreibung der verwendeten externen Tools. Anschließend folgt in Abschnitt 4.3 die Beschreibung der Implementierung des statischen Nagios Plug-Ins, und in 4.4 die Beschreibung der Entwicklung eines Prototyps zur dynamischen Bereitstellung von Metadaten. In Kapitel 5 folgt eine Abschlußbetrachntung der erreichten Ergebnisse, in Abschnitt 5.1 werden die Ergebnisse zusammengefasst und kritisch bewertet, und in Abschnitt 5.2 folgt ein Ausblick auf weitere sinnvolle Arbeitschritte, die innerhalb des gesteckten Zeitrahmens nicht mehr realisiert werden konnten.

1.1 Einführung in die Thematik

Das RHQ Projekt ist ein OpenSource-Projekt unter der Verwaltung der GNU Public License (GPL), es bietet integrierte und erweiterbare Systemmanagement Funktionen für zahlreiche Plattformen, wie z.B. Apache, Tomcat, JBoss Application Server und PostgreSQL. Die RHQ Systemmanagement-Suite wurde von der Firma RedHat RedHat entwickelt und unterstützt die Überwachung und Verwaltung von zahlreichen Anwendungen, wie Applikationsservern, Datenbanksystemen und Netzwerkdiensten. RHQ ist ein verteiltes System mit einem zentralen Server oder Servercluster und einem oder mehreren Agenten, die auf unterschiedlichen Hosts laufen und, mithilfe von Plug-Ins, Informationen auf diesen Hosts sammeln. Innerhalb eines Agenten laufen zeitgleich mehrere Plug-Ins, die Aufgabe der Plug-Ins ist die Überwachung von Ressourcen, wobei als Resource alle Dinge zu verstehen sind, bei denen die Ermittlung von Messdaten möglich ist. Das können sowohl Hard- als auch Softwarekomponenten sein, es muss lediglich eine Schnittstelle existieren, über die das Plug-In mit der Ressource kommunizieren kann. Resourcen, die von einem Plug-In überwacht werden, werden auch als 'ManagedObject' bezeichnet. Abbildung 1 zeigt den Systemaufbau des RHQ System mit seinen Komponenten 'Server', 'Agent' und 'Plug-In'. Der Server wird vom Administrator konfiguriert und ist an die Systemdatenbank angeschlossen, in der die Messdaten und die Konfigurationseinstellungen gespeichert werden. Der Server kommuniziert mit mehreren, verteilt laufenden, Agents, die wiederum mehrere Plug-Ins betreiben. Die Plug-Ins überwachen die 'ManagedObjects' und ermitteln die Monitoring-Daten.

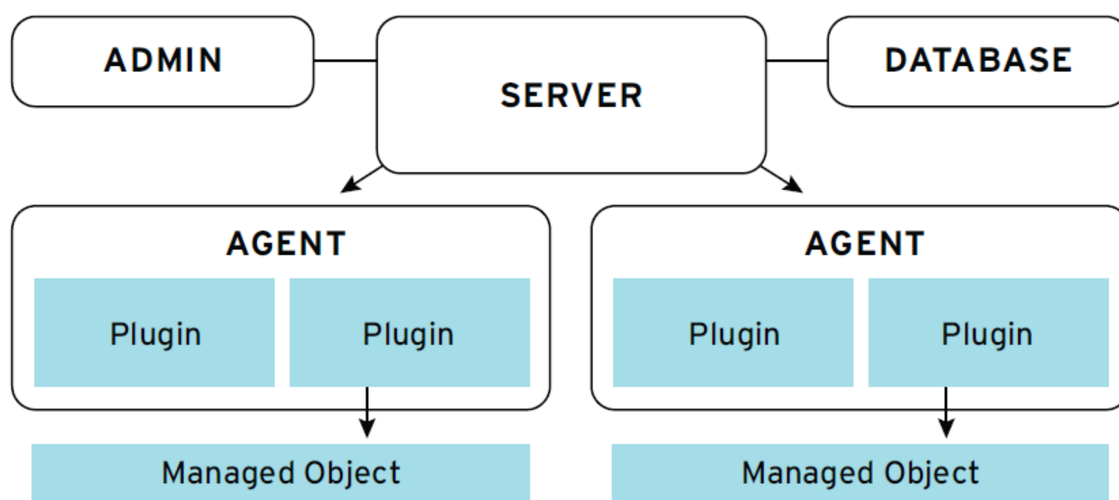


Abbildung 1: Aufbau des RHQ Systems

Jede Resource gehört einem bestimmten Typ an, dem sogenannten ResourceType. Jedes Plug-In wird für einen bestimmten ResourceType entwickelt und kann sämtliche Resourcen dieses Typs managen. Jeder ResourceType gehört einer bestimm-

ten Kategorie, der 'ResourceTypeCategory', an, es gibt die Kategorien 'Platform', 'Server' und 'Service'. Durch die Verwendung der Kategorien können die ResourceTypes hierarchisch gegliedert und Parent-Child-Beziehungen dargestellt werden. Abbildung 2 zeigt die verschiedenen Kategorien und ihre hierarchische Gliederung. Die oberste Kategorie ist die 'Platform', darunter liegen die Kategorien 'Server' und 'Service'. Jede Ressource einer bestimmten Kategorie kann jeweils andere Ressourcen der gleichen oder einer niedrigeren Kategorie betreiben. Die Begriffe 'Resource', 'ResourceType' und 'ResourceTypeCategory' werden im Abschnitt 2.2 detailliert erläutert.

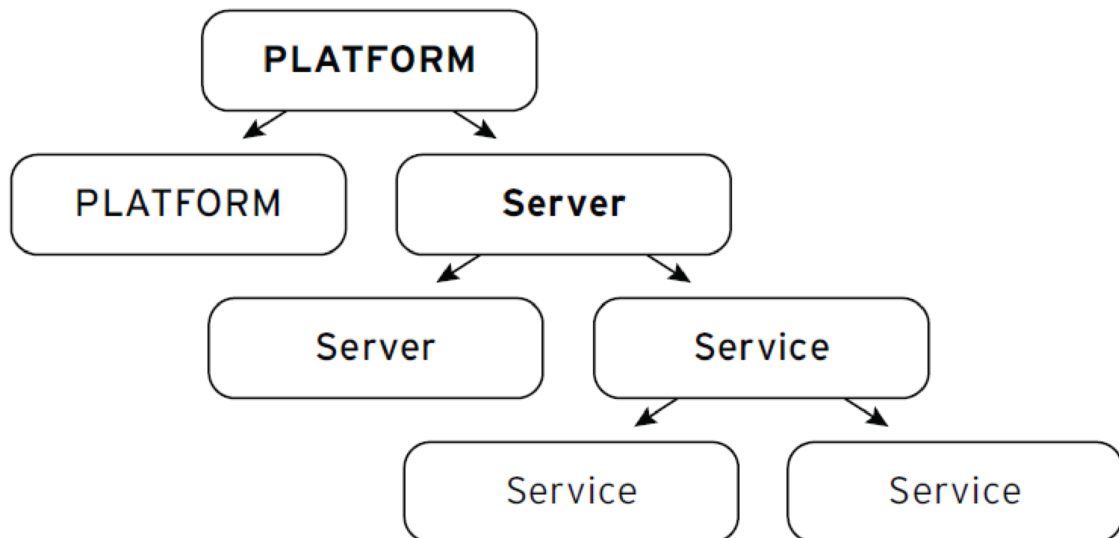


Abbildung 2: Übersicht über die RHQ Resourcenkategorien

Für die anfangs genannten ResourceTypes wie Apache und JBoss, aber auch für viele Weitere, sind standardmässig Plug-Ins im Download-Umfang enthalten, mit denen Resources des jeweiligen Typs gemanaged werden können. Es besteht darüber hinaus die Möglichkeit der Entwicklung eigener Plug-Ins zur Anbindung beliebiger Ressourcetypen. Zu diesem Zweck existiert eine eigene Java Plug-In API, die dem Entwickler eine umfangreiche Bibliothek zur Verfügung stellt: das Advanced Management Plugin System (AMPS). AMPS setzt sich aus 5 Kernelementen zusammen:

- **Plugin Container** : der Plug-In-Container stellt den Manager dar, in dem alle Plug-Ins, und alle zum Management der Plug-Ins notwendigen Objekte, laufen. Der Plug-In-Container läuft innerhalb des Agenten, die Klassen des Moduls befinden sich im RHQ Codebaum unter 'core/plugin-container'.
- **Plugin Components**: In diesem Modul sind sämtliche Interfaces implementiert, sie können von den Component-Klassen der Plug-Ins implementiert werden, die die 'ManagedObjects' des überwachten Systems abbilden. Die Aufgaben der Discovery- und Component-Klassen werden weiter unten im Text

noch genauer beschrieben. Jedes Interface stellt eine andere Eigenschaft eines 'ManagedObjects' dar, man spricht auch von 'Facets'. Der Plug-In-Entwickler muss beim Schreiben der Plug-Ins die benötigten Interfaces implementieren, er erweitert so die Component-Klasse um verschiedene Facets. Die Facet Interfaces befinden sich im RHQ Codebaum unter 'core/plugin-api'.

- **Domain Objects**: Dieses Modul enthält alle individullen Domain Object Klassen, mit denen die beaobachteten Systeme und ihre Ressourcen abgebildet werden können, also Klassen wie 'Resource' und 'ResourceType'. Alle anderen Elemente derPlug-Ins benutzen die Domain-Objekte, um die überwachten Systeme abzubilden. Die Klassen des Domain-Models findet man im RHQ Codebaum unter 'core/domain'.

- **Plugins**: RHQ bietet zahlreiche Out-of-Box Plug-Ins, dies sind einzelne Plug-Ins, die beim Starten des Agenten vom Plug-In-Manager geladen werden. Jedes Plug-In überwacht ein anderes Produkt, es gibt zum Beispiel ein JBoss Plug-In zur Überwachnug von JBoss Applikationservern, oder ein Postgres Plug-In zur Überwachung von Postgres Datenbanken. Die Plug-Ins befinden sich im RHQ Codebaum unter 'plugins'.

- **Native System**: dieses Modul bietet Low-Level-Zugriff auf Betriebssystem-Operationen, man kann dadurch Informationen aus der Prozesstabelle des Betriebssystems ermitteln. Damit kann man ermitteln, ob ein bestimmter Dienst läuft oder nicht, außerdem kann dadurch auf externe Programme zugegriffen werden. (module core/native-system)

Jedes Plug-In besteht aus 3 Kernelementen: dem Plug-In Deskriptor, einer ResourceDiscovery-Klasse und einer ResourceComponent-Klasse. Der Plug-In-Deskriptor ist eine XML Datei, die nach einer genau definierten XML Schemadefinition angelegt wird. Im Plug-In-Deskriptor können folgende Dinge definiert werden:

- Namen und Versionsnummern: Namens- und Versionsinformationen zu den vom Plug-In zu überwachenden Ressourcen

- Plug-In-Konfiguration: Parameterdefinitionen, die für den Verbindungsaufbau zu den vom Plug-In zu managenden Ressourcen notwendig sind

- Metriken: Beschreibung der vom Plug-In zu ermittelnden und in RHQ anzuzeigenden Messdaten

- Methoden: die Methoden, die zum Zugriff auf die 'ManagedResources' benutzt werden

- Resourcenkonfiguration: Beschreibung der Resourcen selbst, nicht der Verbindungsparameter zu den Resourcen
- Resourcenhierarchie: Definition von Parent- und Child-ResourceTypes, um die Beziehung zwischen Ressourcen zu verdeutlichen

Abbildung 1.1 zeigt den einen Ausschnitt des Plug-In-Deskriptors des Nagios Plug-In.

```
<?xml version="1.0"?>
<plugin name="NagiosMonitor"
        displayName="NagiosMonitorPlugin"
        package="org.rhq.plugins.nagios"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns="urn:xmlns:rhq-plugin"
        xmlns:c="urn:xmlns:rhq-configuration"
        description="Interface to Nagios monitoring system. Nagios needs to be equipped with
            mk_livestatus module for this plugin to work"
        version="3.0.0.B04"
        >


    <server name="NagiosMonitor"
        discovery="NagiosMonitorDiscovery"
        class="NagiosMonitorComponent"
        description="Interface to Nagios monitoring system. Nagios needs to be equipped with
            mk_livestatus module for this plugin to work"
        supportsManualAdd="true"
            >

            <plugin-configuration>
                <c:simple-property name="nagiosHost" default="localhost" required="true"
                                   description="Hostname of where the Nagios mk_livestatus is listening" />
```

Die ResourceDiscovery- und ResourceComponent-Klassen bilden die Basis des Plug-Ins, der Plug-In-Entwickler muss für jeden zu überwachenden Resourcentyp diese beiden Klassen implementieren. Die Klassen habe folgende Aufgaben:

- **ResourceDiscovery**: Die ResourceDiscovery-Klasse führt die Suche nach den vom Plug-In zu managenden Ressourcen durch. Die Aufgabe der ResourceDiscovery-Klasse ist es, die Plattform zu scannen und die gefundenen Ressourcen an das System zu melden, sie wird vom Plug-In-Container gesteuert. Dazu übergibt der Plug-In-Container ein ResourceDiscoveryContext-Objekt an die ResourceDiscovery-Klasse, in diesem Kontext-Objekt sind sämtliche Informationen enthalten, die die ResourceDiscovery-Klasse zum Suchen neuer Resourcen benötigt.
- **ResourceComponent**: Eine ResourceComponent-Klasse wird innerhalb des Plug-Ins zur Abbildung einer aktuell überwachten Ressource verwendet. Der Lifecycle einer ResourceComponent-Klasse wird vom Plug-In-Container gemanaged, der Container starter und beendet die ResourceComponent. Beim Start der ResourceComponent verbindet sie sich mit der durch sie repräsentierten Resource, die Verbindung wird solage aufrecht erhalten, bis der Plug-In Container die ResourceComponent beendet. Die ResourceComponent bietet dem

Plug-In-Container die Möglichkeit, die Erreichbarkeit einer überwachten Ressource zu prüfen.

Die Plug-Ins ermöglichen dem RHQ System die Ausführung seiner Monitoring-Funktionen, sie stellen die Komponente dar, die die Überwachung der Ressourcen durchführen und die ermittelten Daten an das Kernsystem übertragen. Die wesentlichen Funktionen des RHQ Systems sind:[1] [2]

- Überwachung von Systemressourcen und Darstellung der Messwerte
- Alarmierung im Fall von auftretenden Fehlern
- Remote-Konfiguration überwachter Ressourcen
- Remote-Ausführung von Operationen
- Logging und Auditing
- Automatisierte Inventarerkennung
- Plug-In-Entwicklung zum Überwachen beliebiger Ressourcentypen

Nagios ist ein weiteres Monitoring-System und dient der Überwachung komplexer IT-Infrastrukturen, es wurde im Gegensatz zum Java-basierten RHQ komplett in C implementiert. Da das Thema der Arbeit mit der Entwicklung eines RHQ Plug-Ins zusammen hängt, wurde das RHQ System ausführlich vorgestellt. Nagios soll nur kurz beschrieben werden, da die Kenntnis seiner genauen Funktionsweise für das Verständnis der Arbeit nicht notwendig ist, es kann als Blackbox betrachtet werden, die notwendigen Daten werden über eine externe Schnittstelle geliefert. Zum Zweck der Überwachung bietet Nagios ebenfalls eine Sammlung von Plug-Ins zur Überwachung von Netzwerken, Hosts und Diensten, diese wurden auch speziell für die Überwachung eines bestimmten ResourceTypes geschrieben. Das Nagios System besteht, wie RHQ, aus einem Server und einem oder mehreren Agenten, dadurch ist die Überwachung verteilter Infrastrukturen möglich. Für unterschiedliche Betriebssysteme existieren unterschiedliche Agenten. Der Nagios Server ist eine freie Software und läuft unter zahlreichen Unix-Distributionen.[3]
Die wichtigsten Nagios Features sind[4]:

- Überwachung von Netzwerkdiensten (SMTP, POP3, HTTP, NNTP, PING, etc.)

[1] RHQ1
[2] RHQ2
[3] NAGIOS1
[4] NAGIOS2

- Überwachung von Host-Ressourcen (Prozessorauslastung, Diskbelegung, usw.)
- Einfache Entwicklung eigener Service-Prüfungen
- Parallelisierte Service-Prüfungen
- Benachrichtigung von Kontakten bei Service- oder Host-Problemen
- Definition von Routinen zur Ereignisbehandlung
- Optionales Web-Interface zur Beobachtung des aktuellen Netzwerkstatus

Die Aufgabenstellung dieser Arbeit setzte sich aus 3 zentralen Punkten zusammen, die realisiert werden sollen. Im ersten Teil sollte die Anbindung der RHQ Systemmanagement-Suite an die bestehende Nagios Systemüberwachung der Abteilung 'Informationstechnik' der Dillinger Hütte ermöglicht werden, zu diesem Zweck musste ein eigenes Nagios Plug-In für das RHQ System entwickelt werden. Im zweiten, eher theoretischen Teil der Arbeit, sollte eine allgemeine Anforderungsnanalyse erstellt werden, in der die Anforderungen an Monitoring-Systeme im Allgemeinen, und speziell bei der Verwendung von dynamischer Metadatenbereitstellung, definiert werden sollten. Zu jeder der einzelnen Anforderungen sollte eine Lösung entwickelt werden, die möglichst von der konkreten RHQ-Problematik abstrahiert werden sollte, um das Konzept auch zur Lösung vergleichbarerer Probleme bei anderen Systemen verwenden zu können. Dieser zweite Teil diente dem Zweck, dem wissenschaftlichen Anspruch einer Masterthesis gerecht zu werden. Im dritten Teil der Arbeit sollte, auf Basis des erstellten Konzepts, am Beipiel RHQ und Nagios, ein Prototyp zur Migration von der statischen Bereitstellung von Metadaten hin zur dynamischen Bereitstellung von Metadaten realisiert werden. Die Beschreibung der von den Plug-Ins zu überwachenden Ressourcentypen geschieht bisher in einem XML-basierten Plug-In-Deskriptor, sie erfolgt statisch im Rahmen der Implementierung des Plug-Ins, Änderungen oder Erweiterungen während des laufenenden Plug-In-Betriebs sind nicht möglich. Wenn Änderungen im Plug-In-Deskriptor vorgenommen werden, bespielsweise die Erweiterung um einen neuen ResourceType, sind Anpassugen im Java Quellcode, ein Re-Build des Plug-Ins und ein Systemneustart notwendig. Bei der Prototypenentwicklung sollte die Funktionsweise des Systems so angepasst werden, dass neue ResourceTypes zur Laufzeit des Systems, ohne Änderugen des XML-Deskriptors, hinzugefügt werden können.

1.2 Motivation

Die Abteilung 'Informationstechnologie' ist zuständig für den Betrieb und die Administration von Rechnern, Datenbanken und Softwareanwendungen, die abteilungsübergreifend benutzt werden. Zum Zweck der Überwachung der eingesezten Rechner und Applikationen wird dort die Nagios Systemüberwachungssoftware verwendet. Die Abteilung 'TI/STW' ist zuständig für die Entwicklung von Softwareapplikationen, die im Bereich des Stahlwerks verwendet werden. Die Applikationen werden ausschließlich in Java EE für die Verwendung auf dem JBoss Applikationsserver der Firma RedHat entwickelt. Innerhalb der Abteilung 'TI/STW' ist man daran interessiert, zur Überwachung der Anwendungen und der Laufzeitumgebung die RHQ Systemmanagement-Suite der Firma RedHat zu verwenden, wofür es mehrere Gründe gibt:

- da die Mitarbeiter der Abteilung 'TI/STW' eine eigene IT-Bereitschaft ausüben müssen, ist es sinnvoll, über eine eigene Überwachungssoftware zu verfügen, um nicht auf die Unterstützung der Abteilung 'IT' angewiesen zu sein
- durch die mehrjährige Verwendung des JBoss Applikationsserver wurde ein entprechendes Know-How bezüglich Benutzung und Adminitration aufgebaut, dieses Know-How kann bei der Verwendung von RHQ ebenfalls genutzt werden, da die Kernkomponente des Systems ein JBoss Applikationsserver ist
- das RHQ Projekt ist ein OpenSource-Projekt und basiert auf der Java Enterprise Edition (JavaEE), man hat Zugriff auf den vollständigen SourceCode und kann, durch das vorhandene JavaEE Know-How, alle notwendigen Anpassungen oder Erweiterungen selbstständig durchzuführen
- da der JBoss Apllikationsserver auf dem Konzept der ManagedBeans(MBeans) basiert, und der RHQ Kernel durch einen JBoss Applikationsserver realisiert wurde, ist eine Integration in das anwendungstechnische Umfeld der 'TI/STW' ohne Probleme möglich
- die Abteilung 'TI/STW' verfügt über JBoss Support und kann so jederzeit professionelle Unterstützung bei Problemen anfordern, was die Wartung und Administration des Systems erleichtert.

Da die Abteilung 'IT', wie bereits erwähnt, das Nagios System zur Überwachung ihrer Laufzeitumgebung verwendet, besteht die Notwendigkeit der Entwicklung eines RHQ Plug-Ins zur Anbindung an das Nagios System. Diese Notwendigkeit ergibt sich aus mehreren Gründen:

- Die Abteilung 'IT' hat für Nagios zahlreiche Plug-Ins entwickelt, die auch Informationen von Rechnern ermitteln, die die Abteilung 'TI/STW' betreffen. Da man den Aufwand der Re-Implementierung dieser Plug-Ins für die RHQ Systemmanagement-Suite vermeiden will, möchte man statt dessen lieber eine Anbindung an das bereits bestehende Nagios System realisieren, um die ohnehin vorhandenen Informationen direkt von dort zu ermitteln.

- Da Nagios ein weit verbreitetes Tool ist, das in zahlreichen IT Umgebungen von Unternehmen verwendet wird, tritt oftmals die Situation ein, dass ein bestehendes Nagios System existiert, aber dennoch der Bedarf der Einführung des RHQ Systems gegeben ist. Dies ist vor allem dann der Fall, wenn eine Java EE Umgebung unter Verwendung des JBoss AS existiert, in die RHQ, aufgrund seines Java EE/JBoss Kernels und zahlreicher, auf Basis der Java EE Technologie geschriebener Plug-Ins, einfacher als Nagios zu integrieren ist. Daher liegt es auch im Interesse der Firma RedHat, dass ein RHQ Plug-In zur Nagios Anbindung entwickelt wird.

Die Migration von der Bereitstellung statischer Metadatne zur Bereitststellung dynamischer Metadaten ist ein zentrales Anliegen der RHQ Entwickler, das bereits seit einiger Zeit existiert. Da sowohl das Unternehmen 'Dillinger Hütte' als auch die Firma 'RedHat' ein Interesse an der Realisierung des Nagios Plug-Ins hatten, entschied man sich, die beiden Dinge zu kombinieren. RedHat stellte den Support für die technische Betreuung der Arbeit zur Verfügung, als Gegenleistung rückte die Entwicklung eines Prototypen zur Bereistellung von dynamischen Metadaten mit in den Focus der Arbeit. Die bisherige statische Beschreibung der Metadaten eines Plug-Ins hat einige entscheidende Nachteile:

- das Hinzufügen neuer Ressourcentypen, oder andere Anpassungen im Plug-In-Deskriptor, bedingen zusätzliche Anpassugen im Java Code, da dort direkter Bezug auf den Deskriptor genommen wird.

- Nachdem die Änderugen vorgenommen wurden, muss das entsprechende Plug-In neu gebaut und in den Applikationsserver des RHQ Systems deployt werden. Die Agenten müssen neu gestartet werden, um die notwendige Synchronisation mit dem Server durchzuführen und die neueste Plug-In Version zu laden, während des Neustarts unterbrechen alle laufenden Plug-Ins ihre Überwachungstätigkeit.

- Da Nagios, ebenso wie RHQ, ein Monitoring-Tool ist, das zur Überwachung von Systemen geschrieben wurde, hat die statische Beschreibung den entscheidenden Nachteil, dass die zu beobachtenden Systeme, durch zur Laufzeit auftretende Veränderungen, einer ständigen Dynamik unterliegen, die vom statischen

System nicht abgebildet werden können, ohne aufwendige Anpassugen vorzunehmen. Daher bot es sich an, die Entwicklung des Prototypen am Beispiel RHQ und Nagios durchzuführen.

Die in der Umsetzung der Migration am Beispiel RHQ und Nagios gewonnenen Erkenntnisse sollen später auch in das Gesamtsystem einfliessen und für alle Plug-Ins übernommen werden, dies ist aber nicht mehr Teil dieser Arbeit und soll hier, nur der Vollständigkeit halber, erwähnt werden.

1.3 Aufgabenstellung

In diesem Abschnitt soll die Aufgabenstellung der Arbeit detailliert beschrieben werden. Die Aufgabenstellung setzte sich aus verschiedenen Anforderungen zusammen. Zunächst ging es darum, eine allgemeine Anforderungsanalyse für die Realisierung eines Monitoring-Systems zu erstellen, und einen, auf die Anforderungen bezogenen, Lösungsansatz zu beschreiben (siehe auch Abschnitt 1.3.2). Desweiteren sollte ein Plug-In für das RHQ System entwickelt werden, das die Anbindung an das Nagios Monitoring-System realisiert (siehe Abschnitt 1.3.1). Als letzter Punkt sollte ein Systemprototyp implementiert werden, der, am Beispiel RHQ und Nagios, die Migration von der Bereitstellung statischer Metadaten zur Bereitstellung dynamischer Metatadaten realisiert(siehe Abschnitt 1.3.3).

1.3.1 Entwicklung eines Plug-Ins für die RHQ Systemüberwachungssuite zur Anbindung des Nagios Systems

Der erste Teil der Aufgabenstellung beinhaltete die Entwicklung eines Plug-Ins für die RHQ Systemanagement-Suite zur Überwachung des Nagios Systems. Die Realisierung dieses Aufgabenteils umfasste 3 Teilaufgaben, die nachfolgend erläutert werden sollen:

1. **Validierung der vorhandenen Schnittstellen zu Nagios**: Im ersten Teil ging es darum, die vorhandenen Schnittstellen zu überprüfen und zu bewerten, die zur Kommunikation mit dem Nagios System existieren. Dazu sollten zunächst geeignete Kriterien festgelegt werden, anhand derer die Bewertung durchgeführt werden konnte. Da es zahlreiche unterschiedliche Möglichkeiten der Anbindung gibt, mussten die verschiedenen Möglichkeiten verglichen werdenund überprüft werden, ob eine dieser Schnittstellen die notwendignen Kriterien erfüllt, oder ob eine Eigenentwicklung notwendig ist. Im Fall mehrerer geeigneter Schnittstellen sollte die Schnittstelle verwendet werden, die die zugrunde gelegten Auswahlkriterien am Ehesten erfüllt. Maßgebliche Kriterien bei der Bewertung waren Aktualität der Schnittstelle, Lebendigkeit des Projektes und die Verwendung von Standardtechnologien.

2. **Entwurf eines Softwaremodels zur Abbildung des Nagios Systems**: Der nächste Teil umfasste die Konzeption und Implementierung einer Softwarearchitektur, die die Abbildung eines zu überwachenden Nagios Systems ermöglicht. Die Implementierung sollte dem Zweck der korrekten Speicherung der Nagios Systemdaten in entsprechenden Softwaremodulen dienen. Hierbei standen vor Allem die korrekte Abbildung der Systemhierarchie des Nagios

Systems, und die Möglichkeit der Darstellung der Mengenrelationen der einzelnen Elemente zueinander, im Focus.

3. **Implementierung der Plug-In-Komponenten zur Realisierung des Nagios Plug-Ins**: Da RHQ ein Plug-In-basiertes System ist, musste zur Anbindung des Nagios Systems ein Plug-In entwickelt werden. RHQ stellt dem Entwickler zu diesem Zweck das Advanced Management Plugin System (AMPS) zur Verfügung. Diese Bibliothek ist eine Ansammlung aus Interfaces und Klassen, die die wichtigsten Aspekte wie Messwerterfassung, Ressourcensuche usw. ermöglichen. Die Aufgabe bestand darin, die Implementierung des Plug-Ins auf Basis der API, unter Einbeziehung der Schnittstelle zu Nagios und des neu entwickelten Datenmodels zur Abbildung des Nagios Systems, zu realisieren.

1.3.2 Anforderungsanalyse und Beschreibung eines Lösungskonzepts

In diesem Teil der Aufgabenstellung ging es darum, zunächst allgemeine Anforderungen zu definieren, die zu erfüllen sind, wenn man ein verteiltes Monitoringsystem realisieren möchte. Dann sollten die speziellen Anforderungen definiert werden, die erfüllt werden müssen, wenn man von statischer zu dynamischer Metadatenbereitstellung migrieren möchte, wie es für Nagios und RHQ geschehen soll. Die Beschreibung der Anforderungen sollte soweit wie möglich von konkreten Problemen abstrahiert werden, um eine allgemeingültige Anforderungsanalyse zu erhalten, die zur Lösung vergleichbarer Probleme innerhalb anderer Syteme herangezogen werden kann. Neben der Definition der Anforderungen sollten allgemein gültige Lösungsansätze beschrieben werden, mit denen die ermittelten Anforderungen erfüllt werden können. Auch hier ist die Abstraktion von der konkreten Anwendung zur abstrakten, von den technischen Gegebenheiten der einzelnen Systeme unabhägigen, Lösungsmethode ein zentrales Ziel, um die Übertragbarkeit der Lösungen auf vergleichbare Probleme zu gewährleisten. Dieser Teil der Arbeit wird dem wissenschaftlichen Anspruch an eine Master-Thesis gerecht und unterscheidet diese klar von der Durchführung einer Bachelor-Thesis, bei der die Abstraktion der konkreten Problematik nicht zwingend Teil der Arbeit sein muss.

1.3.3 Prototypenrealisierung der Migration von statischer zu dynamischer Metadatenbereitstellung im RHQ System

In diesem Teil der Arbeiti sollte ein erster Prototyp entwickelt werden, der die notwendigen Anpassugnen innerhalb der Komponenten Server, Agent und Plug-In realisiert, so dass Metadaten dynamisch in das System aufgenommen werden können.

Auf Basis dieses Prototyps soll dann in einem späteren Projekt die vollständige Systemmigration zur Verwendung dynamischer Metadaten durchgeführt werden. Die statische Metadatenbereitstellung ist sehr unflexibel gegenüber plötzlich auftretenden Änderungen innerhalb eines überwachten Systems, und macht Anpassungen sehr aufwendig. Die Migration von der statischen zur dynmaischen Metadatenbereistellung sollte in 3 aufeinander folgenden Schritten realisiert werden, die nachfolgend beschrieben werden:

1. **Dynamische serverseitige Generierung neuer Ressourcetypen**: Der erste Schritt auf dem Weg zur dynamischen Metadatenbereitstellung sollte die serverseitige Generierung neuer Ressourcetypen zur Laufzeit des Systems sein. Dabei sollten neue Ressourcetypen vom Benutzer angelegt werden können, diese sollten anschließend in der, an den Server angebundenen, Systemdatenbank persistiert werden. Dieser Schritt sollte die Schaffung einer geeigneten Schnittstelle zum Anlegen neuer Ressourcentypen, und die Implementierung des Datenbankzugriffs, beinhalten.

2. **Dynamische agentenseitige Generierung neuer Ressourcetypen**: Im nächsten Schritt sollte die Realisierung des agentenseitigen Anlegens neuer Ressourcetypen implementiert werden. Dieser Schritt sollte die Implementierung einer geeigneten agentenseitigen Benutzerschnittstelle, die Implementierung der Kommunikation mit dem Server, und die abschließende Persitierung der übertragenen Daten durch die serverseitige Datenbankschnittstelle, beinhalten. Die Kommunikation zwische Server und Agent ist notwendig, um die Synchronisation der Agenten mit dem Server zu gewährleisten, damit eine Übermittlung der neuen Daten vom Agenten zum Server stattfinden kann und Datenkonsistenz zwischen beiden Komponenten gewährleistet ist.

3. **Dynamische Generierung neuer Ressourcentypen im Plug-In**: Als letzter Schritt sollte die Realisierung des Anlegens neuer Ressourcetypen aus dem Plug-In erfolgen. Es sollte die Möglichkeit geschaffen werden, neue Ressourcentypen dynamisch, aus dem Plug-In heraus, in das RHQ System aufnehmen zu können, und die Daten der neuen Ressourcentypen über den Agenten zum Server zu übertragen.

2 Ausgangslage

2.1 Projektumfeld

2.1.1 Das Unternehmen Dillinger Hütte

Die Dillinger Hütte ist ein Hüttenwerk mit einer Geschichte von 325 Jahren. Das Unternehmen wurde im Jahr 1685 gegründet und war im Jahre 1809 die erste Aktiengesellschaft in Deutschland. 1962 wurde die erste Stranggussanlage für Brammen in Betrieb genommen. Im Jahre 1998 wurde eine weitere Anlage für bis zu 400 mm dicke Brammen gebaut, dabei handelt es sich um die derzeit dicksten Strangguss-Brammen der Welt. Heute finden Dillinger Bleche in zahlreichen Projekten weltweit Verwendung, unter anderem in Stahlbrücken, Wolkenkratzern, Offshoreprojekten oder Öl- und Gasleitungen. Das Unternehmen ist Marktführer im Bereich der Grobbleche, aufgrund ihrer weltweit einzigartigen Qualitätsstandards werden diese in zahlreichen Großprojekten mit internationalem Renomee verwendet. Die Dillinger Hütte ist die operative Führungsgesellschaft der Dillinger Hütte Gruppe. Dillingen an der Saar ist gleichzeitig Standort für die verbundenen Unternehmen Zentralkokerei Saar (ZKS) und Roheisengesellschaft Saar mbH (ROGESA). Ferner wird auch im Walzwerk von GTS Industries in Dünkirchen, das seit 1992 zur Gruppe gehört, Grobblech produziert. Der in Mühlheim beheimatete Großrohrhersteller Europipe, weltweit führender Anbieter von Rohren für den Erdöl- und Erdgastransport, wurde 1991 als 50-prozentige Tochter der Dillinger Hütte gegründet und verfügt heute über Produktionsstätten in Deutschland, Frankreich, den USA und Brasilien. Im Jahr 2009 beschäftigte die AG der Dillinger Hüttenwerke insgesamt 5907 Mitarbeiter. Die Rohstahlproduktion betrug insgesamt 1922 Kilotonnen, die Grobblechproduktion 1609 Kilotonnen. Die AG der Dillinger Hüttenwerke erzielte Umsatzerlöse in Höhe von 2161 Mio. Euro, die Umsätze der Gesamtgruppe betrugen 2258 Mio. Euro. Die Dillinger Hütte Gruppe erzielte trotz der sehr schwierigen Wirtschaftslage einen Gewinn von 131 Mio. Euro vor Zinsen und Steuern[5]. Der Standort Dillingen gliedert sich in folgende Teilbetriebe:

- **Kokerei:** Die Kokerei stellt den für die Roheisenherstellung notwendigen Koks her. Hierzu wird Steinkohle unter Luftabschluss 24 Stunden lang auf ca. 1.200 °C erhitzt, übrig bleibt der feste und stark porige Koks. Als Nebenprodukte entstehen dabei unter anderem Teer und Benzol, die im Straßenbau bzw. in der chemischen Industrie zum Einsatz kommen. Der Koks selbst ist für den Wärmebedarf des Hochofens und den Entzug des Sauerstoffes aus dem Eisenerz (Reduktion) zuständig.

[5] DH1

- **Sinteranlage:** Die für die Eisenherstellung benötigten Feinerze müssen vor dem Einsatz im Hochofen stückig gemacht werden. Dies geschieht in der Sinteranlage. Beim Sintern werden die feinkörnigen Erze mit Feinkoks und sogenannten Schlackebildern zu kompakten Stücken zusammengebacken. Der so entstandene Rohstoff ist fertig für den Hochofeneinsatz und verspricht aufgrund seiner chemischen und mechanischen Eigenschaften gute Reduzierbarkeit und hohen Eisengehalt.

- **Hochofen:** Im Hochofenverfahren wird flüssiges Roheisen erzeugt. Als Nebenprodukte fallen Schlacke und Gichtgas an. Dabei wird durch Blasformen Heißwind eingeleitet, dieser verbrennt den Koks. Die sich dabei bildenden und aufsteigenden Reaktionsgase erhitzen die Beschickung (Stückerze, Sinter und Pellets) und lösen umfangreiche chemische Reaktionen aus. Dadurch werden die Eisenoxide reduziert. Bei Temperaturen von über 1.500 °C sind das Roheisen und die Schlacke flüssig. In kurzen, regelmäßigen Abständen wird der flüssige Hochofeninhalt abgestochen und weiterverarbeitet.

- **Stahlwerk:** Im Stahlwerk wird das Roheisen zu Stahl verarbeitet. Hierzu sind zwei Schritte notwendig: das Frischen und die sekundärmetallurgische Behandlung. Frischen ist der Fachbegriff für Verbrennen oder Oxidieren. Hohe Anteile von Begleitelementen im Roheisen müssen in diesem Prozess im Konverter verbrannt oder in der Schlacke gebunden werden. Ziel des Frischens ist die Senkung des Kohlenstoffgehalts sowie weiterer Begleitelemente auf geforderte Werte bzw. die komplette Entfernung unerwünschter Elemente. Die moderne Stahlerzeugung wird heute im Wesentlichen durch die Nachbehandlung der Schmelze außerhalb des Konverters mittels Legieren, Spülen und Entgasen geprägt. Diese sogenannte Sekundärmetallurgie zielt auf eine weiter verbesserte Stahlqualität sowie auf eine Stabilisierung des Herstellungsprozesses ab.

- **Walzwerk:** Der im Stahlwerk erschmolzene und vergossene Stahl (entweder im kontinuierlichen Stranggießverfahren oder im Blockguss) wird im Walzwerk in das für den Stahlverbraucher verwendbare Blech umgeformt. In diesem Prozess werden auch die technologischen Eigenschaften des Stahls verändert. Hierzu muss der Stahl weiter behandelt werden. An erster Stelle steht dabei das Walzen. Das Walzen von Stahl ist ein stetiges oder schrittweises Umformen mit Hilfe von mehreren sich drehenden Walzen. Durch den dabei entstehenden Druck kommt es zu plastischen Verformungen, so dass sich das Gefüge des Bleches und somit die mechanischen Eigenschaften positiv ändern.

[6]

[6] DH2

2.1.2 Der IT Bereich der Dillinger Hütte

Der IT Bereich der Dillinger Hütte gliedert sich in 3 Abteilungen, deren Funktionen im Folgenden kurz erläutert werden sollen.

2.1.3 Technische Informatik(TI)

Die Abteilung 'Technische Informatik' hat die Aufgabe der Entwicklung und Pflege der selbst implementierten, oder zugekauften, Software- und Datenbanksysteme für die Betriebe Hochofen, Stahlwerk, Brammenadjustage, Walzwerk, Abnahme und Transportbetrieb. Die Teilbetriebe der TI sind:

- Hochofenrechner: Betrieb und Wartung des Hochofenrechnersystems
- Stahlwerksrechner: Betrieb und Wartung des Stahlwerksrechnersystems
- Gerüstrechner: Betrieb und Wartung des Gerüstrechnersystems
- Betriebsrechner: Pflege und Wartung sämtlicher Software für das Betriebsrechnersystem
- Abnahmerechner: Betrieb und Wartung des Abnahmerechnersystems
- Software Engineering: Erstellung und Pflege von Basis Software zur Nutzung in den einzelnen Rechnerbereichen

2.1.4 Kaufmännische Informatik(KI)

Die Abteilung 'Kaufmännische Informatik' hat die Aufgabe, Informationssysteme zur Verfügung zu stellen, indem sie selbst Software entwickelt und pflegt oder Standardsoftware einsetzt und für Anwenderschulung und -beratung sorgt. Die KI gliedert sich in folgende Bereiche:

- CRM(Customer Relationship Management): Dieser Bereich betreut die Informationssysteme für die Verwaltung der Kundendaten und die Angebots- und Anfragebearbeitung.
- AKIS(Auftrags- und Kundeninformationssystem): Dieser Bereich betreut Informationssysteme für die kaufmännische Auftragsabwicklung, wie z.B. Versand/Distribution, Reklamationen, Auftragsverfolgung, Verkaufsplanung usw.
- Anwendungen Logistik und Rechnungswesen: Dieser Bereich betreut die Informationssysteme für die logistischen Prozesse und für die Anwendungen des internen und externen Rechnungswesens der Dillinger Hütte und der Tochterunternehmen.

- Personalwesen: Dieser Bereich betreut die Informationssysteme für das Personalwesen, wie z.B. Personalmanagement, Personalabrechnung, Veranstaltungsmanagement.

2.1.5 Informationstechnik(IT)

Die Abteilung 'Informationstechnik' hat die Aufgabe, IT-übergreifende, informationstechnische Basisdienstleistungen zu erbringen. Dies sind im Wesentlichen:

- die Auswahl und Standardisierung zentraler und übergreifender Hard- und Softwaresysteme
- der Betrieb zentraler Server- und Speichersysteme
- der Betrieb grundlegender technischer Infrastrukturdienste
- die Betreuung und der Betrieb der beiden zentralen Rechenzentren am Standort Dillingen
- die technische Bereitstellung der SAP-Systeme, von zentralen Datenbanksystemen und von Software-Entwicklungsumgebungen
- die Bereitstellung eines zentralen Informatik Anwenderservice

2.2 Begriffsmodel

In diesem Abschnitt sollen die wichtigsten Begriffe genauer definiert werden, die im Laufe dieser Arbeit immer wieder auftauchen und daher von zentraler Bedeutung sind. Abbildung 3 zeigt das Begriffsmodel der Arbeit. Das Model ist aufgeteilt in die Begriffe zur Beschreibung des RHQ und des Nagios Systems. In den Abschnitten 2.2.1 und 2.2.2 werden die Begriffe des RHQ und Nagios Systems genauer erläutert.

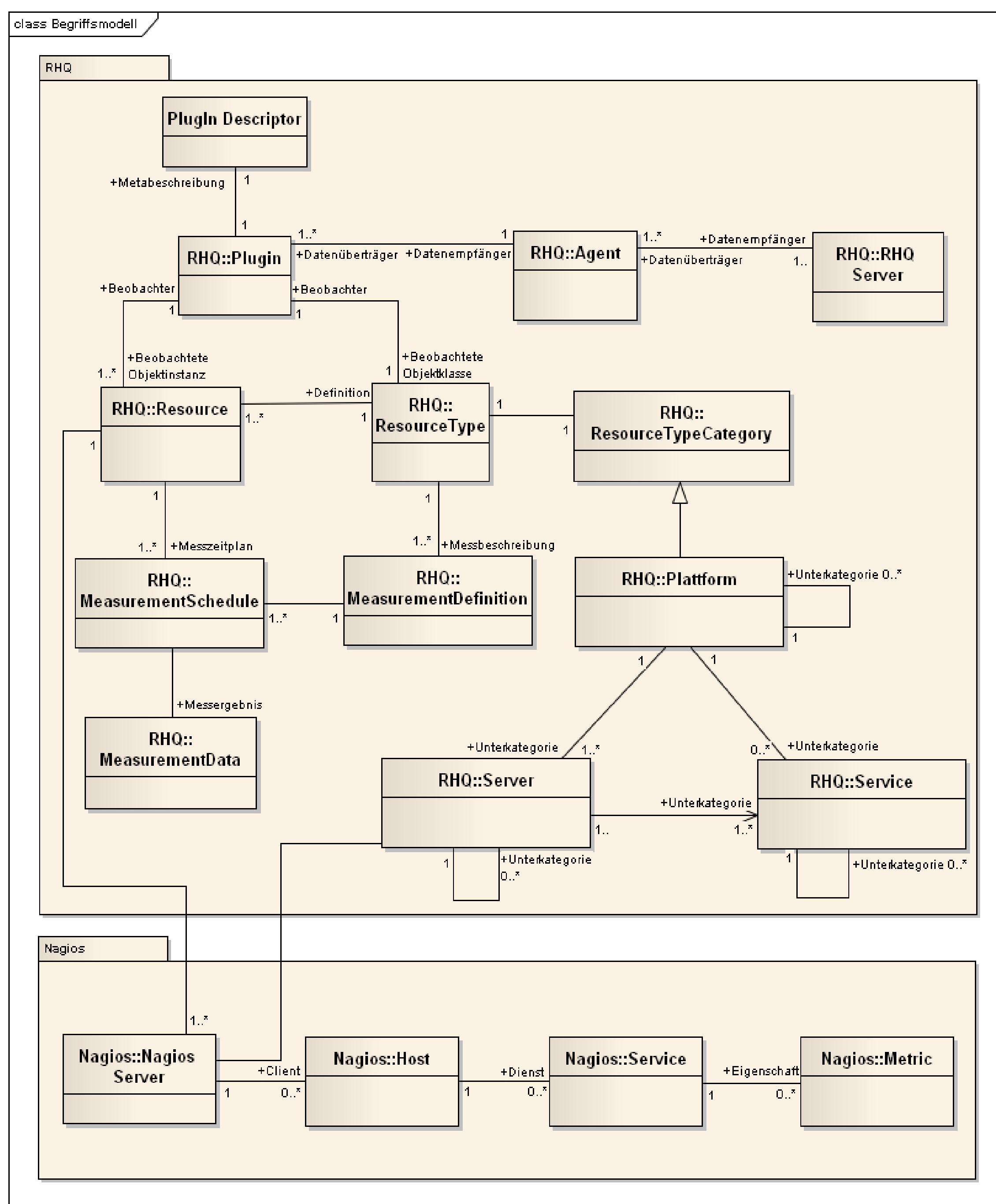


Abbildung 3: Begriffsmodel

2.2.1 RHQ

In diesem Abschnitt werden die wichtigsten Begriffe des RHQ Systems erklärt. Zunächst beginnen wir mit den Architekturkomponenten des RHQ Systems. Das RHQ System setzt sich zusammen aus den folgenden Komponenten:

- **RHQ Server**: Der RHQ Server stellt die zentrale Architekturkomponente des RHQ Systems dar, hier laufen alle Informationen zusammen, die Datenbankanbindung wird realisiert, und die grafische Präsentation der Messdaten erfolgt. Alle Plug-Ins werden vor der Inbetriebnahme durch den Server auf Korrektheit überprüft, dazu wird der, zum jeweiligen Plug-In zugehörige, Plug-In Deskriptor geparsed. Nach erfolgreicher Evaluation wird das Plug-In den Agenten zum Download zur Verfügung gestellt. Der Server bietet außerdem eine Konfigurationsschnittstelle und führt andere wichtige Aufgaben, wie Alerting und Logging, durch.

- **Agent**: Die Agenten stellen die verteilte Komponente des Systems dar. Beim Start eines Agenten erfolgt eine Synchronisation mit dem Server, anschließend werden alle verfügbaren Plug-Ins heruntergeladen. Das bedeutet, dass die Plug-Ins vom Server aus an alle laufenden Agenten verteilt werden, sobald ein Agent hochgefahren wurde und sich mit dem Server synchronisiert hat. Auf jedem, vom RHQ System zu überwachenden, Host läuft ein Agent, dieser interagiert mit den verschiedenen Plug-Ins und erhält so Messdaten der durch die Plug-Ins überwachten Ressourcen. Die Daten werden vom Agenten an den Server weiter geleitet, außerdem führt der Agent weitere Aufgaben wie Logging durch.

- **Plug-In**: Die Plug-Ins implementieren die Schnittstelle zwischen dem RHQ System und dem Fremdsystem, sie übernehmen die Aufgabe der Ressourcenüberwachung und Messdatenermmittlung. Jedes Plug-In wird für einen bestimmten Resourcentyp implementiert und überwacht alle Resourcen dieses Typs. Die vom Plug-In gewonnenen Informationen werden an den jeweiligen Agenten übermittelt, dieser leitet die Daten an den zentralen Server weiter.

  Plug-In-Deskriptor: Zu jedem Plug-In gibt es genau einen Plug-In Deskriptor. Der Plug-In Deskriptor ist eine Metabeschreibung des Plug-Ins in XML, und spezifiziert die Aufgaben und Funktionsweise des Plug-Ins genauer. Er enthält den Name des Plug-Ins, eine allgemeine Beschreibung, die Namen der Discovery und Component Klassen und die Version des RHQ Servers. Außerdem werden im Plug-In-Deskriptor die Ressourcen, die überwacht werden sollen, und die dazugehörigen Metriken beschrieben.

Im Begriffsmodel unterhalb der Architekturkomponenten liegen die Begriffe der Ressourcenbeschreibung und Messwerterfassung. Sie dienen der Abbildung und Beschrei-

bung von zu überwachenden Ressourcen und den bei der Überwachung der Ressourcen gelieferten Messergebnissen, und dienen der Abstraktion des Überwachungsvorgangs. Die wichtigsten Begriffe hierbei sind:

- **ResourceType** Unter einem ResourceType versteht man den Typ einer Resource, also eine Kategorie oder Klasse, in die man diese Ressource einordnen kann. Beispielsweise haben mehrere Netzwerkkarten verschidener Hersteller dennoch alle den gleichen ResourceType 'Netzwerkkarte'.

- **Resource** Angelehnt an den den Begriff des ResourceType ist der Begriff Resource. Während der Begriff ResourceType vergleichbar ist mit einer bestimmten Klasse, stellt der Begriff Resource eine konkrete Instanz dieser Klasse dar. Am Beispiel der Netzwerkkarte stellen die Karte eth0 von Sony und eth1 von Toshiba verschiendene Resourcen vom ResourceType 'Netzwerkarte' dar.

- **ResourceTypeCategory** Jede Resource ist einer bestimmten ResourceType-Category zugeordnet, die Kategorien sind hierachisch gegliedert. So können die verschiedenen Ressourcen in Hierarchien eingegliedert werden, dies ermöglicht die Abbildung von Parent-Child-Beziehungen und ist für die Funktionsweise des Systems sehr wichtig, wie im weiteren Verlauf dieser Arbeit noch erläutert werden wird. Es gibt 3 verschiedene Ressourcenkategorien:

  Platform: Die Kategorie 'Platform' ist die in der Hierarchie oberste Komponente. Sie ist Resourcen zugeordnet, die komplette Betriebssysteme oder betriebssystemähnliche Komponenten wie virtuelle Maschinen abbilden. Alle darin laufenden Anwendungen sind klar klassifiziert und laufen oft nur unter dieser Plattform, z.B. eine speziell für Windows entwickelte Anwendung oder eine Java Anwendung. Innerhalb einer Plattform können weitere Plattformen, Server und Services betrieben werden.

  Server: Die Kategorie 'Server' liegt in der Systemhierarchie unterhalb der Plattform und ist Ressourcen zugeordnet, die Teilsysteme innerhalb kompletter Systemumgebungen darstellen. Beispiele hierfür sind Datenbanksysteme, Applikationsserver und Webserver. Charakteristisch für Ressourcen der Kategorie 'Server' ist es, dass sie der Umgebung, in der sie betrieben werden, andere Ressourcen der Kategorie 'Server' oder 'Service' zur Verfügung stellen.

  Service: Die Kategorie 'Service' stellt die unterste der 3 Hierarchieebenen dar und wird für Ressourcen verwendet, die von Ressourcen der Kategorie 'Server' als Dienst zur Verfügung gestellt werden. Oftmals sind es einzelne Anwendugen, wie z.B eine Webanwendung, die von einem bestimmten Webserver betrieben wird. Innerhalb eines Service können weitere Services angeboten werden.

- **MeasurementDefinition** Eine MeasurementDefinition ist eine genaue Spezifikation eines zu ermittelnden Messwertes für einen bestimmten ResourceType. Beim Parsen des Plug-In Deskriptors wird für jedes XML-Tag 'Metric' ein Objekt vom Typ MeasurementDefinition erzeugt und zu dem dazugehörigen ResourceType hinzugefügt, gleichzeitig wird ein entsprechendes Feld in der Systemdatenbank angelegt. Dadurch wird dem System mitgeteilt, dass für den entsprechende ResourceType genau diese Metrik ermittelt werden soll, und das entsprechende Plug-In wird auf allen Agenten mit der Ermittlung von Messwerten für diese Metrik, von diesem RessourceType, beginnen.

- **MeasurementSchedule** Der MeasurementSchedule ist der Zeitplan, nach dem die Errmittlung der Messwerte für die Metriken des ResourceTypes durchgeführt wird. Der Schedule kann vom Benutzer, mithilfe der Benutzerschnittstelle, für alle vom System überwachten ResourceTypes individuell festgelegt werden, und bestimmt damit, in welchen zeitlichen Abständen die Messwerte der zu überwachenden Ressourcen ermittelt werden. Legt der Anwender nicht explizit Werte fest, so werden Standardzeitintervalle zur Messwerterfassung genommen. Entsprechend dieser Intervalle werden dann die verschiedenen Plug-Ins ihre Messungen in regelmässigen Abständen durchführen und ihre Daten, über den jeweiligen Agenten, an den Server weiterleiten.

- **MeasurementData** Die MeasurementData sind die bei der Überwachung der Ressourcen gemessenen Werte, diese werden von den Plug-Ins ermittelt und an die jeweiligen Agenten weiter geleitet. Von dort werden sie an den Server übermittelt, dort aufbereitet, den einzelnen Ressourcen zugeordnet und auf der Benutzeroberfläche visualisiert.

2.2.2 Nagios

In diesem Abschnitt werden die wichtigsten Begriffe des Nagios Models erklärt. Das Nagios Model ist eine Softwareabbildung des realen Nagios Systems und enstand, im Zuge dieser Arbeit, als vereinfachte Abstraktion von Nagios. Die vereinfachte Darstellung war für den Zweck der Arbeit ausreichend, da es lediglich darum ging, die, über die Schnittstelle zum Nagios System, ermittelten Daten in Software abbilden zu können. Hier werden nur die Elemente beschrieben, die im weiteren Verlauf der Arbeit eine Rolle spielen. Eine komplette Beschreibung des implementierten Models erfolgt bei der Beschreibung der Projektrealisierung in Kapitel 4.3.2. Das Nagios Model besteht aus 4 hierarchisch gegliederten Komponenten:

- Nagios Server: Die Komponente die den Nagios Server des Systems abbildet. Dort laufen alle Informationen der einzelnen Hosts zusammen, der Server verfügt also über alle Hostinformationen und verwaltet diese.
- Host: Diese Komponente bildet einen einzelnen Hosts eines Nagios Systems ab. Jeder Rechner, auf dem ein Nagios Agent läuft, wird als Host abgebildet, auf den Hosts laufen wiederum die einzelnen Services, die dem Host alle bekannt sind.
- Service: Die Service Komponente bildet einen Dienst ab, der auf einem von Nagios überwachten Host läuft. Die relevanten Informationen der Dienste werden durch Messungen ermittelt und als Metriken erfasst.
- Metric: Eine Metric stellt eine Information über einen bestimmten Dienst quantitativ dar, etwa die Bandbreite einer Netzwerkkarte. Sie ist stets einem bestimmten Dienst zugeordnet und wird durch regelmässige Messungen ermittelt.

3 Anforderungsanalyse und Lösungskonzept

In diesem Kapitel erhält der Leser in Abschnitt 3.1 zunächst eine Einführung in den Aufbau und die Funktionsweise einer Systemmanagement-Suite. Danach wird eine Anforderungsanalyse an eine Systemmanagement-Suite erstellt und ein Lösungskonzept für die entsprechenden Anforderungen vorgestellt. Dieses Lösungskonzept soll soweit von der konkreten Problematik abstrahiert werden, dass es konzeptionell auf vergleichbare Probleme übertragen und zu deren Lösung herangezogen werden kann. Dabei werden in Abschnitt 3.2 zunächst die Anforderungen betrachtet, die sowohl bei der Verwendung von statischer als auch dynamischer Metadatenbereitstellung erfüllt sein müssen, in Abschnitt 3.3 folgt dann die Betrachtung der für die Migration zur Verwendung dynamischer Metadaten relevanten Anforderungen. Zum Abschluß des Kapitels werden in Abschnitt 3.4 die im Projektkontext wichtigsten Anwendungsfälle eines solchen Systems beleuchtet, hier efolgt eine Unterscheidung der Anwendungsszenarien, je nachdem ob statische oder dynamische Metadaten verwendet werden.

3.1 Schematischer Aufbau einer Systemüberwachungssuite

Systemmanagement-Suites sind häufig verteilte agentenbasierte Systeme, die aus einer zentralen Servereinheit , einem oder mehreren Agenten bestehen und verschiedenen Plug-Ins bestehen, die auf verschiedenen Hosts bzw. Plattformen laufen. Auf jedem zu überwachenden Host wird ein Agent installiert, der Agent stellt dabei die Kommunikationsschnittstelle zum Server und einige weitere grundlegenden Funktionen wie Logging zur Verfügung. Die eigentliche Überwachung der Laufzeitumgebung auf den einzelnen Hosts wird durch Plug-Ins realisiert, jedes Plug-In wird normalerweise zur Überwachung eines bestimmten Ressourcentyps implementiert. Wie bereits in Kapitel 1.1 erwähnt, entspricht der Ressourcentyp einer Klasse, während eine Ressource der Instanz der Klasse entspricht. Ein Plug-In kann daher sämtliche Ressourcen eines bestimmten Ressourcentyps auf dem Agenten auf dem es läuft überwachen. Die vom Plug-In gesammelten Informationen werden über den Agenten zum Server weitergeleitet, wo sie ausgewertet und aufbereitet werden. Der Vorteil dieser verteilten Systemarchitektur ist die Schlankheit des Kernsystems und die Auslagerung eines großen Teils der Ressourcenbeanspruchung auf die externen Hosts mit den Agenten. Der Server erfüllt nur die zentralen Aufgaben die das Gesamtsystem betreffen, dies sind im Wesentlichen die Bereitstellung einer zentralen Benutzerschnittstelle zur Konfiguration und Wartung des Systems, die Aufbereitung und Präsentation der erfassten Messwerte und die Anbindung an eine Datenbank. Der Server sollte in einer 3-Schichten-Architektur realisiert werden, so dass die Module Benutzerschnitt-

stelle(Presentation), Datenverarbeitung(Logic) und Datenbankzugriff(Data Access) sauber voneinander getrennt sind und Module erweitert oder ausgetauscht werden können. Die Schnittstelle zwischen dem Plug-In und dem Agent sollte standardisiert sein, dies ermöglicht die universelle Benutzbarkeit der Plug-Ins mit allen Agenten und eine standardisierte Plug-In-Entwicklung gegen eine bestimmte Schnittstelle. Die Schnittstelle zwischen den Plug-Ins und den durch sie überwachten Ressourcen ist abhängig vom Ressourcentyp, hier ist eine standardisierte Schnittstelle nicht möglich, da z.B eine Netzwerkarte eine andere Anbindung benötigt als ein Applikationsserver oder ein Sensor. Die Aufgaben der Komponenten Server, Agent und Plug-In werden in den Abschnitten 3.1.1, 3.1.2 und 3.1.3 genauer beschrieben. Abbildung 4 zeigt den schematischen Aufbau eines verteilten Monitoring-Systems.

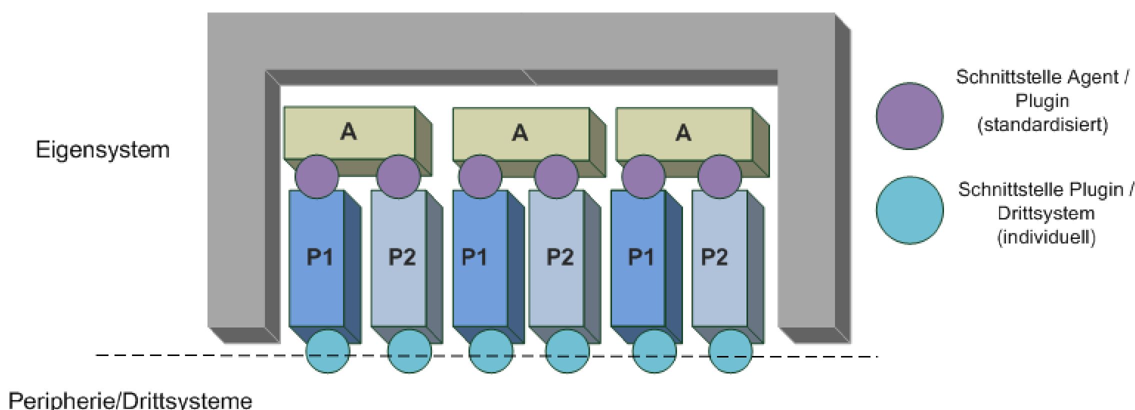


Abbildung 4: Schematischer Aufbau einer Systemüberwachungssuite

3.1.1 Server

Der Server stellt die zentrale Komponente einer Monitoring Suite dar. Seine wichtigsten Aufgaben sind:

- **Konfiguration**: Der Server implementiert die Präsentationsschicht in Form einer Benutzerschnittstelle, mit der für das Gesamtsystem notwendige Konfiguratiosneinstellungen vorgenommen werden können.
- **Monitoring**: Als weiteres Modul der Präsentationsschicht führt der Server die grafische Darstellung der Messdaten aus. Er bietet eine Übersicht über alle vom System beobachteten Ressourcen und ermöglicht eine gezielte Auswahl nach bestimmten Kriterien.
- **Messdatenverabeitung**: Der Server implementiert die Logikschicht, dort werden die ankommenden Messdaten verarbeitet und ausgewertet, bevor sie in die Datenbank geschrieben oder in der GUI dargestellt werden.

- **Persistierung**: Die im laufenden Systembetrieb gesammelten Daten und die vom Systembenutzer vorgenommen Einstellungen müssen in einer Datenbank gespeichert werden, damit sie später wieder abgerufen und ausgewertet werden können. Der Server implementiert die Persistenzschicht als Schnittstelle zur Datenbank und ermöglicht so das Lesen und Schreiben in der Datenbank.

- **Logging**: Der Server muss während des Betriebes den Ablauf seiner wichtigsten Vorgänge mitloggen, um dem Benutzer im Fehlerfall- oder Entwicklungsfall eine gezielte Nachverfolgung der Systemabläufe zu ermöglichen.

- **Scheduling**: Im Server laufen sämtliche Informationen aller Agenten zusammen , daher muss dort die Ressourcenzuweisung für die einzelnen Agenten geschehen.

- **Plug-In Management**: Der Server muss die Funktionalität zum Hinzufügen und Entfernen von Plug-Ins bieten, es muss eine initiale Überprüfung der korrekten Plug-In Funktion durchgeführt werden. Nach erfolgreicher Validierung sind die Plug-Ins zentral verfügbar und können von allen laufenden Agenten geladen werden.

- **Zentrales Inventar**: Der Server verfügt über ein zentrales Inventar mit allen vom gesamten System gemanageten Ressourcen und gleicht dies regelmäßig mit der Systemdatenbank ab.

- **Alarmierung**: Der Server bietet dem Benutzer die Möglichkeit der Definition von Ereignissen, deren Auftreten eine Alarmierung zur Folge hat. Außerdem ermöglicht er dem Benutzer die genaue Definition von Handlungsszenarien im Alarmierungsfall.

3.1.2 Agent

Der Agent stellt die verteilte Komponente der Systemmanagement-Suite dar. Seine wichtigsten Aufgaben sind:

- **Konfiguration**: Wie der Server muss auch der Agent dem Benutzer die Möglichkeit notwendiger Konfigurationseinstellungen bieten, allerdings in geringerem Umfang als beim Server und via Konsole statt GUI.

- **Logging**: Der Agent muss während des Betriebes den Ablauf seiner wichtigsten Vorgänge mitloggen, um dem Benutzer im Fehlerfall- oder Entwicklungsfall eine gezielte Nachverfolgung der Systemabläufe zu ermöglichen.

- **Upgrading**: Der Agent bietet dem Benutzer die Möglichkeit während des Betriebs notwendige Upgrades vom Server zu laden und führt bei jedem Start eine automatische Synchronisation mit dem Server durch.
- **Scheduling**: Da im Agent die Steuerung sämtlicher von ihm betriebenen Plug-Ins vorgenommen wird, muss dort die Ressourcenzuweisung für die Plug-Ins geschehen, so dass eine optimale Ressourcenauslastung erzielt werden kann.
- **Kommunikation**: Der Agent ist für die Kommunikation zum Server verantwortlich und muss alle dafür notwendigen technischen Vorraussetzungen erfüllen.
- **Lokales Inventar**: Der Agent führt ein lokales Inventar der von ihm gemanageten Ressourcen und synchronisiert dieses Inventar regelmäßig mit dem zentralen Inventar des Servers.

3.1.3 Plugin

Die Plug-Ins stellen die Schnittstelle zwischen dem Eigensystem und den zu überwachenden Drittsystemen dar, ihre wichtigsten Aufgaben sind:

- **Überwachung und Datengewinnung von Drittsystemen:** Die Plug-Ins überwachen kontinuierlich die Drittsysteme und ermitteln in festgelegten Zeitabständen Messdaten aller Ressourcen des von ihnen überwachten Ressourcentyps.
- **Abbildung der Messdaten des Fremdsystems in einem geeigneten Datenmodel:** Die ermittelten Messdaten müssen im Plug-In geparsed und in einem Datenmodel, welches das Fremdsystem abbildet, gspeichert werden.
- **Überführung der Messdaten aus dem Datenmodel des Fremdsystems in das Model des Eigensystems:** Das Plug-In hat die Aufgabe die Daten aus dem Model des Drittsystems in das Datenmodel des Eigensystems zu überführen.
- **Übermittlung der Daten an den Agenten:** Die in das Datenmodel des Eigensystems überführten Messdaten müssen an den jeweiligen Agenten übertragen werden.

3.2 Realisierung unter Verwendung statischer Metadaten

In diesem Abschnitt werden die Anforderungen an ein verteiltes Monitoring-System unter Verwendung statischer Bereitstellung von Metadaten beschrieben. Die Anforderungen ergeben sich aus der in Kapitel 3.1 beschriebenen Systemarchitektur und der Aufgabe der einzelnen Systemkomponenten. Es werden dabei die Kernanforderungen beschrieben die an verteilte Monitoring-Systemen gestellt werden müssen. Die in diesem Abschnitt beschriebenen Grundanforderungen an ein Monitoring-System gelten sowohl für Systeme mit statische Metadatenbereitstellung als auch auch für Systeme mit dynamischer Metadatenbereistellung. Zur Realisierung dynamischer Systeme müssen darüber hinaus weitere Anforderungen erfüllt werden, diese werden in Abschnitt 3.3 gesondert beschrieben.

3.2.1 Datenmodel zur Abbildung des Eigensystems

Sowohl bei der Verwendung von statischer als auch von dynamischer Bereitstellung von Metadaten besteht die Notwendigkeit, ein flexibles Datenmodel als Grundlage des eigenen Systems zu implementieren, auf dessen Basis die Aufgaben des Systems wahrgenommen und die datentechnische Abbildung der Informationen gewährleistet werden können. Das Model soll eine hierarchische Systemstruktur abbilden können und gleichzeitig so allgemein gehalten werden ,dass es die auftretenden Daten der beobachteten heterogenen Drittsysteme repräsentieren kann. Sowohl bei der statischen als auch bei der dynamischen Metadatenbereitstellung besteht die Notwendigkeit der Verwendung flexibler Datenstrukturen, da in beiden Fällen das System um neue Ressourcen und Metriken erweitert werden kann. Der Unterscheid besteht lediglich darin, dass im Falle statischer Metadatenbereitstellung ein Systemneustart durchgeführt werden muss und im Falle dynamischer Bereitstellung von Metadaten nicht. Es besteht aber die generelle Notwendigkeit, die Datenstrukturen an den Stellen des Systems, an denen Änderungen vorgenommen werden können, generisch und flexibel zu halten. Dies ist vor allem beim Überführen der Daten aus dem Datenmodel des Fremdsystems in das Datenmodel des Eigensystems der Fall, da hier immer die aktuelle Anzahl der unter Beobachtung stehenden Ressourcen abgebildet werden muss und diese Anzahl sich jederzeit, abhängig von der Erreichbarkeit der Ressourcen, ändern kann. Diese Änderungen müssen softwaretechnisch abgebildet werden können, innerhalb des Kernsystems empfiehlt es sich deshalb, bereits beim Design der Architektur zur statischen Metadatenverwendung eine Struktur zu wählen, die auch in Hinsicht auf etwaige Erweiterungen oder Umstellungen noch verwendet werden kann. Sie muss sowohl die Hierarchie zwischen den Komponenten des Systems abbilden können als auch die Möglichkeit zu nachträglichen Anpassungen aufgrund zur Laufzeit auftretender Veränderungen bieten. Bei einem agentenbasier-

ten Monitoring-System mit Plug-In-Verwendung bietet sich als Grundmodel eine Struktur wie in Abbildung 5 an. Die Struktur bietet die notwendige Flexibilität hinsichtlich der Erweiterbarkeit und kann hierarchisch in Form eines Baums abgebildet werden. Dadurch kann eine Parent-Child-Beziehung zwischen den Komponenten hergestellt werden, was eine nachträgliche dynamische Erweiterung bei gleichzeitiger Abbildung von Beziehungen zwischen Elementen des Systems ermöglicht. Diese Struktur ist einfach gehalten und kann je nach Anforderungen und Systemdesign erweitert werden. Sie beinhaltet aber alle zur funktionalen Beschreibung eines Monitoringsystems benötigten Komponenten und ist daher vollständig für die Abbildung eines solchen Systems geeignet.

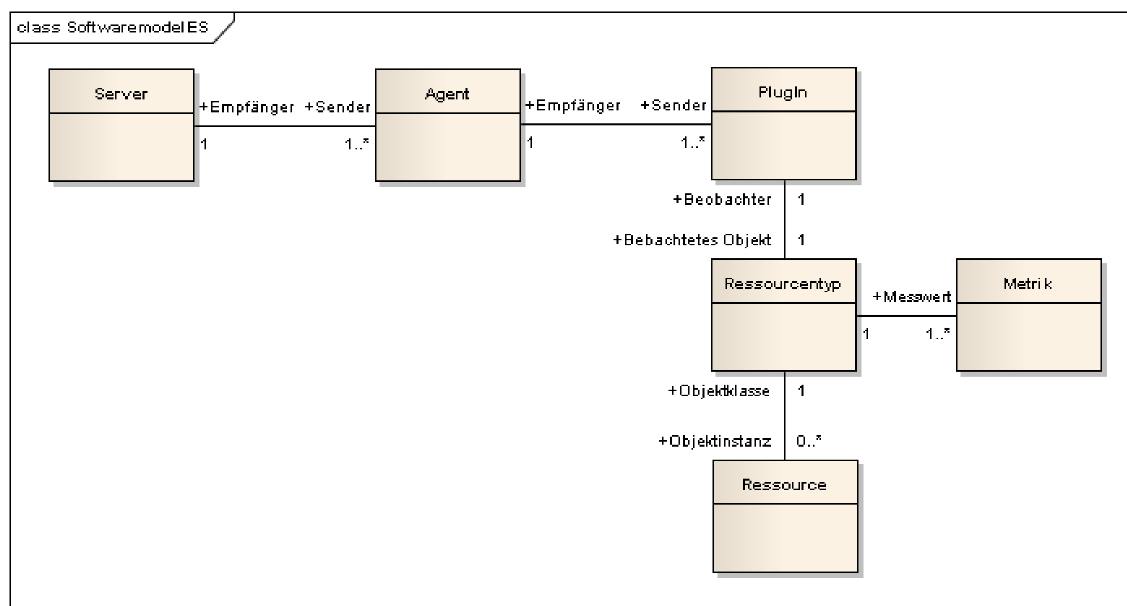


Abbildung 5: Generisches Datenmodel eines agentenbasierten Systems

3.2.2 Beschreibung der zu überwachenden Ressourcentypen

Beim Design einer Systemmanagement-Suite muss vorher entschieden werden, wie der Systemaufbau und die Kernfunktionalität des Systems beschrieben werden kann. Man benötigt ein geeignetes Softwaremodel, welches die Hierarchie und Mengenrelation zwischen den Architekturkomponenten Server, Agent und Plug-In und den von den Plug-Ins zu überwachenden Ressourcentypen darstellen kann. Darüber hinaus benötigt man ein darauf abgestimmtes Datenbankmodel zur Persistierung der gewonnen Daten und der daraus hergeleiteten Informationen in einer Datenbank. Dem System müssen außerdem bei der Inbetriebnahme der einzelnen Plug-Ins Informationen darüber mitgegeben werden, welche Ressourcentypen das Plug-In überwachen soll und welche Metriken für den jeweiligen Ressourcentyp von Interesse sind. Zu diesem Zweck bietet sich eine statische Beschreibung unter Verwendung von Meta-

sprachen an, man spricht daher auch bei der Beschreibung der zu erfassenden Daten von Metadaten, da sie noch keine konkreten Messdatendaten darstellen, sondern lediglich eine Beschreibung der später zu erfassenden Messdaten. Die Verwendung von Metasprachen zur statischen Beschreibung der Systemkomponenten bietet folgende Vorteile:

- Metasprachen wie XML sind standardisiert und dadurch einfach und universell einsetzbar
- die Abbildung auch komplexer Systemstrukturen ist duch die Erstellung eines entsprechende Metasprachenschemas schnell und effektiv möglich
- für die Verwendung von Metasprachen und den zur Auswertung notwendigen Parsing-Prozess existieren APIs in zahlreichen Programmiersprachen
- für die Implementierung der notwendigen Systemfunktionen wie Mapping und Persitierung können daher die gleichen Programmiersprachen wie für das Parsing der Metadaten verwendet werden

Allerdings fehlt dem System durch die statische Metabeschreibung bei der Implementierung oder Erweiterung des Plug-Ins die notwendige Flexibilität bezüglich dynamischer Anpassugen an veränderte Gegebenheiten. Das bedeutet, dass sowohl die jeweiligen Ressourcentypen als auch die zugehörigen Metriken bereits zum Implementierungszeitpunkt des Plug-Ins bekannt sein müssen. Systemmonitoring-Tools unterliegen aber einer gewissen Dynamik und Nicht-Deterministik, so ist es zum Beispiel denkbar, dass zur Laufzeit eines Plug-Ins neue Ressourcentypen hinzukommen und erfasst werden müssen oder bestehende Ressourcentypen um neue Metriken erweitert werden. Dies kann beispielweise durch das Einstecken einer weiteren Netzwerkkarte in einem überwachten Server der Fall sein. Daher ist die statische Beschreibungweise bei der Plug-In-Implementierung unflexibel und hat einige entscheidende Nachteile:

- die Metadatenbeschreibung muss für jedes Hinzufügen oder Entfernen eines neuen Ressourcentyps, oder für Erweiterungen um neue Metriken, vom Entwickler angepasst werden.
- der Quellcode muss bei Änderungen der Metadatenbeschreibung ebenfalls angepasst werden, falls im Code direkter Bezug auf die Metadaten genommen wird
- jede Änderung der Metabeschreibung oder des Quellcodes bedingt einen neuen Build-Prozess und ein anschließendes Re-Deployment der Plug-Ins

- die einzelnen Agenten müssen das neue oder veränderte PLug-In neu vom Server laden und müssen in dieser Zeit die Überwachung der Systemumgebung unterbrechen

Aufgrund dieser Nachteile der statischen Beschreibung bei der Implementierung ist es sinnvoll, eine dynamische Möglichkeit zur Einbindung neuer Ressourcetypen und deren Metriken zu finden. Die statische Metadatenbeschreibung der Plug-Ins sollte sich nur noch auf statische Dinge beziehen die sich zur Laufzeit des Plug-Ins nicht mehr ändern, wie z.B. den Namen des Plug-Ins oder die IP-Adresse des Servers. Neue Ressourcentypen und Metriken sollen zur Laufzeit des Plug-Ins hinzugefügt werden können ohne den Überwachungsprozess unterbrechen zu müssen.

3.2.3 Datenmodel zur Abbildung des Drittsystems

Zur Repräsentation des Drittsystems muss ebenfalls ein Datenmodel implementiert werden, welches genügend Flexibilität besitzt, die sich ändernden Messdaten jederzeit abzubilden und auf Anpassungen zu reagieren. Im Gegensatz zum Datenmodel des Eigensystems, welches die Architektur und Messdatenrepräsentation des Eigensystems implementiert, bildet das Datenmodel die Struktur des zu beobachtenden Drittsystems ab. An der Schnittstelle zum Drittsystem, die als Teil des Plug-Ins implementiert wird, findet eine automatische Erfassung der Drittsystemdaten statt, dort liegen alle verfügbaren Messdaten vor. So besteht jederzeit die Möglichkeit, dynamische Erweiterungen der zu beobachtenden Ressourcentypen vorzunehmen und die entsprechenden Daten des Drittsystems zur Verfügung zu stellen. Handelt es sich beim Dritttsystem beispielsweise um einen Server mit 2 Netzwerkkarten, so ist es sinnvoll, ein Datenmodel im Quellcode zu implementieren, das den Server, die Netzwerkkarten und die Metriken der Netzwerkkarten darstellt. Es sollte gleichzeitig so flexibel sein, dass seine Verwendung auch noch im Falle des Hinzufügens einer dritten Netzwerkkarte oder eines zweiten Servers möglich ist. Die Gewährleistung der Flexibilität ist eine elementare Grundanforderung, sie nimmt maßgeblichen Einfluß auf die Skalierbarkeit des Gesamtsystems und die Möglichkeit späterer Umstellugen des Systemfunktionen auf die Verwendung dynamischer Metadaten. Ein flexibles Softwaremodel des Drittsystems bietet die Möglichkeit der Anpassung auf etwaige Erweiterungen und erleichtert die Überführung der Drittsystemdaten in das Eigensystem. Bei der Implementierung in einer Hochsprache sollte also darauf geachtet werden, entsprechend flexible Datenstrukturen zu verwenden. Als weiterer wichtiger Aspekt beim Design des Softwaremodels zur Repräsentation des Drittsystems muss außerdem berücksichtigt werden, dass die Datenspeicherung innerhalb des Models lediglich als Zwischenstufe vor der Überführung der Messdaten in das Datenmodel des Eigensystems dienen soll. Sie dient also als vorbereitende Tätigkeit

zur endgültigen Überführung ins Eigensystem und soll diesen Vorgang erleichtern. Die Speicherung der aus dem Drittsystem gemessenen Daten in einem geeigneten Softwaremodel geschieht im Plug-In, da dort die Schnittstelle zwischen Dritt- und Eigensystem implementiert wird. Hier erfolgt neben dem Empfang der Daten das Parsen des empfangenen Datenstroms und die Instanziierung der Objekte des implementierten Models. Im Plug-In findet auch die Überführung in das Datenmodel des Eigensystems statt(siehe auch Kapitel 3.2.4). Abbildung 6 zeigt den schematischen Ablauf der Überführung des Datenstroms des Drittsystems in ein geeignetes Datenmodel.

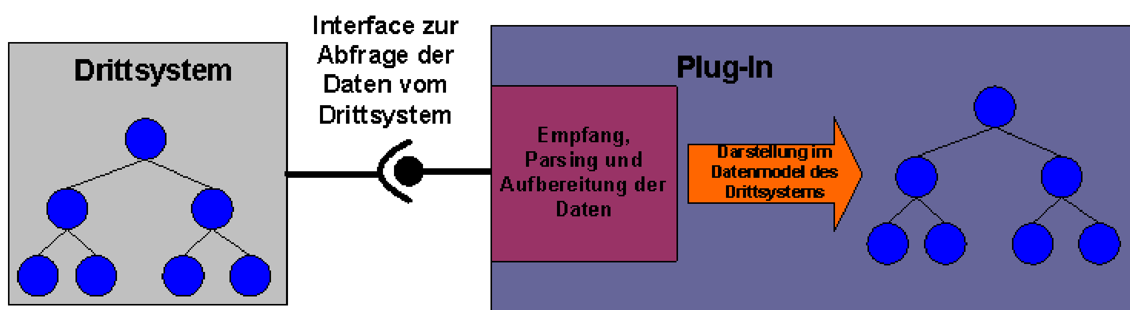


Abbildung 6: Datenmodel des Drittsystems

3.2.4 Mapping der Daten des Drittsystems auf das Datenmodel des Eigensystems

Die beschriebenen Unterschiede der Datenstrukturen des Drittsystems auf der einen Seite, und des Eigensystems auf der anderen Seite, bedingen die Notwendigkeit des Mappings der gewonnen externen Messdaten auf die interne Datenstruktur. Die vom Plug-In ermittelten Messdaten aus dem Fremdsystem werden zunächst in der Datenstruktur abgespeichert die das Fremdsystem abbildet, wie bereits in Abschnitt 3.2.3 beschrieben wurde. Diese Daten müssen anschließend in das im Eigensystem verwendete Datenmodel übertragen werden, um den Datentransport vom Plug-In über den Agent zum Server, die dortige Verarbeitung der Messdaten und das abschliessende Persistieren der Daten in der DB zu ermöglichen. Das Mapping der Daten vom Drittsystemmodell auf auf das Eigensystemmodell wird im Plug-In ausgeführt, da das Plug-In die Schnittstelle zwischen beiden Systemen darstellt. Von dort werden die Daten über den Agenten zum Server weitergeleitet. Daher spielen die Plug-Ins eine zentrale Rolle für die Funktionsweise eines solchen Systems, da dort neben der Datenerfassung auch noch das Parsen des Datenstroms in das Drittsystemmodell und die Überführung in das Datenmodel des Eigensystems stattfindet. Abbildung 7 zeigt den schematischen Ablauf der Kommunikation zwischen den Komponenten Server und Agent des Eigensystems, dem Plug-In als Schnittstelle zwischen den

Systemen und dem Drittsystem als überwachte Ressource.

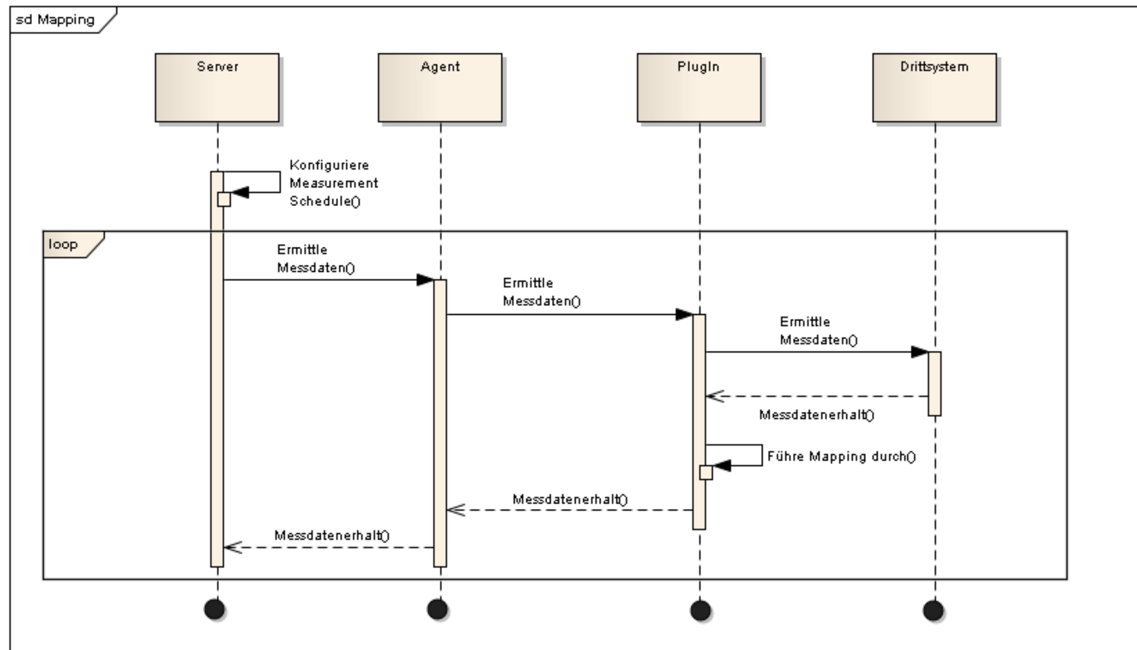


Abbildung 7: Schematischer Ablauf des Datenflußes

3.2.5 Persistierung der Systemdaten

Das Monitoring-System benötigt eine Datenbankanbindung zum Zweck der Speicherung verschiedener längerfristig benötigter Daten. Dies sind im Wesentlichen Konfigurations- und Managementeinstellungen, die von den Anwendern über die GUI eingestellt werden, und Messdaten, die von den verschiedenen Plug-Ins ermittelt und über die Agenten an den Server gesendet werden. Zum Speichern dieser Daten muss ein geeignetes Datenbankschema entwickelt werden, welches an das Softwaremodel des Eigensystems angelehnt ist und die Persistierung der Objekte des Models in der Datenbank realisiert. Bei der Entwicklung des Schemas müssen sämtliche möglichen Relationen, die zwischen den Architekturkomponenten Server, Agent und Plug-In und den Ressourcentypen auftreten können, bedacht werden, damit beim Speichervorgang keine Informationsverluste auftreten. Es muss zum Beispiel möglich sein, zu einer bestimmten Ressource den Ressourcentyp, das für diesen Ressourcentyp zuständige Plug-In und den Agenten, in dem das Plug-In läuft, zu ermitteln. Dies ist notwendig für die Synchronisation der lokalen Inventare der Agenten mit dem zentralen Serverinventar, da sonst nicht mehr ermittelt werden kann, von welchem Plug-In in welchem Agenten die Daten ermittelt wurden. Der Datenbankzugriff soll über eine geeignete Persistenzschicht realisiert werden und modular vom Rest des Quellcodes getrennt sein, um eine saubere Schichtenarchitektur zu gewährleisten. Die Anbindung an die Datenbank wird im Server realisiert, da dort

sämtliche Messdaten zusammenlaufen und die Benutzerschnittstelle zur Systemkonfiguration als Teil des Servers implementiert ist. Abbildung 8 zeigt schematisch den Systemaufbau und die elementaren Schnittstellen zwischen den Systemkomponenten.

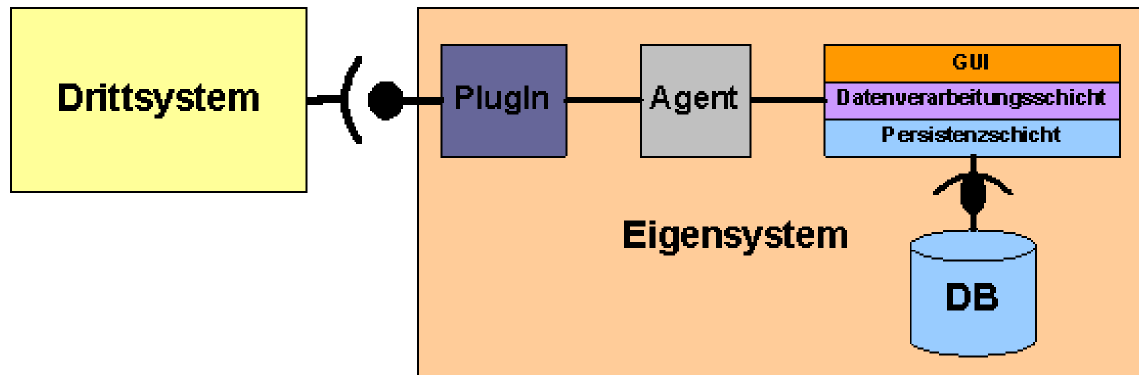


Abbildung 8: Schematischer Ablauf der Persistierung

3.3 Realisierung der Migration von statischer zu dynamischer Metadatenverwendung

In diesem Abschnitt werden die Anforderungen beschrieben, die speziell zur Realisierung der Migration zur Bereistellung dynamischer Metadaten erfüllt sein müssen. Im Gegensatz zu den im vorherigen Abschnitt beschriebenen allgemeinen Anforderungen, die sowohl für die Verwendung statischer als auch dynamischer Bereistellung von Metadaten erfüllt sein müssen, gelten diese Anforderungen nur für die Verwendung dynamischer Metadaten.

3.3.1 Schaffung einer Schnittstelle zum dynamischen Anlegen neuer Resourcetypen

Um neue Ressourcentypen und dazugehörige Metriken zur Laufzeit des Systems anlegen zu können, ist es notwendig, den Anwendern eine Schnittstelle zur Verfügung zu stellen, die die Eingabe entsprechender Daten ermöglicht. Das Interface soll neben der Möglichkeit zur Dateneingabe im Idealfall eine Übersicht aller momentan verfügbaren Ressourcentypen des Drittsystems bieten, um den Anwendern die Auswahl zu erleichtern. Dadurch wird der Auswahlvorgang beschleunigt und die Fehlerwahrscheinlichkeit bei der Eingabe neuer Ressourcentypen reduziert. Im Falle fehlerhafter Eingaben sollen dem Benutzer entsprechende Fehlermeldungen angezeigt werden, dies ist zum Beispiel notwendig, wenn der Benutzer einen Ressourcentyp überwachen möchte, der im Drittsystem nicht existiert. Wenn keine Eingabekontrolle existiert, muss der Anwender genaue Kenntnisse des Drittsystems besitzen, um den Fall der Überwachung eines nicht vorhanden Ressourcentyps zu vermeiden. Die Schnittstelle kann sowohl als Teil des Servers als auch als Teil des Agenten implementiert werden, sinnvoller ist allerdings die Integration in den Server, da dort die Anbindung an die Datenbank realisiert ist und die neu eingegeben Daten direkt persistiert werden können. Da der Server die zentrale Komponente des Systems darstellt, verfügen alle Agenten nach einer Synchronisation mit dem Server in ihren lokalen Inventaren über den gleichen Datenbestand, der auch in der Datenbank vorliegt (siehe auch Abschnitt 3.3.2). Wird die Schnittstelle als Teil des Agenten realisiert, muss zunächst eine Übertragung der Daten an den Server stattfinden, nach der erfolgreichen Persistierung der Daten muss dann eine erneute Synchronisation erfolgen, um die lokalen Agenteninventare in einen konsistenten Zunstand mit der Datenbank des Servers zu bringen. Die Konsistenz muss für die korrekte Funktionsweise des Systems unbedingt gewährleistet sein.

3.3.2 Dynamische Synchronisation der Datenhaltung zwischen Server, Agent und Plug-In

Ein weiterer wichtiger Schritt zur Realisierung der Migration ist die Implementierung der Synchronisation zwischen Plug-In, Agent und Server im Falle des Hinzufügens neuer Ressourcentypen während des Systembetriebs. Bei der statischen Funktionsweise wird beim Systemstart oder bei Inbetriebnahme eines Plug-Ins der zugehörige Plug-In-Deskriptor vom Server geparsed und alle darin beschriebenen Ressourcentypen und die dazugehörigen Metriken in der Datenbank angelegt. Die Ressourcentypen die in der Datenbank existieren werden in das zentrale Inventory des Servers übernommen, eine Synchronisation des Servers mit den einzelnen Agenten erfolgt erst beim Neustart eines Agenten oder beim Durchführen eines Plug-In-Updates, beides muss von den Anwendern des Systems initiiert werden. Um die dynamische Erweiterung des Systems um neue Ressourcentypen zu ermöglichen, muss die Synchronisation nach Veränderungen automatisch initiiert werden. Dazu müssen entsprechend der Eingabe des Benutzers zunächst der neue Ressourcentyp, die dazugehörigen Metriken und das entsprechende Plug-In, das die Überwachung durchführen soll, in die Datenbanktabellen geschrieben werden und in Relation zueinander gesetzt werden, um die Daten später wieder zuordnen zu können. Der Name des neuen Ressourcentyps, seiner Metriken und des für die Überwachung des Ressourcentyps zuständigen Plug-Ins müssen an den Agent übertragen und dort in das lokale Inventar geschrieben werden. Um dies zu realisieren, muss der Ressourcentyp als Kind des Vater-Objektes 'Plug-In' in den Ressourcenbaum gehangen werden, dazu muss ein entsprechender Methodeaufruf erfolgen. Der Methodenaufruf geschieht beim Verwenden statischer Metadaten beim Parsen des Deskriptors, ohne das Einhängen des neuen Typs in den Ressourcenbaum bleibt der neue Ressourcentyp dem Agenten unbekannt und kann nicht überwacht werden. Anschließend kann das Plug-In mit der Überwachung des Ressourcentyps beginnen, hierzu müssen die Methoden des Plug-Ins zum Suchen bestimmter Ressourcen mit den Informationen des neuen Typs gefüttert werden. Die Messdaten zu den Ressourcen des neuen Typs liegen an der als Teil des Plug-Ins realisierten Schnittstelle zum Drittsystem vor und brauchen über die entsprechenden Methoden nur noch abgefragt zu werden. Der Transport der Daten über den Agenten zum Server und die dortige Aufbereitung und Persistierung der Messdaten erfolgt nach dem gleichen Prinzip wie bei der Verwendung statischer Metadaten, eine Anpassung der Verfahrensweise ist hierfür nicht notwendig.

3.4 Anwendungsfälle(Use Cases)

In diesem Abschnitt werden die wichtigsten Anwendungsfälle des Systems und die Abläufe bei ihrer Durchführung beschrieben. Es wird dabei unterschieden zwischen dem Ablauf bei der Verwendung statischer Metadaten und dem Ablauf nach der erfolgreichen Migration zur Bereitstellung von dynamischen Metadaten. Zunächst sollen nachfolgend die Use Cases genannt und knapp erläutert werden, bevor auf die spezifischen Anforderungen eingegangen wird. Die beiden wichtigsten Anwendungsfälle sind:

1. **Hinzufügen neuer Ressourcentypen:** Dieser Anwendungsfall betrifft die Erweiterung des Systems um einen neuen zu beobachtenden Ressourcentyp.

2. **Entfernen nicht mehr benötigter Ressourcetypen:** Dieser Anwendungsfall betrifft die Herausnahme eines bestimmten Ressourcentyps aus der Überwachung durch das System, da die Beobachtung nicht mehr von Interesse ist.

Beide Anwendungsfälle kommen in der Praxis sehr häufig vor, ihre Durchführung unterscheidet sich je nach Verwendung statischer oder dynamischer Metadaten und der jeweiligen Systemfunktionsweise jedoch stark voneinander.

3.4.1 Hinzufügen/Entfernen von Ressourcentypen bei statischer Bereitstellung der Metadaten

Beim Hinzufügen neuer Ressourcentypen unter Verwendung der statischen Bereitstellung von Metadaten muss zunächst der Plug-In-Deskriptor um den entprechenden Ressourcentyp erweitert werden. Dazu muss an entsprechender Stelle im Plug-In-Deskriptor ein neues Feld in der verwendeten Metasprache definiert werden. Die Stelle, an der der Deskriptor erweitert wird, ist abhängig davon, welcher Ressourcenkategorie der neue Ressourcentyp angehört und ob er Parent- oder Child-Type ist. Für den neuen Ressourcentyp müssen anschließend die entsprechenden Metriken definiert werden, daher muss man vor der Erweiterung des Deskriptors sorgfältig planen, welche Metriken für den neuen Ressourcentyp von Interesse sind. Nach der Anpassung des Plug-In-Deskriptors müssen zusätzlich Anpassugen im Java Quellcode des Plug-Ins vorgenommen werden, da der Code direkten Bezug auf den Plug-In-Deskriptor nimmt. Nachdem die Anpassungen sowohl im Deskriptor als auch im Quellcode durchgeführt wurden, müssen die Bestandteile des Plug-Ins in eine Archivdatei gepackt und in den laufenden Systemserver deployt werden. Danach werden die Metadaten des erweiterten Plug-In-Deskriptors vom Server geparsed und auf Korrekheit geprüft. Nach erfolgreicher Prüfung wird das neue Plug-In auf dem Server zum Download bereit gestellt und kann von den Agenten geladen weden. Damit die einzelnen Agenten das erweiterte Plug-In vom Server laden können, muss für jeden einzelnen Agenten entweder ein Neustart oder ein Plug-In-Update durchgeführt werden. In beiden Fällen wird die Kommunikation zum Server und somit auch die Übertragung der Messdaten an den Server unterbrochen. In diesem Zeitraum können wichtige Informationen verloren gehen oder verspätet eintreffen, in beiden Fällen können daraus bei sensiblen oder zeitkritischen Daten erhebliche Probleme entstehen. Nachdem ein Agent das erweiterte Plug-In herunter geladen hat, beginnt er mit dem Parsing des Plug-In-Deskriptors und generiert aus der im Plug-In-Deskriptor stehenden Metabeschreibung entsprechende Objekte, die die im Deskriptor beschriebenen ResourceTypes, ihre Metriken und ihre Parent-Child-Beziehungen abbilden. Anschließend wird das lokale Inventar des Agenten um die neuen ResourceTypes erweitert, indem diese als weitere Kinder des Parent ResourceTypes in den Ressourcenbaum eingehangen werden. Die Informationen über das lokale Inventar werden vom Agenten an den Server übertragen, dort werden die übermittelten Daten in der zentralen Systemdatenbank persitiert und das zentrale Inventar des Servers mit der Datenbank abgeglichen. Dies ist notwendig, um Konsistenz in der Datenhaltung des Systems und die korrekte Systemfunktionalität zu gewährleisten. Anschließend sind die Informationen über den neuen Typ im System verteilt, und das Plug-In kann mit der Überwachung der Ressourcen dieses Typs beginnen. Beim Entfernen

von Ressourcentypen aus der Beobachtung durch das System müssen ebenfalls die entsprechenden Deskriptoren und Java Klassen angepasst und eine neue Java Archivdatei gebaut werden. Auch hier ist ein anschließendes Deployment notwendig und ein Neustart oder Update der Agenten erforderlich, was in beiden Fällen eine Unterbrechung der Systemfunktionalität nach sich zieht. Diese Unterbrechung der Systemüberwachung ist der große Nachteil bei der Verwendung von statischer Metadatenbereitstellung. Abbildung 9 veranschaulicht die beschriebenen Abläufe und die partizipierenden Akteure in Form eines UseCase-Diagramms.

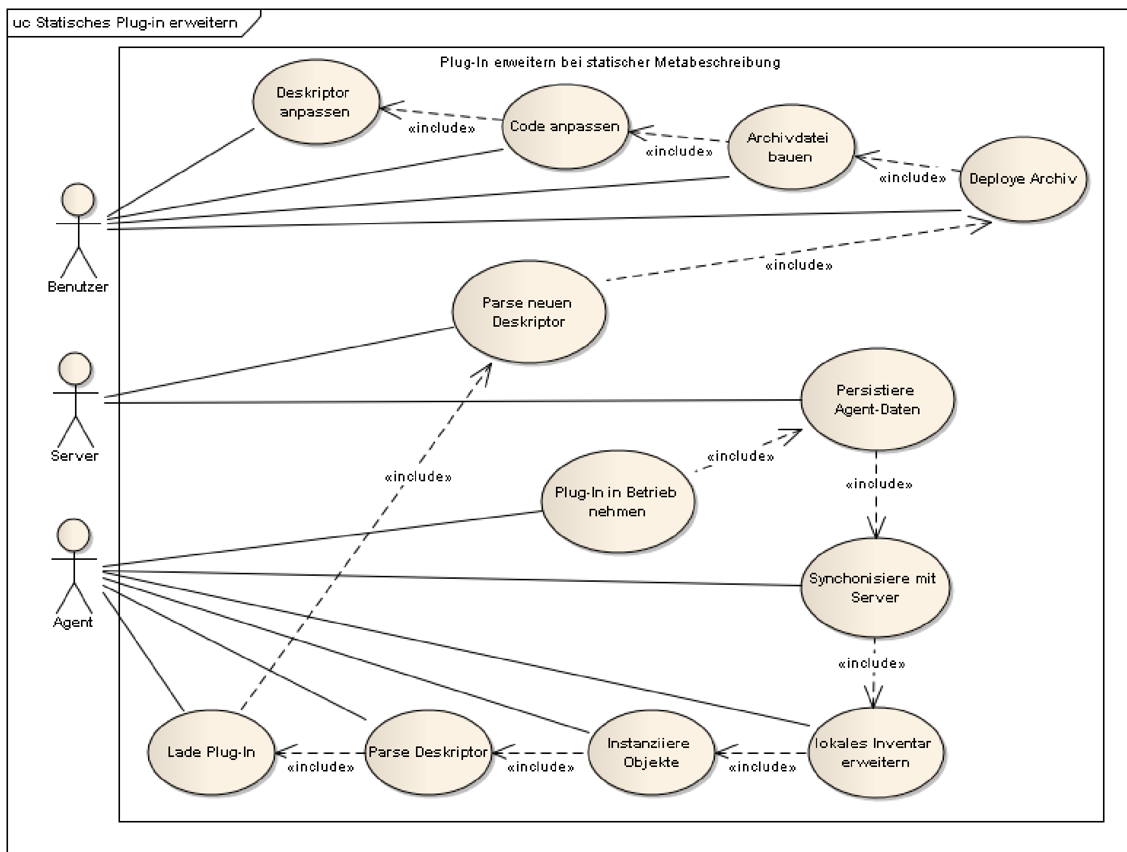


Abbildung 9: Erweitern eines statischen Plug-Ins

3.4.2 Hinzufügen/Entfernen von Ressourcetypen bei dynamischer Bereitstellung der Metadaten

Die Migration zur dynamischen Bereitstellung von Metadaten hat das Ziel, die Systemfunktionalität zu verbessern und die entscheidenden Nachteile der statischen Metadatenbereitstellung zu beheben: hoher Anpassungsaufwand im Plug-In-Deskriptor und im Java Quellcode, Notwendigkeit des Re-Builds von Java Archivdateien, Re-Deployment nach jedem Re-Build und Unterbrechung der Systemüberwachungsfunktionalität beim Download der erweiterten Plug-Ins durch die Agenten. Es muss daher eine Lösung gefunden werden, die an den entscheidenden Schwachstellen der statischen Metadatenbereitstellung ansetzt und diese behebt. Zunächst gilt es, den Anpassungsaufwand im Plug-In-Deskriptor bei jeder Erweiterung oder Anpassung zu beheben. Ein Grundgerüst an Ressourcetypen und Metriken kann weiterhin statisch aus der Beschreibung im Plug-In-Deskriptor angelegt werden, wenn bekannt ist, dass diese sich im weiteren Verlauf nicht mehr ändern, und daher keine Anpassungen notwendig sind. Da in diesem Fall das Parsing des Deskriptors, und die damit verbundenen Abläufe, nur einmal beim Start des Agenten geschieht, ist keine Beeinträchtigung der Überwachungsfunktionalität gegeben. Sollen weitere Ressourcentypen hinzugefügt werden, so soll dies nicht mehr über den Deskriptor geschehen, sondern über eine geeignete Benutzerschnittstelle, mit der der neue Ressourcentyp, und die dazugehörigen Metriken, definiert werden können. Nach der Dateneingabe erfolgt eine Überprüfung, ob es bereits einen gleichnamigen Ressourcentyp gibt, um die Eingabe des Benutzers zu überprüfen und sinnlose Systemaktionen zu verhindern. Nach der Prüfung müssen die Objekte für den neuen Ressourcentyp und die dazugehörigen Metriken erzeugt und mit den eingegeben Metadaten initialisiert werden. Dies geschieht nicht mehr, wie es bei der statischen Vorgehensweise üblich war, beim Parsen der Metadadaten des Plug-In-Deskriptors, sondern direkt aus der Eingabe des Benutzers. Das neu angelegte Objekt vom Typ 'ResourceType' wird dann in den Ressourcenbaum des Systems, als weiteres Kind des Plug-Ins, eingehängt, und gelangt so in das lokale Inventar des Agenten. Danach erfolgt die Synchronisation mit dem Server, dort erfolgt die Persistierung der neuen Objekte in der Datenbank und die Erweiterung des zentralen Inventars. Anschließend kann das Plug-In die erweiterte Funktionalität nutzen, entsprechend der Messbeschreibungen werden die Monitoring-Daten der neu hinzugefügten Ressourcentypen und deren Metriken ermittelt und, über den Agenten, an den Server übertragen, so wie es auch für die im Plug-In Deskriptor definierten Resourcentypen geschieht. Dort erfolgt dann die Persistierung der Messdaten in der Datenbank und die Visualisierung über die GUI. Alle zur Erweitwerung der Plug-In-Funktionalität notwendigen Schritte werden während des Systembetriebs durchgeführt, es ist kein Neustart und somit auch keine

Unterbrechnung der Systemüberwachung notwendig. Der Anpassungsaufwand des Deskriptors, des Quellcodes und das Re-Build und Re-Deployment der Java Archive entfällt. Das System kann kontinuierlich ohne Unterbrechung laufen und sammelt parallel zu seiner Erweiterung Informationen, so dass keine Informationslücken auftreten. Abbildung 10 zeigt eine Übersicht über die bei der dynamischen Erweiterung des Plug-Ins notwendigen Schritte.

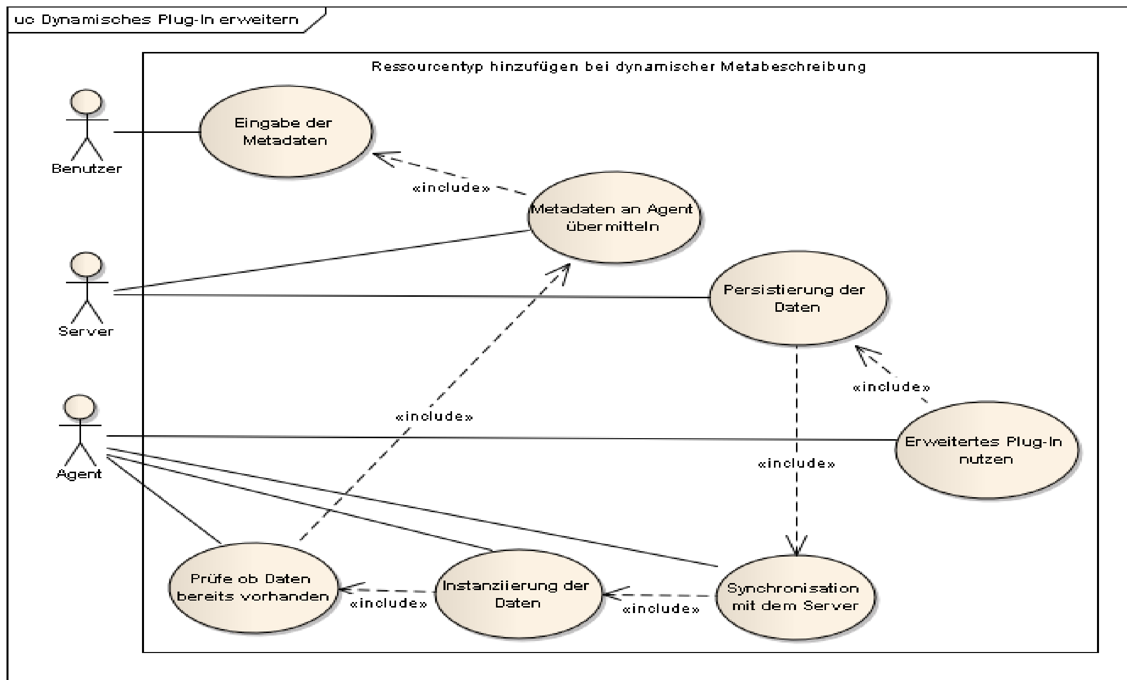


Abbildung 10: Erweitern eines dynamischen Plug-Ins

4 Umsetzung

In diesem Kapitel wird der konkrete Lösungsweg zur Realisierung der Projektziele beschrieben. Zunächst wird in Abschnitt 4.1 die Planung der einzelnen Projektschritte beschrieben. In Abschnitt 4.2 werden die bei der Lösungsrealisierung verwendeten externen Tools beschrieben. Abschnitt 4.3 beschreibt die einzelnen Teilabschnitte bei der Realisierung der Nagios Anbindung, und Abschnitt 4.4 abschließend die Teilabschnitte bei der Realsierung der Migration von der Verwendung der Bereitstellung von statischen hin zu dynamischen Metadaten.

4.1 Projektplanung

Nach der Definition der genauen Aufgabenstellung mit allen Teilaufgaben wurde zunächst ein Fahrplan erstellt, den es bei der Realsierung der Lösung abzuarbeiten galt. Da das Projekt per Definition aus mehreren Teilprojekten bestand, wurden die einzelnen Teilprojekte als Projektphasen definiert. Daraus ergaben sich in der Planung zunächst die beiden Hauptphasen des Projektes, für die jeweils 3 Monate vorgesehen wurden. Der schriftliche und theoretische Teil wurde in der Zeitplanung der Implementierungsphase mit einbezogen, da die Dokumetation und Abstraktion der durchgeführten Aufgaben immer zeitnah erfolgen sollte. Das Projekt wurde in folgende Phasen unterteilt:

4.1.1 Entwicklung eines RHQ Plug-Ins für die Nagios Integration unter Verwendung statischer Metadaten

In dieser Phase des Projektes sollte ein einfaches RHQ Plug-In zur Anbindung des Nagios Systems unter Verwendung der statischen Metadatenbeschreibung realisiert werden. Zur besseren Kontrolle des Projektfortschritts und eines einfacher zu schätzenden Zeitbedarfs wurde diese Projektphase in weitere Teilphasen unterteilt:

- **Evaluation vorhandener Schnittstellen zur Realisierung der Nagios Anbindung:** In dieser Phase des Projektes sollten zunächst die vorhandenen Schnittstellen zu Nagios auf ihre Tauglichkeit überprüft werden, die wichtigsten Kriterien hierbei waren Verfügbarkeit des Quellcodes, möglichst hoher Grad an Standardisierung, Aktualität der Schnittstelle und Lebendigkeit des Projektes. Mithilfe dieser Kriterien sollten die vorhandene Schnittstellen oder Schnittstellen-APIs überprüft werden, und eine Entscheidung getroffen werden, ob eine vorhandene Schnittstelle den Anforderungen entspricht ob eine komplette Neuentwicklung einer Schnittstelle zum Nagios System erfoderlich sein würde. Der Zeitbedarf hierfür wurde auf 2 Wochen geschätzt.

- **Entwicklung eines Softwaremodels zur Abbildung des zu überwachenden Nagios Systems:** Nach der Schnittstellenevaluation sollte, angelehnt an die Architektur des Nagios Systems, ein Softwaremodel entwickelt werden, mit dem die über die Schnittstelle empfangenen Nagios Systemdaten innerhalb des Plug-Ins abgebildet werden konnten. Das Model sollte möglichst modular sein und den Anforderungen an moderne objektorientierte Software genügen. Als Zeitrahmen hierfür wurden 4 Wochen geschätzt.

- **Implementierung der Plug-In-Komponenten zur Realisierung des Nagios Plug-Ins:** Im dritten Teilabschnitt der Phase 1 sollte ein erstes RHQ Plug-In entwickelt werden, welches die Anbindung an Nagios unter Verwendung der Nagios Schnittstelle und des Nagios Softwaremodels realisiert. Das Plug-In sollte auf der Basis des AdvancedManagementPluginSsystem (AMPS) realisiert werden und in der ersten Version die statische Bereitstellung von Metadaten über den XML-Deskriptor nutzen. Für diese Phase des Projektes wurden 6 Wochen eingeplant.

4.1.2 Entwicklung eines Prototypen für die Migration zur Bereitstellung dynamischer Metadaten

In dieser Phase sollte das entwickelte Plug-In und die relevanten Systemkomponenten so angepasst werden, dass die Migration zur Verwendung dynamischer Metadaten durchgeführt werden konnte. Auch diese Projektphase wurde aus Gründen der besseren Planbarkeit und Übersichtlichkeit in mehrere Teilabschnitte aufgeteilt:

- **Dynamische serverseitige Generierung von Metadaten:** In diesem ersten Abschnitt der zweiten Projektphase sollten zunächst die serverseitigen Anforderungen der Migration zur Verwendung dynamischer Metadaten umgesetzt werden. Es sollte eine Benutzeroberfläche geschaffen werden, die das Anlegen neuer Ressourcentypen und dazugehöriger Metriken erlaubt, nach der Eingabe sollte eine korrekte Persistierung der neuen Daten in der Systemdatenbank durchgeführt werden. Dazu sollten entweder die bereits vorhandene Methoden des Serverquellcodes verwendet oder die Module entsprechend erweitert werden, falls notwendig. Für die Phase wurden 4 Wochen vorgesehen.

- **Dynamische agentenseitge Generierung von Metadaten:** Im zweiten Abschnitt der zweiten Projektphase sollten, aufbauend auf die Erkenntnisse des ersten Abschnitts, die agentenseitigen Anforderungen der Migration zur Verwendung dynamischer Metadaten umgesetzt werden. Es sollte eine Konsolenschnittstelle geschaffen werden, mit deren Hilfe neue Ressourcentypen und dazugehöriger Metriken angelegt werden konnten. Der Quellcode des Agenten

sollte dabei so erweitert/angepasst werden, dass nach dem Anlegen der neuen Typen eine Weiterleitung der Informationen an den Server dortige Persitierung geschehen konnte, wie auch bei der serverseitigen Ressourcetypengenerierung. Nach erfolgreicher Persitierung sollte eine Synchronisation mit dem lokalen Inventar der Agenten durchgeführt werden, so dass keine Inkonsistenzen zwischen der Datenbank des Servers und den einzelnen Agenten auftreten konnten.

- **Dynamische Generierung von Metadaten durch ein Plug-In:** Im dritten Abschnitt sollte, aufbauend auf die beiden vorhergehenden Abschnitte, der letzte Schritt zur Migration gegangen werden. Ein vom Plug-In gelieferter neuer Ressourcentyp und die zugehörigen Metriken sollten zunächst an den Agenten übermittelt werden. Dort sollte nach einer Prüfung, ob dieser Ressourcentyp bereits existiert, eine Weiterleitung der Informationen an den Server realisiert werden.

4.2 Technische Rahmenbedingungen bei der Umsetzung

In diesem Abschnitt werden die technischen Rahmenbedingungen beschrieben, unter denen der programmiertechnische Teil der Arbeit realisiert wurde. Inbesondere werden die Komponenten beschrieben, die das technische Umfeld der Arbeit bildeten, wie z.B. Betriebsystem, Datenbanken, IDE und verwendete Build-Tools.

4.2.1 Betriebssystemumgebung

Bei der Wahl des Betriebssystems entschied man sich für die Verwendung einer Linux Distribution (OpenSuse), da Nagios nur unter Linux Distributionen läuft, und man verschiedene Unix Dienste, wie z.B. ssh und xinetd, für die Lösungsrealisierung benötigte. Da der Einfachheit halber während der Implementierungsphase kein verteiltes, sondern lediglich ein lokales System verwendet wurde, liefen der Nagios Server und alle von ihm zu überwachenden Ressourcen auf dem gleichen Host. Das RHQ System wurde ebenfalls unter OpenSuse betrieben, da es hauptsächlich unter Linux entwickelt wird, und in solch einer Umgebung am Stabilsten läuft, obwohl es als Java Anwendung auch unter anderen Betriebssystemen betrieben werden kann.

4.2.2 Datenbanksystem

Um das RHQ System zu betreiben, bedarf es der Installation eines Datenbanksytems zur Persitierung der Systemdaten. Bei der Wahl des Datenbanksystems standen zwei Alternativen zur Auswahl, mit denen das RHQ System betrieben werden konnte: PostgresQL und H2. H2 ist ein in RHQ integriertes Datenbanksystem und dient lediglich zu Entwicklungs- und Testzwecken, da es über keine sehr hohe Leistungsfähigkeit verfügt. Daher entschied man sich für die Verwendung von PostgresQL, da dieses System kostenlos verwendet werden kann, einfach zu administrieren ist, wenn entsprechende Grundkenntnissen von Datenbanken vorhanden sind, und über die nötige Leistungsfähigkeit verfügt, um es auch für den Einsatz im Produktivsystem zu nutzen.

4.2.3 Entwicklungsumgebung

Als Entwicklungsumgebung wurde Eclipse verwendet. Eclipse bietet dem Entwickler zahlreiche Plug-Ins, die z.B. eine einfache Anbindung an Quellcoderepositories oder verschiedene Build-Tools zum automatisierten Durchführen des Build-Prozesses bieten. Die Verwendung von Eclipse war obligatorisch da, in der Abteilung 'Technische Informatik' der Dillinger Hütte, speziell unter Java, ausschließlich Eclipse als Entwicklungsumgebungverwendet wird.

4.2.4 Versionskontrolle

Zur Versionskontrolle während der Implementierungsphase der Arbeit wurden die Versionierungstools Subversion und Git verwendet. Das RHQ Projekt steht unter Git Verwaltung und kann unter git.fedorahosted.org über verschiedene Protokolle wie ssh oder http heruntergeladen werden. Die IT Abteilung der Dillinger Hütte benutzt Subversion zur Versionsverwaltung, alle im Rahmen der Arbeit entwickelten Module wurden neben der OpenSource Versionierung über Git noch unter Subversionskontrolle gestellt, da an das SVN Repository verschiedene andere Tools wie SonarJ angehangen sind, die z.B. die Qualität der Software messen

4.2.5 Build-Tools

Als Tool zur automatischen Durchführung des Build-Prozesses wurden Maven von Apache und der Hudson Server verwendet. Maven ist ein kostenloses Tool, das zahlreiche Aufgaben innerhalb des Build-Prozesses durchführen kann, wie z.B. Generierung ausführbarer Bibliotheken und Generierung und Download benötigter Referenzpakete. Darüber hinaus gibt es für Maven zahlreiche Plug-Ins, die diese Kernfunktionalitäten um Aspekte der Versionskontrolle oder der Qualitätskontrolle erweitern. Es exisiert ein Plug-In für Eclipse, das es dem Anwender möglich macht, direkt aus Eclipse den Maven Build Prozess anzustoßen. Der Hudson Server stößt auf Basis des unter Subversionsverwaltung stehenden Codes einen allnächtlichen Build-Prozess an, die daraus genenierten Artefakte werden entsprechend ihrer Version auf dem Nexus abgelegt. Der Nexus ist ein an Maven angebundener Server, auf dem sämtliche vom Hudson gebauten Bibiliotheken abgelegt werden, und somit auch von anderen Projekten referenziert werden können. Darüber hinaus laufen auf dem Hudson Server noch Tools zur Qualitätssicherung des Quellcodes, die diesen auf Redundanzen und andere Schwachstellen hin untersuchen und bei schwerwiegenden Fehler den Build-Prozess verhindern.

4.2.6 Advanced Management Plugin System (AMPS)

AMPS ist die RHQ Plug-In-API, auf deren Verwendung die Implementierung der RHQ Plug-Ins basiert. Sie bietet dem Plug-In-Entwickler alle zur Implemenierung notwendigen Komponenten und wurde bereits in Abschnitt 1.1 ausführlich beschrieben.

4.3 Umsetzung der Nagios Anbindung

4.3.1 Validierung der vorhandenen Schnittstellen zu Nagios

Vor dem Start der Implementierungsphase für das Plug-In zur statischen Anbindung des RHQ Systems an das Nagios System sollte zunächst überprüft werden, ob bereits Schnittstellen zum Abfragen der Nagios Systeminformaionen existierten, deren Verwendung eine Alternative gegenüber der Entwicklung einer eigenen Schnittstelle darstellte. Wenn dies der Fall war, sollten die existierenden Schnittstellen anhand grundlegender Bewertungskriterien evaluiert und verglichen werden, ansonsten sollte die Eigenimplementierung einer geeigneten Datenabfrage-Schnittstelle zum Nagios System realisiert werden. Bei der Überprüfung der Schnittstellen sollten folgende Bewertungskriterien zur Beurteilung der Eignung herangezogen werden:

- **Performanz**: Die Schnittstelle sollte eine möglichst schnelle Datenabfrage gewährleisten und möglichst wenig Systemressourcen in Anspruch nehmen.
- **Verwendung von verbreiteten Standard-Technologien**: Es sollte angestrebt werden, dass die existierende Schnittstelle auf einer Standard-Technologie basiert, die weit verbreitet und akzeptiert ist. Ein Vorteil dabei ist, dass für verbreitete Standard-Technologien in allen gängigen Programmiersprachen entsprechende APIs zum Verarbeiten der Schnittstellendaten exisitieren, die die Nutzung der Schnittstelle stark vereinfachen. Ein weiterer Vorteil ist die Unabhängigkeit gegenüber Wartung oder Support von Dritten, die im Falle der Verwendung proprietärer Systeme von Fremdanbietern nicht gegeben ist.
- **Aktualität des Schnittstelle**: Die Schnittstelle sollte möglichst aktuell sein, im Falle der Realisierung der Schnittstelle in Form eines OpenSource-Projektes sollte darauf geachtet werden, dass das Projekt noch lebendig ist und dass der Quellcode noch verbessert bzw. gewartet wird. Ein lebendiges Projekt bietet den Vorteil der Verfügbarkeit der Schnittstelle auch in Zukunft, dadurch werden etwaige Anpassungen der Schnittstellenanbindung innerhalb des eigenen Quellcodes vermieden.
- **Verfügbarkeit des Quellcodes**: Bei Verwendung von Schnittstellen von Drittanbietern muss der Quellcode und die Beschreibung der Standards frei zugänglich sein. Dadurch wird das Verständnis des Systems im Falle notwendiger Wartungen oder Anpassungen auf veränderte Anforderungen erheblich erleichtert.
- **Keine Lizenzgebühren**: Als weiteres Kriterium sollten Lizengebühren durch die Verwendung von Software oder Technologien vermieden werden.

Bei der Suche nach existierenden Technololgien und Schnittstellen zur Abfrage der Nagios Systemdaten stieß ich auf die Nagios EventBrokerAPI. Die Nagios Event Broker (NEB) bietet die Möglichkeit, interne Nagios Systemevents zur Behandlung durch externe Routinen verfügbar zu machen. NEB ist Modul-basiert, die NEB Module werden beim Systemstart in den Nagios Kern eingehangen. NEB verwendet Callback-Routinen, die von den NEB Modulen implementiert werden müssen. Die Routinen werden beim Auftreten der verschiendenen Systemevents innerhalb des Nagios Servers ausgeführt und ermitteln die Systemdaten aus den Events. Das Nagios Monitoring System generiert zahlreiche Events, die Events sind die Ergebnisse der Überwachung von Anwendungen, Datebbanken, Geräten, Diensten und Hosts. Außerdem werden von Nagios Performancedaten und Eventmeldugen generiert. Die durch das Einhängen der NEB Module in den Nagios Kern registrierten Routinen können alle vom Nagios Prozess generierten Events abfangen, wenn die entprechenden Routinen bzw. Module existieren. Die Nagios Event Broker API ist in der Programmiersprache C implementiert, sie bietet zahlreiche Funktionen, die die Events, die innerhalb eines Nagios Sytems auftreten können, auffangen und weiterreichen können. Durch die Verwendung der Programmiersprache C für die Implementierung der EventBrokerAPI ist die Datengewinnung vom Nagios System performant und ressourcenschonend. Es ist nicht notwendig, die Eventdaten durch die Implementierung einer eigenen Schnittstelle zur EventBrokerAPI abzufragen, da bereits Systeme exisitieren, die dies erledigen und komfortablere Schnittstellen zur Datenabfrage bieten. Für die Verwendung innerhalb des Projektes kamen letztlich die auf der NEB aufsetzenden Systeme 'NDO Utils' und 'MK Livestatus' in Betracht, die im Folgenden kurz beschrieben werden.[7]:

1. **NDO_Utils**: NagiosDataOut(NDO) ist eine vom Nagios Core Team entwickelte, auf der Verwendung der Nagios Event Broker API aufsetzende, Datenbank-Abstraktionsschicht. NDO ist ist in der Lage, alle Events und Konfigurationsdaten, die durch die Routinen der NEB aufgefangen und ausgewertet werden, in einer Datenbank zu speichern. NDO Utils setzt sich aus den beiden Kompontenten 'ndomod' und 'ndo2db' zusammen. NDOMOD ist ein NagiosEventBroker Modul, das in den Nagios Prozess geladen wird. Es leitet die Informationen an Dienste weiter, die TCP or UNIX Sockets verwenden. Die zweite Komponente ist NDO2DB, sie stellt einen solchen Dienst dar. NDO2DB empfängt die Informationen von der NDOMOD Komponente und schreibt sie in eine relationale Datenbank, wobei momentan nur die Verwendung von MySQL unterstützt wird. Die persistierten Nagios Monitoringdaten können über eine geeignete Datenbankschnittstelle abgefragt werden. Die in der Datenbank

[7] NEB

anfallende Datenmenge wird allerdings mit der Zeit sehr groß, darüber hinaus erzeugen die regelmäßigen Datenbankabfragen relativ große Systemlast. Die Datenbankabfragen können die Nagios Prozesse blockieren, dies ist einer der wesentlichen Nachteile bei der Verwendung von NDO Utils. Außerdem muss neben der MySQL Datenbank zur Verwendung mit NDO Utils ein weiteres Datenbanksystem zur Speicherung der Daten des RHQ Systems installiert werden, da RHQ nur die Verwendung von Oracle, Postgres oder eingebetteten H2 Datenbanken unterstützt. Abbildung 11 verdeutlicht den Aufbau und die Funktionsweise von NDO_Utils. [8] [9]

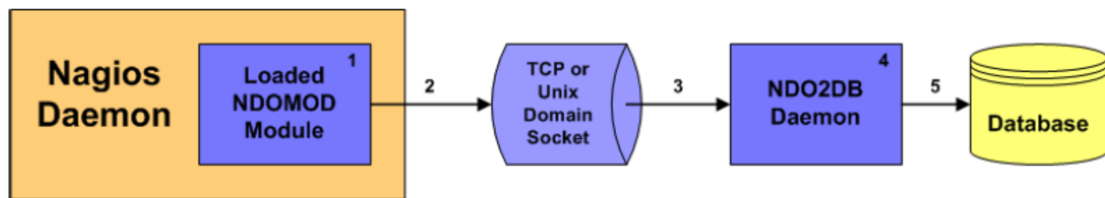


Abbildung 11: Funktionsweise von NDO Utils

2. **MK Livestatus**:

   MK Livestatus ist ein NagiosEventBroker Modul und setzt wie die NDO Datenbank auf der Event Broker API von Nagios auf, anders als bei NDO Utils werden aber keine Daten aktiv in eine Datenbank kopiert. Die Daten werden über einen UNIX-Socket nach außen bereitgestellt, allerdings nur dann, wenn sie aktiv angefragt werden. Der große Vorteil des Livestatus Moduls liegt darin, dass es direkten Zugriff auf Monitoring-Daten im Nagios Kern liefert, ohne irgendwelche Daten in einer Datenbank oder Datei zu speichern. Dadurch, dass lediglich CPU Zeit benötigt wird, um die gewünschten Statusinformationen abzufragen, ist Livestatus sehr performant. Livestatus kann sämtliche Statusdaten direkt aus dem Speicher des Nagios Prozesses abfragen und benötigt keine weiteren Speicher- oder Filesystemzugriffe. Der Focus von Livestatus liegt darin, die aktuelle auftretenden Status Events nach außen zu liefern, es werden keine historischen Informationen gesammelt. Da Livestatus keine Informationen in externen Speicher schreiben muss, wird der Nagios Prozess nicht geblockt, darin liegt der Hauptvorteil gegenüber der Verwendung der NDO Utils. Es exisitiert eine eigene, leicht verständliche Abfragesprache zur Abfrage der Statusinformationen über einen Unix Socket(Livetstatus Query Language, LQL), durch entsprechende LQL Kommandos können über die Schnittstelle sämtliche verfügbaren Statusinformationen abgefragt werden. Das Format der

[8]NDO1
[9]NDO2

Livetstatus Antworten ist standardmäßig CSV, durch Anpassugen der Request-Queries können aber auch problemlos andere Delimiter als Kommas verwendet werden. Der Unix Socket kann durch die Verwendung von Diensten wie XINETD, NETCAT oder SSH als Remote Schnittstelle verwendet werden und über TCP angesprochen werden. Der Quellcode ist frei verfügbar und unter Gnu Public License(GPL) Verwaltung, er kann zu eigenen Zwecken beliebig angepasst werden. Durch seine Verwendung entstehen keine Lizenzgebühren. Das Livestatus Projekt ist ein aktives OpenSource Projekt, das ständig weiter entwickelt wird. Das Modul erfüllt sämtliche definierten Anforderungen an eine Schnittstelle und wird daher als Schnittstelle zum Nagios System verwendet. Abbildung 12 zeigt die Funktionsweise des Systems unter Verwendung von MK Livetstatus.[10] [11]

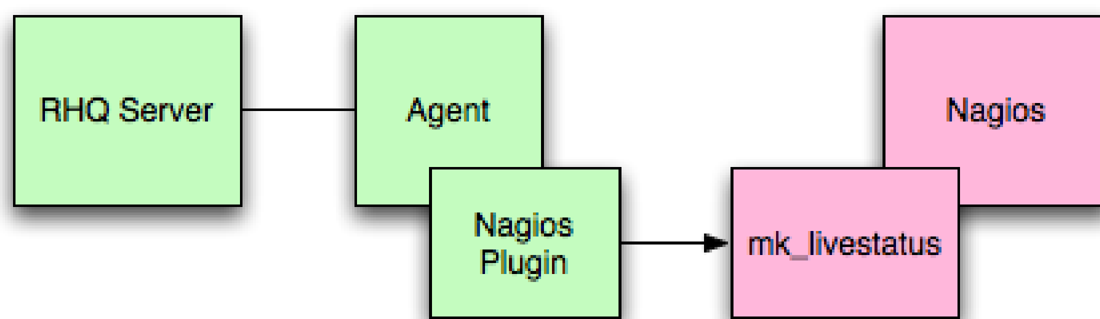


Abbildung 12: Funktionsweise des Nagios Plug-In

[10] MK1
[11] MK2

4.3.2 Entwurf eines Softwaremodels zur Abbildung des Nagios Systems

Um die Anbindung an das Nagios System zu realisieren, musste ein Softwaremodel entwickelt werden, mit dessen Hilfe es möglich ist, ein komplettes Nagios System abzubilden. Ein Nagios System besteht aus einem zentralen Server, einem oder mehreren verteilt laufenden Hosts, den auf den einzelnen Hosts laufenden Services und den zu den Service gehörenden Metriken. Die abzubildenen Elemente stehen also in Relation zueinander, die Relationsinformationen der Objekte zueinander müssen ebenso abgebildet werden können wie die Dateninformationen der aus der Abfrage gewonnenen Daten. Wie bereits in Abschnitt 3.2.3 beschrieben, muss das Datenmodel so flexibel gehalten werden, dass es jede auftretende Änderung innerhalb des überwachten Systems abbilden kann, wie z.B. den Ausfall eines Hosts oder Services oder das Hinzufügen einer neuen Ressource. Zur Abbildung des Nagios Systems wurde als Teil des Plug-In Quellcodes deshalb ein eigenes Paket 'data' implementiert, die darin enthaltenen Klassen dienen ausschließlich der Abbildung des Nagios Systems, sie basieren grundlegend auf dem Design des in Abschnitt 3.2.3 entworfenen, abstrakten Datenmodels zur Abbildung von verteilten Monotring Systemen. Das Paket 'data' enthält die Klassen, die die Nagios Systemarchitektur darstellen, sie bilden die Komponenten des Nagios Systems ab und dienen der Speicherung der von Nagios empfangenen Daten. Abbildung 13 beschreibt die Klassen des Moduls 'data' als UML Diagramm. Die Klasse 'NagiosSystemData' repräsentiert einen Nagios Systemserver und enthält die Datenstrukturen mit den Informationen über alle mit dem Server interagierenden Hosts. Außerdem enthält sie Datenstrukturen mit allgemeinen Statusinformationen wie Availabillity, Performance usw. Die Host- und Statusinformationen werden in den Attributen 'hostData' vom Typ HostData und 'statusData' vom Typ 'StatusData' abgebildet. Beide Typen sind Containertypen, sie enthalten jeweils sämtliche Host- bzw. Statusinformationen des gesamten beobachteten Nagios Systems. Es gibt innerhalb eines Nagios Systems genau einen Systemstatus, daher enthält der Container vom Typ 'StatusData' lediglich ein Status Objekt. Die den Status genauer beschreibenden Teilinformationen sind in Form von Objekten des Typs Metrik realisiert, die notwendige Flexibilität gegenüber Änderungen oder Erweiterungen wird durch die Verwendung von Hashtables als Datenstruktur zur Verwaltung der Metrikobjekte erreicht. Diese bieten die notwendige Flexibilität zur Abbildung der Objektrealationen innerhalb des Nagios Systems, außerdem können in Hashtables große Datenmengen gespeichert werden, und der Zugfriff auf die Daten ist aufgrund der eindeutigen Schlüssel-Objekt-Relation sehr performant. Da es nicht nur einen, sondern mehrere Hosts geben kann, muss zur Abbildung der Relation HostData-Host ebenfalls eine entsprechend flexible Datenstruktur gewählt werden, die diese Relation abbilden kann. Auch in diesem Fall erfolgt die Abbildung der Re-

lationen aufgrund der genannten Eigenschaften durch Hashtables. Die Objekte vom Typ 'Host' stellen je einen Nagios Host dar und beinhalten neben der IP Adresse und dem Namen des Hosts ein Attribut 'serviceData' vom Typ 'ServiceData'. Dieser Typ ist eine weitere Container Klasse, er enthält die Informationen über alle laufenden Services des jeweiligen Host. Die drei Containerklassen implementieren jeweils das Interface 'NagiosData' und definieren die im Interface deklarierten Methoden 'fillWithData()' und 'getSingleMetricForRessource()'. Die Methode 'fillWithData()' dient dem Füllen der Datenstrukturen nach dem Parsen und erhält als Übergabeparameter die Informationen für die Host-, Service- und Metrikobjekte übergeben. Die Methode 'getSingleMetricForRessource()' dient dem Suchen einer bestimmten Metric zu einer Ressource und bekommt Host-, Service- und Metricinformationen als Suchparameter übergeben. Der Container vom Typ 'ServiceData' enthält wiederum ein oder mehrere Objekte vom Typ 'Service', diese stellen einen einzelnen Dienst des ensprechenden Hosts dar. Wie bei den bereits beschriebenen Objekten vom Typ 'Status' beinhalten auch Objekte vom Typ 'Service' eine oder mehrere Metriken, die die Teilinformation abbilden, die Abbildung der Metrikobjekte erfolgt auch hier durch Hashtables. Die Implementierung des Softwaremodels zur Abbildung des Nagios Systems innerhalb des RHQ Plug-Ins basiert auf dem in Abschnitt 3.2.3 entworfenen Konzept, im Rahmen der Implementierung wurde das Model auf die Fähigkeiten von MK_LIvestatus angepasst und in Bezug auf die abtrakte Beschreibung stellenweise erweitert. So sind in der abstrakten Beschreibung in Abschnitt 3.2.3 keine Containerklassen vorgesehen, bei der realen Implementierung zeigte sich aber die Notwendigkeit, einen gemeinsamen Obertyp für die Datenklassen zu haben. Dadurch werden einige Aufgaben, wie das in Abschnitt 4.3.3 beschriebene Parsing, erleichert, da die Methoden und Funktionen lediglich mit Parametern des Obertyps arbeiten und je nach Subtyp interne Entscheidungen treffen. Auch Statusinformatioen sind im abtrakten Model nicht vorgesehen, da diese nicht zwangsläufig zur Abbildung eines Monitoring Systems benötigt werden. Da Livetstatus die Möglichkeit der Abfrage dieser Informationen bietet, wurde das Softwaremodel der Vollständigkeit halber um dieses Feature erweitert. Die Komponenten Server, Host, Service und Metrik entsprechen vollständig dem Konzeptmodel und stehen in den gleichen Entitätsrelationen wie die Komponenten innerhalb des Modelkonzepts.

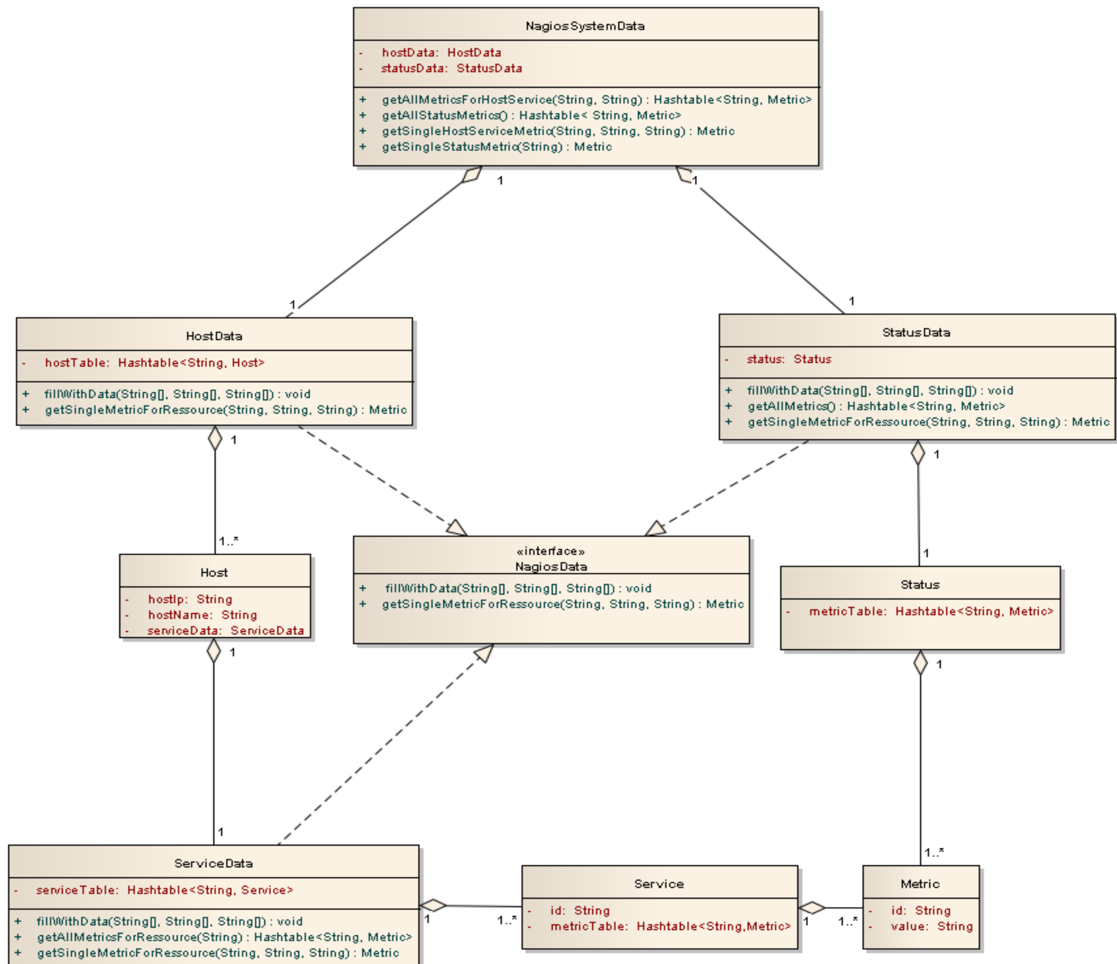


Abbildung 13: Klassen des Moduls 'Data'

4.3.3 Entwicklung eines RHQ Plugins für die Nagios Integration unter Verwendung statischer Metadaten

Nachdem die Evaluation einer geeigneten Schnittstelle zur Abfrage der Nagios Systemdaten und die Implementierung eines geeigneten Softwaremodels zur Abbildung der über die Schnittstelle abgefragten Daten abgeschlossen waren, musste als nächster Schritt die Realisierung des RHQ Plug-Ins unter Verwendung der Schnittstelle und des Softwaremodels angegangen werden. Beim Design des RHQ Nagios Plug-Ins war darauf zu achten, dass die Elemente des Plug-Ins möglichst modular entwickelt wurden, so dass im Falle späterer Anpassungen problemlos Teilfunktionen aus dem Gesamtmodel herausgenommen oder bestimmte Module erweitert werden konnten. Die Notwendigkeit der Implementierung verschiedener Module entstand aus den verschieden Aufgaben, die das Nagios Plug-In abzudecken hatte:

- Suchen von Ressourcen des durch das Plug-In zu überwachenden Typs
- Definition von Livetstatus Abfragen unter Verwendung der Livetstatus Query

Language(LQL)

- Bereitstellung einer TCP-Schnittstelle zur Übermittlung der Abfragen an MK_Livetstatus und zum Empfang der Antwortdaten
- Speicherung der Antwortdaten in geeigneten Objekten
- Parsing der Antwortobjekte und Instanziierung von Objekten unter Verwendung des in Abschnitt 4.3.2 beschriebenen Softwaremodels
- Mapping der Objektinstanzen des Nagios Softwaremodels auf das Softwaremodel des RHQ Systems

Aus den beschrieben Anforderungen entstanden mehrere Softwarepakete, die wiederum einzelne oder mehrere Klassen enthalten. Alle Klassen eines Pakets dienen dem gleichen Zweck, daher erfolgte die Beschreibung der Funktionalität paketweise. Folgende Pakete wurden realisiert:

- **rhqNagiosPlugin**: Das Paket 'rhqNagiosPlugin' enthält die für das Auffinden der zu überwachenden Ressourcen und die Instanziierung der Ressourcen notwendigen Discovery- und Component-Klassen. Die prinzipiellen Aufgaben der Discovey- und Component-Klassen innerhalb der RHQ Plug-Ins wurde bereits in Abschnitt 1.1 ausführlich beschrieben, diese Kernaufgaben müssen für alle RHQ Plug-Ins erfüllt werden. Das Paket 'rhqNagiosPlugin' enthält die Klassen 'NagiosMonitorDiscovery' und 'NagiosMonitorComponent', Erstere ist für das Auffinden der Nagios Ressourcen zuständig, Letztere für die Instanziierung je eines Component-Objektes für jede gefundene Ressource. Listing 1 zeigt einen Codeausschnitt der Methode 'discoverResources()' der Klasse 'NagiosMonitorDiscovery', die für das Auffinden neuer Ressourcen zuständig ist. Der Code zeigt die Stelle, an der die Informationen zu einer gefundenen Ressource in ein Objekt der Klasse 'DiscoveredResourceDetail' geschrieben wird. Dies geschieht für jede gefundene Ressource, alle erstellten 'DiscoveredResourceDetail'-Objekte werden in ein HashSet gepackt und dieses wird zurück gegeben.

Listing 1: Objekterzeugung

```
1 public Set<DiscoveredResourceDetails> discoverResources(
    ResourceDiscoveryContext discoveryContext) throws
    Exception {
2
3         DiscoveredResourceDetails detail = new
              DiscoveredResourceDetails(
```

```
4                  wanted, "nagiosKey@" + "Nr:" + i + ":" +
                       resourceTypeReply.getLqlReply().get(i), "
                       Nagios@" + "Nr:" + i
5                       + ":" + resourceTypeReply.getLqlReply().get(
                           i), null, "NagiosService: "
6                       + resourceTypeReply.getLqlReply().get(i),
                           null, null);
7             discoveredResources.add(detail);
8
9     return discoveredResources;
10 }
```

Anschließend wird für jedes Objekt der Klasse 'DiscoveredResourceDetail' eine Instanz der Klasse 'NagiosMonitorComponent' erstellt und mit den Informationen gefüllt. Auf diese Weise läuft dann innerhalb des Systems eine Instanz vom Typ 'Resource' für alle gefundenen Objekte vom Typ 'NagiosMonitor'. Um die Ressourcen im System mit Informationen füllen zu können, muss die Methode 'start' innerhalb der Klasse 'NagiosMonitorComponent' implementiert werden, die nach der Instanziierung der 'NagiosMonitorComponent'-Klasse aufgerufen wird. Listing 2 zeigt die Defintion der Methode 'start()'. Aus dem übergebenen Context-Objekt wird die Plug-In Konfiguration ausgelsen, dadurch kann man an Informationen aus dem Konfigurationsteil des Plug-In-Deskriptors gelangen, falls man diese benötigt. Dann werden die Portnummer und die IP-Adresse aus der Konfiguration ausgelesen und an ein Objekt vom Typ 'NagiosManagementInterface' übergeben. Dadurch wird die Verbindung zu MK_Livetstatus initialisiert, die Funktionsweise der Klasse 'NagiosManagementInterface' und der anderen, als Teile des Plug-Ins, implementierten Klassen, werden später in diesem Abschnitt noch genauer beschrieben.

Listing 2: Objekterzeugung

```
1 public void start(ResourceContext context) throws
      InvalidPluginConfigurationException, Exception {
2           this.context = context;´
3
4   Configuration conf = context.getPluginConfiguration();
5   nagiosHost = conf.getSimpleValue("nagiosHost",
        DEFAULT_NAGIOSIP);
6   String tmp = conf.getSimpleValue("nagiosPort",
        DEFAULT_NAGIOSPORT);
7   nagiosPort = Integer.parseInt(tmp);
8
```

```
9           nagiosManagementInterface = new
                NagiosManagementInterface(nagiosHost, nagiosPort)
                ;
10  }
```

- **managementInterface**: Das Paket 'managementInterface' enthält die für die Steuerung der Abläufe der Datenermittlung und Datenweiterverwertung zuständige Klasse 'NagiosManagementInterface'. Die Klasse 'NagiosManagementInterface' ist die zentrale Steuerklasse der Nagios Schnittstelle und enthält als Attribute die IP Adresse und die Portnummer von MK Livetstatus, eine Instanz der Klasse 'NetworkConnection' zur Abfrage der Daten von MK_Livetstatus und eine Instanz der Klasse 'Controller'. Sie ist zuständig für die Initiierung der Datenabfrage von Livetstatus und die anschließende Initiierung des Parsingprozesses auf oberstem Managementlevel. Die internen Abläufe der anderen Pakete und ihrer Klassen werden durch Aufruf der Methode 'createNagiosSystemData()' angetriggert und die ermittelten Systemdaten in ein Attribut der Klasse 'NagiosSystemData' geschrieben, welches den Nagios Server mit allen Systemdaten abbildet. Das Paket dient also der Abstraktion der komplexen Abläufe auf eine höhere Ebene, um die Datengewinnung für die Entwickler einfacher zu gestalten. Abbildung 14 zeigt die Klassen des Pakets 'managementInterface' in Form eines UML Diagramms.

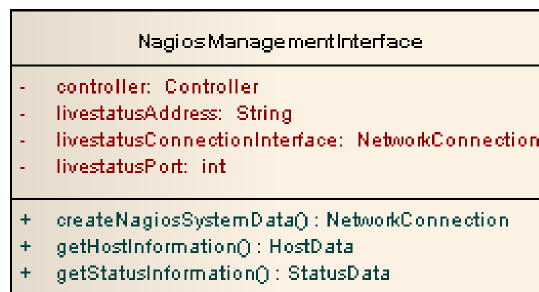


Abbildung 14: Klassen des Pakets 'ManagementInterface'

- **controller**: Das Paket 'controller' enthält die Klassen, die zur Auswahl der korrekten Parsing-Funktionen notwendig sind. Je nach Abfragekontext werden andere Informationen zurück geliefert. Abbildung 15 beschreibt die Klassen des Moduls 'Controller' in UML. Die Klasse 'Controller' enthält ein Attribut 'lqlReplyParser', dieses Attribut ist eine Instanz der, für das Parsing der Livestatus Antworten zuständigen, Klasse 'LqlReplyParser'. Darüber hinaus enthält die Klasse die Methode 'createDataModel()', mit der die Antwortobjekte vom Typ 'LqlReply' auf ihren Kontext geprüft und an die entsprechende Funktion des

Parsers übergeben werden. Der Kontext gibt Auskunft darüber, ob die LqlReply eine Antwort auf einen Host-, Service- oder Statusrequest enthält, und instanziiert das Attribut 'nagiosData' vom allgemeinen Obertyp 'NagiosData' mit dem entsprechenden Untertyp 'HostData', 'ServiceData' oder 'StatusData'. Die 'createDataModel()'-Methode der Klasse Controller wird innerhalb der Methoden 'getHostData()' und 'getStatusData()' der Klasse 'NagiosManagementInterface' aufgerufen, um die über die Netzwerkschnittstelle empfangenen Daten zu Parsen und daraus die korrekten Objekte des Datenmodels zu intanziieren. Die beiden letzteren Methoden werden wiederum in der Methode 'createNagiosSystemData()' aufgerufen, diese übergibt die HostData- und StatusData-Informationen an den Konstruktor der Klasse 'NagiosSystemData' und gibt eine Instanz dieser Klasse mit allen Systeminformationen zurück.

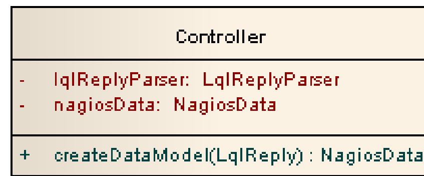


Abbildung 15: Klassen des Moduls 'Controller'

- **request**: Das Paket 'request' enthält die Klassen, die die verschiedenen möglichen Anfragen an Livetstatus implementieren. Mögliche Anfragen an Livestatus sind HostRequest, ServiceRequest und StatusRequest. Ein HostRequest dient der Abfrage von Informationen eines bestimmten Hosts, ein ServiceRequest dient der Abfrage von Informationen eines bestimmten Services, und ein StatusRequest dient der Abfrage von allgemeinen Statusinformationen des Nagios Servers. Die Basis des Pakets 'request' bildet das Interface 'LqlRequest', in diesem Interface werden die Methoden getRequestQueryList(), setRequestQueryList(), getRequestType() und setRequestType() deklakriert. Die Klassen LqlHostRequest, LqlServiceRequest und LqlStatusRequest implementieren das Interface 'LqlRequest' und definieren die Methoden nach ihrem jeweiligen Bedarf. Die drei Klassen enthalten jeweils die Attribute 'requestQueryList' und 'requestType'. Das Attribut 'requestQueryList' enthält das, je nach je nach Abfragetyp, notwendige LQL Kommando für die Abfrage der Host-, Service- oder Statusinfomation. Eine LQL Abfrage ist meistens eine Kombination mehrerer Teilkommandos, daher ist die Speicherung der Teilkommandos in Form einer ArrayList realisiert, aus deren Elementen das Gesamtkommando zusammengesetzt werden kann. Dadurch kann durch Hinzufügen neuer Attribute in der ArrayList eine bequeme Anpassung der LQL Kommandos vorgenommen werden, dies erhöht die Flexibilität gegenüber Änderungen. Das Attribut 're-

questType' bestimmt die Art der Anfrage an Livestatus, also, ob es ein Host-, Service odet Statusrequest ist. Das Attribut ist vom Typ 'NagiosRequestType', dieser Typ enthält als Attribute einen EnumerationType, der die möglichen Arten von Requests aufzählt, und ein Attribut 'nagiosRequestType', das mit einem der Enumeration Werte initialisiert wird. Der RequestType spielt eine wichtige Rolle beim Senden der Anfrage über die Socket Schnittstelle, da je nach RequestTyp ein entsprechendes RequestType Flag im Reply Objekt gesetzt wird, in das die Antwort auf die Anfrage geschrieben wird. Das bedeutet, dass bei der Instanziierung der Antwortobjekte eine Information in diese Objekte geschrieben wird, auf welche Art von Anfrage das Objekt die Antwort darstellt. Diese Typinformation wird beim Parsen der Reply Information benötigt, um die geparsten Monitoring-Information den richtigen Objekten des Nagios Softwaremodels aus dem Paket 'data' zuzuweisen und das Nagios System so korrekt abzubilden. Abbildung 16 zeigt eine Übersicht über die Klassen des Pakets 'request' in Form eines UML Diagramms.

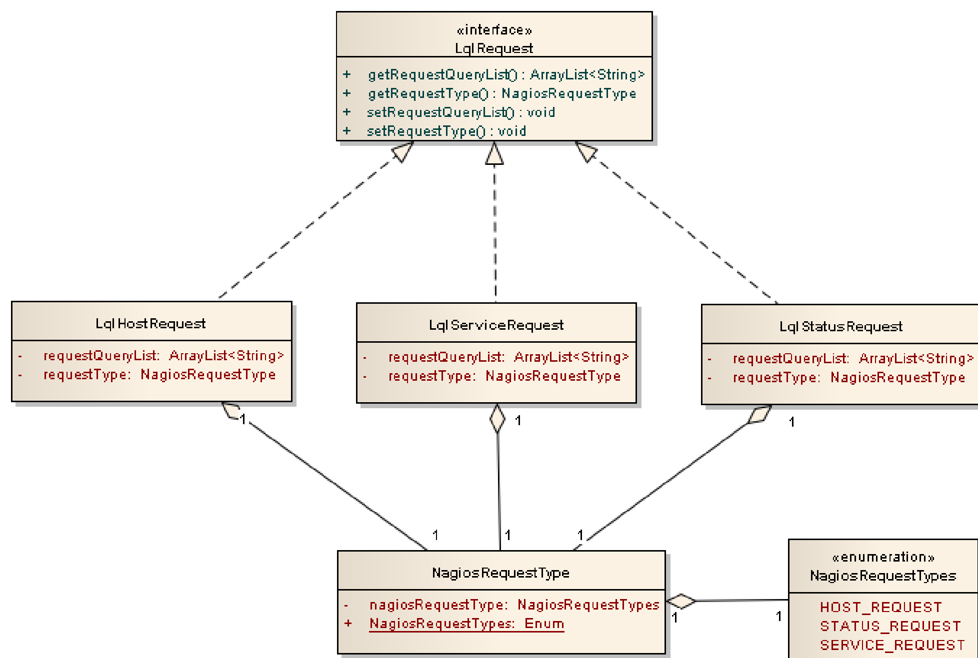


Abbildung 16: Klassen des Moduls 'Request'

- **network**: Das Paket 'network' enthält die Klassen, die die TCP Schnittstelle realisieren, die zur Kommunikation mit dem MK_Livestatus-Modul benötigt wird, von dem die Monitoring Daten von Nagios abgefragt werden können. Abbildung 17 beschreibt die Klassen des Moduls 'Network' als UML Diagramm.

Die Klasse 'NetworkConnection' enthält die Attribute 'destinationAdress' und 'destinationPort' mit der IP Adresse und der Portnummer von MK Livetstatus, außerdem das Socket-Attribut über das die Verbindung aufgebaut werden kann. Die Klasse enthält außerdem die benötigten Stream- und Reader-Objekte zum Lesen und Schreiben des Datenstroms auf dem Socket. Die Klasse implementiert die Methoden 'openConnection()' und 'closeConnection()' zum Öffnen und Schließen der Reader-, Writer und Socket-Objekte. Außerdem enthält die Klasse die Methode 'sendAndReceive()' zum Senden der Anfragen an MK_Livetstatus und zum Empfangen der entsprechenden Antworten über die Socket Schnittstelle. Die Methode 'sendAndReceive()' bekommt als Übergabeparameter Anfragen an MK_Livetstatus in Form von Objekten des Typs 'LqlRequest' und schreibt die Antwortdaten in Objekte des Typs LqlReply. Die Objekte des Typs LqlReply werden innerhalb der Methode 'sendAndReceive()' erstellt und mit dem Kontext der übergebenen LqlRequest initialisiert, um das spätere Parsing vorzubereiten, so wie es bei der Beschreibung des Pakets 'request' bereits erklärt wurde.

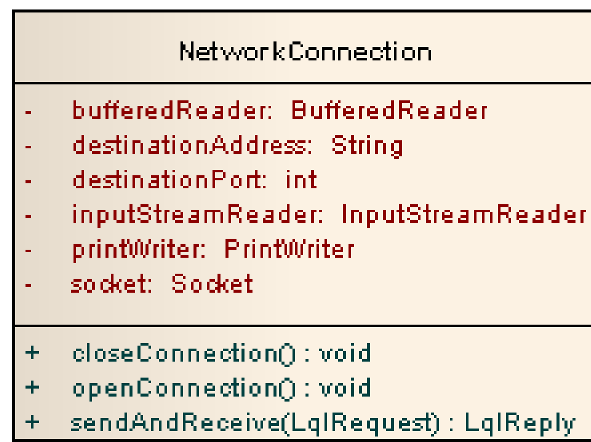


Abbildung 17: Klassen des Moduls 'Network'

- **reply**: Das Paket 'reply' enthält die Klasse 'LqlReply', die eine Antwort auf eine MK_Livestatus Datenanfrage abbildet. Abbildung 18 zeigt eine UML Darstellung der Klasse 'LqlReply'. Die Klasse enthält ein Attribut 'lqlReply' vom Typ ArrayList<String>, da die Antwort auf verschiedene Anfragen mehr als eine Zeile lang ist, und daher eine Datenstruktur verwendet werden muss, in der eine mehrzeilige Antwort problemlos gespeichert werden kann. Außerdem ist ein Attribut 'context' vom Typ 'NagiosRequestType' enthalten, in diesem Feld wird beim Aufruf des Konstruktors der Klasse 'LqlReply' die Kontextinformation reingeschrieben. Die Kontextinformation wird beim Parsen benötigt und gibt Auskunft darüber, ob das Reply-Objekt eine als Antwort auf eine Host-,

Service- oder Statusrequst erstellt wurde.

Abbildung 18: Klassen des Moduls 'Reply'

- **parser**: Das Paket 'parser' enthält die Klasse 'LqlReplyParser', die zum Parsen der von der Netzwerkschnittstelle gelieferten Reply Objekte verwendet wird. Die Reply Objekte werden mit den von der Netzwerkschnittstelle als Antwort auf die Anfragen an Livestatus gelieferten Daten initialisiert und von den Klassen des Pakets 'controller' an die richtigen Methoden des Pakets 'parser' übergeben. Abbildung 19 beschreibt die Klasse 'LqlReplyParser' als UML Diagramm. Die Klasse implementiert die Methoden 'parseLqlHostReply', 'parseLqlServiceReply' und 'parseLqlStatusReply', die zum Parsen der Antworten auf Host-, Service- und Statusrequest dienen. Für das Parsing der verschiedenen Reply Objekte sind unterschiedliche Methoden notwendig, da je nach Kontext des Reply Objektes andere Instanzen der Klassen des Pakets 'data' erstellt werden müssen, um das überwachte System korrekt abzubilden. Jede der drei Methoden erwartet als Übergabeparameter das zu parsende Reply Objekt und ein Objekt vom im Paket 'data' definierten Typ 'NagiosData'. Das Objekt vom Typ 'NagiosData' ist ein Containerobjekt, in dieses Objekt werden die jeweiligen, beim Parsing des Reply Objektes gewonnenen Systeminformationen reingeschrieben und intern in die entsprechenden Datenstrukturen verteilt, so wie es bei der Bescheribung des Pakets 'data' erläutert wurde. Die Klasse 'LqlReplyParser' und ihre Methoden werden innerhalb der Klasse Controller verwendet, dort wird der Kontext der Reply ermittelt und die korrekte, dem Kontext entsprechende Parsingmethode aufgerufen.

Abbildung 19: Klassen des Moduls 'Parser'

- **error**: Das Paket 'error' enthält die Klassen, die im Fall des Auftretens von Fehlern verwendet werden. Die Klassen sind Unterklassen des Typs 'Exception' werden im Falle auftretender Fehler als Exception geworfen. Jede der Klassen wurde für einen anderen Fehlerfall entwickelt. Die Klasse 'InvalidHostRequestException' wird geworfen, wenn bei der Abfrage von Hostdaten ungültige oder fehlerhafte Parameter mitgegeben wurden, die Klasse 'InvalidServiceRequestException' wird geworfen, wenn bei der Abfrage von Servicedaten ungültige oder fehlerhafte Parameter mitgegeben wurden, und die Klasse 'InvalidMetricRequestException' wird geworfen, wenn bei der Abfrage von Metrikdaten ungültige oder fehlerhafte Parameter mitgegeben wurden. Die Klasse 'InvalidReplyTpeException' wird geworfen, wenn beim Parsing ein Reply Objekt an eine nicht dafür geeignete Methode übergeben wurde. Abbildung 20 zeigt die UML Notaion der Klassen des Moduls 'error'.

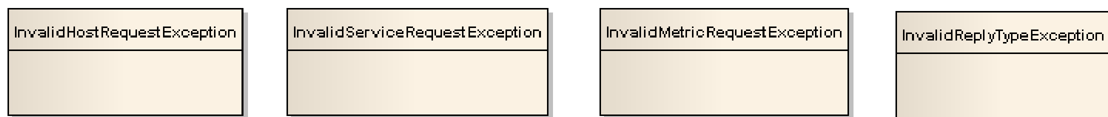


Abbildung 20: Klassen des Moduls 'Error'

Abbildung 21 verdeutlicht die Arbeitsweise des Nagios Plug-Ins auf mittlerer Abstraktionsebene, es werden sowohl die Aufgaben der einzelnen Module als auch die Interaktionen der Module in Form eines UML Kommunikationsdiagramm dargestellt. Die Arbeitsweise des Plug-Ins ist folgende: nachdem die zu überwachenden Ressourcen von der Klasse 'NagiosMonitorDiscovery' ermittelt wurden, wird für jede gefundene Ressource eine eigene Instanz vom Typ 'NagiosMonitorComponent' erstellt. Der zu überwachende Ressourcentyp heißt 'NagiosMonitor' und ist von der Ressourcenkategorie 'Server'. Es gibt genau eine Ressource vom Typ 'NagiosMonitor', nämlich der auf dem System installierte Nagios Server. Die Component Klasse wird gestaret und beginnt mit der Datenermittlung zum Füllen der zu überwachenden Services und den zugehörigen Metriken. Sie ruft die Methode 'createNagiosSystemData()' der Klasse 'NagiosManagementInterface' aus dem Paket 'managementInterface' auf, mit der die Ermittlung der Monitoringdaten des Nagios System angetriggert wird (Schritt 1). Im Modul 'managementInterface' wird für die entsprechende Abfrage der Livestatus Daten ein Request Objekt erstellt (Schritt 2) und an die Methode 'sendAndReceive()' der Klasse 'NetworkInterface' im Modul 'network' übergeben (Schritt 2.1). Die Methode 'sendAndReceive' sendet eine Datenabfrage an das MK_Livestatus Modul (Schritt 3), dort wird die Anfrage bearbeitet und die ermittelten Daten an das Modul 'network' zurück geschickt(Schritt 4). Dort werden die Daten in ein Reply Objekt gepackt (Schritt 5), dieses wird anschließend an das

Modul 'managementInterface' zurück geliefert (Schritt 5.1). Das Modul 'managementInterface' übergibt das vom Modul 'network' gelieferte Reply Objekt mit den Antworten der Livestatus Abfrage an das Modul 'controller' (Schritt 6). Das Modul 'controller' überprüft den Kontext des Reply Objektes (Schritt 7) und ruft die dem Kontext entsprechende Methode des Moduls 'parser' auf, der er das Reply Objekt übergibt (Schritt 7.1). Das Modul 'parser' zerlegt die im Reply Objekt gespeicherten Monitoring Informationen und instanziiert die entsprechenden Objekte des Nagios Softwaremodels (Schritt 8). Das mit den Messdaten gefüllte Softwaremodel wird vom Modul 'parser' an das Modul 'controller' zurückgeliefert (Schritt 8.1), dieses liefert die Daten weiter an das Modul 'managementInterface' (Schritt 9), von wo die Messdaten schließlich als Return-Wert zum initialen Aufruf der Methode 'createNagiosSystemData() an die Component-Klasse des Moduls 'rhqNagiosPlugin' zurückgeliefert werden. In der Component-Klasse liegen dann alle Nagios-Monitoringdaten vor, diese können einfach über entsprechende Methodenaufrufe abgefragt und den einzelnen Service bzw. deren Metriken zugeordnet werden.

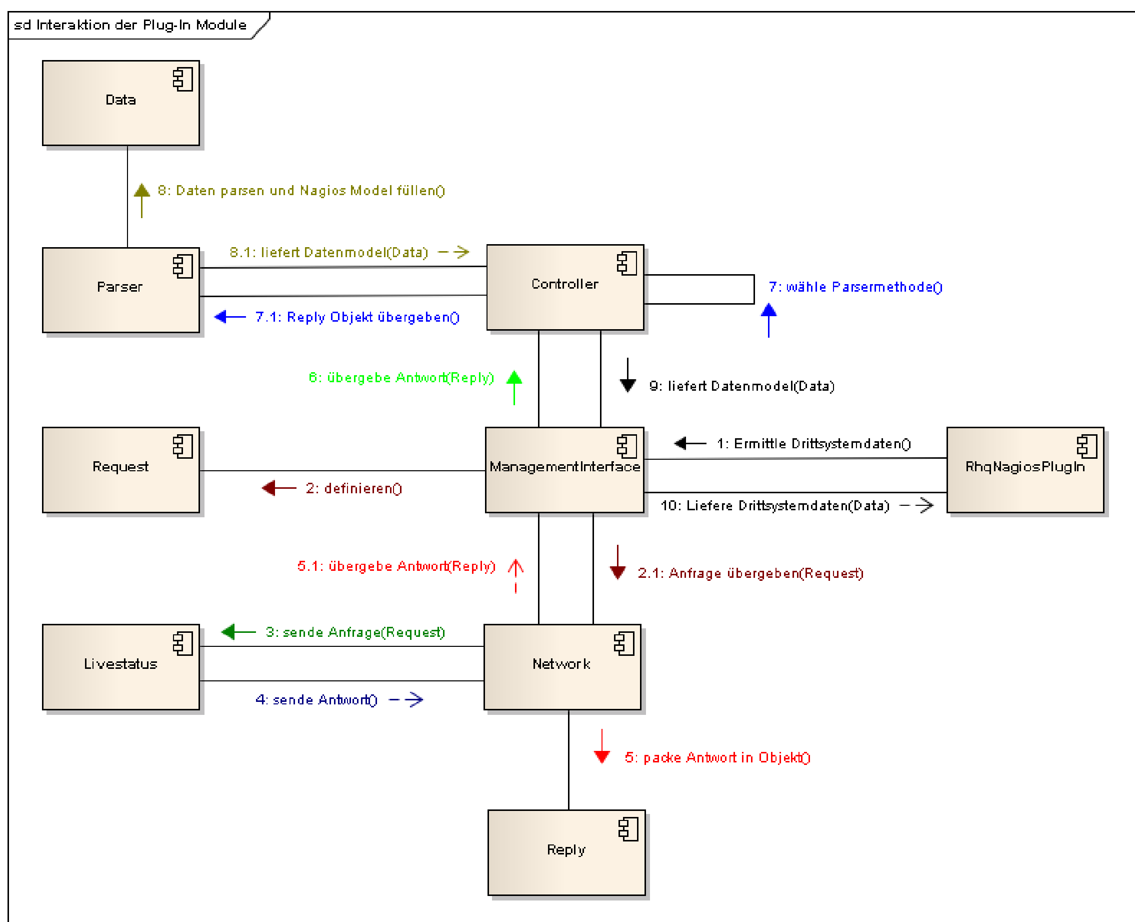


Abbildung 21: Interaktion der Module des Nagios Plug-Ins

4.4 Umsetzung der Migration zur Bereitstellung dynamischer Metadaten

In diesem Abschnitt wird die Realisierung der Arbeitsschritte beschrieben, die für die Migration zur Verwendung dynamischer Metadaten notwendig sind. Die Realisierung wurde in drei aufeinander folgenden Schritten vollzogen, die funktional aufeinander aufbauen, dabei wurden nacheinander die notwendigen Schritte in den Archtitekturkomponenten Server, Agent und Plug-In umgesetzt. In den folgenden Abschritten werden die notwendigen Anpassungen in den einzelnen Modulen beschrieben. Abbildung 22 zeigt die einzelnen Architekturkomponenten und die jeweilgen, für die Realisierung des Prototyps notwendigen, den Architekturkomponenten zugeordneten Klassen und deren Methoden. Es soll eine erste Übersicht bieten und das Verständnis der in den fogenden Abschnitten beschriebenen Abläufe vereinfachen.

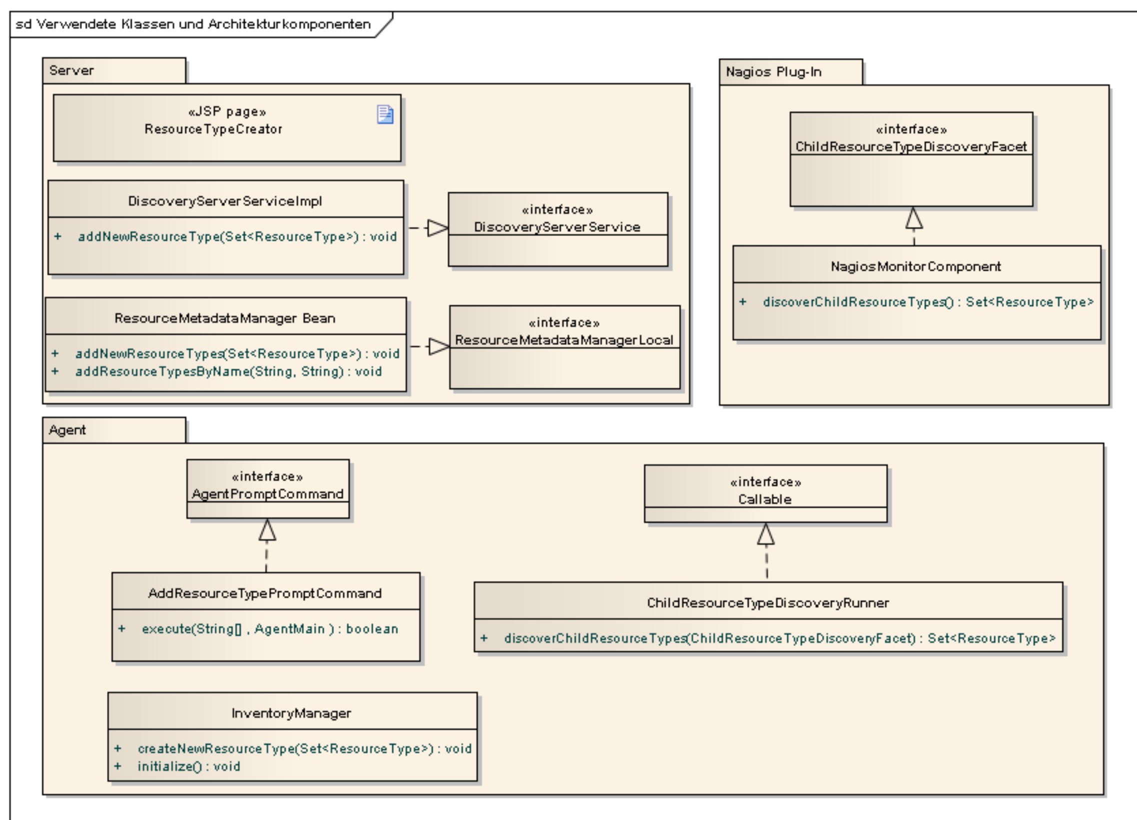


Abbildung 22: Verwendete Klassen und Architekturkomponenten

4.4.1 Dynamische serverseitige Generierung neuer Resourcetypen

Zunächst wurden die für die Migration notwendigen Anpassungen innerhalb des Servers vorgenommen, da der Server die zentrale Komponente des Systems darstellt und dort die Persistierung der Daten erfolgt. Die Persitierung stellt einen essentiellen Bestandteil des Auswertungsprozesses der Metadaten dar, nur in der Datenbank persistierte Daten werden auch in der Benutzeroberfläche des Servers angezeigt und können beobachtet werden. Alle weiteren zur Migration notwendigen Schritte, egal ob innerhalb der Architekturkomponenten 'Agent', 'Server' oder 'Plug-In', schließen mit der Persistierung der Daten in der Serverdatenbank ab und beinhalten daher, dass die für die Persistierung notwendigen Systemfunktionen vorher angepasst oder neu geschrieben wurden. Um einen ersten Prototypen zu realisieren, musste zunächst eine Schnittstelle implementiert werden, die zur Eingabe von Metadaten für die neuen Ressourcentypen und die zugehörigen Metriken verwendet werden konnte. Die Schnittstelle wurde in die GUI des Servers eingebunden, so dass sie vom Anwender zur Eingabe neuer Metadaten genutzt werden konnte. Die Implementierung der Schnittstelle erfolgte in Form einer JSP-Page, die in das Web-Modul des Servers eingehangen wurde. Nachdem das erweiterte Web-Modul neu kompiliert und in den, als Kern des RHQ Systems, laufenden JBoss Applikationsserver deployt worden war, konnte die JSP-Seite über die GUI des Servermoduls aufgerufen werden. Sie beinhaltete ein Feld zur Eingabe des Namens des neuen Ressourcentyps und ein weiteres Feld zur Eingabe des Namens einer zugehörigen Metrik. Die JSP-Seite musste dann an die entsprechenden Backend-Klassen angebunden werden, um die abschließende Persistierung der Ressourcendaten in der Systemdatenbank durchführen. Dazu musste überprüft werden, welche Klassen und Methoden bei der Persitierung der aus dem Plug-In-Deskriptor generierten Objektinstanzen benutzt werden, das Verständnis der komplexen Systemabläufe war hierfür unverzichtbar. Um die Abläufe zu verstehen war es notwendig, das laufende System an einen geeigneten Debugger anzuschließen und einen Einstiegspunkt zur Systembeobachtung zu finden. RHQ bietet die Möglichkeit, sowohl den Server als auch den Agent im laufenden Betrieb über Remote-Zugriff zu debuggen. Dazu mussten entsprechende Anpassungen innerhalb der Startskripte des jeweiligen Architekturmoduls gemacht werden, das Debugging des Quellcodes konnte von Eclipse aus erfolgen. Abbildung 23 zeigt die zum Einschalten des Remote Debuggings notwendigen Anpassungen innerhalb des Startskripts.

```
RHQ_SERVER_JAVA_OPTS="-Xmx256m -XX:MaxPermSize=256m \
-Djava.net.preferIPv4Stack=true \
-agentlib:jdwp=transport=dt_socket,address=8787,server=y,suspend=n \
-Djboss.platform.mbeanserver"
```

Abbildung 23: Einschalten des Remote Debuggings

Durch das Debugging der Systemabläufe wurde klar, dass die Persistierung der Metadaten von der Klasse 'ResourceMetadataManagerBean' durchgeführt wird, die im Anschluss an den Parsing Prozess des Deskriptors aufgerufen wird. Die Klasse implementiert das Interface 'ResourceMetatdataManagerLocal' und ist eine typisches EJB3 Bean, bestehend aus einem lokalem Interface mit Methodendeklaration und der zugehörigen Interfaceimplementierung in einer Klasse. Das Interface wurde um zwei Methoden erweitert, die in der zugehörigen Klasse implementiert wurden, diese Methoden dienen der Persistierung der über die JSP-Seite eingegebenen Metadaten. Listing 3 zeigt die Methodenrümpfe der neuen Methoden.

Listing 3: Methoden zum Persitieren der Metadaten

```
1 void addNewResourceTypeByNames(String newResourceTypeName, String
      metricName);
2 void addNewResourceType(Set<ResourceType> resourceTypes);
```

Die Methode 'addNewResourceTypeByNames' erhält als Übergabeparameter zwei Stringobjekte mit dem Namen des neuen Ressourcentyps und dem Namen einer zugehörigen Metrik. Bevor die Persistierung in der Datenbank erfolgen kann, muss zunächst ein Objekt vom Typ 'ResourceType' für den neuen Ressourcentyp und ein Objekt vom Typ 'MeasurementDefinition' für die zugehörige Metrik erstellt werden. Der Konstruktor der Klasse 'ResourceType' benötigt folgende Übergabeparameter

- den Name des neuen Ressourcentyps
- den Name des entsprechenden Plug-Ins
- die Ressourcenkategorie des neuen Typs
- den Ressourcentyp des Parent-Objekts

Da das RHQ System modular aufgebaut ist, werden beim Build des Gesamtsystems die einzelnen Module zu zusammenhängenden Beans kompiliert und in den im RHQ Kern laufenden JBoss Applikationsserver deployt. Um eine Bean zu verwenden, muss daher ein entsprechender Naming Service aufgerufen werden, der eine Instanz der deployten Bean zurück liefert, deren Funktionalität man nutzen kann. RHQ bietet die WrapperKlasse 'LookupUtil', die den Aufruf des Naming Services verhüllt und über Methoden Instanzen verschiedener Bean-Klassen liefert, so dass der Entwickler sich nicht selber darum kümmern muss. Die Klasse 'LookupUtil' wird benötigt, um an die zuvor genannten Informationen für den neuen Ressourcentyp zu gelangen. Listing 4 zeigt die Aufrufe unterschiedlicher Methoden der Klasse 'LookupUtil', mit deren Hilfe man an die zur Objekterzeugung notwendigen Informationen gelangt.

Listing 4: Beschaffen der notwendigen Informationen

```
1 Plugin plugin = LookupUtil.getResourceMetadataManager().getPlugin
    ("NagiosMonitor");
2 ResourceType parentResourceType = LookupUtil.
    getResourceTypeManager().getResourceTypeByNameAndPlugin(
3             "NagiosMonitor", "NagiosMonitor");
```

Nachdem alle benötigten Informationen gesammelt wurden, können die Objekte der Klassen 'ResourceType' und 'MeasurementDefinition' angelegt werden. Die Klasse 'MeasurementDefinition' bildet eine Metrik ab und benötigt den neuen Ressourcentyp als Parameter, damit die Metrik einem Ressourcentyp zugewiesen werden kann. Listing 5 zeigt den Aufruf der Konstruktoren der beiden Klassen.

Listing 5: Objekterzeugung

```
1 ResourceType newResourceType = new ResourceType(
    newResourceTypeName, plugin.getName(), ResourceCategory.
    SERVICE, parentResourceType);
2 MeasurementDefinition measurementDef = new MeasurementDefinition(
    newResourceType, metricName);
```

Um diesen ersten Migrationsschritt abzuschließen, musste innerhalb der JSP-Page noch der Aufruf der Methode 'addResourceTypeByName' erfolgen. Auch hier wird die Klasse 'LookupUtil' verwendet, um an die im Applikationsserver laufende Bean und deren benötigte Methode zu gelangen, wie Listing 6 zeigt.

Listing 6: Beispielcode

```
1 LookupUtil.getResourceMetadataManager().addNewResourceTypeByNames
    (resourceTypeName, metricName);
```

Abbildung 24 zeigt den Ablauf des Prozesses von der Eingabe durch den Benutzer auf der JSP Seite über den Methodenaufruf zur Persistierung.

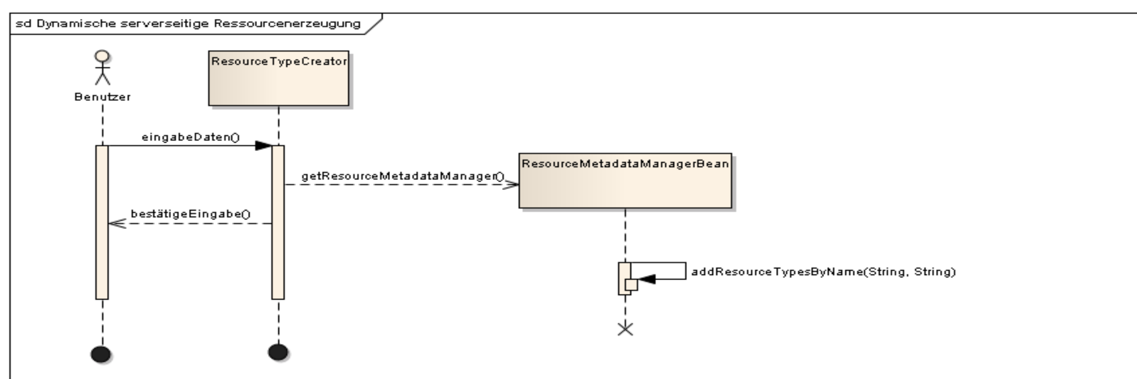


Abbildung 24: Ablauf der Serverkommunikation

4.4.2 Dynamische agentenseitige Generierung neuer Resourcetypen und Synchronisation mit dem Server

Nachdem in einem ersten Prototyp die Anpassung der severseitgen Klassen und Methoden realisiert wurde, das Anlegen neuer Ressourcentypen und die abschließende Persistierung in der Systemdatenbank durchgeführt wurde, musste die Anpassung der agentenseitigen Mechanismen vollzogen werden, um den nächsten Schritt in Richtung erfolgreicher Migration zu gehen. Dieser Schritt umfasste mehrere Anforderungen, die alle erfüllt werden mussten, um die agentenseitigen Softwaremodule den Anforderungen eines dynamischen Systems anzupassen. Folgende Schritte mussten realsiert werden:

- Schaffung einer Kommandozeilenschnittstelle zur agentenseitigen Metadateneingabe
- Auswertung der Metadaten und Instanziierung der neuen Objekte
- Übermittlung der neuen Objekte an den Server

Die Schaffung der Kommandozeilenschnittstelle zur Eingabe der Metadaten sollte dem Zweck dienen, die Abläufe bei Kommandozeilenoperationen mit dem Agenten zu verstehen, darüber hinaus war es sinnvoll, auch direkt am Ageten neue Metadaten eingeben zu können, ohne den Server einzubeziehen. Da zur Persitierung die Daten sowieso zum Server übertragen werden müssen, ist es nicht notwendig, erst die Daten vom Server an den Agenten und dann wieder zurück zu übermitteln, sondern man kann diesen Schritt im Arbeitsablauf sparen. Um das Anlegen neuer ResourceTypen per Kommandozeile zu initialisieren, musste ein entsprechendes Kommando implementiert werden. Dazu wurde eine neue Klasse 'AddResourceTypePromptCommand' geschrieben, die in Abbildung 22 in UML Form dargestellt ist. Diese Klasse musste das Interface 'AgentPromptCommand' und die darin deklarierte Methode 'execute()' implementieren, um als Kommandozeilenbefehl verwendet werden zu können. In der Methode 'execute()' mussten folgende Abläufe realisiert werden:

- Erstellung einer Instanz des Plug-In-Containers
- Erstellung einer Instanz des InventoryManagers
- Aufruf einer entsprechenden Methode des InverntoryManagers zum Anlegen eines neuen Ressourcentyps

Die Erstellung einer Instanz des Plug-In-Containers ist notwendig, da der Plug-In-Container die Informationen über den zuständigen InventoryManager enthält. Der

InventoryManager ist eine zentrale Klasse des Agenten, er enthält die Informationen über das lokale Inventar, also darüber, welche Ressourcen momentan von welchem Plug-In des Agenten gemanaged werden. Ein laufender Agent verfügt über die Informationen über seinen zuständigen Plug-In-Conatiner, mithilfe der statischen Methode 'getInstance' der Klasse 'PluginContainer' kann eine Instanz des Plug-In-Conatiners zurückgeliefert werden. Durch den Aufruf der Methode 'getInventoryManager()' der neuen Instanz der Klasse 'PluginContainer' wird eine Instanz des InventoryManager zurückgeliefert. Von dieser Instanz muss dann die Methode 'createNewResourceType()' aufgerufen werden, der dann die Namen des neuen ResourceType und seiner zugehörigen Metrik mitgegeben werden. Da es sich um einen Prototypen handelt, erfolgt die Übergabe der Parameter hardcodiert und nicht als Parameter des Kommandozeilenbefehls. Listing 7 zeigt die beschriebenen Abläufe innerhalb der Methode 'execute()'.

Listing 7: Beispielcode

```
1 PluginContainer pc = PluginContainer.getInstance();
2 InventoryManager inventoryManager = pc.getInventoryManager();
3 inventoryManager.createNewResourceType("AlexTestType", "
    AlexTestTypeMetric");
```

Damit der Agent den Kommandozeilenbefehl ausführen kann, muss der Befehl innerhalb der Methode 'getPromptCommandString()' definiert werden, diese Methode ist ebenfalls im Interface 'AgentPromptCommand' deklariert und muss in jeder Klasse, die einen Kommandozeilenbefehl repräsentiert, implementiert werden. Listing 8 zeigt die Implementierung der Methode innerhalb der Klasse 'AddResourceTypePromptCommand'.

Listing 8: Beispielcode

```
1 public String getPromptCommandString() {
2         String addResourceTypeCommand = "addType";
3         return addResourceTypeCommand;
4 }
```

Damit die Klasse innerhalb des Agenten aufgerufen werden kann, und der Kommandozeilenbefehl korrekt ausgeführt wird, muss innerhalb der Methode 'setupPromptCommandsMap' der Klasse 'AgentMain' noch eine Instanz der Klasse 'AddResourceTypePromptCommand' angelegt werden. Listing 9 zeigt den Aufruf des Konstruktors der Klasse 'AddResourceTypePromptCommand' innerhalb der Methode 'setupPromptCommandsMap'.

Listing 9: Beispielcode

```
AgentPromptCommand[] all\_cmds = new AgentPromptCommand[] {new
    AddResourceTypePromptCommand()};
```

Die bereits erwähnte Methode 'createNewResourceType()' musste innerhalb der Klasse 'InventoryManager' neu geschrieben werden und sollte die Aufgaben der Instanziierung der neuen Objekte und der Übermittlung der neuen Objekte an den Server erfüllen. Die Instanziierung neuer Domain-Objekte vom Typ 'ResourceType' und 'MeasurementDefinition' mit den eingegebene Informationen direkt im Agenten war sinnvoll, um die Methoden zur Synchronisation mit dem Server auf die Verwendung der entsprechenden Objekte anzupassen. Im ersten Schritt erfolgte dieser Schritt vor der Persitierung im Server, dort wurden die Objekte vom Typ 'ResourceType' und 'MeasurementDefinition' aus String-Informationen erstellt. Diese Variante war für den zweiten Schritt nicht mehr ausreichend, da spätestens im letzten Migrationschritt, der Anpassung der Plug-In-Abläufe, die Domain-Objekte direkt vom Plug-In an den Agent geliefert werden sollten. Deshalb war eine Anpassung auf diese Anforderung bereits in diesem Schritt sinnvoll, um die Grundlagen für den letzen Schritt zu schaffen und die Übertragung der Objekte vom Plug-In über den Agent zum Server vorzubereiten. Der Ablauf beim Anlegen der Objekte der Klassen 'ResourceType' und 'MeasurementDefinition' wurde bereits in Abschnitt 4.4.1 beschrieben und soll hier nicht nochmal erläutert werden. Zur Übermittlung der Objektdaten an den Server mussten innerhalb der Methode 'createNewResourceType()' die notwendigen Mechanismen implementiert werden, die die Kommunikation mit dem Server ermöglichen sollten. Zunächst wird über das Attribut 'configuration' der Klasse 'PluginContainerConfiguration' eine Instanz der Klasse 'DiscoveryServerServiceImpl' zurückgeliefert, die das Interface 'DiscoveryServerService' implementiert. Die Instanz dieser Klasse abstrahiert den laufenden Server, über geeignete Methoden kann man an die, innerhalb der Servers implementierten, Bean-Klassen gelangen und deren Methoden verwenden.

Listing 10: Beispielcode

```
DiscoveryServerService serverService = configuration.
    getServerServices().getDiscoveryServerService();
serverService.addNewResourceType(resourceTypes);
```

Im Interface musste die Methode 'addNewResourceType()' deklariert werden, die in der zugehörigen Klasse definiert wurde. Listing 11 zeigt die Methodendeklaration innerhalb des Interface.

Listing 11: Beispielcode

```
void addNewResourceType(Set<ResourceType> resourceTypes);
```

Die Methode wurde in der Klasse 'DiscoveryServerServiceImpl' definiert und hat die Aufgabe, über einen Aufruf der Methode 'getResourceMetadatemanager' der Klasse 'LookupUtil' eine Instanz der, im Applikationserver laufenden, Bean 'ResourceMetadataManagerBean' zu liefern. Von dieser Bean-Klasse muss dann die Methode 'addNewResourceType' aufgerufen werden, die die Persistierung in der Datenbank durchführt wie in Abschnitt 4.4.1 beschrieben. Listing 12 zeigt die Definition der Methode 'addNewResourceType()' innerhalb der Klasse 'DiscoveryServerServiceImpl'.

Listing 12: Beispielcode

```
1 public void addNewResourceType(Set<ResourceType> resourceTypes) {
2         LookupUtil.getResourceMetadataManager().
              addNewResourceType(resourceTypes);
3 }
```

Abbildung 25 veranschaulicht nochmal die Abläufe beim Hinzufügen neuer Metadaten über´die Kommandozeile des Agenten. Nach der Eingabe des Befehls 'addType' auf der Konsole wird die Methode 'excute()' der Klasse 'AddResourceTypePromptCommand' aufgerufen. Innerhalb der Methode 'execute()' wird durch den Aufruf der Methode 'getInstance()' eine Instanz der Klasse 'PluginManager' erzeugt, von dieser Instanz wird die Methode 'getInventoryManager()' aufgerufen, die eine Instanz des InventoryManagers liefert. Dann wird die Methode 'createNewResourceType' des InventoryManagers aufgerufen, in dieser Methode wird über den Aufruf der Methode 'getDiscoveryServerService()' eine Instanz der Klasse 'DiscoveryServerServiceImpl' erstellt, die den Server repräsentiert. Über den innerhalb der Methode 'addNewResourceType()' implementierten Aufruf der Methode 'getResourceMetadataManager()' der Klasse 'LookupUtil' wird dann die Instanz der Klasse 'ResourceMetadataManagerBean' geliefert, der Aufruf der Methode 'addResourceType()' führt die abschließende Persistierung durch.

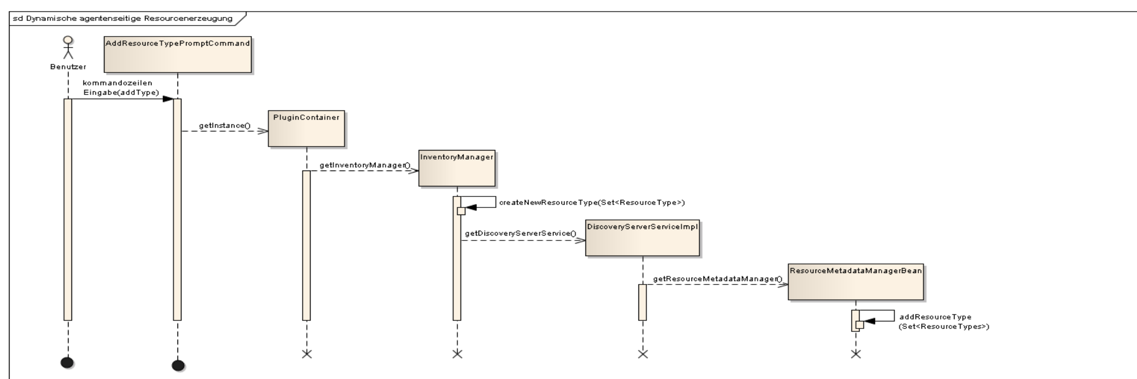


Abbildung 25: Abläufe auf Server und Agent

4.4.3 Dynamische Generierung neuer Resourcetypen im Plug-In

Als letzer Schritt zur Fertigstellung eines ersten Prototypen für die Migration zur dynamischen Bereitstellung von Metadaten mussten noch die notwendigen Anpassugen im Quellcode des Plug-Ins vorgenommen werden. Da es sich um einen Prototyp handelt, sollte der neue Ressourcentyp, und seine Metriken hardcodiert, vom laufenden Plug-In angelegt werden und, durch ensprechende Mechanismen, an den Agent, und von dort an den Server, weiter gegeben werden. Die Methode für das Anlegen neuer ResourceTypes sollte in einem eigens dafür erstellten Interface deklariert werden und von der ResourceComponent des Nagios Plug-In implementiert werden. Das neue Interface wurde mit dem Namen 'ChildResourceTypeDiscoveryFacet' angelegt und darin die Methode 'discoverChildResourceTypes()' deklariert. Listing 13 zeigt das Interface mit der Deklaration der Methode.

Listing 13: Beispielcode

```
1 public interface ChildResourceTypeDiscoveryFacet {
2     Set<ResourceType> discoverChildResourceTypes();
3 }
```

Die Klasse 'NagiosMonitorComponent' des Nagios Plug-In implementiert dieses Interface und definiert die Methode. Innerhalb der Methode wird ein neues Objekt vom Typ 'ResourceType' angelegt und ein weiteres Objekt vom Typ 'MeasurementDefinition', welches dem Ressourcentyp zugeordnet wird. Anschließend wird der neue Ressourcentyp in ein HashSet geschrieben und zurück geliefert. Der Einfachheit halber wird nur ein ResourceType in das HashSet geschrieben, im Praxiseinsatz sind aber mehrere ResourceTypes möglich, daher wird eine entsprechend flexible Datenstruktur verwendet. Listing 14 zeigt die Definition der Methode 'discoverChildResourceTypes()' innerhalb der Klasse 'NagiosMonitorComponent'.

Listing 14: Beispielcode

```
1  public Set<ResourceType> discoverChildResourceTypes() {
2              ResourceType parentType = this.context.getResourceType
                   ();
3              ResourceType resourceType = new ResourceType("
                   NewNagiosChild", parentType.getPlugin(),
4                   ResourceCategory.SERVICE, parentType);
5
6              MeasurementDefinition measurementDef = new
                   MeasurementDefinition(resourceType, resourceType.
                   getName()
7                   + "Metric");
8              resourceType.addMetricDefinition(measurementDef);
```

```
9
10          Set<ResourceType> resourceTypes = new HashSet<
                ResourceType>();
11          resourceTypes.add(resourceType);
12
13          return resourceTypes;
14  }
```

Um die Methode im laufenden Systembetrieb zu nutzen, und somit zur Laufzeit des Plug-Ins neue Ressourcentypen zu erstellen, waren weitere Anpassugen des System notwendig. Die Klasse 'InventoryManager' ruft bei der Inbetriebnahme des Agenten die Methode 'initialize()' auf, in der verschiendene Threads gestartet werden. Diese Threads führen die Scan-Operationen für die verschiedenen Ressourcenkategorien durch, also Platform-, Server- und Service-Scans, sie werden von einem Thread-Pool-Objekt verwaltet, das den einzelnen Threads die Systemressourcen zuordnet, die sie benötigen. Es sollte eine eigene Thread-Klasse implementiert werden, die ebenfalls beim Start des Agenten gestartet werden soll, und die Suche nach Child-ResourceTypes durchführt. Dazu wurde die Klasse 'ChildResourceTypeDiscovery-Runner' implementiert, die die Interface 'Callable' und 'Runable' implementiert und daher als Thread-Klasse verwendet werden kann. Die Klasse musste die Methoden 'call()' und 'run()' implementieren. Die Methode 'call()' führt die Hauptaufgaben durch und wird innerhalb der Methode 'run()' aufgerufen. Die Methode 'call()' hat verschiedene Aufgaben zu erfüllen. Zunächst muss sie eine Instanz des InventoryManagers ermitteln, dies geschieht über den Plug-In-Container auf die gleiche Art und Weise wie bereits mehrfach beschrieben wurde. Listing 15 zeigt die entsprechende Code-Zeile.

Listing 15: Beispielcode

```
1  InventoryManager im = PluginContainer.getInstance().
       getInventoryManager();
```

Anschließend muss über einen entsprechenden Methodenaufruf die Plattform ermittelt werden, auf der der InventoryManager läuft, dies geschieht durch den Aufruf der Methode 'getPlatform()' der Klasse 'InventoryManager'. Über die zurückgelieferte Plattform-Ressource kann man dann mit der Methode 'getChildResources()' sämtliche Kinder der Plattform abfragen. Listing 16 zeigt den Aufruf der Methoden 'getPlatform()' und 'getChildResources()'.

Listing 16: Beispielcode

```
1  Resource platform = im.getPlatform();
2  Set<Resource> children = platform.getChildResources();
```

Anschließend muss für jeden der gefundenen Kind-Ressourcentypen der Platform geprüft werden, ob er der ResourcenCategory 'Server' angehört. Dies ist deshalb notwendig, da das Nagios Plug-In dieser Ressourcenkategorie angehört, die Überpüfung erspart, dass alle Resourcen den weiteren Prüfungen unterzogen werden, obwohl sie gar nicht die Grundbedingung erfüllen. Wenn der Kind-Ressourcentyp der Kategorie 'Server' angehört, wird anschließend versucht, eine Instanz der Component-Klasse des Plug-Ins zurückzuliefern, welches das Interface 'ChildResourceTypeDiscoveryFacet' implementiert. Da dies ist nur beim Nagios Plug-In der Fall ist, wird so das Plug-In gefunden, ohne dass, wie in Abschnitt 4.4.1 beschrieben, hardcodiert nach dem Name des Plug-Ins gesucht werden musste. Ist dies nicht der Fall, so wird einen Exception geworfen und der nächste Typ wird überprüft. Listing **??** zeigt den die entsprechenden Codezeilen.

Listing 17: Ermitteln der Kind-Ressourcentypen

```
1 if (child.getResourceType().getCategory() == ResourceCategory.
    SERVER) {
2         ChildResourceTypeDiscoveryFacet discoveryComponent =
              ComponentUtil.getComponent(child.getId(),
              ChildResourceTypeDiscoveryFacet.class,
              FacetLockType.READ, 30 * 1000, true, true);
3         resourceTypes = discoverChildResourceTypes(
              discoveryComponent);
4 }
```

Wird, wie beim Nagios Plug-In, das Interface implementiert, kann die Methode 'discoverChildResourceTypes()' aufgerufen werden und die soeben gelieferte Instanz der Component-Klasse an die Methode übergeben werden. Im Falle des Nagios Plug-In ist dies eine Instanz der Klasse 'NagiosMonitorComponent'. Intern wird dann die Methode 'discoverChildResourceTypes()' aufgerufen, die von jeder Klasse implementiert wird, die das Interface 'ChildResourceTypeDiscoveryFacet' implementiert. Da dies in diesem Fall nur bei der Klasse 'NagiosMonitorComponent' der Fall ist, wird die in dieser Klasse definierte 'discoverChildResourceTypes()'-Methode aufgerufen und liefert HashSet mit dem neuen Ressourcentyp zurück. Listing 18 zeigt den Ablauf innerhalb der Methode 'discoverChildResourceTypes()' der Klasse 'ChildResourceTypeDiscoveryRunner'.

Listing 18: Beispielcode

```
1 private Set<ResourceType> discoverChildResourceTypes(
    ChildResourceTypeDiscoveryFacet discoveryComponent) {
2     Set<ResourceType> resourceTypes = null;
3     try {
```

```
4         long start = System.currentTimeMillis();
5         resourceTypes = discoveryComponent.
              discoverChildResourceTypes();
6         long duration = (System.currentTimeMillis() - start);
7
8         if (duration > 2000) {
9             log.info("[PERF] Discovery of childResourceTypes for
                  [" + discoveryComponent + "] took [" + duration
10                + "ms]");
11        }
12    } catch (Throwable t) {
13
14        log.warn("Failure to discover childResourceType data -
              cause: " + ThrowableUtil.getAllMessages(t));
15    }
16
17    return resourceTypes;
18 }
```

Wenn die neu hinzu zu fügenden Ressourcentypen zurückgeliefert wurden, müssen alle bestehenden Child-ResourceTypes des Nagios Plug-Ins ermittelt werden. Dann werden alle bestehenden Kind-Ressourcentypen mit allen neuen Kind-Ressourcentypen verglichen, es wird also geprüft ob für das aktuelle Plug-In bereits ein Typ mit gleichem Namen existiert. Ist dies für einen Typ der Fall, so wird er nicht weiter berücksichtigt, ansonsten wird der neue Typ in ein HashSet hinzugefügt. Abschließend wird das HashSet mit den neu hinzuzufügenden Typen an die Methode 'createNewResourceType()' des InventoryManagers übergeben und wird dort nach dem gleichen Ablauf wie in 4.4.2 weiter verabeitet. Listing 19 zeigt die Überprüfung der ResourceTypes und die Übergabe des HashSet an den InventoryManager. Damit ist der Weg des Ressourcentyps vom Plug-In über den Agent zum Server abgeschlossen, die notwendigen Anpassugen am System wurden vorgenommen und ein erster Prototyp realisiert.

Listing 19: Beispielcode

```
1
2 Set<ResourceType> currentChildTypes = container.getResource().
      getResourceType().getChildResourceTypes();
3
4 for (ResourceType alreadyExistingType : currentChildTypes) {
5         if (newTypetoAdd.getName().equals(alreadyExistingType.
              getName())
```

```
6                       && newTypetoAdd.getPlugin().equals(
                            alreadyExistingType.getPlugin())) {
7                   childAllreadyExists = true;
8                   }
9           }
10
11          if (!childAllreadyExists) {
12              newTypesToAdd.add(newTypetoAdd);
13          }
14
15   im.createNewResourceType(newTypesToAdd);
16 }
```

Abbildung 26 verdeutlicht nochmal den schematischen Ablauf innerhalb der Methode 'call()' der Klasse 'ChildResourceTypeDiscoveryRunner'. Der Thread wird vom InventoryManager gestartet, ermittelt über den InventoryManager die Platform und die ChildResourceTypes und führt für die ChildResourceTypes alle im Text beschriebenen Prüfungen durch. Wenn das Kind-Objekt alle Tests besteht, ist es ein Kind des Nagios Plug-In und kann die in der Plug-In Component-Klasse implementierte Methode 'discoverChildResourceTypes()' aufrufen, die die neuen Ressourcentypen liefert. Existieren die Typen noch nicht, wird die Methode 'createNewResourceTypes()' des InventoryManager aufgerufen, und die neuen Typen werden an diese Methode übergeben. Anschließend entspricht der Ablauf dem in Abbildung 25 beschriebenen Szenario.

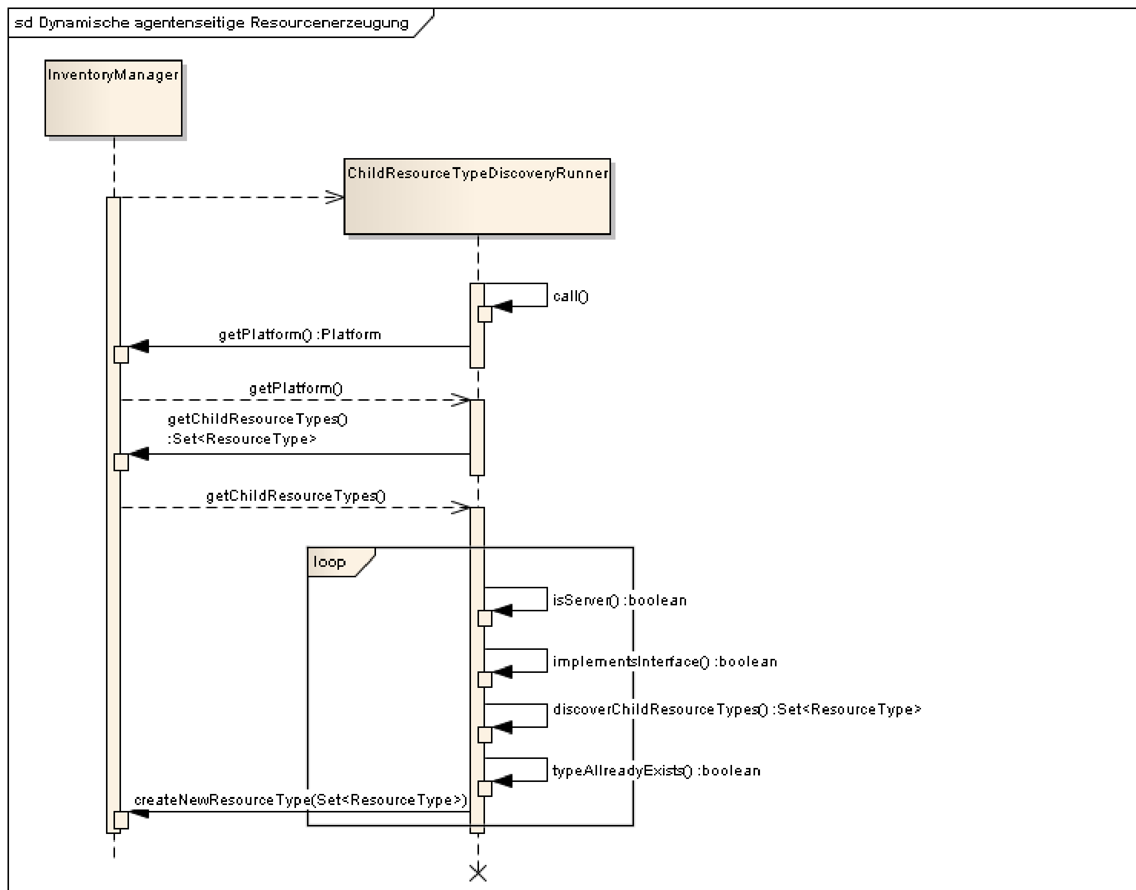


Abbildung 26: Ablauf der Threadklasse

5 Resumee

5.1 Zusammenfassung

Die Arbeit hat mehrere Ergebnisse hervor gebracht, die im Folgenden abschließend betrachtet und bewertet werden sollen. Die Arbeit setzte sich aus drei Teilaufgaben zusammen: der Implementierung eines Plug-Ins zur Überwachung des Nagios Systems unter Verwendung statischer Metadatenbereitstellung, dem Entwurf eines Konzepts zur Migration zu dynamischer Metadatenbereitstellung, und der Entwicklung eines dynamischen Prototypen durch Umsetzung des Konzepts. Im ersten Schritt sollte die Entwicklung des Plug-Ins auf der Basis der Verwendung statischer Metadatenbereitstellung realisiert werden. Die Metadaten wurden aus einer XML-Datei, dem Plug-In-Deskriptor, durch Parsing herausgelesen und in entsprechenden Datenobjekten des RHQ Systems, den Domain-Objekten, abgebildet. Die Metadaten eines Plug-In-Deskriptors beschreiben die vom Plug-In zu überwachenden Ressourcentypen und definieren zugehörige Metriken. Änderungen am XML-Deskriptor, z.B. das Hinzufügen eines weiteren Ressourcentyps oder einer weiteren Metrik, bedingen ein Re-Build und Re-Deployment des Plug-In. Die Entwicklung des statischen Plug-Ins war einer der drei Kernaspekte der Arbeit und konnte vollständig umgesetzt werden. Die Realisierung beinhaltete die Validierung vorhandener Schnittstellen zum Nagios System, die Entwicklung eines Datenmodels zur Abbildung des Nagios Systems, und die Entwicklung eines Nagios Plug-Ins unter Verwendung der AMPS Plug-In Bibliothek. Dabei sollten die Schnittstelle zu Nagios und das entwickelte Datenmodel in das Plug-In integriert werden. Alle Anforderungen zur Realiserung dieses ersten Teils der Arbeit konnten vollständig umgesetzt werden, dazu waren keinerlei Anpassugen am System notwendig, da die bisherige Funktionsweise des Systems auf statischer Bereitstellung von Metadaten basiert. Durch das entwickelte Plug-In ist es möglich, ein laufendes Nagios System zu beobachten und in der RHQ Systemmonitoring-Suite darzustellen. Das Plug-In ermittelt Messdaten vom Nagios System, persitiert diese in der Systemdatenbank und zeigt die Daten über die grafische Oberfläche des Servers an. Das zweite zentrale Thema der Arbeit war die Entwicklung eines Konzepts, auf dessen Basis die Migration von statischer zu dynamischer Metadatenbereistellung bei Monitoring-Systemen durchgeführt werden konnte. Das Konzept sollte möglichst weit von der konkreten Problematik des Nagios Systems abtrahiert werden, es sollte vielmehr einen allgemeinen Leitfaden für die Lösung ähnlicher Probleme bieten. Diese Anforderung wurde ebenfalls umgesetzt. Es erfolgte dabei eine Beschreibung der allgemeinen Anforderungen an ein Monitoring-System, die sowohl bei statischer als auch bei dynamischer Bereitstellung von Metadaten erfüllt sein müssen. Darüber hinaus wurden die Anforderungen, die für die Realiserung der Mi-

gration zu dynamischer Metadatenbereitstellung notwendig sind, gesondert betrachtet. Für alle definierten Anforderungen wurden dabei Lösungsvorschläge mitgeliefert, ohne auf RHQ-spezifische Realisierungsdetails einzugehen. Dadurch ist das Migrationskonzept auch auf vergleichbare Probleme anderer Systeme anwendbar. Als dritte zentrale Aufgabe sollte ein Prototyp eines Plug-Ins entwickelt werden, das unter Verwendung dynamischer Metadatenbereitstellung arbeitet. Dabei sollte das System so angepasst werden, dass das Hinzufügen neuer Resourcetypen nicht mehr durch Erweiterung des Deskriptors, sondern direkt aus dem Plug-in möglich ist. Dieses Ziel wurde ebenfalls erreicht, allerdings ist nur ein erster Prototyp realisiert worden, der noch nicht für den produktiven Einsatz geeignet ist, die Migration wurde also skizziert, aber nicht vollständig abgeschlossen. Das Servermodul wurde um geeignete Methoden erweitert, die die Persitierung neuer ResourceTypes in der Datenbank durchführen, diese Methoden wurden mit einer einfachen Benutzerschnittstelle gekoppelt, über die die Eingabe neuer Metadaten erfolgen kann. Das Agentenmodul wurde an mehreren Stellen angepasst, so dass, über die Eingabe eines Kommandozeilenbefehls im Agent, neue Ressourcentypen zur Laufzeit des Systems angelegt und an das Server Modul übertragen werden können, wo dann die Persitierung erfolgt. Darüber hinaus wurden weitere Anpassugen innerhalb des Agentenmoduls gemacht, so dass alle laufenden Plug-Ins ständig darauf geprüft werden, ob sie über Funktionen zum dynamischen Hinzufügen neuer ResourceTypes verfügen. Wird ein solches Plug-In gefunden, so werden die entsprechende Funktionen aufgerufen und die neuen ResourceTypes aus dem Plug-In an den Agenten übermittelt. Dort werden die neuen Ressourcentypen dann in das Inventar des Agenten geschrieben und an den Server übertragen. Die Component-Klasse des Nagios Plug-In wurde um eine Methode erweitert, die neue Ressourcentypen generiert. Dazu wurde ein eigenes Interface geschrieben, das von der Component-Klasse implementiert wird, der Agent kann das Plug-In darauf prüfen, ob es dieses Interface implementiert. Ist dies der Fall, wird die neue Methode aufgerufen, die die neuen ResourceTypes vom Plug-In an den Agent liefert. Damit ist ein erster Durchstich realisiert worden, der dynamisch zur Laufzeit des Plug-Ins neue Resourcetypes anlegt, sie im System bekannt macht und abschließend persistiert.

5.2 Ausblick

Obwohl alle in der Aufgabenstellung festgelegten Ziele erreicht wurden, gibt es einige Dinge, die in Zukunft angegangen werden müssen, um die dynamische Funktionalität des Systems weiter zu verbessern, und es für den Einsatz in Produktivumgebungen zu rüsten. Die Migration zur Verwendung dynamischer Metadaten ist noch nicht vollständig abgeschlossen. Zwar wurden die Architekturkomponenten Server, Agent und Plug-In so angepasst, dass das dynamische Hinzufügen von neuen Resourcen möglich ist, aber noch werden keine echten Messdaten vom Nagios System ermittelt. Die Schnittstelle zu MK_Livestatus, die für das im ersten Schritt implementierte statische Plug-In implementiert wurde, muss noch an das dynamische Plug-In angebunden werden, um echte Messdaten zu liefern, so wie es beim statischen Plug-In bereits funktioniert. Dies wurde aus Zeitmangel nicht mehr realisiert. Desweiteren ist es sinnvoll, über die Nutzung einer, auf MK_Livestatus basierenden, WebService Schnittstelle mit dem Namen LivestatusSlave nachzudenken. Bisher wird Xinetd in Kombination mit einem Unix Socket genutzt, um mit Livestatus zu kommunizieren. Xinetd kann über TCP Sockets angesprochen werden, die in den meisten Programmiersprachen einfach zu implementieren sind. Es wäre dennoch von Vorteil, den Livestatus Socket beispielsweise über HTTP ansprechen zu können, so könnten Anfragen zum Beispiel über XMLHttpRequest realisiert werden. Zum Zeitpunkt der Realisierung der Schnittstelle war dieses Tool noch in Entwicklung, andernfalls hätte es eine Alternative zur jetzigen Lösung darstellen können. Als weiterer Punkt ist zu nennen, dass die Anpassugen, die am System gemacht wurden, nur an einem System mit einem Agenten und einem Server getestet wurden. Daher muss das System in Zukunft so angepasst werden, dass Änderungen über den Server an alle Agenten übertragen werden, ohne dass die Überwachung der Ressourcen durch das System unterbrochen wird. Es müssen also weitere Funktionen zur Systemsynchronisierung implementiert werden, die Daten über das gesamte, aus einem Server und mehreren Agenten bestehende, System verteilen können. Das Erstellen neuer ResourceTypen geschieht bereits zur Laufzeit des Plug-Ins, ohne dass die Funktion des Systems unterbrochen wird, aber die neuen Resourcentypen werden momentan noch hardcodiert im Quellcode angelegt. In Zukunft sollte eine Schnittstelle geschaffen werden, über die neue Typen und zugehörige Metriken angelegt werden können, so wie es im Prototyp bereits realisiert wurde, als die Anpassungen an Server und Agent gemacht wurden. Die Eingabe der Metadaten sollte über Server oder Agent erfolgen, die eingegebenen Daten sollten zum Plug-In übermittelt werden, dort sollten dann neue Instanzen erstellt werden und mit den, über die Schnittstelle zu MK_Livestatus ermittelten, Messdaten aus dem System gefüllt werden. Abschließend bleibt noch zu nennen, dass die im Zuge der Anpassungen des Servers implementierte JSP-Page

nicht mehr als Schnittstelle verwendet werden kann, da während des Projektverlaufes die gesamte GUI des RHQ Projektes von der Verwendung von JSP (JavaServerPages) nach GWT (GoogleWebToolkit) portiert wurde. Um die Portierung der im Projektkontext implementierten Server-Schnittstelle umzusetzen, reichte die Zeit leider nicht mehr aus.

6 Anhang

6.1 Quellenverzeichnis

DH1 http://www.dillinger.de/dh/unternehmen/daten/index.shtml.de
DH2 http://www.dillinger.de/dh/unternehmen/produktion/index.shtml.de
NA1 http://de.wikipedia.org/wiki/Nagios
NA2 http://nagios3docs-de.svn.sourceforge.net/viewvc/nagios3docs-de/trunk/pdf/nagios3c
RH1 http://www.pro-linux.de/cgi-bin/DBApp/check.cgi?ShowApp..16152.100
RH2 http://rhq-project.org/display/RHQ/Home
NEB http://nagios.larsmichelsen.com/nagios-event-broker/
ND1 http://nagios.larsmichelsen.com/ndoutils-nagios-data-out/0
ND2 http://nagios.sourceforge.net/docs/ndoutils/NDOUtils.pdf
MK1 http://nagios.larsmichelsen.com/mklivestatus-and-nagvis-making-the-ndo-needless/
MK2 http://mathias-kettner.de/checkmk_livestatus.html

Abbildungsverzeichnis

1 Aufbau des RHQ Systems 4
2 Übersicht über die RHQ Resourcenkategorien 5
3 Begriffsmodel 20
4 Schematischer Aufbau einer Systemüberwachungssuite 26
5 Generisches Datenmodel eines agentenbasierten Systems 30
6 Datenmodel des Drittsystems 33
7 Schematischer Ablauf des Datenflußes 34
8 Schematischer Ablauf der Persistierung 35
9 Erweitern eines statischen Plug-Ins 40
10 Erweitern eines dynamischen Plug-Ins 42
11 Funktionsweise von NDO Utils 50
12 Funktionsweise des Nagios Plug-In 51
13 Klassen des Moduls 'Data' 54
14 Klassen des Pakets 'ManagementInterface' 57
15 Klassen des Moduls 'Controller' 58
16 Klassen des Moduls 'Request' 59
17 Klassen des Moduls 'Network' 60
18 Klassen des Moduls 'Reply' 61
19 Klassen des Moduls 'Parser' 61
20 Klassen des Moduls 'Error' 62
21 Interaktion der Module des Nagios Plug-Ins 63
22 Verwendete Klassen und Architekturkomponenten 64
23 Einschalten des Remote Debuggings 65
24 Ablauf der Serverkommunikation 67
25 Abläufe auf Server und Agent 71
26 Ablauf der Threadklasse 77