INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Projektarbeit

Architektur eines komponenten-orientierten Autorisierungsframeworks und zeitliche Evaluierung verschiedener Implementierungsvarianten

Daniel Reichel

Aufgabensteller:	Prof. Dr. Rolf Hennicker
Betreuer:	Ludwig Adam
Abgabetermin:	09. Oktober 2009

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

München, den 09. Oktober 2009

. . . .

(Daniel Reichel)

Zusammenfassung

Autorisierungssysteme sind ein zentraler Bestandteil vieler kritischer Anwendungen. Hierbei kommen meist hochspezialisierte, für einen bestimmten Einsatzzweck erstellte Systeme zum Einsatz. Da damit aber auch die Wartbarkeit und Erweiterbarkeit eines solchen Systems nicht immer gegeben ist, wird aktuell im Rahmen eines IuK-Projekts, gefördert vom bayerischen Wirtschaftsministerium ([iuk09]), am Lehrstuhl PST der LMU München in Zusammenarbeit mit der petaFuel GmbH an dem komponenten-orientierten, regelbasierten Autorisierungsframework Ruleset Based Authorization Framework in Java (RAJA) ([raj09]) geforscht.
Im ersten Teil dieser Arbeit wird ein Überblick über den aktuellen Stand des Autorisierungsframeworks gegeben und auf Basis von [Ada08] ein Vorschlag für eine Spezialisierung dieses Frameworks für den Einsatz zur Autorisierung von Kreditkartentransaktionen vorgenommen.
Im Laufe der Projektarbeit wurde ein Managementinterface für den Betrieb des Frameworks im Allgemeinen und für den Einsatz zur Autorisierung von Kreditkartentransaktionen im Speziellen implementiert. Diese Managementfunktionen werden vorgestellt und ihre Integration in das Framework beschrieben.
Zum Abschluss dieser Arbeit wird ein Implementierungsvorschlag für die möglichst effiziente Bearbeitung von Autorisierungsnachrichten erstellt. Hierbei wird darauf eingegangen, wie sich unterschiedliche Kommunikationsmöglichkeiten der Komponenten auf die Effizienz des Systems auswirken.

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Einleitung / Motivation 1
1.2 Vorteile eines komponenten-orientierten Autorisierungssystems 2
1.3 Ziele dieser Arbeit 2

2 Komponenten-orientierte Modellierung des Systems **5**
2.1 Struktur des Autorisierungsframeworks 5
2.2 Externe Komponentenspezifikation 6
2.3 Verwendung des Frameworks für konkrete Autorisierungsszenarien 7
2.4 Erweiterung des Frameworks zur Autorisierung von Kreditkartentransaktionen 8

3 Framespezifikationen für den Ablauf einer Standard-Autorisierung **9**
3.1 Beschreibung einer Standardautorisierung 9
3.2 Zuordnung zu Komponenten 9
3.2.1 Schnittstelle für die Kommunikation mit dem Client 9
3.2.2 Koordination der Bearbeitung 10
3.2.3 Bearbeitung der Anfrage 10
3.3 Framespezifikation 10

4 Ausbau des bestehenden Implementierungsframeworks um Management Klassen **15**
4.1 Bestimmung benötigter Managementfunktionen 15
4.1.1 Allgemeine Managementfunktionen des Autorisierungsframeworks . . 15
4.1.2 Managementfunktionen zur Autorisierung von Kreditkartentransaktionen 16
4.2 Integration der Managementfunktionen in das bestehende Framework 17

5 Zeitliche Evaluierung verschiedener Implementierungsansätze **23**
5.1 Beschreibung des zur Evaluierung genutzten Testfalls 23
5.2 Vorstellung der möglichen Implementierungstechniken 24
5.3 Beschreibung der Implementierungsmöglichkeiten innerhalb des Frameworks . 27
5.4 Untersuchung der Implementierungsmöglichkeiten 29
5.5 Bewertung der Ergebnisse 29

6 Zusammenfassung **33**

Abbildungsverzeichnis **35**

Literaturverzeichnis **37**

1 Einleitung

In diesem Kapitel wird die Motivation für den Entwurf eines regelbasierten, komponentenorientierten Autorisierungsframeworks dargestellt. Es werden die Vorteile eines solchen Systems erläutert und auf die Ziele dieser Arbeit eingegangen.

1.1 Einleitung / Motivation

Autorisierungssysteme nehmen in vielen Anwendungsbereichen eine wichtige und zentrale Rolle ein. So haben sich mit der Zeit verschiedene Autorisierungssysteme entwickelt, die alle auf ihren jeweiligen Einsatz spezialisiert sind. Mit dieser Spezialisierung und neuen Anforderungen an diese Systeme ergeben sich aber Probleme: So werden diese Systeme mit den wachsenden Anforderungen immer komplexer und damit auch immer schwerer wartbar und erweiterbar.
Allgemein lassen sich in einem Autorisierungssystem zwei Rollen identifizieren: eine anfragende und eine autorisierende Partei. Die anfragende Partei stellt hierbei eine Autorisierungsanfrage an die autorisierende Partei und erwartet eine Entscheidung über diese Anfrage. Für die anfragende Partei ist nach [Ada08] vor allem wichtig, dass die autorisierende Partei die Entscheidung unabhängig, deterministisch, nachvollziehbar und verbindlich trifft. Daraus ergeben sich vier Eigenschaften für die autorisierenden Partei:

1. Unabhängigkeit
   Die Entscheidung darf nur auf Basis der vorhandenen Daten getroffen werden.

2. Determinismus
   Wenn zwei gleichartige Anfragen auf Grundlage der gleichen Daten bearbeitet werden, müssen auch die Ergebnisse für beide Anfragen gleich sein.

3. Nachvollziehbarkeit
   Die Entscheidung muss für die anfragende Partei nachvollziehbar sein.

4. Verbindlichkeit
   Eine getroffene Entscheidung ist endgültig und darf nachträglich nicht mehr verändert werden.

Ziel des aktuellen IuK-Projekts der LMU in Zusammenarbeit mit der petaFuel GmbH ist es nun, ein Autorisierungsframework zu entwickeln, das die vier genannten Eigenschaften erfüllt. Zudem soll dieses Framework flexibel und einfach in verschiedenen Szenarien einsetzbar sein.
Um diese Flexibilität zu erreichen, muss in dem Framework eine Trennung zwischen der Kommunikation mit der anfragenden Partei, der Koordination der Bearbeitung einer Anfrage und der Entscheidungsfindung stattfinden.
Während die Koordination der Bearbeitung einer Anfrage unabhängig von dem gewünschten

Einsatzszenario geschehen kann, muss zum Einen die Kommunikation mit der anfragenden Partei konkret auf die Einsatzumgebung angepasst werden und zum Anderen wird die Entscheidungsfindung durch die Integration eines Regelsystems weiter vereinfacht. Dieser Ansatz bietet die oben genannte Flexibilität bei der Anpassung des Systems an neue Anforderungen.

1.2 Vorteile eines komponenten-orientierten Autorisierungssystems

Wie in Kapitel 1.1 bereits angedeutet und in [Ada08] genauer beschrieben, lassen sich in einem Autorisierungssystem verschiedene funktionelle Bereiche finden:

1. Es gibt eine Schnittstelle, über die die anfragende Partei eine Autorisierung durch das System beauftragen kann.
2. Es gibt eine koordinierende Instanz, die innerhalb des Autorisierungssystems verschiedene Anfragen gemäß ihrer Art bearbeitet.
3. Es gibt eine oder mehrere Instanzen, die abhängig von der Art der Anfrage, eine Entscheidung für die Anfrage treffen.

Diese Struktur des Autorisierungssystems kann mit Hilfe eines komponenten-orientierten Entwurfs sehr gut dargestellt werden, deshalb fiel die Designentscheidung für die Implementierung des Frameworks innerhalb des Projekts auf eine komponenten-orientierte Architektur nach dem Java/A Komponentenmodell ([KJH+07], [HJK08a], [BHH+05], [HJK08b]).
Vorteil einer Architektur nach diesem Modell ist vor allem, dass einzelne Komponenten nur über definierte Schnittstellen, sogenannte Ports, miteinander kommunizieren und somit die Funktion der einzelnen Komponenten / Bereiche des Frameworks strikt von der Architektur des Gesamtsystems getrennt werden kann. Hierzu stellt jeder Port einer Komponente eine Reihe von Operationen bereit, über die er angesprochen werden kann (provided Interface). Im Gegenzug besitzt jeder Port aber auch Anforderungen an Operationen, so dass er selbst mit einer anderen Komponente kommunizieren kann (required Interface). Hierdurch wird die geforderte Wartbarkeit des Systems erreicht, denn einzelne funktionelle Bereiche können unabhängig und ohne Einfluss auf das Gesamtsystem verändert werden. Insbesondere wird aber durch die Modellierung und Spezifikation des System nach dem Java/A Komponentenmodell auch das Verhalten der Komponenten / Bereiche des Systems formal beschrieben. Damit wird es möglich, einzelne Systemeigenschaften formal und unabhängig zu verifizieren.

1.3 Ziele dieser Arbeit

Ziel dieser Arbeit ist es, dem Leser zunächst einen Überblick über die Architektur des Autorisierungsframeworks RAJA zu geben. Hierbei werden die vorhandenen Komponenten beschrieben und deren Portverhalten spezifiziert.
Des Weiteren wird das Frameworks für den konkreten Einsatz zur Autorisierung von Kreditkartentransaktionen erweitert. Es werden zusätzlich benötigte Komponenten und der Ablauf einer Autorisierung einer Standard-Kreditkartentransaktion dargestellt.
Im dritten Teil der Arbeit werden daraufhin verschiedene Managementfunktionen für das

gesamte Autorisierungsframework und für den konkreten Einsatzzweck zur Autorisierung von Kreditkartentransaktionen identifiziert und eine mögliche Integration dieser Funktionen in das bestehende System beschrieben.
Innerhalb des Implementierungsframeworks werden verschiedene Arten der Komponentenkommunikation unterstützt. So ist es möglich, dass diese Kommunikation synchron oder asynchron (nach [HJK08a]) und single-threaded oder multi-threaded geschehen kann. Im vierten Teil der Arbeit wird nun abschließend auf diese verschiedenen Arten der Komponentenkommunikation eingegangen, mit dem Ziel hier eine möglichst effiziente und schnelle Bearbeitung von Anfragen zu erreichen.

2 Komponenten-orientierte Modellierung des Systems

In diesem Kapitel wird zunächst ein Überblick über den aktuellen Stand des Autorisierungsframeworks RAJA geben. So werden die vorhandenen Komponenten innerhalb des Frameworks, deren Funktion und Beziehungen zueinander beschrieben. Zudem wird eine Erweiterung des Frameworks für den konkreten Einsatzzweck zur Autorisierung von Kreditkartentransaktionen vorgenommen. Anschließend werden die bereitgestellten Ports der einzelnen Komponenten näher beschrieben.

2.1 Struktur des Autorisierungsframeworks

Basis dieser Arbeit bildet das bestehende Autorisierungsframework RAJA ([raj09]) im aktuellen Stand. Deshalb wird zunächst ein Überblick über die in diesem Framework vorhandenen Komponenten und deren Funktion gegeben:

- **Komponente** `Acceptor`
  Die Komponente `Acceptor` stellt die Schnittstelle zur anfragenden Partei bereit. Es werden eingehende Anfragen entgegengenommen und Entscheidungen des Autorisierungssystems an die anfragende Partei zurück gemeldet.

- **Komponente** `CommunicationControlComponent`
  Die Komponente `CommunicationControlComponent` bildet die zentrale, koordinierende Komponente zwischen Anfrageeingang und Entscheidungsfindung. Dazu nimmt diese Komponente Anfragen von der Komponente `Acceptor` entgegen und reicht diese an die Entscheidungskomponente weiter. Zudem gibt sie die Antwort nach einer getroffenen Entscheidung von der Entscheidungskomponente wieder an die Komponente `Acceptor` zurück.

- **Komponente** `RulesetAuthComponent`
  Die Komponente `RulesetAuthComponent` bestimmt wie im Kapitel 1.1 beschrieben anhand eines festgelegten Regelsatzes und der Art der Anfrage, welche spezialisierten Einzelentscheidungskomponenten für die aktuelle Anfrage benötigt werden. Somit nimmt sie Anfragen von `CommunicationControlComponent` entgegen und koordiniert auf Basis des hinterlegten Regelsatzes die Entscheidungsfindung für die Anfrage. Anschließend reicht dieses Komponente die Antworten an `CommunicationControlComponent` zurück.

- **Komponenten vom Typ** `DecisionComponent`
  `RulesetAuthComponent` delegiert zur Entscheidungsfindung innerhalb des Regelsatzes

Anfragen an eine oder mehrere Entscheidungskomponenten, die alle die Verhaltensspezifikation der abstrakten Komponente `DecisionComponent`[1] implementieren. Diese Komponenten fällen eine spezialisierte Entscheidung über die aktuelle Anfrage und melden das Ergebnis jeweils an `RulesetAuthComponent` zurück.
Damit müssen diese Komponenten je nach Einsatzzweck des Autorisierungsframeworks speziell für das jeweilige Szenario spezifiziert und implementiert werden. An dieser Stelle befindet sich somit der benötigte Erweiterungspunkt des Frameworks, der es ermöglicht das System einfach auf neue Einsatzszenarien anzupassen.

2.2 Externe Komponentenspezifikation

Nachdem nun die Kernkomponenten des Autorisierungsframeworks und die Beziehungen der einzelnen Komponenten bekannt sind, lassen sich die verschiedenen Ports der Komponenten darstellen.
Es folgt nun eine Aufstellung für jede Komponente, welche Ports diese Komponente besitzt:

1. **Komponente** `Acceptor`
   a) Port: `IOFacade`
      Über diesen Port findet die Kommunikation mit der Komponente `CommunicationControlComponent` statt.
2. **Komponente** `CommunicationControlComponent`
   a) Port: `ClientConnector`
      Über diese Port-Beziehung findet die Kommunikation mit der Komponente `Acceptor` statt.
   b) Port: `AuthorizationDispatcher`
      Über diese Port-Beziehung findet die Kommunikation mit der Komponente `RulesetAuthComponent` statt.
3. **Komponente** `RulesetAuthComponent`
   a) Port: `AuthorizationProvider`
      Über diese Port-Beziehung findet die Kommunikation mit der Komponente `CommunicationControlComponent` statt.
   b) Port: `DecisionDispatcher`
      Über diese Port-Beziehung findet die Kommunikation mit Komponenten vom Typ `DecisionComponent` statt.
4. **Komponenten vom Typ** `DecisionComponent`
   a) Port: `SingleDecisionProvider`
      Über diese Port-Beziehung findet die Kommunikation mit der Komponente `RulesetAuthComponent` statt.

Da nun die Komponenten des Autorisierungsframeworks und deren Ports bekannt sind, kann auch in Abbildung 2.1 ein Gesamtüberblick für das System angegeben werden. Die Methoden der provided und required Interfaces werden in Kapitel 3 näher spezifiziert, auf die Angabe dieser Methoden wird deshalb hier verzichtet.

[1] Zur Definition des Begriffs der abstrakten Komponente wird auf [Ada09a] verwiesen.

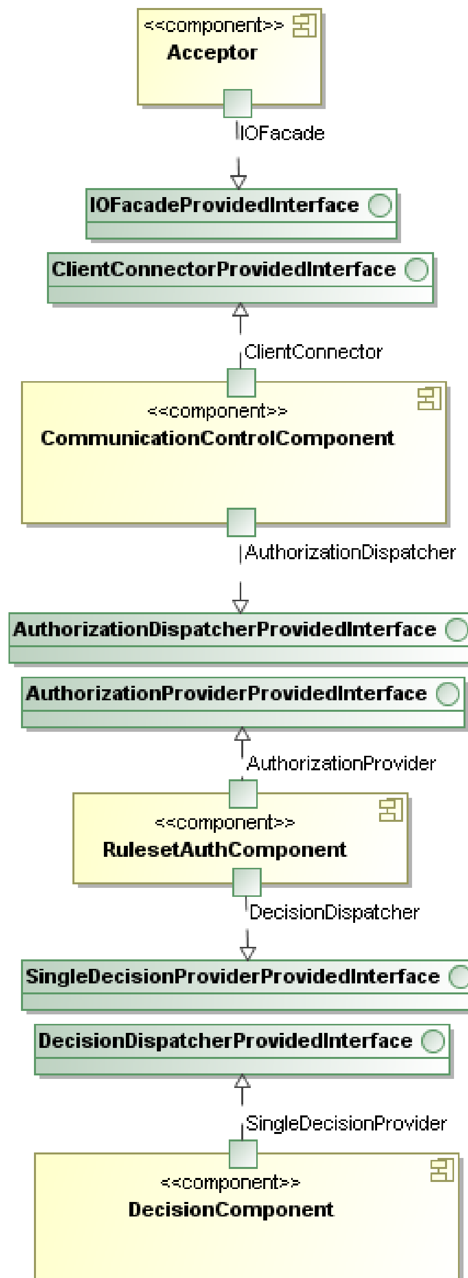

Abbildung 2.1: Kernkomponenten des Autorisierungsframeworks

2.3 Verwendung des Frameworks für konkrete Autorisierungsszenarien

Die beschriebenen Kernkomponenten bilden das Autorisierungsframework. Für den Einsatz des Frameworks in einem konkreten Szenario sind nun zwei Erweiterung nötig: Zum Einen muss ein Regelsatz angelegt werden, anhand dessen die Komponente `RulesetAuth` die Anfrage zur Entscheidungsfindung an spezialisierte Komponenten vom Typ `DecisionComponent`

delegiert (siehe Kapitel 2.1). Zum Anderen müssen diese Entscheidungskomponenten so implementiert werden, dass sie die vom Regelsatz benötigten Fakten berechnen und bereitstellen.

2.4 Erweiterung des Frameworks zur Autorisierung von Kreditkartentransaktionen

Der unter [Ada09b] veröffentlichte Regelsatz im DRL-Format beschreibt die notwendigen Entscheidungen für das Szenario einer Kreditkartenautorisierung. Darin lassen sich folgende Entscheidungskomponenten identifizieren:

- **Komponente** `BalanceCheckDecisionComponent`
  Diese Komponente überprüft, ob der Anfragebetrag eine Kreditkartentransaktion von dem aktuellen Verfügungsrahmen der betroffenen Kreditkarte abgedeckt wird.
- **Komponente** `CardValidDecisionComponent`
  Diese Komponente prüft, ob die Kreditkartentransaktion mit der betroffenen Kreditkarte zulässig ist (z.B. ob die Karte aktiv ist).
- **Komponente** `MagStripeDecisionComponent`
  Diese Komponente prüft die übermittelten Kartendaten auf ihre Gültigkeit.

Unabhängig von der eigentlichen Autorisierung von Anfragen sind zudem weitere Komponenten nötig, die keine Kreditkartentransaktionen im eigentlichen Sinne autorisieren. So gibt es administrative Nachrichten, die zur Verwaltung der Kommunikation und Synchronisation nötig sind, es gibt Netzwerknachrichten zur An- / Abmeldung des Autorisierungssystems am gesamten Netzwerk und es gibt spezielle Nachrichten, die vorhergehende Nachrichten widerrufen bzw. deren Effekt rückgängig machen[2].

- **Komponente** `AdministrativMessageComponent`
  Diese Komponente nimmt administrative Nachrichten des Netzwerkes entgegen und behandelt diese entsprechend.
- **Komponente** `NetworkManagementComponent`
  Diese Komponente nimmt Netzwerknachrichten entgegen und behandelt diese entsprechend.
- **Komponente** `ReversalComponent`
  Diese Komponente nimmt Nachrichten entgegen, die vorhergehende Nachrichten widerrufen, und behandelt diese entsprechend.

[2] Diese Komponenten sind aufgrund der MasterCard-Spezifikationen zur Autorisierung von Kreditkartentransaktionen notwendig.

3 Framespezifikationen für den Ablauf einer Standard-Autorisierung

In diesem Kapitel wird die Verhaltensspezifikation des Systems anhand des Ablaufes einer Standard-Autorisierung beschrieben. Zunächst wird ein Überblick über den Ablauf einer Autorisierung gegeben und eine mögliche Einteilung des Ablaufes in funktional zusammenhängende Bereiche erstellt. Daraufhin werden die in Kapitel 2 beschriebenen Komponenten den gefundenen funktionalen Bereichen zugeordnet und ein detaillierter Ablauf einer Autorisierung über die Komponenten beschrieben.

3.1 Beschreibung einer Standardautorisierung

Der Ablauf einer Autorisierung kann wie folgt beschrieben werden:
Ein Client stellt eine Anfrage an das Autorisierungssystem. Für diese Anfrage stellt das Autorisierungssystem eine spezifizierte Schnittstelle bereit, über welche die gesamte Kommunikation mit dem Client stattfindet. Nachdem die Anfrage entgegengenommen ist, wird untersucht, um welche Art von Anfrage es sich handelt und die Anfrage gemäß ihrer Art behandelt. Sobald die Entscheidung zu der Anfrage feststeht, wird diese Entscheidung wieder über die definierte Schnittstelle an den Client zurückgemeldet und die Bearbeitung ist abgeschlossen.

Somit lassen sich folgende funktionalen Bereiche innerhalb des Autorisierungssystems finden (Abbildung 3.1):

1. Eine Schnittstelle für die Kommunikation mit dem Client.
2. Ein Bereich zur Koordination, welcher über die Art und Bearbeitung der Anfrage entscheidet.
3. Ein Bereich zur Bearbeitung der Anfrage gemäß seiner Art.

3.2 Zuordnung zu Komponenten

Die in Kapitel 3.1 beschriebenen Bereiche einer Standardautorisierung lassen sich den in Kapitel 2.1 beschriebenen Komponenten zuordnen:

3.2.1 Schnittstelle für die Kommunikation mit dem Client

Zugeordnete Komponente: `Acceptor`
Die Komponente `Acceptor` nimmt Anfragen von Clients entgegen und reicht diese zur Bearbeitung weiter. Zudem meldet die Komponente `Acceptor` das Ergebnis der Bearbeitung wieder zurück an den Client.

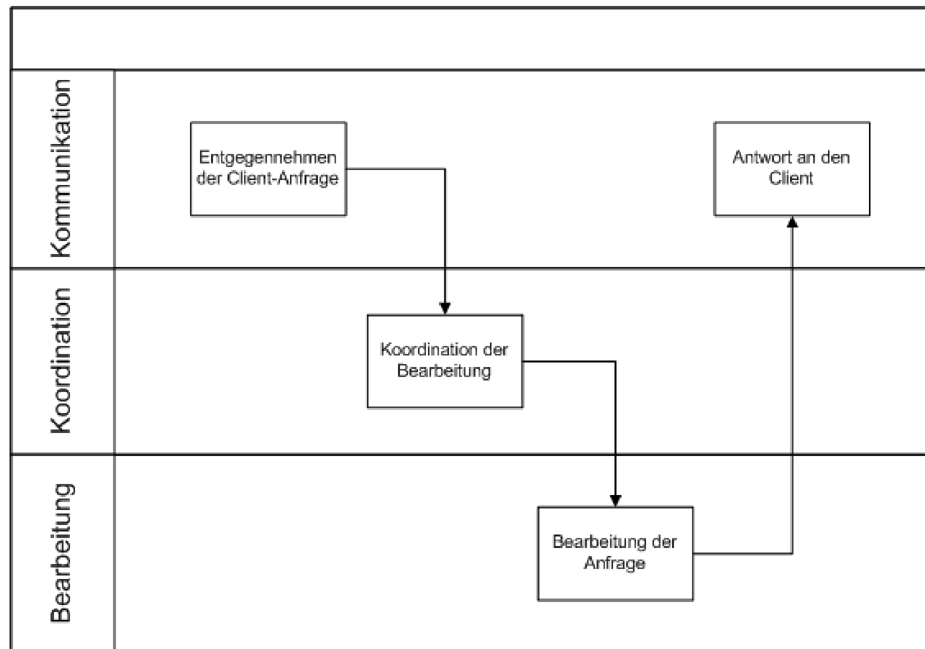

Abbildung 3.1: funktionale Bereiche des Autorisierungssystems

3.2.2 Koordination der Bearbeitung

Zugeordnete Komponente: `CommunicationControlComponent`
Die Komponente `CommunicationControlComponent` empfängt Anfragen von der Komponente `Acceptor` und entscheidet auf Grund der Art der Anfrage über dessen Bearbeitung. Hierzu reicht die Komponente die Anfrage an definierte Komponenten weiter und nimmt deren Entscheidung über die Anfrage wieder entgegen. Das Ergebnis der Bearbeitung der Anfrage wird anschließend wieder an die Komponente `Acceptor` zurückgegeben.

3.2.3 Bearbeitung der Anfrage

Zugeordnete Komponenten: `RulesetAuthComponent`, `DecisionComponent`
Da in diesem Bereich eine Entscheidung über die Anfrage getroffen werden muss und diese Entscheidung von der Art der Anfrage abhängig ist, sind hier mehrere Komponenten zu finden. Die Komponente `RulesetAuthComponent` nimmt zunächst Anfragen von der Komponente `CommunicationControlComponent` entgegen und delegiert diese Anfragen gemäß des hinterlegten Regelsatzes an mindestens eine Komponente vom Typ `DecisionComponent`. Nachdem die Entscheidung zu einer Anfrage feststeht wird das Ergebnis von der Komponente `RulesetAuthComponent` an die Komponente `CommunicationControlComponent` zurückgemeldet.

3.3 Framespezifikation

Der Ablauf einer Standardautorisierung über die Komponenten und lässt sich nun unabhängig von den speziellen Operationen der Komponenten zunächst wie in Abbildung 3.2

vereinfacht darstellen.

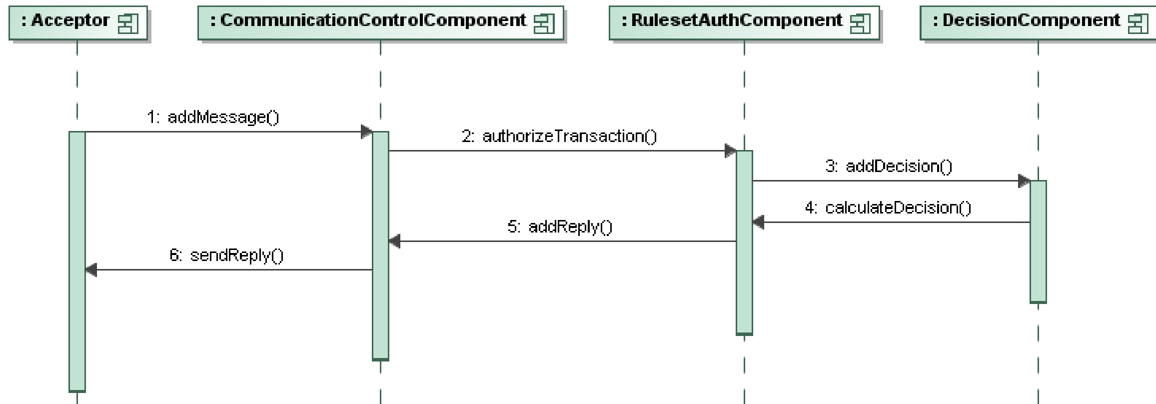

Abbildung 3.2: Sequenzdiagramm: Ablauf einer Autorisierung

Es zeigt sich bereits hier, dass die Kommunikation zwischen zwei Komponenten nach einem einfachen Request-Reply-Mechanismus stattfindet. Eine anfragende Komponente erwartet auch immer eine Antwort der angefragten Komponente. Somit kann für jede Kommunikationsbeziehung zwischen zwei Komponenten zunächst ein spezifisches Portprotokoll angegeben werden. Die Darstellung der Portprotokolle erfolgt hier mit Hilfe des LTSA-Tools ([MK09])

- Portprotokoll für die Kommunikation zwischen `IOFacade` und `ClientConnector`
  Eine Anfrage wird über die Methode `addMessage()` vom Port `IOFacade` an den Port `ClientConnector` übergeben. Nach der Bearbeitung meldet die Komponente `ClientConnector` über den Aufruf der Methode `sendReply()` das Ergebnis zurück. Das Portprotokoll ist in Abbildung 3.3 dargestellt.

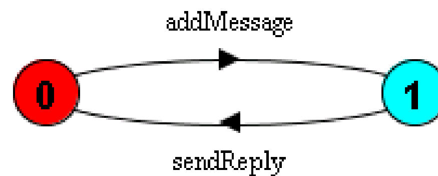

Abbildung 3.3: Portprotokoll: IOFacade - ClientConnector

- Portprotokoll für die Kommunikation zwischen `AuthorizationDispatcher` und `AuthorizationProvider`
  Durch den Aufruf von `authorizeTransaction()` wird die Anfrage vom Port `AuthorizationDispatcher` an den Port `AuthorizationProvider` übergeben. Das Ergebnis wird über die Methode `addReply()` zurückgemeldet. Das Portprotokoll ist in Abbildung 3.4 dargestellt.
- Portprotokoll für die Kommunikation zwischen `DecisionDispatcher` und `SingleDecisionProvider`

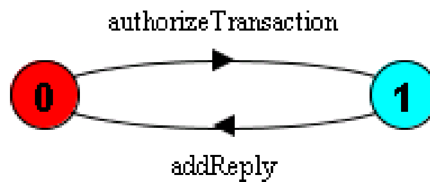

Abbildung 3.4: Portprotokoll: AuthorizationDispatcher - AuthorizationProvider

Hier wird die Anfrage durch den Aufruf von `addDecision()` vom Port `DecisionDispatcher` an den Port `SingleDecisionProvider` übergeben. Die Rückmeldung erfolgt über die Methode `calculateDecision()`. Das Portprotokoll ist in Abbildung 3.5 dargestellt.

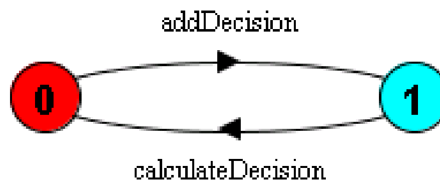

Abbildung 3.5: Portprotokoll: DecisionDispatcher - SingleDecisionProvider

Auf Basis dieser Portprotokolle lässt sich nun für jede Komponente eine spezifische Framespezifikation angeben. Hierbei werden die Protokolle der Ports einer Komponente kombiniert und somit der Ablauf der Anfragebearbeitung pro Komponente beschrieben[1].

- Framespezifikation `Acceptor`
  Die Komponente `Acceptor` reicht über ihren Port `IOFacade` mit Hilfe der Methode `addMessage()` eine Anfrage zur Bearbeitung weiter. Im Gegenzug erhält sie über diesen Port durch die Methode `sendReply()` die Antwort auf eine Anfrage. Die Framespezifikation für die Komponente `Acceptor` ist in Abbildung 3.6 dargestellt.

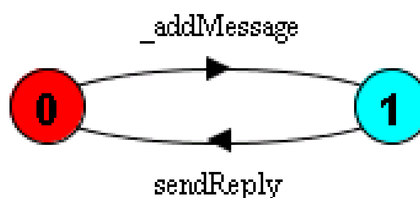

Abbildung 3.6: Framespezifikation: Acceptor

- Framespezifikation `CommunicationControlComponent`
  Die Komponente `CommunicationControlComponent` empfängt zunächst durch die Methode `addMessage()` von ihrem Port `ClientConnector` Anfragen zur Bearbeitung.

[1]Komponenten-interne Operationen werden hier nicht dargestellt.

Diese Anfragen werden über den Port `AuthorizationDispatcher` und die Methode `authorizeTransaction()` weitergereicht. Rückmeldungen erhält diese Komponente ebenfalls zunächst über den Port `AuthorizationDispatcher` über den Aufruf von `addReply()` und reicht diese Antworten dann wieder über den Port `ClientConnector` und die Methode `sendReply()` weiter. Die Framespezifikation für die Komponente `CommunicationControlComponent` ist in Abbildung 3.7 dargestellt.

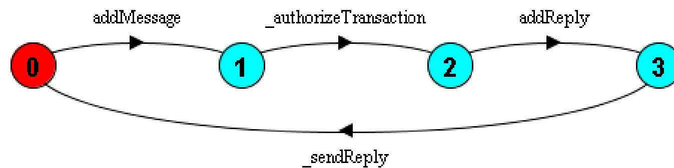

Abbildung 3.7: Framespezifikation: CommunicationControlComponent

- Framespezifikation `RulesetAuthComponent`
  Die Komponente `RulesetAuthComponent` empfängt über die Methode `authorizeTransaction()` an ihrem Port `AuthorizationProvider` Anfragen zur Bearbeitung. Diese Anfragen werden wiederum über den Port `DecisionDispatcher` und den Aufruf der Methode `addDecision()` weitergereicht. Rückmeldungen erhält diese Komponente ebenfalls zunächst über den Port `DecisionDispatcher` über die Methode `calculateDecision()` und reicht diese Antworten dann wieder über den Port `AuthorizationProvider` und die Methode `addReply()` weiter. Die Framespezifikation für die Komponente `RulesetAuthComponent` ist in Abbildung 3.8 für die Kommunikation mit einer einzelnen Entscheidungskomponente dargestellt.

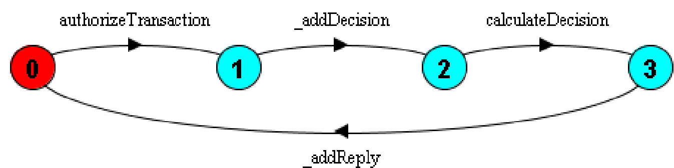

Abbildung 3.8: Framespezifikation: RulesetAuthComponent

- Framespezifikation `DecisionComponent`
  Die Komponente `DecisionComponent` erhält über ihren Port `SingleDecisionProvider` mit Hilfe der Methode `addDecision()` eine Anfrage zur Bearbeitung. Im Gegenzug meldet sie über diesen Port durch die Methode `calculateDecision()` die Antwort auf eine Anfrage zurück. Die Framespezifikation für die Komponente `DecisionComponent` ist in Abbildung 3.9 dargestellt.

Der Gesamtablauf einer Autorisierung und damit die Framespezifikation des Systems bei einer Autorisierung lässt sich nun wie in Abbildung 3.10 darstellen.

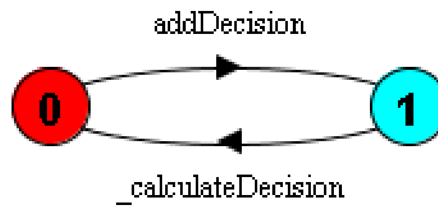

Abbildung 3.9: Framespezifikation: DecisionComponent

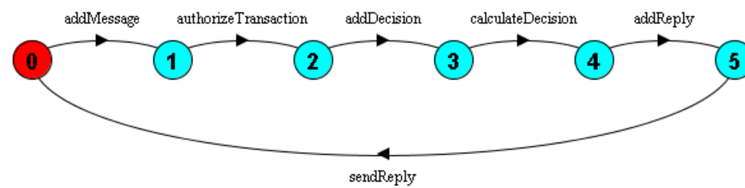

Abbildung 3.10: Framespezifikation für den gesamten Ablauf einer Autorisierung

4 Ausbau des bestehenden Implementierungsframeworks um Management Klassen

Für den Betrieb des Autorisierungssystems werden verschiedene Managementfunktionen benötigt. Neben Funktionen, die allgemein für das Autorisierungsframework nötig sind, gibt es auch hier wieder Funktionen, die speziell für den Einsatz des Systems zur Autorisierung von Kreditkartentransaktionen verfügbar sein müssen. Im Folgenden werden diese Funktionen nun kurz vorgestellt und die Integration dieser Funktionen in das bestehende Framework beschrieben.

4.1 Bestimmung benötigter Managementfunktionen

4.1.1 Allgemeine Managementfunktionen des Autorisierungsframeworks

Um die Wartbarkeit und Verfügbarkeit des Autorisierungssystems gewährleisten zu können, werden vor allem Managementfunktionen benötigt, die es ermöglicht einzelne Komponenten während des Betriebs zu manipulieren.

- **Aktualisierung von Komponenten während des Betriebs**
  Einzelne Komponenten sollen während des Betriebs ausgetauscht werden, zu bearbeitende Anfragen dürfen hiervon aber nicht beeinträchtigt werden. Nach [Ada09a] ist es hierzu nötig, dass zunächst die aktualisierte Komponente gestartet wird und parallel zu der zu ersetzenden Komponente verfügbar ist. Daraufhin werden die Port-Verbindungen der alten Komponente aufgelöst und entsprechende Verbindungen an der neuen Komponente erstellt. Sobald die alte Komponente nicht mehr mit anderen Komponenten in Verbindung steht, kann diese Komponente abgeschaltet werden und der Austausch der Komponente ist erfolgt (Abbildung 4.1).

- **Verschieben von Komponenten zwischen verschiedenen Servern**
  In einem verteilten System soll es möglich sein Komponenten zwischen verschiedenen Systemen zu verschieben. In gleichem Zusammenhang soll es möglich sein das gesamte Autorisierungssystem zwischen verschiedenen Serversystemen zu verschieben, um so z.B. Wartungsarbeiten an einem Serversystem durchführen zu können.
  Diese Funktion ist ähnlich dem Aktualisieren von Komponenten während des Betriebs. Zunächst wird die neue Komponente auf dem neuen System bereitgestellt. Anschließend werden die Port-Verbindungen der zu ersetzenden Komponente getrennt und auf die neue Komponente übertragen. Daraufhin kann die alten Komponente beendet werden. Analog kann dieser Vorgang auf ein komplettes System angewandt werden.

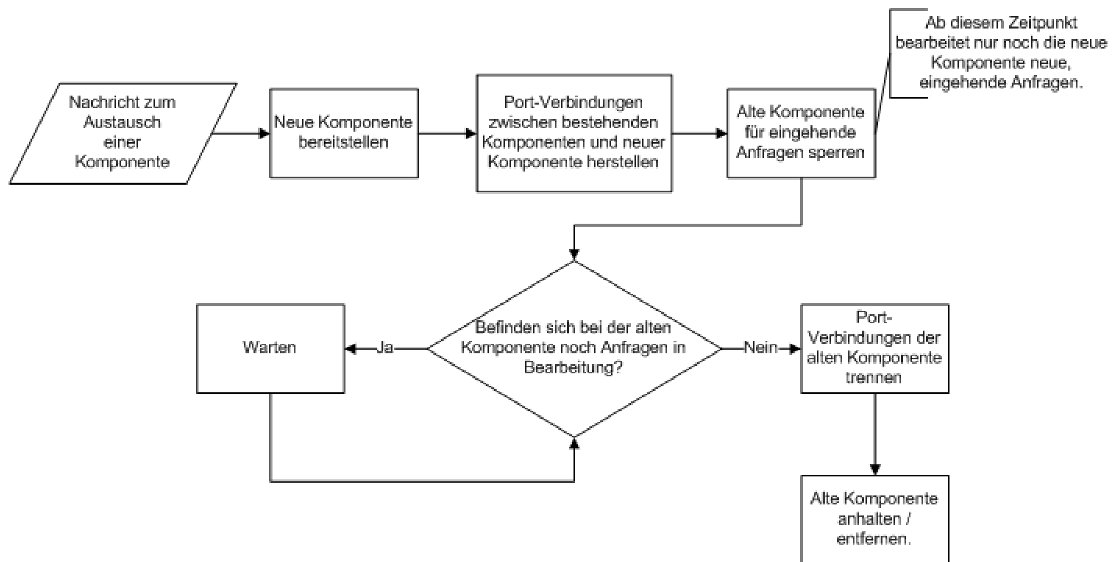

Abbildung 4.1: Flussdiagramm: Austausch einer Komponente

- **Anpassung von Autorisierungsregeln der Komponente `RulesetAuthComponent` während des Betriebs**
  Die Komponente `RulesetAuthComponent` greift auf definierte Regelsätze zu, um eine Entscheidung zu einer Anfrage zu treffen. Diese Regelsätze können dynamisch geändert werden, um Komponenten vom Typ `DecisionComponent` hinzuzufügen oder zu entfernen. So können Parameter für die Entscheidung über eine Anfrage während der Laufzeit verändert werden. Diese Regelsätze können mit dieser Managementfunktion bearbeitet werden.

4.1.2 Managementfunktionen zur Autorisierung von Kreditkartentransaktionen

Beim Einsatz des System zur Autorisierung von Kreditkartentransaktionen werden spezielle Managementfunktionen benötigt, die das Verhalten des Systems vor allem in Bezug zu der anfragenden Partei beeinflussen.

- **Starten des Systems**
  Nach dem Laden der Software befindet sich das Autorisierungssystem ein einem Zustand, in dem es noch keine Client-Anfragen entgegennimmt. Erst durch das explizite Starten des Systems wird es in diesen Zustand versetzt. Hierbei stellt das System eine Verbindung zu einem übergeordneten System her, es führt also eine „Anmeldung“ an einem anderen System durch.

- **Stoppen des Systems**
  Analog zum Starten des Systems muss es möglich sein, das System in einen Zustand zu versetzen, in dem keine weiteren Client-Anfragen mehr entgegengenommen werden. Wichtig ist hierbei, dass Anfragen, die sich gerade in Bearbeitung befinden, noch fertig bearbeitet und beantwortet werden. Beim Stoppen des Systems darf also eine eventuell bestehende Verbindung zu einem Client nicht getrennt werden, sondern es muss zunächst dafür gesorgt werden, dass keine neuen Anfragen entgegengenommen

werden. Erst wenn sich keine Anfragen mehr in Bearbeitung befinden, können alle Verbindungen getrennt werden und eine „Abmeldung“ von einem anderen System erfolgen (Abbildung 4.2).

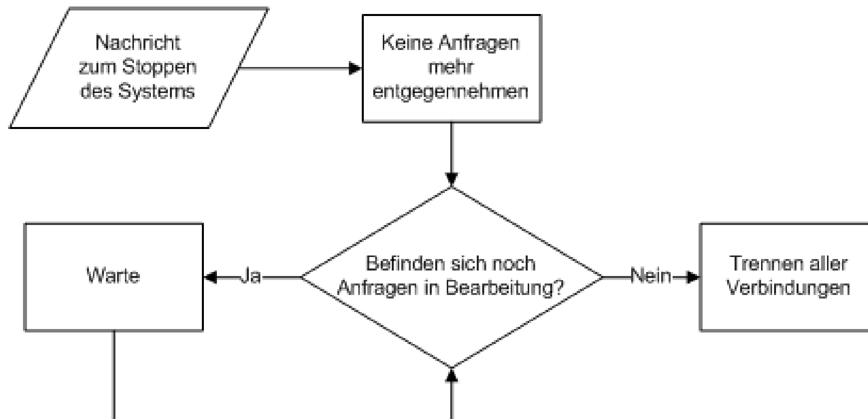

Abbildung 4.2: Flussdiagramm: Stoppen des Systems

4.2 Integration der Managementfunktionen in das bestehende Framework

Für die Umsetzung der gewünschten Managementfunktionen bietet sich Nutzung der Java Management Extensions (JMX) Technologie ([jmx09]) an. Diese Technologie ist seit Java 1.5 in der Sun JVM integriert und stellt bereits grundlegende Funktionen zur Verfügung, insbesondere bietet die JMX-Technologie bereits definierte Schnittstellen zur Remote-Kommunikation mit Management-Klassen (sog. Managed Beans oder MBeans) und Mechanismen zur Kontrolle des Zugriffs auf Management-Funktionen. So können Benutzer definiert werden, die Zugriff auf Management-Funktionen haben und die Remote-Verbindung im Allgemeinen kann über Zertifikate abgesichert / verschlüsselt werden.
Die JVM stellt hierzu einen MBean-Server bereit, in dem alle benötigten Management-Klassen (MBeans) registriert werden. Der Zugriff auf den MBean-Server erfolgt über spezielle JMX Kommunikationsadapter (JMX Connector), welche die Schnittstelle für den Client bilden. Die Adapter ermöglichen es den Zugriff auf den MBean-Server zu kontrollieren. Eine Übersicht über die JMX-Architektur ist in Abbildung 4.3 gegeben.

Das Bereitstellen von neuen MBeans ist einfach, es muss nur ein Interface und eine Implementierung dieses Interfaces mit den gewünschten Management-Funktionen erstellt werden:

```
public interface MyManagementClassMBean {
    //Managementfunction "doSomething"
    public void doSomething();
}

public class MyManagementClass implements MyManagementClassMBean {
    public void doSomething() {
        ...
```

```
    }
}
```

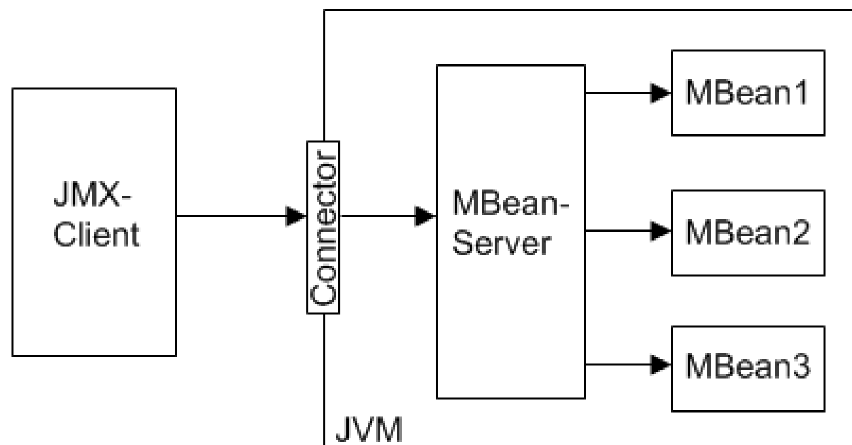

Abbildung 4.3: JMX: Übersicht

Diese Implementierung wird in dem MBean-Server der JVM registriert:

```
...
MBeanServer mbs = ManagementFactory.getPlatformMBeanServer();
ObjectName name =
    new ObjectName("com.example.mbeans:type=MyManagementClass");
MyManagementClass mbean = new MyManagementClass();
...
```

Nun steht diese Managementfunktion sofort zur Verfügung und kann aufgerufen werden.

Durch die Nutzung der JMX-Technologie bietet sich vor allem der Vorteil von standardisierten Mechanismen zum Remote-Zugriff, der Authentifizierung von Benutzern und der Absicherung der Verbindung. Es kann somit auf den Entwurf eines eigenen Management-Protokolls verzichtet werden.

Die in Kapitel 4.1. identifizierten Managementfunktionen lassen sich ein drei Bereiche gliedern:

- **Systemfunktionen**
  In dem Bereich der Systemfunktionen lassen sich das Aktualisieren von Komponenten und das Verschieben von Komponenten zwischen verschiedenen Servern einordnen.
- **Funktionen zur Beeinflussung der Entscheidungsfindung**
  Hier lässt sich die Funktion zur Bearbeitung der Regelsätze einordnen.
- **Netzwerkfunktionen**
  In den Bereich der Netzwerkfunktionen fallen das Starten und Stoppen des Autorisierungssystems.

Diese drei Bereiche können nun in entsprechenden Managementklassen implementiert werden. Hierbei kann es nötig sein, dass diese Managementklassen selbst auch direkte Verbindungen zu verschiedenen Komponenten haben, um somit direkten Einfluss auf die Komponenten nehmen zu können. Die drei Klassen werden nun kurz beschrieben:

- `NetworkManagement`
  Diese Klasse implementiert Funktionen aus dem Bereich „Netzwerkfunktionen". Hierzu benötigt diese Klassen eine Verbindung zur Komponente `Acceptor`. Über diese Verbindung wird die Komponente `Acceptor` veranlasst, Client-Verbindungen entgegen zunehmen oder abzulehnen. Das Klassendiagramm ist in Abbildung 4.4 angegeben.

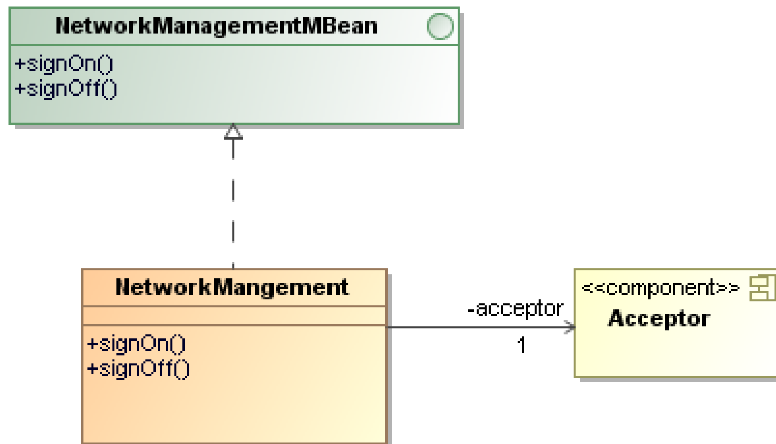

Abbildung 4.4: Klassendiagramm: NetworkManagement

- `SystemManagement`
  Diese Klasse implementiert Funktionen aus dem Bereich „Systemfunktionen". Diese Klasse muss Verbindungen zu allen bestehenden Komponenten haben. Dort kann sie Port-Verbindungen zwischen verschiedenen Komponenten auf- und abbauen. Zudem können über diese Klasse neue Komponenten bereitgestellt werden. Das Klassendiagramm ist in Abbildung 4.5 angegeben.

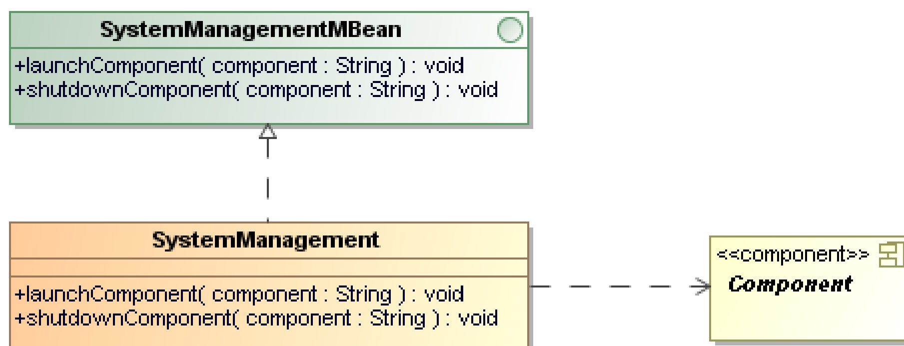

Abbildung 4.5: Klassendiagramm: SystemManagement

- `RulesetManagement`
  Diese Klasse implementiert Funktionen zur Beeinflussung der Entscheidungsfindung. Diese Klasse benötigt Zugriff auf die zur Entscheidungsfindung verfügbaren Regelsätze. Eine direkte Verbindung zu einer Komponente ist hierbei nicht nötig. Das Klassendiagramm ist in Abbildung 4.6 angegeben.

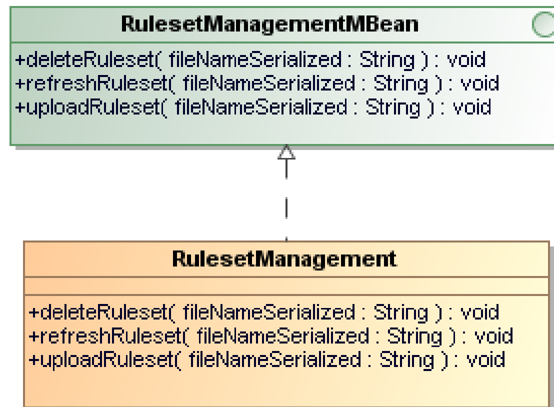

Abbildung 4.6: Klassendiagramm: RulesetManagement

Diese Klassen müssen nun direkt beim Start des Autorisierungssystems im MBean-Server der JVM registriert werden und stehen dann zum Aufruf bereit. Dieser Aufruf kann auf verschiedene Arten erfolgen:

- **jconsole**
  Mit jeder Standard JDK wird bereits das Programm „jconsole“ geliefert. Mit Hilfe dieses Programms kann man zu einer beliebigen laufenden JVM eine JMX/RMI-Verbindung herstellen und die dort bereitgestellten MBeans einsehen bzw. deren Funktionen aufrufen.

- **Eigener JMX-Client**
  Alternativ kann relativ einfach ein eigener Client JMX-Client erstellt werden. Dieser verbindet sich ebenfalls über eine JMX/RMI-Verbindung zu einer JVM und ruft dort Funktionen der bereitgestellten MBeans auf. In einem eigenen Client können komplexe Prozesse wie das Austauschen von Komponenten während des Betriebs automatisiert und für den Benutzer einfach bereitgestellt werden.
  Ein solcher Client wurde für den konkreten Einsatz des Systems zur Autorisierung von Kreditkartentransaktionen bei der petaFuel GmbH erstellt. Hier wurden auch die drei genannten Bereiche zum Netzwerk- (Abbildung 4.7), System- (Abbildung 4.8) und RulesetManagement (Abbildung 4.9) umgesetzt.

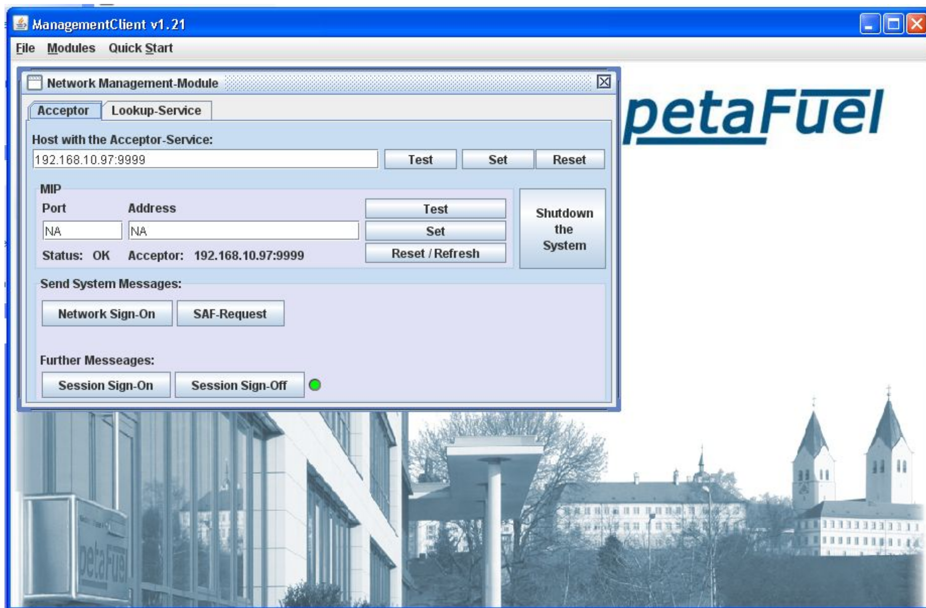

Abbildung 4.7: ManagementClient: NetworkManagement

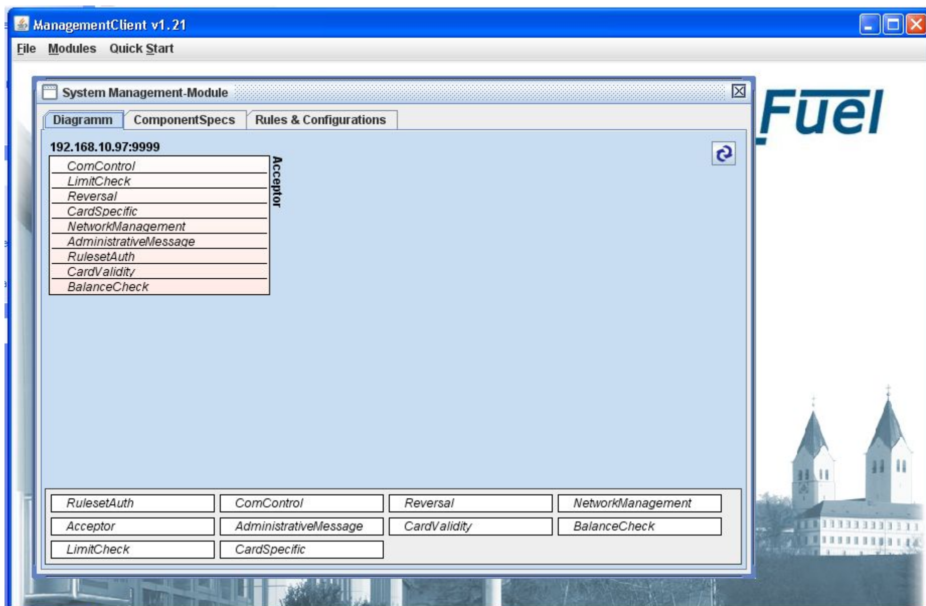

Abbildung 4.8: ManagementClient: SystemManagement

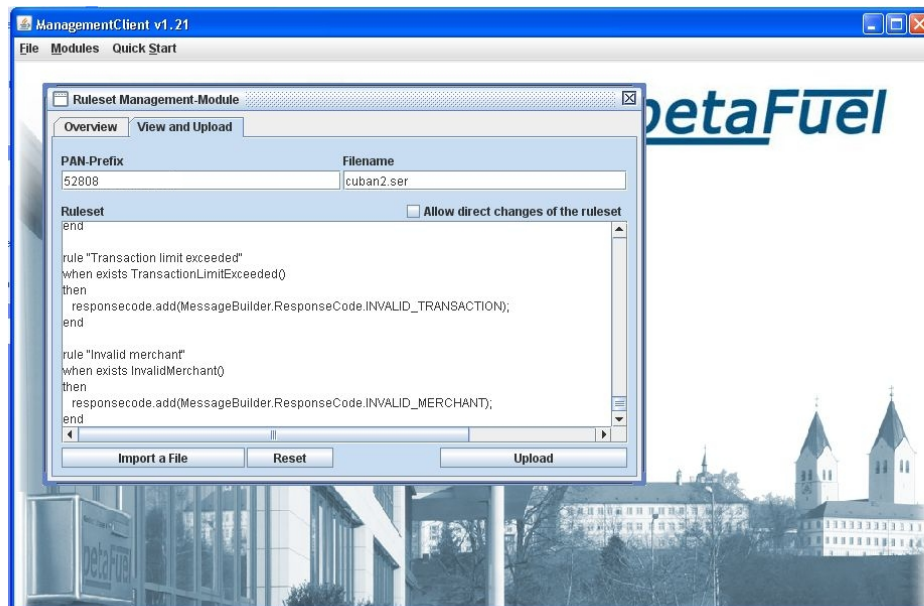

Abbildung 4.9: ManagementClient: RulesetManagement

5 Zeitliche Evaluierung verschiedener Implementierungsansätze

In diesem Kapitel werden verschiedene Implementierungsmöglichkeiten für das Autorisierungsframework vorgestellt. So kann die Kommunikation zwischen verschiedenen Komponenten synchron oder asynchron erfolgen und es können verschiedene Anfragen innerhalb einer Komponente sequentiell oder parallel bearbeitet werden. Hierbei wird vor Allem auf das zeitliche Vorhalten des System bei der Nutzung verschiedener Implementierungsansätze geachtet. So wird zunächst ein Testfall für eine Standardautorisierung festgelegt, dann werden die verschiedenen Implementierungsmöglichkeiten vorgestellt und abschließend das Verhalten des Systems unter Berücksichtigung der verschiedenen Technologien untersucht und verglichen.

5.1 Beschreibung des zur Evaluierung genutzten Testfalls

Zur Untersuchung des Systems wird die Kommunikation zwischen zwei einzelnen Komponenten beobachtet. Die Komponente `ComponentA` übergibt hierzu eine Anfrage an die Komponente `ComponentB`. Nach Bearbeitung der Anfrage in `ComponentB` gibt es eine Rückmeldung an `ComponentA`.
Komponente `ComponentA` besitzt einen Port `PortA` mit dem provided Interface `PortAProvidedInterface` und dem required Interface `PortBProvidedInterface`.
Komponente `ComponentB` hingegen besitzt den Port `PortB` mit dem provided Interface `PortBProvidedInterface` und dem required Interface `PortAProvidedInterface`.
Für die Kommunikation stellt `PortA` die Methode `serviceReply()` und `PortB` die Methode `service()` bereit. Das entsprechende Portprotokoll ist in Abbildung 5.1 dargestellt.

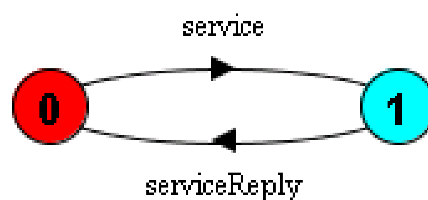

Abbildung 5.1: Portprotokoll: Testsystem

Die Bearbeitung einer Anfrage wird in der Komponente `ComponentA` durch die Komponenteninterne Methode `someInternalComputation()` simuliert. Diese Methode ist wie folgt implementiert:

```
protected void someInternalComputation() {
   try {
```

```
      //simulierte Bearbeitungsdauer: 100ms
      Thread.sleep(100);
   } catch (InterruptedException e) {}
}
```

In der Komponente `ComponentB` erfolgt die Bearbeitung einer Anfrage analog. Hier wurde eine Bearbeitungsdauer von 200ms gewählt.

Das Testsystem lässt sich somit wie in Abbildung 5.2 gegeben darstellen.

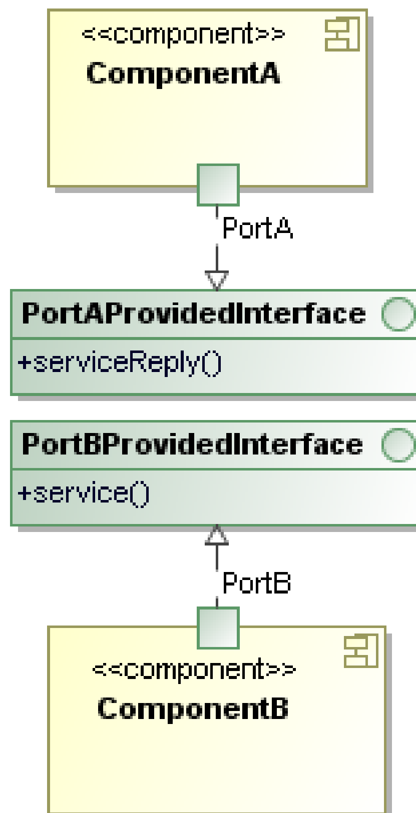

Abbildung 5.2: Übersicht über das Testsystem

5.2 Vorstellung der möglichen Implementierungstechniken

Für die Ausführung der Portoperationen des provided Interfaces stehen zunächst zwei grundlegend verschiedene Techniken zur Auswahl: das Standard Java/A Komponentenmodell nach [HJK08a] sieht eine single-threaded Ausführung vor, in [Ada09a] wird jedoch eine multithreaded Ausführung gefordert. Das Implementierungsframework von RAJA unterstützt beide Möglichkeiten. In diesem Zusammenhang muss auch entschieden werden, ob die Kommunikation der Komponenten Java-synchron oder asynchron mit Hilfe von Queues erfolgen soll. Es müssen somit vier unterschiedliche Implementierungsmöglichkeiten untersucht werden:

1. single-threaded und synchron
2. single-threaded und asynchron
3. multi-threaded und synchron
4. multi-threaded und asynchron

Zudem stellt sich die Frage, auf welche Bereiche des Autorisierungssystems diese Implementierungstechniken angewandt werden sollen. Mögliche Bereiche sind:

1. Komponenten global
   Die Bearbeitung einer Anfragen erfolgt über alle Operationen hinweg in einem Thread. Das heißt, dass alle zur Bearbeitung nötigen Operationen in dem selben Thread aufgerufen werden.
2. Komponenten lokal
   Zur Bearbeitung von Anfragen wird pro Komponente mindestens ein Thread erstellt. Innerhalb dieses Threads werden alle Operationen der Komponente ausgeführt.
3. System global (pro System in einem verteilten System)
   Die Bearbeitung einer Anfragen erfolgt pro System über alle Operationen hinweg in einem Thread. Das heißt, dass alle zur Bearbeitung nötigen Operationen des Systems in dem selben Thread aufgerufen werden.

Aus der Kombination der möglichen Implementierungstechniken und der Bereiche ergeben sich insgesamt 12 Fälle, die betrachtet werden können. Es lassen sich aber bereits hier erste Einschränkungen der zu untersuchenden Fälle vornehmen. So müssen folgende Fälle nicht genauer betrachtet werden, da sie sich auf andere Fälle zurückführen lassen:

- **Implementierung:** single-threaded und synchron
  Bereich: Komponenten intern
  Falls die Kommunikation der Komponenten synchron geschieht, werden alle nötigen Operationen sequentiell aufgerufen. Hierdurch ist dieser Fall aber gleich der synchronen single-threaded Implementierung im Bereich „Komponenten global".
- **Implementierung:** single-threaded und asynchron
  Bereich: Komponenten global
  Bei einer single-threaded Implementierung des gesamten Systems ist keine asynchrone Kommunikation möglich. Da innerhalb eines Threads alle Operationen nur sequentiell aufgerufen werden können, lässt sich dieser Fall auf die synchrone single-threaded Implementierung zurückführen.
- **Implementierung:** multi-threaded und synchron
  Bereich: Komponenten global
  In diesem Fall können mehrere Autorisierungsanfragen parallel bearbeitet werden. Es werden alle zur Bearbeitung einer Anfrage nötigen Operationen innerhalb dem selben Thread sequentiell aufgerufen. Da die parallele Bearbeitung mehrerer Anfragen aber auch durch die asynchrone multi-threaded Implementierung im Bereich „Komponenten intern" erreicht werden kann, wird auf die genauere Untersuchung dieses Falls verzichtet.

- **Implementierung:** multi-threaded und synchron
  Bereich: Komponenten intern
  In diesem Fall können mehrere Autorisierungsanfragen parallel bearbeitet werden. Es werden zwar zur Bearbeitung der Anfrage pro Komponente eigene Threads genutzt, da die Kommunikation dieser Threads aber synchron ist, werden alle Operationen sequentiell aufgerufen. Dieser Fall lässt sich somit auf die synchrone multi-threaded Implementierung im Bereich „Komponenten global" zurückführen.

- **Implementierung:** multi-threaded und asynchron
  Bereich: Komponenten global
  Da in diesem Fall alle nötigen Operationen innerhalb eines Threads aufgerufen werden, kann keine asynchrone Kommunikation erfolgen. Dieser Fall lässt sich somit auf die synchrone multi-threaded Implementierung im Bereich „Komponenten global" zurückführen.

- **Bereich:** System global
  Der Bereich „System global" lässt sich generell auf andere Bereiche zurückführen. Wenn es nur ein System gibt, entspricht dieser dem Bereich „Komponenten global". Falls aber mehrere Systeme vorhanden sind, kann unterschieden werden, wie viele Komponenten pro System bereitgestellt werden. Im Idealfall wird pro System nur eine Komponente bereitgestellt, dann entspricht dieser Bereich aber dem Bereich „Komponenten intern". Falls aber mehrere Komponenten pro System bereitgestellt werden, entspricht dieser Bereich einer Mischung aus den Bereichen „Komponenten intern" und „Komponenten global". Da hier aber vor allem eine ideale Implementierung gefunden werden soll, wird dieser Bereich bei einer Verteilung von mehreren Komponenten pro System nicht weiter untersucht, sondern nur der Idealfall von einer Komponenten pro System betrachtet. Damit muss der Bereich „System global" aber nicht gesondert untersucht werden, sondern man kann sich auf den Bereich „Komponenten intern" beschränken.

Durch diese Einschränkungen ergeben sich effektiv drei Fälle, die genauer untersucht werden müssen:

- **Fall 1**
  Implementierung: single-threaded und synchron
  Bereich: Komponenten global
  Die Bearbeitung einer Anfrage erfolgt über alle nötigen Operationen hinweg innerhalb eines Threads. Eingehende Nachrichten werden sequentiell bearbeitet. Ein entsprechendes Sequenzdiagramm ist in Abbildung 5.3 angegeben. Die dargestellte Ausführung entspricht dabei nicht dem natürlichen Verständnis der Komponentenkommunikation nach dem Portprotokoll. Diese Implementierung ist jedoch die einzige Möglichkeit einer single-threaded, Java-synchronen Kommunikation.

- **Fall 2**
  Implementierung: single-threaded und asynchron
  Bereich: Komponenten intern
  Pro Komponente steht ein Thread zur Ausführung aller Operationen der Komponente zur Verfügung. Es können mehrere Operationen unterschiedlicher Komponenten parallel ausgeführt werden. Anmerkung: Dieser Fall entspricht dem asynchronen Fall

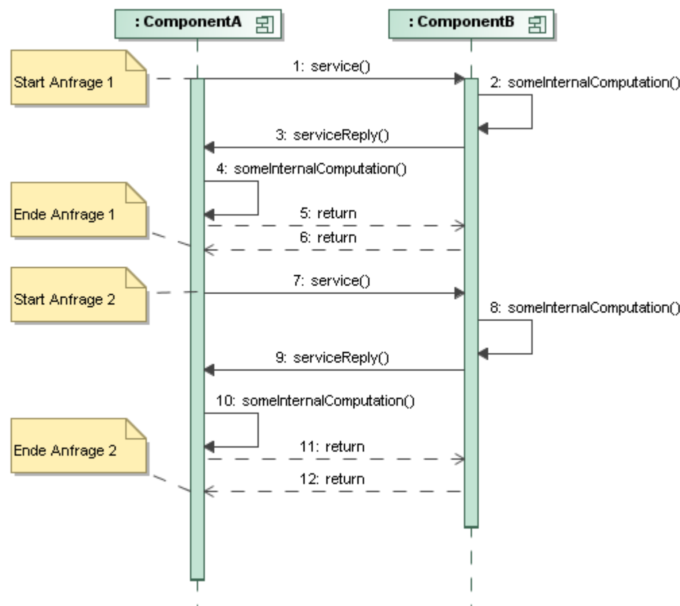

Abbildung 5.3: Sequenzdiagramm Fall 1

des Java/A Modells mit aktiven Komponenten nach [HJK08a]. Ein entsprechendes Sequenzdiagramm ist in Abbildung 5.4 angegeben. Auf die Modellierung der Portqueues und die entsprechenden put- und take-Operationen wird hier verzichtet. Der dargestellte Fall impliziert mehrere Portinstanzen, damit `ComponentA` bereits ein neues service-Signal an `ComponentB` schicken kann, während die entsprechende serviceReply-Operation für das erste Signal noch nicht eingetroffen ist. Die an den Portqueues anliegenden Signale werden jedoch von der Komponente single-threaded verarbeitet.

- **Fall 3**
  Implementierung: multi-threaded und asynchron
  Bereich: Komponenten intern
  Pro Operationsaufruf auf einem Port wird ein Thread gestartet. Es können somit auch Operationen der selben Komponente parallel ausgeführt werden. Ein entsprechendes Sequenzdiagramm ist in Abbildung 5.5 angegeben.

5.3 Beschreibung der Implementierungsmöglichkeiten innerhalb des Frameworks

Fall 1:
In diesem Fall wird beim Systemstart ein einzelner Thread initialisiert. Eingehenden Anfragen werden innerhalb dieses Threads entgegengenommen und bearbeitet. Zur Bearbeitung der Anfrage werden nun die benötigten Komponenten sequentiell / synchron aufgerufen. Sobald die Entscheidung zu der Anfrage feststeht und das Ergebnis an den Client gemeldet wurde, kann die nächste Anfrage bearbeitet werden.

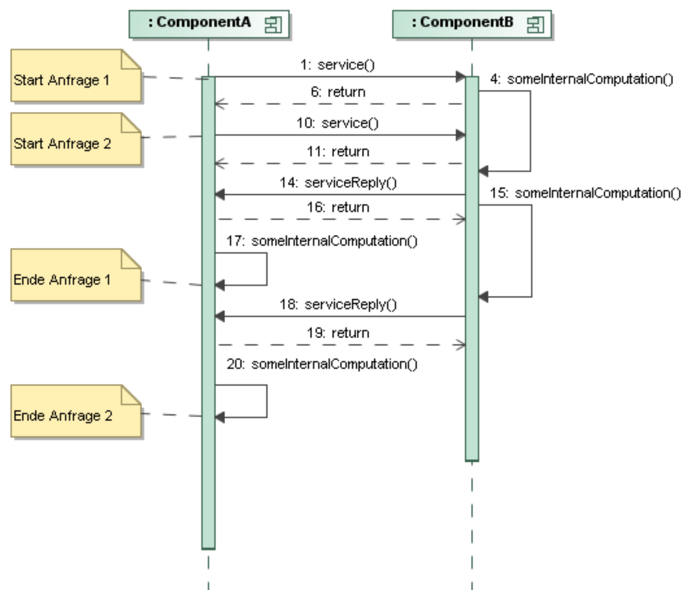

Abbildung 5.4: Sequenzdiagramm Fall 2

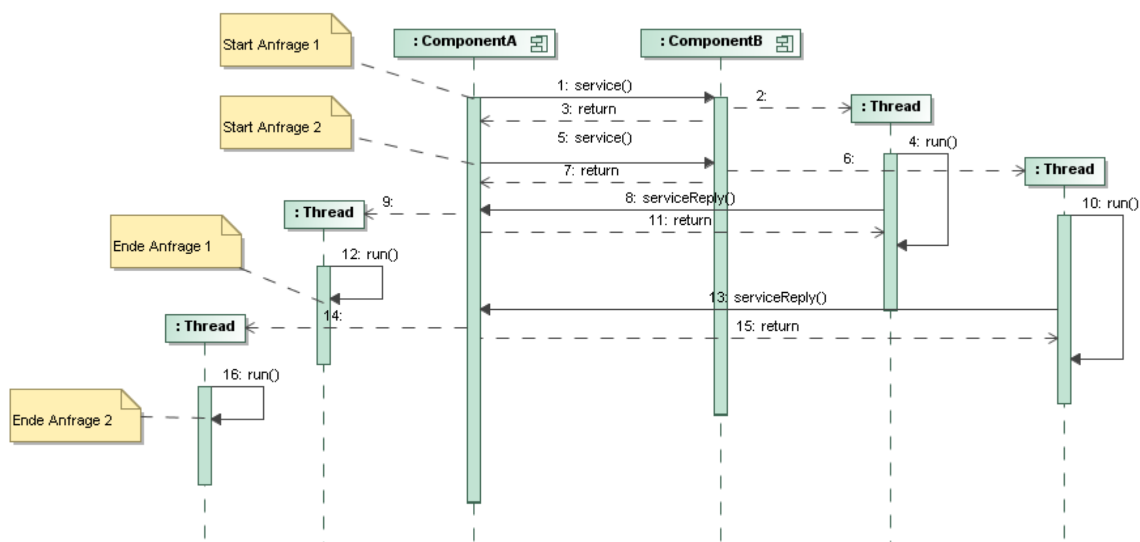

Abbildung 5.5: Sequenzdiagramm Fall 3

Fall 2:
Es wird pro Komponente ein Thread initialisiert, in dem die Operationen der Komponente ausgeführt werden. Eingehende Nachrichten werden in der Komponente in einer Warteschlange abgelegt und von dem Thread zur Bearbeitung entnommen. Nach Bearbeitung einer Anfrage wird das Ergebnis bei der nächsten Komponente ebenfalls in einer Warteschlange abgelegt und die nächste Anfrage zur Bearbeitung aus der eigenen Warteschlange entnommen. Somit können die aufrufende und die aufgerufene Komponente unabhängig voneinander parallel verschiedene Anfragen bearbeiten.

Fall 3:
Fall 3 ist prinzipiell analog zu Fall 2, aber es werden mehrere Threads pro Komponente initialisiert. Diese Threads greifen auf dieselbe Warteschlange zu und Operationen derselben Komponente können somit auch parallel ausgeführt werden.

5.4 Untersuchung der Implementierungsmöglichkeiten

Im Folgenden wird nun das zeitliche Verhalten des Systems unter Berücksichtigung der Implementierungsmöglichkeiten untersucht. Hierzu werden die drei zu betrachtenden Fälle in dem beschriebenen Testsystem umgesetzt und das zeitliche Verhalten des System bei der Bearbeitung von Anfragen beobachtet. Bei diesen Tests können zwei Parameter variiert werden:

- Die Anzahl der parallel zu bearbeitenden Anfragen.
  Die Anzahl der Anfragen wurde hier zwischen 10 und 300 Anfragen variiert.
- Die Anzahl der verfügbaren Threads bei einer multi-threaded Implementierung (nur Fall 3).
  Hierzu wurde Fall 3 weiter unterteilt:
  - **Fall 3.1**: Es stehen pro Komponente 10 Threads zur Bearbeitung bereit.
  - **Fall 3.2**: Es stehen pro Komponente 20 Threads zur Bearbeitung bereit.
  - **Fall 3.3**: Es stehen pro Komponente 30 Threads zur Bearbeitung bereit.

Gemessen wurde bei den Tests zunächst die Gesamtlaufzeit (Abbildung 5.6) bis alle Anfragen bearbeitet waren. Aus dieser Gesamtlaufzeit lässt sich ein Durchsatz (Abbildung 5.7) von bearbeiteten Anfragen pro Sekunde errechnen.

Es zeigt sich, dass Fall 1 die schlechtesten Ergebnisse liefert. Fall 2 ist mit steigender Anzahl an Anfragen dem Fall 1 deutlich überlegen. Fall 3 im Allgemeinen liefert hier die besten Ergebnisse. Bei steigender Anzahl an verfügbaren Threads sinkt die Gesamtlaufzeit und damit der Durchsatz an bearbeiteten Anfragen pro Sekunde.

5.5 Bewertung der Ergebnisse

Anhand der Untersuchung der Implementierungsmöglichkeiten zeigt sich, dass aus theoretischer Sicht die Umsetzung eines multi-threaded, asynchronen Systems im Bereich „Komponenten intern“ die beste Lösung darstellt. Allerdings ergibt sich bei einer solchen Implementierung in der Praxis ein neues Problem: Es kann eine Synchronisierung zwischen einzelnen Anfragen nötig sein.
Je nach Einsatzzweck des Autorisierungssystems kann es möglich sein, dass zwischen einzelnen Anfragen synchronisiert werden muss. Beim Einsatz des Systems z.B. zur Autorisierung von Kreditkartentransaktionen dürfen Anfragen, die dieselbe Kreditkarte betreffen nicht parallel autorisiert werden. Bei einer solchen Anfrage kann z.B. das verfügbare Guthaben der Kreditkarte als einzelne Ressource gesehen werden, die zur Entscheidungsfindung benötigt wird. Diese Ressource muss geschützt werden und darf zu einem Zeitpunkt nur von einer

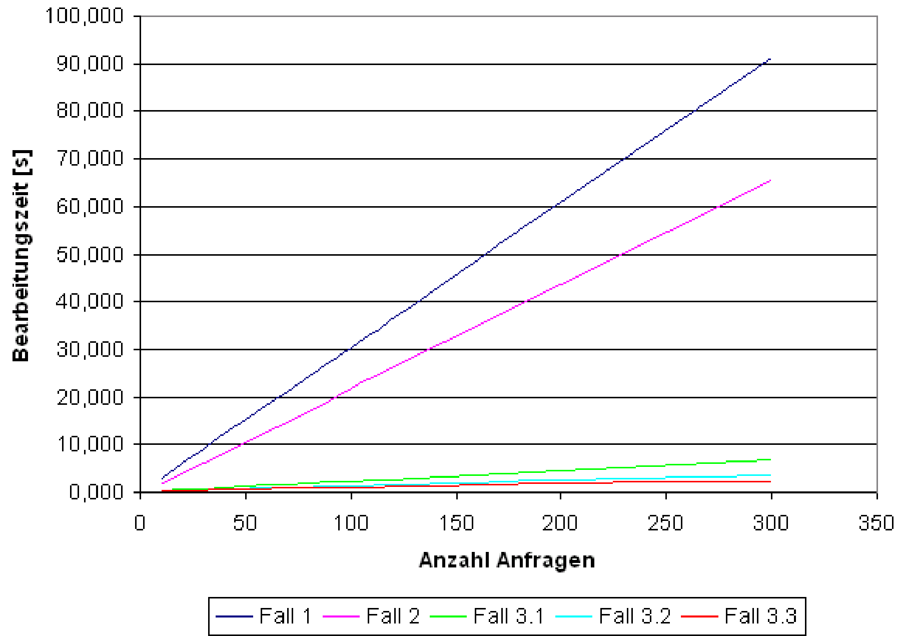

Abbildung 5.6: Auswertung: Gesamtlaufzeit

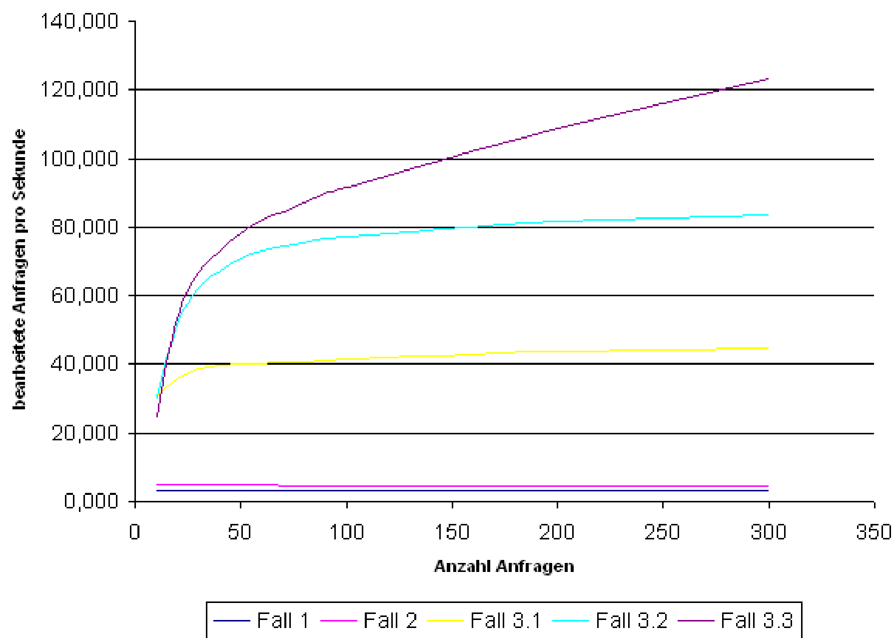

Abbildung 5.7: Auswertung: Durchsatz

Anfrage genutzt werden.
Diese Problem kann bei einer einzelnen betroffenen Ressource einfach in der Implementierung innerhalb der betroffenen Komponente gelöst werden. Dort muss die betroffene Ressource

dann mit Hilfe eines Monitors gesichert und somit ein exklusiver Zugriff sichergestellt werden. Falls aber mehrere Ressourcen über mehrere Komponenten hinweg geschützt werden müssen, ergeben sich weiterhin Probleme. Da die Entscheidungsfindung für eine Anfrage innerhalb des Systems nicht in einer Komponenten festgelegt ist, sondern durch die Nutzung des Regelsystems auf mehrere spezialisierte Entscheidungskomponenten verteilt wird, ergibt sich das Problem sogar konkret. Hier ist es nun sehr aufwändig mit Hilfe von Monitoren und einer Deadlock-Erkennung innerhalb der einzelnen Komponenten exklusive Ressourcen zu schützen, deshalb ist es sinnvoll dieses Problem direkt beim Design des Autorisierungssystems zu beachten.
Somit bietet es sich an das Autorisierungsframework nicht komplett in einer der oben gefundenen Implementierungsmöglichkeiten umzusetzen. Vielmehr lassen sich die Komponenten innerhalb des Frameworks in verschiedene Bereiche unterteilen, in denen verschiedene Anforderungen herrschen. So können problemlos mehrere Anfragen parallel entgegengenommen und zur Koordination der Bearbeitung weitergereicht werden. Die Entscheidungsfindung für die Anfragen muss aber synchron geschehen. Konkret ergibt sich also auf Komponentenbasis folgender Implementierungsvorschlag:

- **Komponenten** `Acceptor`, `CommunicationControlComponent` **und** `RulesetAuthComponent`
  Diese Komponenten können multi-threaded und asynchron implementiert werden. Diese Komponenten arbeiten in ihrer Funktion unabhängig voneinander und exklusive Ressourcen innerhalb einer Komponente können bei Bedarf einfach geschützt werden.

- **Komponenten vom Typ** `DecisionComponent`
  Diese Komponenten müssen single-threaded implementiert werden. Da diese Komponenten bei der Entscheidungsfindung auf exklusive Ressourcen zugreifen muss hier eine Synchronisierung stattfinden. Verschiedene Komponenten vom Typ `DecisionComponent` können problemlos parallel arbeiten, da diese Komponenten zwar auf exklusive Ressourcen zugreifen, diese Ressourcen aber nur von dieser einen Komponente genutzt werden.

Zusammenfassend kann man nun sagen, dass es keinen allgemeinen und für das gesamte Framework gültigen Implementierungsvorschlag gibt. Vielmehr muss eine Differenzierung geschehen und eine auf die Problemstellung angepasste Implementierung der einzelnen Komponenten geschehen.

6 Zusammenfassung

Mit der Entwicklung des komponenten-orientierten, regelbasierten Autorisierungsframeworks RAJA ergeben sich viele Vorteile: Der komponenten-basierte Ansatz ermöglicht eine Gliederung des Frameworks in verschiedene, funktionelle Bereiche, welche nur über definierte Ports miteinander kommunizieren. Zudem ist es durch diesen Ansatz möglich die Bereiche des Systems getrennt zu spezifizieren und zu validieren. Der regelbasierte Ansatz hingegen bietet die nötige Flexibilität beim Einsatz des Frameworks in einem konkreten Szenario. Dieser Vorteil zeigt sich bereits bei der Nutzung des Frameworks zur Autorisierung von Kreditkartentransaktionen.
Das Management des Autorisierungsframeworks wird vor allem durch den Einsatz der standardisierten JMX-Technologie vereinfacht. Hier können benötigte Managementfunktionen einfach bereitgestellt werden und der Zugriff auf diese Funktionen geschieht geschützt.
Das Autorisierungsframework unterstützt verschiedene Arten der Komponentenkommunikation, diese Kommunikation kann sowohl synchron, wie auch asynchron geschehen. Zudem können die Operationen der Komponenten sequentiell (single-threaded) oder parallel (multi-threaded) ausgeführt werden. Durch die parallele Ausführung der Operationen und die asynchrone Kommunikation der einzelnen Komponenten wird eine hohe Effizienz des Systems erreicht.
Zusammenfassend entsteht durch das Autorisierungsframework RAJA ein sehr flexibles System, dass durch seine Architektur einfach auf die Anforderungen vieler unterschiedlicher Szenarien anpassbar ist. Die in dieser Arbeit vorgestellte Implementierung stellt dabei aber nur einen ersten Prototypen dar. Für den im IuK-Projekt geforderten Einsatz in einer Echtzeitumgebung ist es zusätzlich notwendig zu untersuchen, wie sich die unterschiedlichen Implementierungsmöglichkeiten unter einem Echtzeit-Scheduler umsetzen lassen und ob die hier beobachteten Ergebnisse dann noch gültig sind.

Abbildungsverzeichnis

2.1 Kernkomponenten des Autorisierungsframeworks 7

3.1 funktionale Bereiche des Autorisierungssystems 10
3.2 Sequenzdiagramm: Ablauf einer Autorisierung 11
3.3 Portprotokoll: IOFacade - ClientConnector 11
3.4 Portprotokoll: AuthorizationDispatcher - AuthorizationProvider 12
3.5 Portprotokoll: DecisionDispatcher - SingleDecisionProvider 12
3.6 Framespezifikation: Acceptor 12
3.7 Framespezifikation: CommunicationControlComponent 13
3.8 Framespezifikation: RulesetAuthComponent 13
3.9 Framespezifikation: DecisionComponent 14
3.10 Framespezifikation für den gesamten Ablauf einer Autorisierung 14

4.1 Flussdiagramm: Austausch einer Komponente 16
4.2 Flussdiagramm: Stoppen des Systems 17
4.3 JMX: Übersicht 18
4.4 Klassendiagramm: NetworkManagement 19
4.5 Klassendiagramm: SystemManagement 19
4.6 Klassendiagramm: RulesetManagement 20
4.7 ManagementClient: NetworkManagement 21
4.8 ManagementClient: SystemManagement 21
4.9 ManagementClient: RulesetManagement 22

5.1 Portprotokoll: Testsystem 23
5.2 Übersicht über das Testsystem 24
5.3 Sequenzdiagramm Fall 1 27
5.4 Sequenzdiagramm Fall 2 28
5.5 Sequenzdiagramm Fall 3 28
5.6 Auswertung: Gesamtlaufzeit 30
5.7 Auswertung: Durchsatz 30

Literaturverzeichnis

[Ada08] ADAM, LUDWIG: *Komponenten-orientierte Entwicklung eines regelbasierten Autorisierungssystems.* Diplomarbeit, Ludwig Maximilian Universität München, Februar 2008.

[Ada09a] ADAM, LUDWIG: *Realtime-contraints for component-based enterprise architectures demonstrated on the RAJA case study.* Doktorarbeit, Ludwig Maximilian Universität München, 2009. noch nicht veröffentlicht.

[Ada09b] ADAM, LUDWIG: *Regelsatz*, 2009. `http://www.pst.ifi.lmu.de/~adaml/pub/raja/default.drl`.

[BHH+05] BAUMEISTER, H., F. HACKLINGER, R. HENNICKER, A. KNAPP und M. WIRSING: *A Component Model for Architectural Programming.* In: *Proc. FACS'05, International Workshop on Formal Aspects of Component Software*, Seiten 75–96, 2005.

[HJK08a] HENNICKER, R., S. JANISCH und A. KNAPP: *Component Specification and Asynchronous Communication.* In: *Monterey Workshop, Budapest*, 2008.

[HJK08b] HENNICKER, R., S. JANISCH und A. KNAPP: *On the Observable Behaviour of Composite Components.* In: *FACS'08*, 2008.

[iuk09] *IuK*, 2009. `http://www.iuk-bayern.de`.

[jmx09] *Java Management Extensions (JMX)*, 2009. `http://java.sun.com/javase/technologies/core/mntr-mgmt/javamanagement/index.jsp`.

[KJH+07] KNAPP, ALEXANDER, STEPHAN JANISCH, ROLF HENNICKER, ALLAN CLARK, STEPHEN GILMORE, FLORIAN HACKLINGER, HUBERT BAUMEISTER und MARTIN WIRSING: *Modelling the CoCoME with the JAVA/A Component Model*, Juli 2007. `http://www.pst.ifi.lmu.de/Research/current-projects/cocome/cocome-extended.pdf`.

[MK09] MAGEE, JEFF und JEFF KRAMER: *LTSA - Labelled Transition System Analyser.* http://www.doc.ic.ac.uk/ltsa/, 2009.

[raj09] *RAJA: Entwicklung eines echtzeitfähigen und regelbasierten Autorisierungsframeworks auf Java-Basis.*, 2009. `http://www.pst.ifi.lmu.de/Forschung/laufende-projekte/raja/projektbeschreibung`.