Design Document

Etienne Buschong Jens Kiefer Fabian Kreppel

8. Dezember 2015

Inhaltsverzeichnis

Teil I

Architectural Design

Kapitel 1

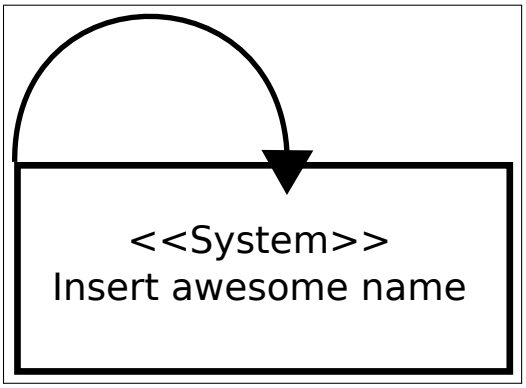
Einleitung

In diesem Dokument wird die Struktur unseres Systems anhand verschiedener graphischer Darstellungen (oftmals auf Pattern beruhend) dargestellt. Es werden sowohl Hierarchien der verschiedenen Komponenten des Systems erklärt, als auch ihr Zusammenspiel. Für die benutzten Patterns werden außerdem Sequenz-Diagramme angegeben, welche an beispielhafte Szenarien angelehnt sind, um ihre Funktionsweise anschaulich zu machen. Vom Ablauf her wird das System zuerst in sehr große Einzelteile aufgeteilt, welche dann jeder für sich immer detaillierter beschrieben werden. Somit erreicht man ein „Herein-Zoomen“ in das System.
Alle Verweise beziehen sich auf Functional Requirements der Requirements Solution vom 20.11.2015.

Kapitel 2

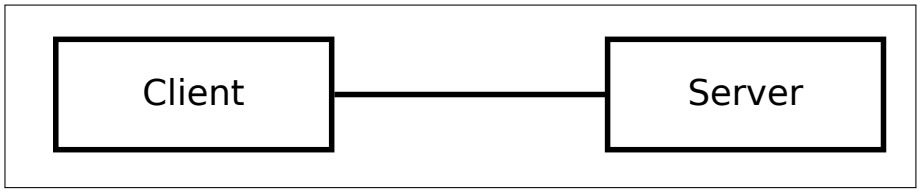
Komponenten

Context-Modell:



Das Context-Modell zeigt, dass das System keine weiteren Beziehungen zu anderen System hat. Es kann lediglich zu anderen Instanzen des Systems Beziehungen haben, was mittels eines Pfeils zu sich selbst symbolisiert wird.

Client-Server:



Als gröbste Einteilung kann man das System in Client und Server aufteilen, da diese beiden Komponenten die Hauptbestandteile des Systems sind. Hier bietet sich das Client-Server-Pattern an, welches genau unseren Anforderungen genügt. Es beschreibt die Verteilung von Aufgaben in einem Netzwerk, in unserem konkreten Fall dem World Wide Web, wobei die Bearbeitung dieser Aufgaben auf zwei Komponenten, einerseits dem Client und andererseits dem Server, stattfindet.

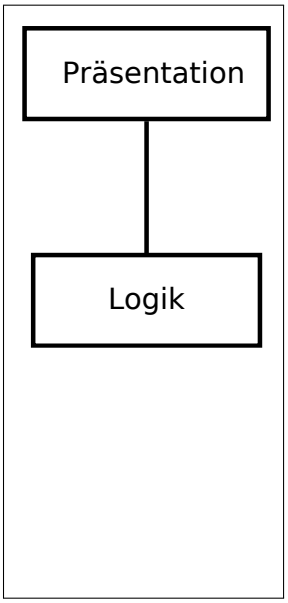
Der Client wird bei uns durch das Frontend repräsentiert. Der Nutzer hat hier die Möglichkeit eine Aufgabenanfrage zu stellen, welche daraufhin durch den Server bearbeitet und erledigt wird. Es kommt hierbei das „Thin-Client-Model“ zum Einsatz, sprich der Client führt keinerlei Berechnungen durch, sondern ist lediglich für die Visualisierung zuständig. In unserem Fall ist dies die HTML-Seite, welche durch einen Web-Browser erreichbar ist. Zusätzlich kann das System durch Programme benutzt werden, welche in der Lage sind REST-konforme HTTP-Anfragen zu stellen. Somit findet im Client nur die Dateneingabe und die Datenausgabe statt. Jegliches Prozessieren und Speichern von Daten geschieht auf dem Server (vgl. FR020).

Die Wahl viel auf das „Thin-Client-Model“, da es viele Probleme erspart, wenn der Nutzer keine Software installieren muss, welche zu Kompatibilitätsproblemen führen könnte. Außerdem entfällt von der Seite des Entwicklers der Aufwand vorhandene Software zu warten. Es ist somit sicher gestellt, dass jeder der einen funktionierenden Internetbrowser besitzt, Zugriff auf unser System hat.

Ein Nachteil, der auf der Hand liegt, ist natürlich das hohe Maß an Daten, welche über das Netzwerk ausgetauscht werden müssen, da keinerlei Berechnung lokal stattfindet. Die plattformunabhängige und einfache Nutzung ist jedoch ein Vorteil, der in unserem Fall stark überwiegt.

Der Server arbeitet wie im klassischen Modell passiv, das heißt, er wartet auf die aktiven Anfragen des Clients und bearbeitet diese daraufhin.

Client:



Bei näherer Betrachtung des Clients ist eine weitere Unterteilung desselben in mehrere Teilkomponenten möglich. Dies wird in diesem Fall mit einem Layer-Pattern umgesetzt. Das Ziel des Layer-Pattern ist es, eine Separation und somit die Unabhängigkeit verschiedener Komponenten zu schaffen.

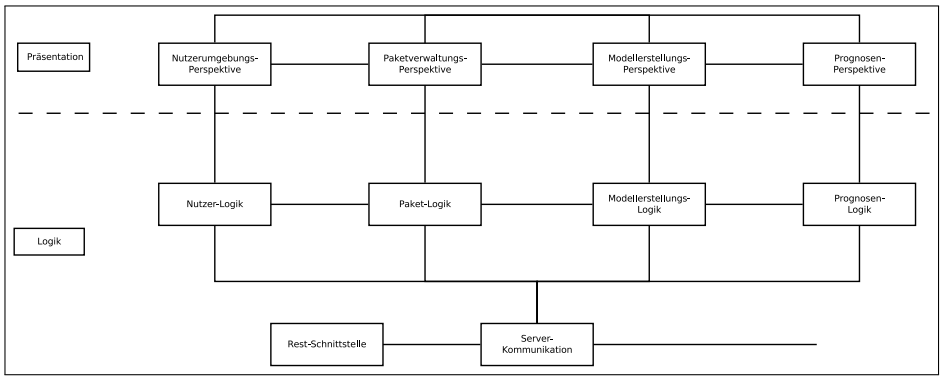
Der Vorteil besteht darin, dass die Trennung von Komponenten größer wirkende Aufgaben in kleinere Teilaufgaben aufspaltet. Diese Teilaufgaben können dann separat von einander entwickelt, gewartet und modifiziert, beziehungsweise sogar ausgetauscht werden. Dies macht die Weiterentwicklung unseres Systems wesentlich einfacher, da man, wenn man eine Schicht verändern oder neu implementieren möchte, nur schauen muss, dass der Input aus der unteren Schicht weiterhin richtig verarbeitet wird und immer noch eine korrekte Ausgabe für die Schicht darüber stattfindet. Alle anderen Schichten funktionieren weiterhin unverändert und benötigen keine weitere Betrachtung. Man kann außerdem erkennen, welche Schritte ein gewisser Vorgang durchlaufen muss, was für eine gewisse Transparenz in diesem sorgt.

Ein nicht ganz unwesentlicher vorteilhafter Aspekt ist auch die eigentliche Entwicklung eines in Schichten geteilten Systems: Verschiedene Personen oder Teams können ohne jede Interaktion die Schichten parallel zu einander entwickeln, wenn die Anforderungen, welche an jede Schicht gestellt werden, klar sind.

Offensichtliche Nachteile sind zum Beispiel zum einen die in der Praxis nicht ganz triviale Separation einer Komponente in verschiedene Teilkomponenten oder das Problem, dass höhere Schichten bei der Implementierung direkt mit niederen Ebenen kommunizieren müssen ohne die dazwischenliegenden Ebenen zu nutzen.

In unserer Implementierung des Clients ist die oberste Schicht die graphische Präsentation, welche dem Nutzer durch eine HTML-Seite bereitgestellt wird. Hier kann er Eingaben tätigen, welche daraufhin von der nächsten Schicht, nämlich der clientseitigen Logik verarbeitet werden. Diese Eingaben werden dann an die Datenschicht weitergesendet.

Eine etwas detaillierte Ansicht des Clients:



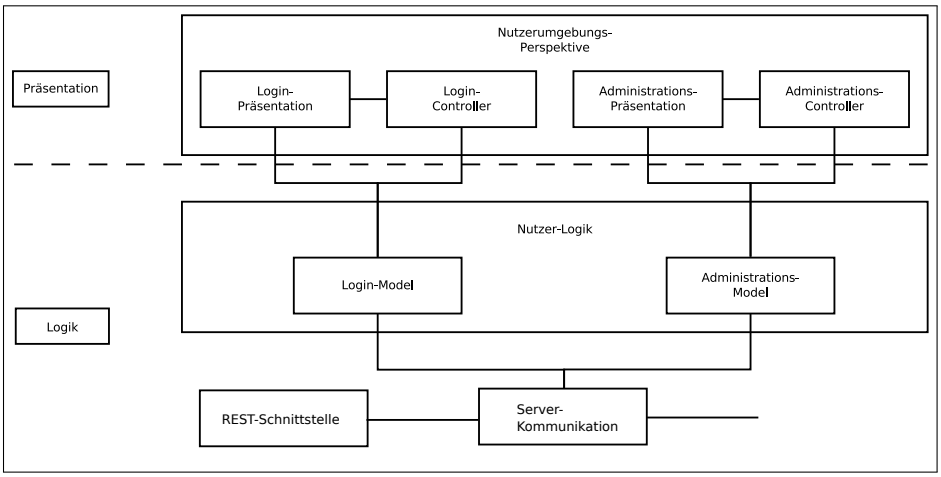
Nun betrachten wird das zuvor beschriebene Layer-Pattern des Clients detaillierter, indem wir die Präsentation, die Logik und die Datenschicht in Teilkomponenten aufteilen, um die verschiedenen Aufgaben und Kompetenzen besser aufzuzeigen.

In der Präsentations-Schicht unterscheiden wir zwischen den verschiedenen Visualisierungen für den Nutzer. Zunächst existiert eine Nutzerumgebungsperspektive: Sie ist die Visualisierung der Webapplikation und beinhaltet das grundlegende Gerüst der Seite. Von dieser Visualisierung kann man zu den anderen gelangen. Die Präsentation wird in der Logik-Schicht von der Nutzer-Logik gesteuert, welche die Aktionen des Nutzers an die Serverkommunikation weiterleitet.

Eine weitere Perspektive ist die Paketverwaltungsperspektive: Betritt der Nutzer über die Nutzerumgebungsperspektive einen Bereich, in welchem Pakete verwaltet werden, wird die Paketverwaltungsperspektive dynamisch eingebunden, damit der Nutzer auf dem Inhalt von Paketen arbeiten kann. Die Kommunikation zwischen den Eingaben des Nutzers und der Serverkommunikation wird in der Logik-Schicht von der Paket-Logik übernommen.

Auch für die Modellerstellung und die generierten Prognosen ist eine eigene Visualisierung vorhanden, welche dynamisch eingebunden werden kann. Diese beiden Perspektiven verfügen ebenfalls über jeweils eine Logik in der Logik-Schicht, welche die getätigten Eingaben an die Serverkommunikation weiterleiten. Somit erreichen alle Aktionen des Benutzers die Serverkommunikation, welche die Eingaben an eine REST-Schnittstelle weiterleitet, in welcher diese passend kodiert werden (vgl. FR017).

Nutzerumgebung:



Da es sich bei unserem System um eine interaktive Applikation handelt, ist ein weiteres Pattern, welches sich als sehr nützlich gestaltet, das Model-View-Controller-Pattern. Dieses besteht aus den drei Komponenten Model, View und Controller.

Bei dem Model-View-Controller-Pattern übernehmen die einzelnen Komponenten folgende Aufgaben:
Model:

- kapselt den Applikationsstatus
- benachrichtigt die View über Statusänderungen

View:

- generiert die Visualisierung des Models
- fordert Änderungen des Models an
- sendet die Nutzereingaben an den Controller

Controller:

- verarbeitet die Nutzereingaben zu Model-Veränderungen
- wählt die Visualisierung

Ein großer Vorteil, welcher sich ebenfalls im Layer-Pattern gefunden hat, ist die Modularität: Die einzelnen Komponenten können wieder leicht ersetzt werden. Das besondere Augenmerk liegt bei diesem Pattern auf der View. Durch die Modularität kann es nicht nur eine View geben, sondern beliebig viele, welche zur

Laufzeit dynamisch gewechselt werden können, um dem Nutzer die bestmögliche Visualisierung für die zu präsentierenden Informationen zu ermöglichen (vgl. FR019).

Die Nutzerumgebung des Clients wird dabei in die verschiedenen Anforderung des Nutzers untergliedert. Dadurch wird gewährleistet, dass jeder Nutzer eine seinen Anforderungen entsprechende Präsentation des Systems erhält.

Zum Beispiel bekommt der Nutzer beim Login ein Graphical User Interface, also eine HTML-Seite, für den Login angezeigt, das von der View visualisiert wird und den Nutzer die Eingabe seiner Login-Daten erlaubt. Von der View werden die Login-Daten an den Login-Controller weitergeleitet und Änderungen des Login-Modells abgefragt.

Der Login-Controller verarbeitet die Login-Daten des Nutzers, welche er von der Login-View erhält und gibt diese an das Login-Modell weiter. Zudem wählt er die passende Visualisierung für die Login-View aus, zum Beispiel separierte Präsentationen für einen noch nicht getätigten, erfolgreichen oder fehlgeschlagenen Login.

Das Login-Modell, das Bestandteil der Nutzer-Logik ist, sorgt für eine Kapselung des Applikationsstatus und benachrichtigt, falls er eine Anfrage der Login-View über ein Status-Update erhalten hat, die Login-View mit den Änderungen. Es ist ebenfalls dafür zuständig, um die Login-Daten des Nutzers an die Datenschicht des Clients weiter zu geben.

Ein weiteres Beispiel ist der Administrator. Dieser bekommt von der Administrator-View ein Graphical User Interface angezeigt, dass ihn erlaubt, Änderungen am System vorzunehmen, für die nur er die Befugnisse besitzt. Von der Administrator-View werden diese Daten an den Administrator-Controller weitergeleitet und Änderungen des Adminstrator-Modells abgefragt.

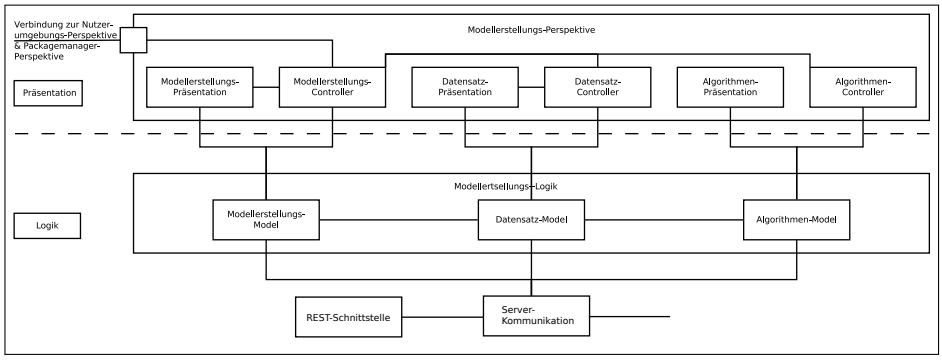
Der Administrator-Controller verarbeitet die Eingaben des Administrators, welche er von der Administrator-View erhält und gibt diese an das Administrator-Modell weiter. Zudem wählt er die passende Visualisierung für die Administrator-View aus, zum Beispiel separierte Präsentationen für Paket- und Nutzerverwaltung.

Das Administrator-Modell, das Bestandteil der Nutzer-Logik ist, sorgt für eine Kapselung des Applikationsstatus und benachrichtigt, falls er eine Anfrage der Administrator-View über ein Status-Update erhalten hat, die Administrator-View mit den Änderungen. Es ist ebenfalls dafür zuständig, um die Eingaben des Administrators an die Datenschicht des Clients weiter zu geben.

Die Wahl fiel auf dieses Pattern, da es unsere Anforderungen erfüllt, dem Nutzer verschiedene Visualisierungsmöglichkeiten für Daten zu bieten. Außerdem

sollte es in Zukunft einfach sein, weitere Visualisierungsmöglichkeiten schnell und unkompliziert hinzuzufügen. Weiterhin ist die Repräsentation der Daten im Modell und die eigentliche Präsentation der Daten in der View komplett unabhängig von einander, was erneut der Implementierung zu Gute kommt.

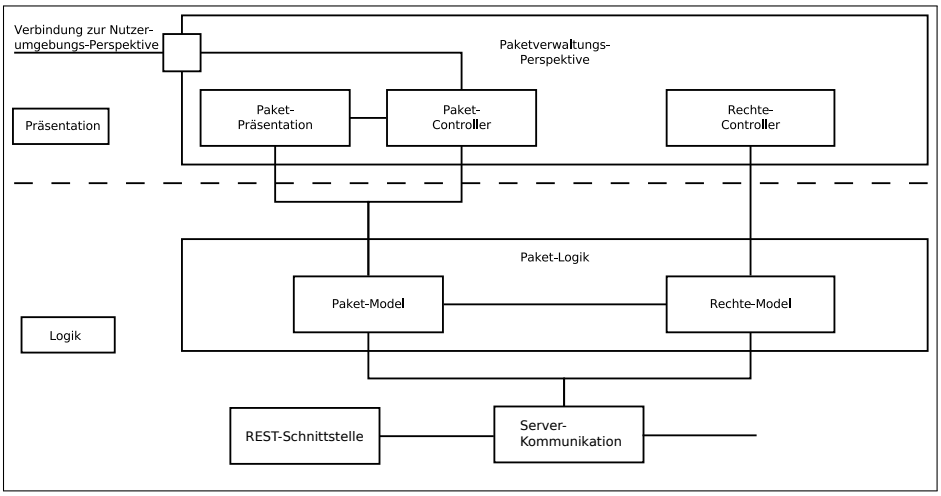
Modellerstellung:



In diesem Diagramm wird der clientseitige Anteil der Modellerstellung unter Benutzung des Model-View-Controller-Patterns näher ausgeführt (vgl. FR001, FR021).

In der Präsentationsschicht liegen verschiedene Visualisierungen für die Modellerstellung, den Datensatz und die Algorithmen, welche durch ihren jeweiligen Controller gesteuert werden. Die Daten werden von drei zugehörigen Modellen in der Logik-Schicht verwaltet, welche alle miteinander kommunizieren können. Dadurch haben die Modelle jeweils auch Zugriff auf die anderen beiden Controller. Außerdem können sich die Controller auch untereinander austauschen. Die drei Modelle leiten ihre Informationen in die Datenschicht des Clients weiter.

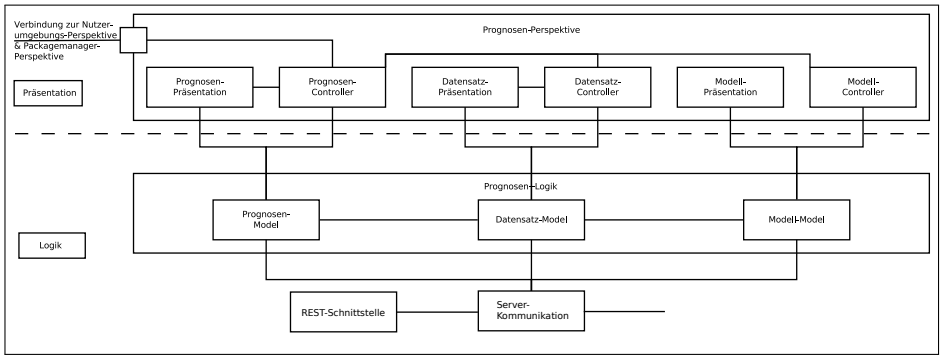
Pakete:



In diesem Diagramm wird der clientseitige Anteil der Paketverwaltung unter Benutzung des Model-View-Controller-Pattern näher ausgeführt (vgl. FR011, FR027).

In der Präsentationsschicht liegt die Visualisierung der Pakete. Änderungen an den Paketen werden vom Paketcontroller prozessiert und an das Paket-Modell gesendet. Das Paket-Modell kooperiert mit dem Rechte-Modell, welches aber lediglich über einen Rechte-Controller und keine View verfügt. Anhand der Änderungen auf dem Rechte-Modell kann der Rechte-Controller, welcher mit dem Paket-Model kommunizieren kann, zum Beispiel dafür sorgen, dass einem Nutzer ohne Berechtigung der Zugriff auf das Paket „ausgegraut“ wird und er somit keinen Zutritt hat (vgl. FR013).

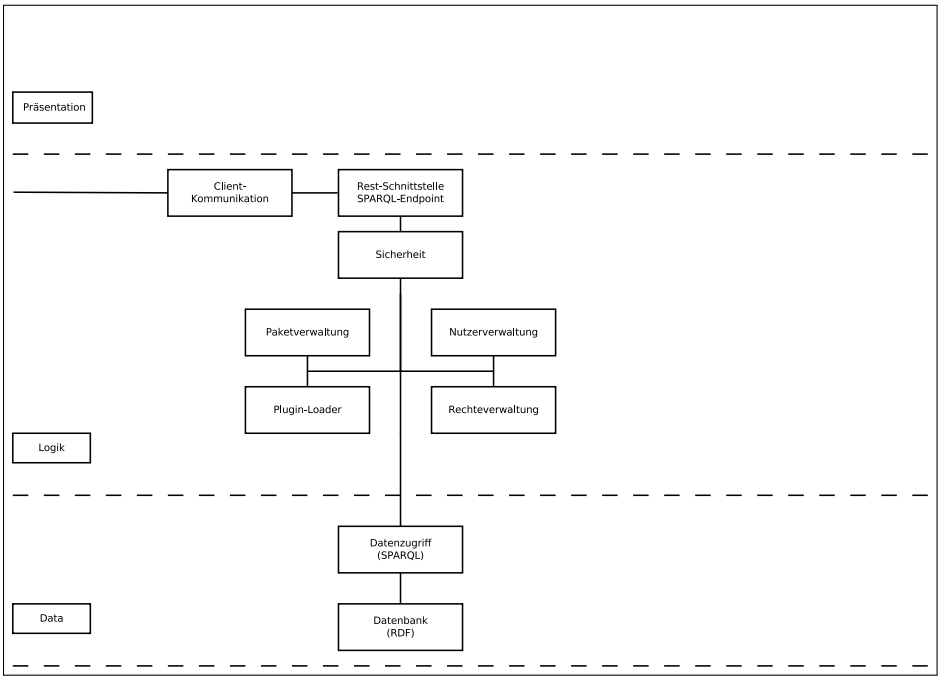
Prognosen:



In diesem Diagramm wird der clientseitige Anteil der Prognosenerstellung unter Benutzung des Model-View-Controller-Pattern näher ausgeführt.

Diese ist ähnlich zur Modellerstellung. Es gibt die Visualisierung von Prognosen, Datensätzen und Modellen, welche erneut jeweils über einen Controller und ein zugehöriges Model verfügen und ebenfalls alle untereinander kommunizieren. Durch die Weiterleitung der Informationen der drei Modelle, insbesondere des Prognosenmodells, in die Datenschicht des Clients, wird gewährleistet, dass die Ergebnisse der Prognosen in der Datenbank gespeichert werden (vgl. FR009).

Server-Logik:



Auch die Logik des Servers kann durch ein Layer-Pattern beschrieben werden. Wie auch beim Client bietet sich hier eine Modularisierung der Aufgaben an, um die oben genannten Vorteile, wie erheblich einfachere Entwicklung und Wartung, zu erreichen.

Die Schicht am weitesten oben in der Logik ist die REST-Schnittstelle und der SPARQL-Endpoint. Die REST-Schnittstelle erhält die Anfragen der Client-Logik, interpretiert diese richtig und reicht diese Informationen daraufhin eine Schicht weiter nach unten in die Sicherheit (vgl. FR017, FR029).

In dieser Schicht wird geklärt, ob es sich um eine legitime Anfrage handelt. Dies ist eine weitere Stärke des Layer-Pattern: Komponenten können über Zwischenschichten von einander getrennt werden, um sicherzustellen, dass bestimmte Ebenen nicht direkt miteinander kommunizieren können. Das ist in diesem Fall genau unser Bestreben, da es sinnvoll ist, zunächst die Anfragen auf ihre Legitimität zu prüfen, bevor man sie unkontrolliert in die Kern-Logik, genauer gesagt die verschiedenen Verwaltungsstrukturen unseres Servers, weiterleitet. So kann beispielsweise frühzeitig geprüft werden, ob die hochgeladenen Dateien den unterstützenden Typen entspricht oder der Nutzer darauf hingewiesen werden, dass er ein Modell erstellt, das schon einmal mit identischen Daten und Parametern erstellt worden ist (vgl. FR002, FR015).

In der untersten Schicht unserer Server-Logik befindet sich die Kern-Logik. Sie beinhaltet fundamentale Funktionen des Systems.

Zum einen hat man die Paketverwaltung, welche sich um alles kümmert, was Pakete betrifft. So wird zum Beispiel nach der Neuregistrierung für den neuen Nutzer automatisch ein erstes Paket angelegt. Sie kooperiert mit der Nutzerverwaltung, der Rechteverwaltung und dem Plugin-Loader, um zum Beispiel Informationen über die Rechte von zugreifenden Nutzern zu erfahren (vgl. FR011, FR014, FR016, FR025, FR027, FR031).

Die Nutzerverwaltung ist ebenfalls mit den anderen drei Komponenten in der Kern-Logik verbunden. Sie kann zum Beispiel Rechte für Benutzer anfragen (beispielsweise auf welche Pakete der Nutzer Zugriff hat, wie viel Speicher und Rechenzeit ihm zur Verfügung steht) oder wird bei Login- und Registrierungsvorgängen benötigt (vgl. FR022, FR023, FR024, FR025, FR027).

Die Rechteverwaltung verwaltet, wie der Name bereits suggeriert, die Rechte von Nutzern und hilft den anderen Komponenten der Kern-Logik, falls diese Anfragen über Rechte von Nutzern stellen (vgl. FR013, FR031).

Der Plugin-Loader kümmert sich um das bereitstellen von Plugins oder Modulen für die anderen Komponenten der Kern-Logik (vgl. FR014, FR016, FR021, FR030).

In der Datenschicht der Logik des Servers ist eine SPARQL-Komponente für den Datenzugriff auf die darunterliegende Datenbank, welche auf dem RDF-Format operiert (vgl. FR029).

Szenario Login-Sequenz-Diagramm (Client Seite):

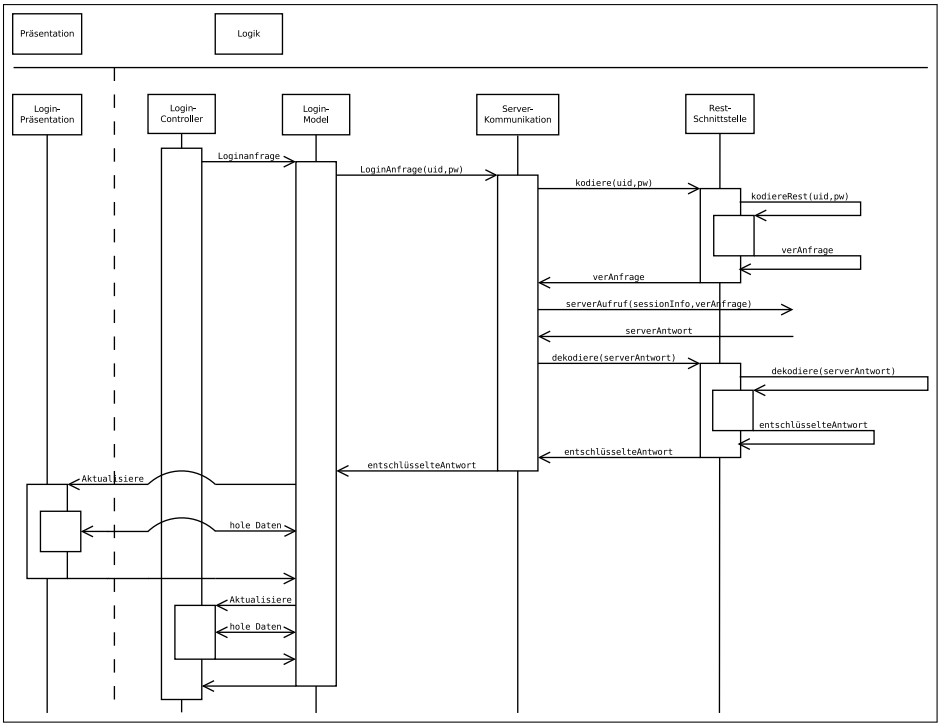


Abbildung 2.1: Ein Sequenzdiagramm, das beispielhaft einen Loginvorgang von der Clientseite aus zeigt.

Szenario Login-Sequenz-Digramm (Server Seite):

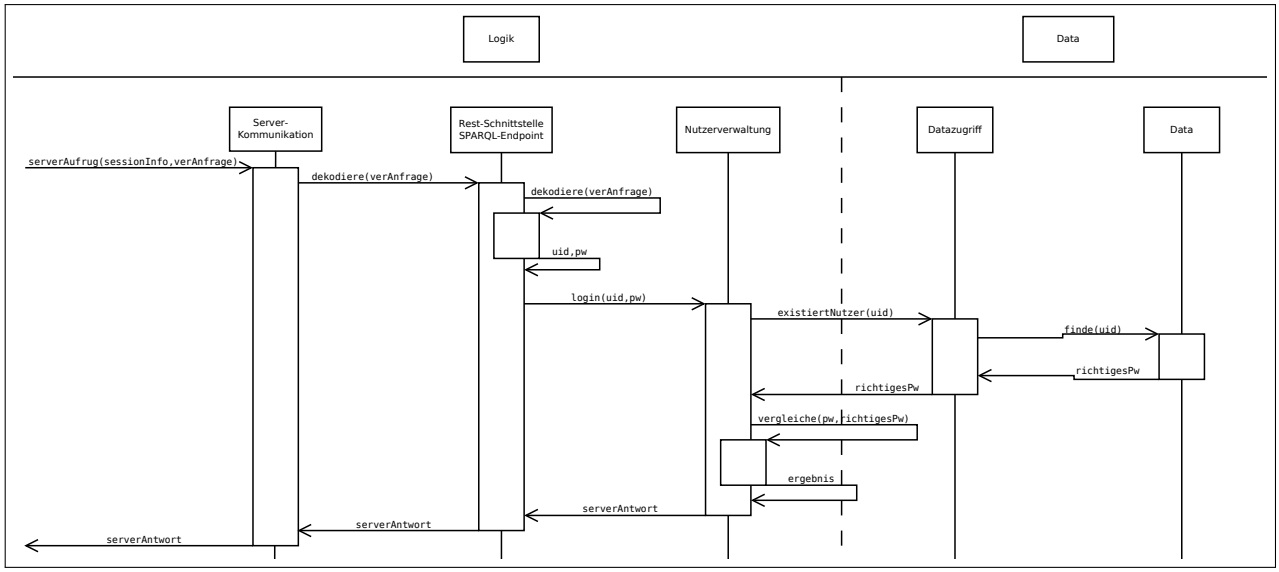
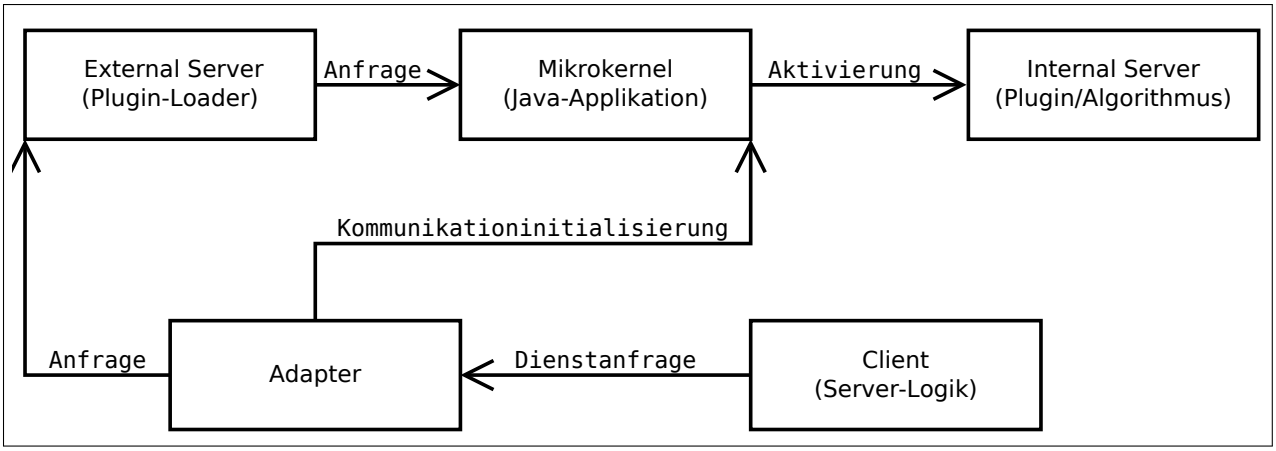


Abbildung 2.2: Ein Sequenzdiagramm das Beispielhaft einen Loginvorgang von der Serverseite aus zeigt.

Mikrokernel:



Das Mikrokernel-Pattern kommt in Systemen zum Einsatz, die ihre Anforderungen dynamisch, zum Beispiel durch das Hinzukommen von neuen Komponenten wie Erweiterungen (Plugins) oder ähnlichem ändern. Dabei ist die Idee, den funktionalen Kern des Systems so klein wie möglich zu halten und Funktionen, die lediglich ab und zu auf die Kernfunktionalität zugreifen müssen, auszulagern. Diese Kernfunktionalität sorgt für die korrekte Einbindung und Kollaboration von Erweiterungen.

Das Mikrokernal-Pattern besteht aus den fünf Komponenten Mikrokernel, External Server, Internal Server, Adapter und Client. Dabei übernehmen die einzelnen Komponenten folgende Aufgaben:
Mikrokernel:

- stellt Kernfunktionalitäten und Kommunikationseinrichtungen bereit
- kapselt Systemabhängigkeiten
- verwaltet und kontrolliert Systemressourcen

External Server:

- stellt Programmierschnittstelle für seine Clients bereit

Internal Server:

- implementiert zusätzliche Funktionen
- kapselt einige Systemspezifikationen
- lädt zur Laufzeit des Systems Algorithmen
- intern mittels Reflection Pattern umgesetzt

Adapter

- verbirgt Systemabhängigkeiten wie Kommunikationseinrichtungen vom Client
- aktiviert Methode auf dem External Server im Auftrag des Clients

Client:

- repräsentiert eine Anwendung der Logik
- übergibt eine Aufgabe an den Adapter

In unserem System kommt das Mikrokernel-Pattern im Zusammenhang mit den Algorithmen der WEKA-Bibliothek vor.

Der Internal Server wird bei uns durch die eigentliche Bibliothek von Algorithmen repräsentiert, welche auf Anfrage den richtigen Algorithmus bereitstellt (vgl. FR004, FR007).

Der External Server ist der Teil des Systems, auf dem die eigentliche Anwendung des Algorithmus geschieht (vgl. FR020).

Gesteuert werden die Prozesse vom Mikrokernel, welcher durch unsere JAVA-Implementierung repräsentiert wird.

Sendet also der „Client“, in unserem Fall die Server-Logik, welche die Daten aus dem Client erhält, eine Anfrage, wird diese vom Adapter interpretiert. Der Adapter sendet die Anfrage weiter an den External Server und initialisiert außerdem die Kommunikation zwischen dem Mikrokernel und dem External Server.

Der External Server fragt daraufhin die Funktionalität des Mikrokernels an, welcher dafür sorgt, dass die Anfrage mithilfe der Erweiterungen des Internal Servers auf dem External Server prozessiert wird.

Reflection

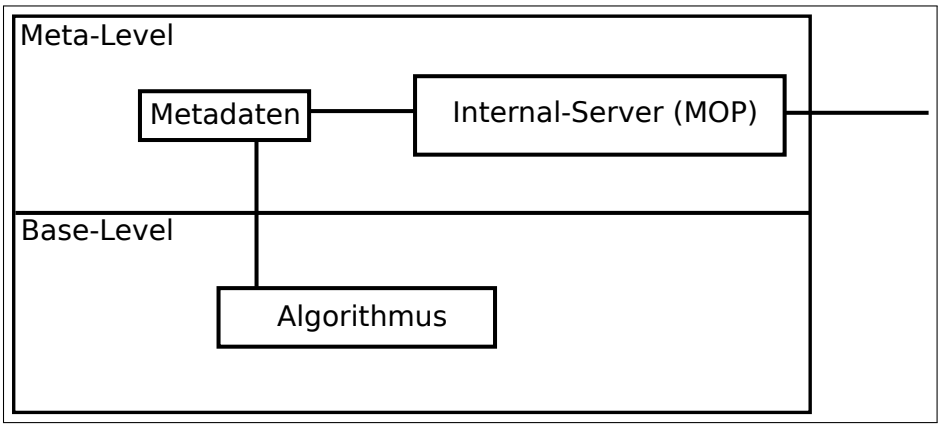


Abbildung 2.3: Vereinfachte Version des Reflection Patterns.
Das Reflection Pattern wird verwendet um dynamisch dem System zu Beginn unbekannte Algorithmen zu laden und dessen mögliche Parameter auszulesen. Der Internal Server des Mikrokernel agiert in dieser Version des Patterns als Meta-Object-Protocol und ist die Instanz, die Zugriff auf die einzelnen Algorithmen hat.

Mikrokernel-Sequenzdiagram:

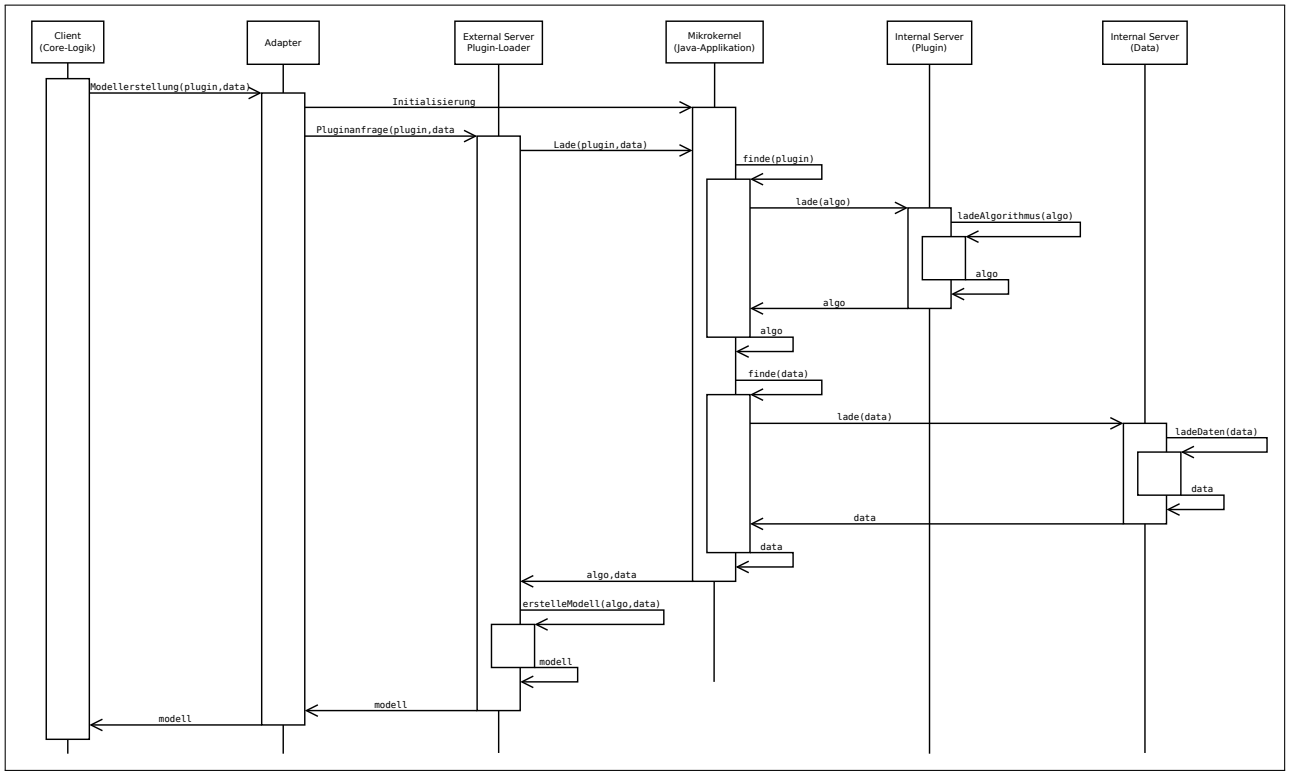


Abbildung 2.4: Ein Sequenzdiagramm, das eine beispielhafte Anfrage eines Clients zeigt.

Kapitel 3

Nicht benutzte Pattern

Blackboard
Da unser System offensichtlich komplett deterministisch arbeitet, wird kein Pattern für nicht-deterministische Anwendungen benötigt.

Broker
Das Broker-Pattern würde im System eine Anwendung finden, wenn die Bearbeitung der Anfragen des Nutzers auf verschiedene Server aufgeteilt werden würde. Jedoch wird in unserem System die Anfrage des Servers stets an den Hauptserver weitergeleitet, was eine Aufteilung der Nutzeranfrage nicht erforderlich macht.

Pipes and Filters
Das Pipes-and-Filters-Pattern wird genutzt, um in einem System einen Strom von Daten zu prozessieren. Dabei werden die Daten durch eine „Röhre“ geleitet und auf dem Weg von einer beliebigen Anzahl von Filtern manipuliert. Eine mögliche Anwendung in unseren System wäre die Verschlüsselung von Daten gewesen, zum Beispiel beim Login-Vorgang, was aber die einzige sinnvolle Verwendung für dieses Pattern gewesen wäre. In der weiteren Entwicklung des Systems, wären keine weiteren Filter hinzugekommen, was das Pattern redundant gemacht hätte. Außerdem kann jeder Filter nur ein Datenformat bearbeiten. Da in dem System verschiedenen Datenformate hochgeladen werden können (vgl. FR002, FR021), hätten Transformationen der Datenformate stattfinden müssen, was zu einer vermeidbaren Belastung des Servers geführt hätte.

Presentation-Abstraction-Control
Das Presentation-Abstraction-Control-Pattern teilt das System in die drei Einheiten Presentation, Abstractation und Control auf. Diese Aufteilung ist Ähnlich der Aufteilung View, Model und Control im Model-View-Controller-Pattern, das in diesem System diese Aufgabe übernimmt. Zudem wird im Presentation-Abstraction-Control-Pattern eine Hierarchie verschiedener Teile angelegt, die alle eine Aufgabe des Systems übernehmen. Dies wird in unserem System bereits

durch das Layer-Pattern realisiert. Somit wird das Presentation-Abstraction-Control-Pattern nicht benötigt.

Repository
Wenn die Kommunikation nur über das Repository laufen würde, würde dies einen Kommunikationsfalschenhals darstellen, was negative Auswirkungen auf die Laufzeit des System hätte. Deswegen wurde das Repository Pattern nicht verwendet.