Technische Hochschule Ingolstadt

Fakultät Wirtschaftsingenieurwesen
Studiengang Wirtschaftsingenieurwesen

Masterarbeit

Thema: Entwicklung eines Frameworks für die serverseitige Steuerung eines Cobots auf Grundlage von REST am Beispiel einer ontologiebasierten KI-Anwendung

Vor- und Zuname: Dennis Pal

Matrikelnummer: 00104973

ausgegeben am: 12.09.2024
abgegeben am: 08.11.2024

Erstprüfer: Prof. Dr. Jürgen Bock
Zweitprüfer: Prof. Dr.-Ing. Roland Meyer

Abstract

Im Labor für Produktionstechnik der Technischen Hochschule Ingolstadt ist ein Cobot KUKA LBR iiwa installiert. Für dieses Modell wurde bereits eine REST-API entworfen und auf der Robotersteuerung implementiert. Im Rahmen der vorliegenden Arbeit wurde dieser Client um einen geeigneten Server ergänzt. Auf Grundlage einer Anforderungsanalyse wurde dafür eine Plug-in-Architektur entworfen, welche die Integration verschiedener Steuerungssysteme mit KI-Funktionen für den Roboter ermöglicht. Die Applikation wurde in Python implementiert und abschließend anhand einiger beispielhafter Plug-ins im Zusammenspiel mit dem Roboter getestet.

Die Applikation ist verfügbar unter: https://github.com/EngDePal/REST_AI_Server

A KUKA LBR iiwa cobot is installed inside the laboratory for production technology at Technische Hochschule Ingolstadt. Previously a REST-API has been designed and implemented on the robot controller. This thesis aims to develop the appropriate server to this client. Following an initial requirements analysis a plug-in architecture was designed, to allow for the integration of different control systems with an option for AI functionality. The application was implemented in Python and tested on the robot using some exemplary plug-ins.

The application is available at: https://github.com/EngDePal/REST_AI_Server

This work was conducted using Protégé.

Danksagung

Ein großer Dank gebührt meinem Betreuer Prof. Dr. Jürgen Bock, welcher mich trotz vieler Startschwierigkeiten bis zur Abgabe meiner Masterarbeit unterstützt hat. Letztendlich durfte ich mich also doch noch erfolgreich in ein für mich vollkommen neues und spannendes Thema einarbeiten. Ebenfalls bedanken möchte ich mich bei Prof. Dr.-Ing. Roland Meyer für seine Rolle als Zweitprüfer und bei Thomas Flieger für die Betreuung im Labor. Unersetzlich war auch die Vorarbeit meines Kommilitonen Andreas Ziegltrum, welcher den Grundstein für dieses Projekt gelegt hat. In diesem Sinne bin ich äußerst gespannt, wie sich „unsere“ gemeinsame Anwendung durch den Einsatz zukünftiger Studenten weiterentwickeln wird.

Eigenständigkeitserklärung nach § 30 Abs. 4 Nr. 7 APO THI

Ich erkläre hiermit, dass ich die Arbeit selbstständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen oder Hilfsmittel benützt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Ingolstadt, _____

_____

Dennis Pal

Abkürzungsverzeichnis

API	Programmierschnittstelle
CIRC	Kreisförmige Roboterbewegung
CORA	Core Ontology for Robotics and Automation
DB	Datenbank
DM	Data Manager
FA	Funktionale Anforderung
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
IoT	Internet of Things
JSON	Java Script Object Notation
KG	Knowledge Graph
KI	Künstliche Intelligenz
LIN	Linearbewegung
MQTT	Message Queuing Telemetry Transport
MRK	Mensch-Roboter-Kollaboration
NFA	Nichtfunktionale Anforderung
OPC UA	Open Platform Communications Unified Architecture
OWA	Open World Assumption
OWL	Ontology Web Language
PTP	Point-to-Point
RDF	Resource Description Framework
REST	Representational State Transfer
RLM	Robot Logic Manager
ROS	Robot Operating System
S	Soll-Anforderung

SQL	Structured Query Languague
THI	Technische Hochschule Ingolstadt
TM	Token Manager
UI	User Interface
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
W	Wunsch(-Anforderung)
WWW	World Wide Web

Inhaltsverzeichnis

Danksagung I

Eigenständigkeitserklärung II

Abkürzungsverzeichnis III

Inhaltsverzeichnis V

1 Einleitung 1

2 Technische Grundlagen 4

2.1 Robotik 4

2.1.1 Verbreitung und Varianten 4

2.1.2 Kinematik 4

2.1.3 Mensch-Roboter-Kollaboration 6

2.2 Softwaredesign 7

2.2.1 Architektur 7

2.2.2 REST-API 8

2.2.3 HTTP-Verben 9

2.2.4 Struktur der Client-Server-Kommunikation 11

2.3 Künstliche Intelligenz 13

2.3.1 Der Begriff KI 13

2.3.2 Ontologien 14

2.3.3 Ontology Web Language 16

2.3.4 KI in der Robotik 17

2.4 Verwandte Arbeiten 18

3 Methodik 19

4 IST-Analyse 22

5 Anforderungsmanagement 25

5.1 Server 25

5.2 KI-Applikation 26

5.3 Systemarchitektur 26

5.4 Testing 27

5.5 Release 27

5.6 Rahmenbedingungen ... 28
6 Entwurf der Architektur ... 30
7 Technology Stack ... 33
8 Implementierung des Servers ... 35
8.1 Übersicht der Module ... 35
8.1.1 Server ... 36
8.1.2 TokenManager ... 36
8.1.3 DataManager ... 37
8.1.4 RobotLogicManager ... 37
8.1.5 CockpitWidget ... 38
8.2 Design der Plug-in-Architektur ... 40
8.3 Betrachtung des Programmablaufs ... 43
8.3.1 Hochfahren des Servers ... 43
8.3.2 Login ... 43
8.3.3 Bereitstellung eines Befehls ... 45
8.3.4 Bestätigung und Generierung eines Befehls ... 46
8.3.5 Übermittlung von Roboterdaten ... 48
8.3.6 Abschalten des Servers ... 48
9 Implementierung der KI-Applikation ... 49
9.1 Analyse der Domäne und Plug-in-Anforderungen ... 49
9.2 Entwurf der Ontologie ... 51
9.3 Integration der Ontologie in ein Server-Plug-in ... 53
10 Integrationstest im Labor ... 57
10.1 Planung des Experiments ... 57
10.2 Durchführung und Ergebnisse des Tests ... 59
11 Bewertung der Applikation ... 61
11.1 Verifizierung ... 61
11.2 Validierung ... 63
12 Fazit und Ausblick ... 66
Abbildungsverzeichnis ... VIII

Tabellenverzeichnis IX

Literaturverzeichnis X

Quellcode XV

1 Einleitung

Heutzutage sind Roboter vielerorts fester Bestandteil der industriellen Produktion. Aufgrund ihrer Fähigkeit vordefinierte Bewegungsabläufe auch in Gefahrensituationen fehlerfrei und unermüdlich zu wiederholen, können sie menschliche Arbeitskräfte in geeigneten Anwendungsfällen ersetzen [1, S. 112]. Ein anschauliches Beispiel stellt hierbei die Automobilindustrie dar. Insbesondere im Karosseriebau, in dem schwere Bauteile bei einem Produktionsvolumen von mehreren hundert Stück pro Tag mit hoher Präzision an einer Vielzahl von Schweißpunkten zusammengefügt werden müssen, sind Industrieroboter in großer Stückzahl vorzufinden.

Vollständig übernommen haben Roboter menschliche Arbeitsplätze bisher jedoch nicht. Im Kontext des Fahrzeugbaus wären die hohen Stückzahlen nicht ohne den breiten Einsatz von Personal in den Montagelinien der Hersteller denkbar. Die Komplexität, welche durch eine Vielzahl an Teilen, Modellen und Einbaupositionen entsteht, erfordert ein Maß an Flexibilität und Lernfähigkeit in der Entscheidungsfindung, welches in dieser Form derzeit nur Menschen einbringen können [2, S. 235].

Stattdessen entwickelt sich schon seit einigen Jahren der Trend, Roboter nicht als Ersatz, sondern als Unterstützung für Mitarbeiter in solche Prozesse zu integrieren. Aufgrund der besonderen Anforderungen, welche aus diesen Anwendungen resultieren, hat sich eine neue Klasse von Industrierobotern entwickelt: die Cobots. Diese kollaborativen Roboter weisen insbesondere einen erhöhten Umfang an Sicherheitsfunktionen auf, um im Gegensatz zu ihren klassischen Gegenstücken in unmittelbarer Nähe der Mitarbeiter betrieben zu werden. Dies wird als Mensch-Roboter-Kollaboration (MRK) bezeichnet. Auch die THI verfügt im Produktionslabor über einen Cobot des Typs KUKA LBR iiwa 7 R800. [1, S. 127]

Einen sinnvollen Einsatz von Cobots zur Unterstützung von Mitarbeitern mit Behinderungen oder gesundheitlichen Probleme demonstriert das Ford Werk in Köln (vgl. Abbildung 1). Die dortige Montage von Motorabdeckungen erfordert eine hohe Präzision und ist einigen Mitarbeitern aufgrund eingeschränkter Mobilität in der Schulter oder im Handgelenk nur schwer möglich. Nun stellen diese dem Roboter Teile bereit, welcher sie präzise und mit der nötigen Kraft einsetzt. Der Mitarbeiter verschraubt die Komponenten anschließend. [3]

Neben MRK kann das Forschungsfeld der Künstlichen Intelligenz (KI) weitere Ansätze liefern, um die Einsatzmöglichkeiten von Industrierobotern zu erweitern. Prinzipiell ist der kombinierte Einsatz zahlreicher Technologien wie Ontologien oder Machine Learning

denkbar. Um sowohl im Training als auch bei Inferenz im Betrieb Ergebnisse in einem akzeptablen Zeitrahmen zu erreichen, ist leistungsstarke Hardware notwendig.

Abbildung 1 Mitarbeiter im Ford Werk Köln mit einem KUKA LBR iiwa Cobot
Bildquelle: [3]

Roboter nutzen zur Ausführung von Programmen ihre eigene Steuerung. Für übliche Roboterprogramme, welche beispielsweise mit dem KUKA smartPAD erstellt werden, ist dies ausreichend. Für die genannten KI-Anwendungen kann die vorhandene Hardware jedoch einen Engpass darstellen.

Deshalb wurde in einer vorangegangen Bachelorarbeit eine Programmierschnittstelle (API) gemäß dem Architekturstil Representational State Transfer (REST) definiert. Diese wurde anschließend in Java auf der Robotersteuerung des KUKA LBR iiwa implementiert. Damit ist es dem Roboter nun möglich, in der Rolle eines Clients Daten mit einem Server auszutauschen. Folglich kann die gesamte Befehlslogik nun auf Letzteren ausgelagert werden, welcher einzelne Kommandos definiert und diese dem Client anschließend über die API zur Verfügung stellt.

Diesem Ansatz folgend wird im Rahmen der vorliegenden Arbeit ein KI-Server entworfen und implementiert. Dieser besteht zunächst aus einer Kernapplikation, welche den Datenaustausch regelt. Zusätzlich soll die Architektur neben diesem Kern die Erweiterbarkeit durch Dritte mit ihren eigenen KI-Anwendungen ermöglichen.

Als Beispiel für eine solche Erweiterung wird ebenfalls eine konkrete Anwendung in Form einer Ontologie erstellt. Diese soll Arbeitsgänge in Form von Subprozessen und Bedingungen modellieren. Dadurch ist ein *Reasoner* in der Lage aus den verschiedenen Bedingungen eine Reihenfolge der Einzelschritte abzuleiten, ohne dass diese wie bei der Roboterprogrammierung üblich explizit angegeben ist. Somit zielt die Anwendung darauf ab, die Steuerung im Vergleich zu statischen Programmen zu flexibilisieren.

Abschließend wird das Gesamtsystem verifiziert und validiert. Unter anderem wird die Integration von KI-Anwendung, Server und Client im Produktionslabor unter Einsatz des

Roboters getestet, um dessen Funktionalität sicherzustellen. Hierbei wird auch das Potenzial des ontologiebasierten Ansatzes zur Robotersteuerung experimentell evaluiert.

2 Technische Grundlagen

Die vorliegende Abschlussarbeit beschäftigt sich mit drei wesentlichen Themenbereichen: Robotik, Softwaredesign und Künstlicher Intelligenz. Diese unterschiedlichen Teilbereiche werden hinsichtlich des aktuellen Stands der Technik beleuchtet. Ebenfalls werden verwandte Ansätze zur Robotersteuerung in Industrie und Forschung vorgestellt.

2.1 Robotik

2.1.1 Verbreitung und Varianten

Auch in Hinblick auf die demographischen Herausforderungen in den hochentwickelten Volkswirtschaften sind Industrieroboter von unschätzbarem Wert. So ist Deutschland neben Südkorea und Singapur gemessen an der Roboterdichte (Anzahl der Roboter pro 10.000 Mitarbeiter in der fertigenden Industrie) eines der drei Länder mit dem höchsten Automatisierungsgrad auf der Welt. Das Wachstum der globalen Roboterdichte von 2017 bis 2022 um 12% jährlich deutet auf deren steigende Bedeutung hin. [4]

Die beiden wichtigsten Kunden für Roboterhersteller sind mit großem Abstand die Elektronik- und die Automobilindustrie [5, S. 12]. Über Branchengrenzen hinweg stellen Handling, Schweißen, Montage, der Einsatz in Reinräumen und die Dosierung von Material die häufigsten Anwendungsbereiche dar [5, S. 14].

Abhängig von den individuellen Anforderungen eignen sich einige Robotertypen für eine bestimmte Anwendung besser als andere. So sind beispielsweise SCARA-Roboter bei Pick-and-Place Aufgaben sehr schnell. Den flexibelsten Typ stellen jedoch jene mit Knickarm-Kinematik wie der Cobot KUKA LBR iiwa im Labor der THI dar. Da der dortige Bestand ausschließlich dieser Kategorie angehört, wird im weiteren Verlauf nur diese Kinematik betrachtet und vereinfacht als Industrieroboter bezeichnet.

2.1.2 Kinematik

Für die Fernsteuerung eines Industrieroboters über einen KI-Server ist ein Grundverständnis der Roboterkinematik notwendig. Generell haben die meisten Industrieroboter sechs Achsen, wobei zusätzliche Achsen zu einer Verbesserung der Feinbewegungen im Austausch gegen einen erhöhten Steuerungsaufwand führen [1, S.

65]. Ein Beispiel für letzteren Typ ist der bereits genannte KUKA LBR iiwa mit sieben Achsen [6].

Für den Roboter wird ein Weltkoordinatensystem in der Roboterbasis definiert [1, S. 104], wobei auch jedes Armglied ein eigenes Koordinatensystem erhält (vgl. Abbildung 2). Roboterbewegungen werden durch eine Reihe von Koordinatentransformationen entlang der Bewegungskette berechnet. Bei einer Vorwärtsrechnung wird aus den Gelenkstellungen beginnend an der Basis die Handstellung im Weltkoordinatensystem berechnet. Dagegen werden bei der Rückwärtsrechnung aus der gewünschten Handstellung die korrekten Drehwinkel berechnet. [1, S. 66]

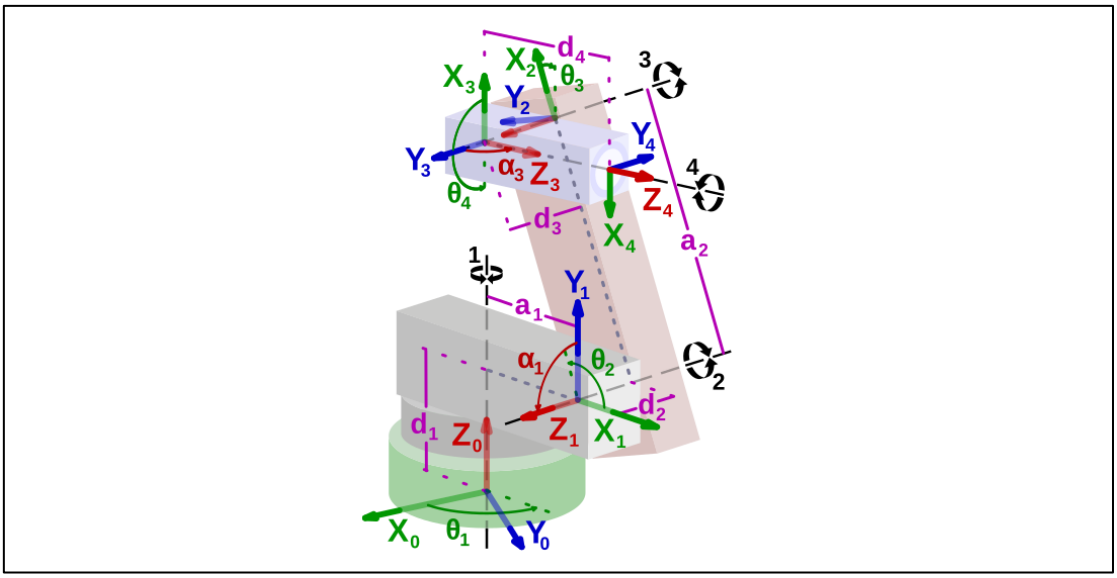

Abbildung 2 Bewegungskette eines Roboters mit eingezeichneten Koordinatensystemen
Bildquelle: [7]

Bei der Programmierung des Roboters wird einem Punkt im Raum ebenfalls ein eigenes Koordinatensystem, auch Frame genannt, zugewiesen, welcher dessen Position und Orientierung beschreibt [8, S. 21]. Ein Frame besteht aus sechs Informationen: die Werte X, Y und Z beschreiben die Verschiebung entlang der Achsen eines Referenzkoordinatensystems in Millimetern. Die Werte A, B und C beschreiben hingegen die Rotation des Frames in Grad um die Achsen des Referenzkoordinatensystems. [8, S. 103]

Um einen Zielpunkt anzufahren, sind verschiedene Bewegungsarten verfügbar (vgl. Abbildung 3). KUKA definiert drei grundlegende Varianten: Point-to-Point (PTP), eine lineare Bewegung (LIN) und eine Kreisbahn (CIRC). PTP führt den Industrieroboter auf der schnellsten Bahn zum Ziel, welche in der Regel keine Gerade ist. LIN dagegen verfährt den Arm entlang einer Gerade und somit auf der kürzesten Strecke im Raum.

Die Kreisbahn bei CIRC wird durch einen Startpunkt, einen Hilfspunkt und das Ziel definiert. [8, S. 385-387]

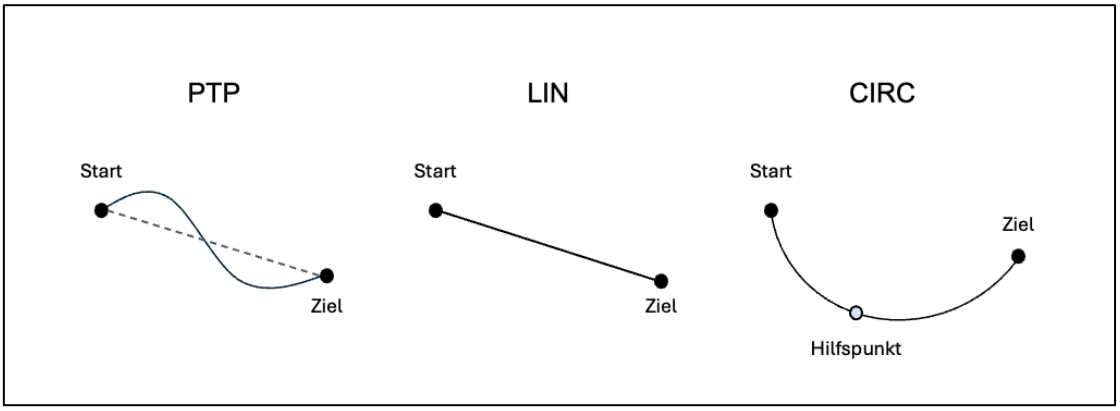

Abbildung 3 Bewegungsarten bei KUKA-Robotern
In Anlehnung an [8, S. 385-387]

2.1.3 Mensch-Roboter-Kollaboration

Wie einleitend beispielhaft skizziert (vgl. Abbildung 1), stellt MRK einen großen Trend im Bereich der Automatisierung dar. Der globale Bestand an Cobots wächst stark und diese Produktkategorie erreichte 2022 einen Marktanteil von 10% [5, S. 15]. Während traditionelle Industrieroboter hinter Schutzzäunen arbeiten, interagieren Cobots direkt mit Mitarbeitern. Ermöglicht wird dies durch eine Vielzahl konstruktiver und funktionaler Designentscheidungen.

Zunächst weisen diese Roboter ein deutlich verbessertes Verhältnis von Traglast zu Eigengewicht auf. Die geringere Masse senkt das Verletzungsrisiko beträchtlich. So weist der LBR iiwa 7 R800 eine maximale Traglast von 7 kg bei einem Gewicht von lediglich 23,9 kg auf [9]. Eine weitere Reduktion erfolgt durch ein meist abgerundetes und abgeschlossenes Design (vgl. Abbildung 1). [10, S. 3]

Neben der Konstruktion spielt auch die verbaute Sensorik eine große Rolle. Beispielsweise ist der LBR iiwa dank der Drehmomentsensoren in seinen Gelenken in der Lage, Zusammenstöße zu erkennen und sowohl seine Kraft als auch Geschwindigkeit sofort zu reduzieren. Dies schützt nicht nur Mitarbeiter vor Verletzungen, sondern verschafft dem Roboter auch Vorteile bei der Arbeit mit filigranen Teilen. [6]

2.2 Softwaredesign

2.2.1 Architektur

"Architecture is the fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution."
[11]

Im Rahmen des Softwareentwicklung wird zunächst eine Analyse zur Ermittlung der Anforderungen an ein System durchgeführt. Dabei wird zwischen funktionalen Anforderungen, welche die Funktionalität des Programms beschreiben, und nicht-funktionalen Anforderungen unterschieden, welche beispielsweise die Performance und geltende Einschränkungen abdecken [12, S. 4].

Eine wichtige Rolle nimmt hierbei die Software-Architektur ein. Als Abstraktion eines Systems beschreibt sie dessen Anwendungsfunktionen sowie deren Interaktionen. Dadurch ist sie im Wesentlichen für nicht-funktionale Anforderungen verantwortlich. Sie stellt das Ergebnis der Entwurfsphase im Entwicklungsprozess dar. [12, S. 3]

Durch eine gute Architektur wird die spätere Implementierung enorm erleichtert. Dies gilt insbesondere für große und modulare Projekte. Wichtige Prinzipien sind hierbei *Loose Coupling* und *Strong Cohesion*. Ersteres beschreibt eine lose Kopplung zwischen unterschiedlichen Software-Modulen, letzteres eine starke Bindung innerhalb dieser Bestandteile. Somit werden ausschließlich relevante Daten ausgetauscht und die innere Logik versteckt. Dieses Prinzip ist als *Information Hiding* bekannt. [12, S. 5, S. 34]

In der Literatur werden zahlreiche Architekturmuster mit ihren jeweiligen Vor- und Nachteilen beschrieben. Besonders interessant hinsichtlich ihrer flexiblen Erweiterbarkeit ist die sogenannte Plug-in-Architektur (vgl. Abbildung 4), welche im Kontext von Betriebssystemen auch als Mikrokernel-Architektur bezeichnet wird. Diese ermöglicht einer Kernapplikation im laufenden Betrieb ergänzende Softwaremodule zu laden. Für eine Implementierung sind ein Plug-in-Manager zur Verwaltung der Erweiterungen und die Definition von Erweiterungspunkten notwendig. Dadurch ist lediglich Kenntnis über die Schnittstelle erforderlich, um eigene Plug-ins zu entwickeln. [12, S. 348-350], [13, S. 21-22]

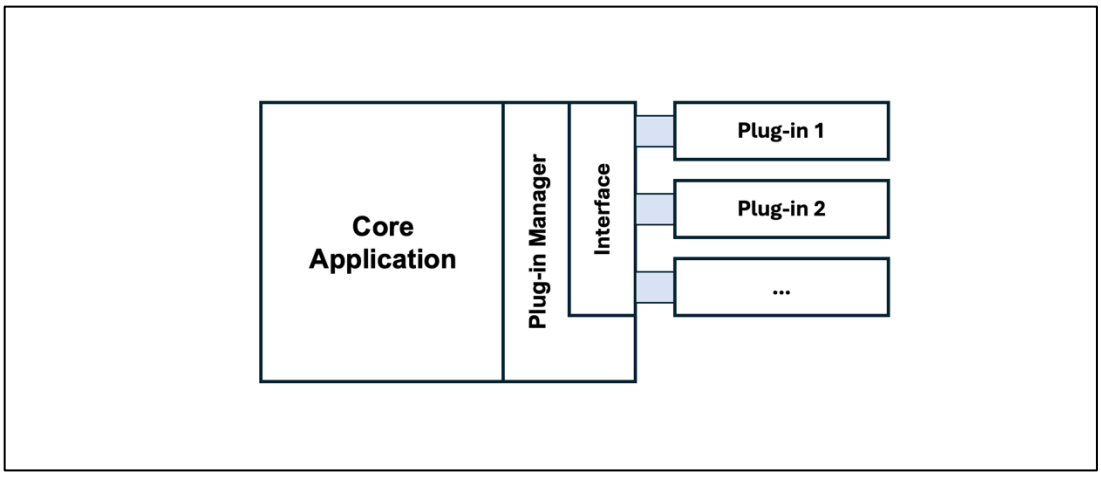

Abbildung 4 Schematische Darstellung der Plug-in-Architektur
In Anlehnung an [13, S. 22] und [12, S. 347]

2.2.2 REST-API

In vielen Softwareprojekten wie auch in dieser Arbeit besteht das Gesamtsystem aus mehreren Modulen. Dafür werden sogenannte APIs definiert, welche Regeln für die Kommunikation zwischen den verschiedenen Bestandteilen definieren. Somit ermöglichen sie auch die Erweiterbarkeit bestehender Software, indem lediglich relevante Daten übermittelt werden, während das Innenlebens des Systems gemäß dem Prinzip *Information Hiding* versteckt ist. [14]

Einen weit verbreiteten Architekturstil für APIs stellt REST dar. Dieser wurde im Rahmen der Entwicklung des World Wide Web (WWW) erstmals im Jahr 2000 von Roy Fielding beschrieben. REST basiert auf einigen wesentlichen Grundprinzipien, wobei die für diese Arbeit Relevantesten im Folgenden erläutert werden (vgl. Abbildung 5, in blau gekennzeichnet). Aus Verständnisgründen werden englische Fachbegriffe verwendet, wie sie auch in der Literatur Anwendung finden.

Die Grundlage bildet eine Client-Server-Architektur. Hierbei sendet ein Client Requests an den Server, welcher diese anschließend verarbeitet und eine Response zurückschickt [15, S. 78].

Da der Server keinen Kontext für die Verarbeitung zukünftiger Requests speichert, werden REST-APIs als zustandslos bezeichnet [16]. Für die korrekte Verarbeitung der Requests müssen die vom Client gesendeten Daten ausreichen. Zustandslosigkeit trägt zur Skalierbarkeit und Flexibilität der Applikationen bei [17].

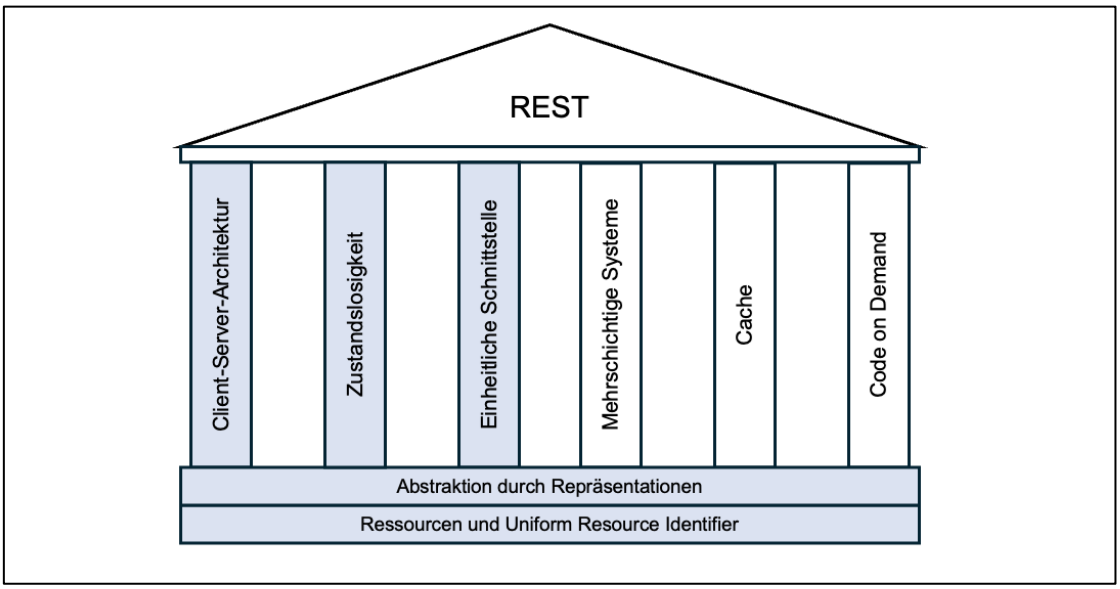

Abbildung 5 Prinzipien des REST-Architekturstils
In Anlehnung an [15, S. 78-92]

Ein weiteres wesentliches Prinzip dieses Architekturstils ist zudem die Definition einer einheitlichen Schnittstelle. Obwohl der standardisierte Datenaustausch unabhängig von den Bedürfnissen einer spezifischen Applikation nicht besonders effizient ist, ermöglicht diese Entkopplung die individuelle Weiterentwicklung von Client und Server. [15, S. 81-82]

In REST werden Daten in Form von Ressourcen abstrahiert, welche mit Hilfe von Uniform Resource Identifiers (URI) eindeutig identifiziert werden [15, S. 88-90]. Zur Interaktion mit diesen Ressourcen werden sogenannte Repräsentationen genutzt, welche konkrete Informationen und Metadaten zu deren Beschreibung enthalten. Es wird also nicht die Ressource an sich manipuliert, sondern deren Zustand in Form der Repräsentation. [15, S. 90-92]

Heutzutage bieten zahlreiche Onlineplattformen öffentlich zugänglich APIs an, welche Entwicklern eine Anbindung an ihre eigenen Projekte ermöglichen. Beispielhaft zu nennen ist etwa eBay, welches bereits 2002 als einer der Vorreiter auf diesem Gebiet tätig war [18].

2.2.3 HTTP-Verben

Der REST-Stil stellt eine Abstraktion dar, welche theoretisch mit beliebigen Protokollen umgesetzt werden kann. Eine konkrete Implementierung von REST auf Basis des

Hypertext Transfer Protocol (HTTP) wird auch als *RESTful HTTP* bezeichnet. [19, S. 10-11]

Eine einheitliche Schnittstelle für den Zugriff auf alle Ressourcen wird durch die sogenannten HTTP-Verben - auch Methoden genannt - festgelegt [19, S. 53]. Sind diese für alle Ressourcen korrekt implementiert, ermöglichen sie zahlreichen Clients Zugriff auf einen Server, was einen großen Vorteil dieses Architekturstils darstellt [19, S. 17]. Gemäß der Spezifikation für HTTP existieren acht Standardmethoden (vgl. Tabelle 1). Die beiden Verben *TRACE* und *CONNECT* werden aufgrund ihrer geringen Relevanz nicht betrachtet.

Zwei besondere Eigenschaften der Verben sind in Tabelle 1 hervorgehoben. Eine Standardmethode ist sicher, wenn keine Zustandsänderung durch den Client angefordert wird [19, S. 53-54]. Da zumindest aus Clientsicht lediglich eine Datenabfrage initiiert wird, fordert dieser also nicht explizit Nebenwirkungen an, welche potenziell gefährlich sein können. Ebenfalls ist eine Methode idempotent, wenn eine Anfrage unabhängig von der Anzahl an Ausführungen das gleiche Ergebnis liefert. [20]

Name	Beschreibung	Sicher	Idempotent
GET	Ruft eine Repräsentation unter der angegebenen URI ab	x	x
PUT	Aktualisiert eine Ressource oder erzeugt eine Repräsentation, falls unter der angegebenen URI noch keine vorhanden ist		x
POST	Anlegen einer neuen Ressource unter einer vom Server bestimmten URI. In der Praxis für die Implementierung beliebiger Funktionalität genutzt		
DELETE	Löschen einer Ressource unter der angegebenen URI		x
OPTIONS	Liefert Informationen über die unterstützten Methoden einer bestimmten Ressource	x	x
HEAD	Analog zu GET, jedoch werden ausschließlich Metadaten übertragen	x	x

Tabelle 1 Übersicht der HTTP-Verben
In Anlehnung an [20] und [19, S. 53-58]

Diese Spezifikation garantiert insbesondere vor dem Hintergrund des WWW ein erwartbares Verhalten jeder Standardmethode, welches die Entwicklung in diesem

Umfeld erleichtert. Die tatsächliche Sicherstellung dieser Eigenschaften ist jedoch Aufgabe des jeweiligen Entwicklers während der Implementierung.

Für die Kommunikation mit Industrierobotern existieren zahlreiche etablierte Technologien, welche sich als potenzielle Alternativen zur Entwicklung einer REST-API auf HTTP-Basis anbieten. Während der ursprünglichen Entwicklung der Schnittstelle wurden diese jedoch aus verschiedenen Gründen nicht betrachtet. Das Robot Operating System (ROS) ist ein Open-Source Projekt, welches für den KUKA LBR iiwa lediglich bis zu einer unvollständigen Beta-Version implementiert und seitdem nicht mehr weiterentwickelt wurde. Die Open Platform Communications Unified Architecture (OPC UA) und Message Queuing Telemetry Transport (MQTT) sind populäre Technologien im Bereich Internet of Things (IoT) [21]. Erstere ist wie ROS nicht für diesen Cobot implementiert, während letztere Größenbeschränkungen bei der Datenübertragung aufweist. [22, S. 10-11]

2.2.4 Struktur der Client-Server-Kommunikation

Neben REST und den HTTP-Verben ist eine Betrachtung der Kommunikationsstruktur zwischen Server und Client sinnvoll. Deren Nachrichten – Requests von Client zu Server und Responses in die entgegengesetzte Richtung – zeichnen sich gemäß HTTP-Spezifikation durch einen standardisierten Aufbau aus. Im Folgenden wird dieser für *HTTP 1.1* beschrieben, welches auch für die clientseitige Implementierung der REST-API verwendet wurde.

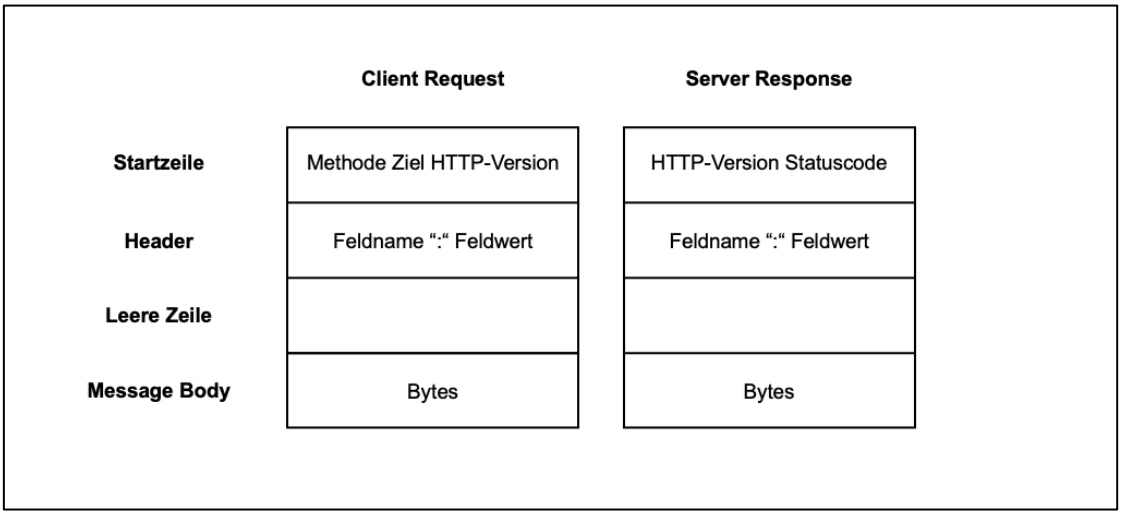

Abbildung 6 Aufbau von HTTP-Nachrichten
In Anlehnung an [23]

Abbildung 6 zeigt schematisch den Inhalt der beiden Nachrichtenarten auf. Diese beginnen mit einer Startzeile, deren Inhalt von der Übertragungsrichtung abhängig ist. Bei Requests enthält sie als *Request Line* das HTTP-Verb, das Ziel der Nachricht in Form einer URI und abschließend die HTTP-Version, in diesem Fall *HTTP/1.1*. Bei Responses wiederum beginnen sie als *Status Line* mit der Protokollversion, woraufhin ein Statuscode folgt. Dieser gehört zu einer von 5 Kategorien und beschreibt das Ergebnis einer Request (vgl. Tabelle 2) [20]. Optional ist eine sogenannte *Reason Phrase*, ein Erklärungstext zum Code. Der *Header* enthält mehrere Schlüssel-Wert-Paare, welche die Nachricht detaillierter beschreiben. Enthält diese einen *Message Body*, umfasst der Header ein sogenanntes *Content Length Field*, welches die Größe des Inhalts zur Bestimmung des Nachrichtenendes beschreibt. Der *Message Body* selbst stellt den eigentlichen Inhalt der Nachricht dar. [23]

Die Inhalte oder Repräsentationen können eine Vielzahl von Formaten annehmen. Es wird zwischen fünf grundlegenden Medientypen unterschieden: *Text*, *Image*, *Audio*, *Video* und *Application* [24]. In der Kommunikation mit dem Server führen Clients in ihren Requests einen *Accept-Header* an, welcher die akzeptierten Dateiformate spezifiziert, wohingegen ein *Content-Type Header* bei Responses sowie *POST* und *PUT*-Requests das tatsächlich gewählte Format wiedergibt [19, S. 87-88]. Codiert werden die Formate wie beispielsweise JSON nach dem Muster *application/json*. Der *Accept-Header* und weitere Felder ermöglichen die sogenannte *Content Negotiation* zur Ermittlung des besten Repräsentationsformats.

Code	Kategorie	Beschreibung
1xx	Informational	Request vom Server erhalten, in Verarbeitung
2xx	Successful	Request erfolgreich erhalten und verarbeitet
3xx	Redirection	Aktion erforderlich, um fortzufahren
4xx	Client Error	Request enthält fehlerhafte Syntax oder kann nicht erfüllt werden
5xx	Server Error	Interner Fehler des Servers bei korrekter Request

Tabelle 2 Übersicht der HTTP-Statuscodes
In Anlehnung an [20]

2.3 Künstliche Intelligenz

2.3.1 Der Begriff KI

Im Wesentlichen wird unter KI die Fähigkeit eines Computers verstanden, Aufgaben zu erledigen, welche üblicherweise die kognitiven Fähigkeiten menschlicher Intelligenz erfordern. Als deren Komponenten gelten in der Psychologie die Bereiche Lernen, logisches Schlussfolgern, Problemlösung, Wahrnehmung und Sprache. [25]

Obwohl KI spätestens mit der Veröffentlichung von ChatGPT durch OpenAI im Jahr 2022 einen regelrechten Hype erlebt, ist das Grundkonzept deutlich älter [26]. Bereits 1950 schätzte Alan Turing, dass gegen Ende des 20. Jahrhunderts Computer existieren würden, welche einen Menschen glaubhaft imitieren können [27]. Einige Jahre zuvor beschrieben McCulloch und Pitts ein mathematisches Modell künstlicher Neuronen (1943), doch als Bezeichnung für ein eigenes Forschungsfeld tauchte der Begriff KI erstmals 1955 auf [28]. Große Fortschritte wurden jedoch erst durch die steigende Leistungsfähigkeit von Computerhardware in den vergangenen 15 Jahren erzielt [29]. So besiegte 2017 mit Alpha GO ein Modell von Google DeepMind den weltbesten Go-Spieler Ke Jie [30].

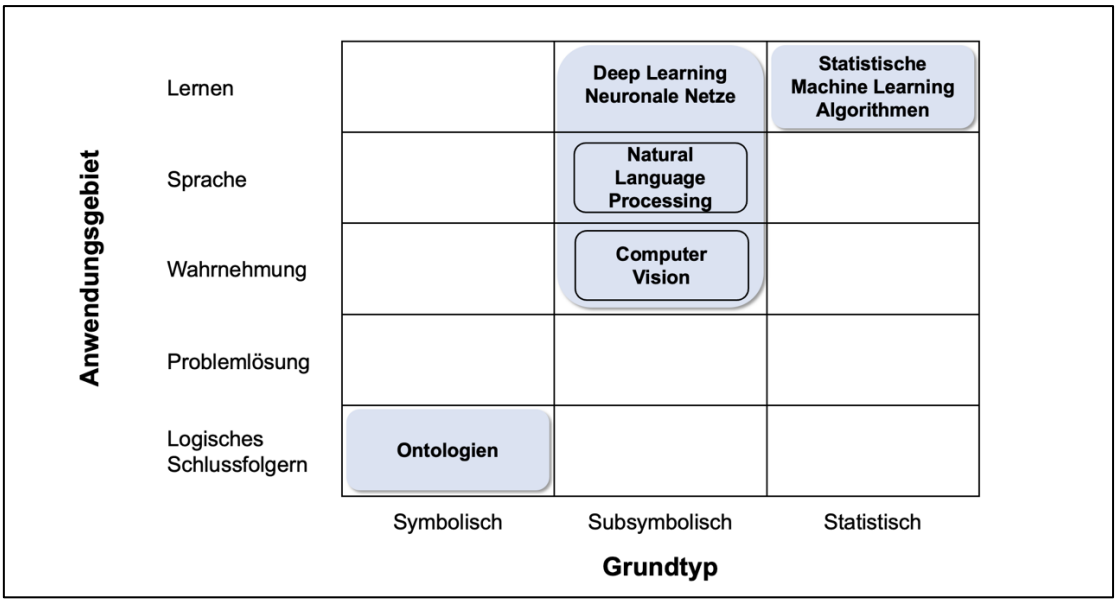

Abbildung 7 Ausgewählte KI-Technologien und Forschungsfelder nach Grundtyp und Anwendungsgebiet
In Anlehnung an Vorlesungsunterlagen [31]

Unter dem Oberbegriff KI existieren verschiedene Ansätze und Technologien, welche darauf abzielen, die verschiedenen Teilbereiche menschlicher Intelligenz nachzuahmen

(vgl. Abbildung 7). Diese können in die drei Kategorien symbolisch, subsymbolisch und statistisch unterteilt werden.

Symbolische KI repräsentiert Expertenwissen. Eine Domäne wird strukturiert und abstrakt mit einer standardisierten Sprache modelliert. Auf Grundlage dieser Modellierung ist es möglich, konkrete Probleme und Zustände zu beschreiben. Durch die festgelegten Regeln und Fakten können diese KI-Systeme bei Inferenz logische Schlussfolgerungen ziehen. Vertreter der symbolischen KI und Fokus der Arbeit sind Ontologien und die zugehörigen *Reasoner*. [32, S. 24-25]

Subsymbolische KI verfolgt einen anderen Ansatz. Anstatt die Modellierung menschlichen Wissens zu nutzen, lernen diese Systeme Zusammenhänge zwischen Input- und Output-Variablen aus Trainingsdaten. Kern dieser Verfahren bildet die Abstraktion des menschlichen Gehirns durch künstliche neuronale Netze. [33, S. 2-3]

Statistische KI ähnelt sub-symbolischer KI, da es sich hierbei ebenfalls um einen datengetriebenen Ansatz handelt. Statt neuronaler Netzte kommen jedoch Algorithmen zum Einsatz, welche durch die Anwendung mathematischer und insbesondere statistischer Grundlagen wie der Wahrscheinlichkeitstheorie entwickelt wurden [34]. Zu den Methoden des statistischen Lernens gehört etwa lineare Regression.

KI-Technologien finden weltweit Anwendung in zahlreichen Branchen und Produkten. Dennoch ist der Begriff mit Vorsicht zu betrachten, da Modelle für spezifische Anforderungen entwickelt werden. Beispielsweise ist ChatGPT nicht in der Lage, einen anderen Zweck als die kontextbasierte Generierung von Text zu erfüllen. Ebenfalls weist es kein Verständnis der dargestellten Informationen auf. Während ein Modell in seiner spezifischen Domäne also eine beindruckende Performance bieten kann, sind diese Systeme in Hinblick auf ihre Vielseitigkeit weit davon entfernt, mit der Intelligenz eines Menschen zu konkurrieren.

2.3.2 Ontologien

Als Vertreter symbolischer KI bildet eine Ontologie eine gemeinsame, formalisierte Abstraktion real existierender Objekte, Konzepte und ihrer Beziehungen innerhalb einer bestimmten Domäne [35, S. 908-909]. Weitere Begriffe, welche häufig im Zusammenhang mit Ontologien auftreten, sind Taxonomien und Knowledge Graphs (KG). Während eine Taxonomie Objekte ausschließlich klassifiziert, modelliert eine Ontologie zusätzlich deren semantische Bedeutung [36, S. 61]. KGs dienen dagegen der Visualisierung von Daten mit Hilfe von Knoten und Kanten sowie der Speicherung

großer Datenmengen (vgl. Abbildung 8), wobei eine Ontologie als Blaupause fungiert [37, S. 169-170].

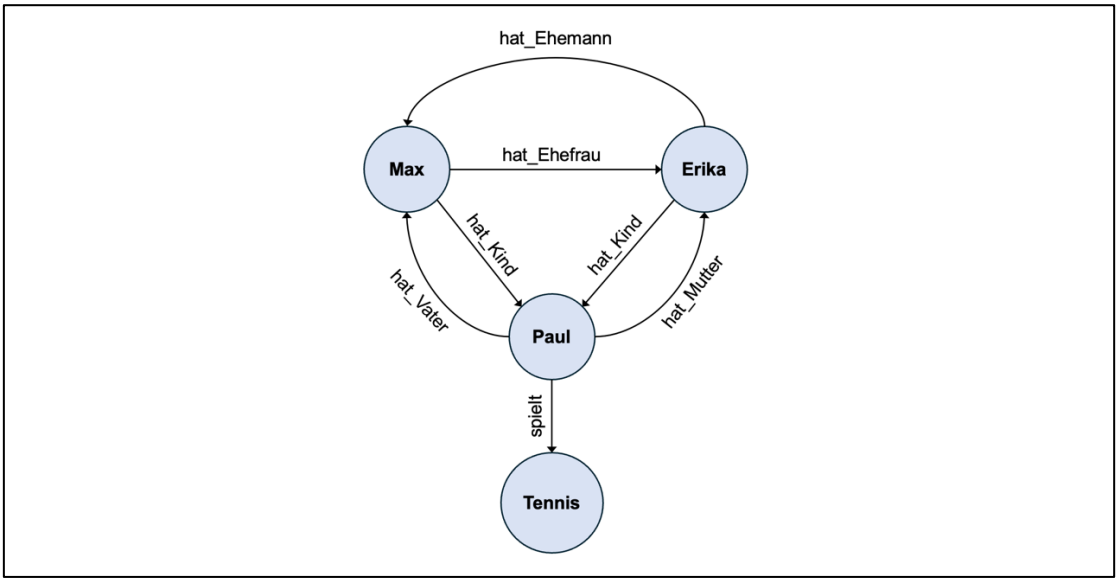

Abbildung 8 Beispielhafter Knowledge Graph zur Darstellung von Familienverhältnissen

Für die formalisierte Repräsentation von Wissen ist die Definition einer standardisierten Modellierungssprache und einer zugrundeliegenden Logik notwendig. Die verschiedenen Logiksysteme bewegen sich in einem Spannungsfeld zwischen Ausdruckskraft und Komplexität [37, S. 165]. So ermöglich der Einsatz der Prädikatenlogik erster Ordnung die Modellierung von Beziehungen zwischen Objekten, ist jedoch auch unentscheidbar. Unentscheidbarkeit beschreibt das Fehlen eines Algorithmus, welcher innerhalb einer finiten Anzahl an Schritten in der Lage ist, die Korrektheit einer beliebigen Aussage zur Ontologie zu prüfen. Zur Lösung dieses Problems wurden entscheidbare Beschreibungslogiken entwickelt, welche auch implizites Wissen aus einer Ontologie ableiten können. [32, S. 25-26]

Symbolische KI wie Ontologien sind für klar definierte und strukturierte Probleme geeignet [32, S. 29]. Eine Erweiterung kann durch die Modellierung von Unsicherheiten erfolgen. Anstelle der absoluten Zustände Wahr und Falsch, kann ein Grad an Unsicherheit in der Bewertung einer Aussage beschrieben werden. Daneben kann eine Aussage in einer Ontologie auch als unbekannt gelten. Da diese nur einen Ausschnitt der Realität modelliert, wird unbekanntes Wissen nicht zwangsläufig als wahr oder falsch eingestuft. Dieser Umstand wird als Open World Assumption (OWA) bezeichnet, welche im Gegensatz zur Closed World Assumption steht. Hierbei wird unbekanntes Wissen prinzipiell als falsch eingestuft. [32, S. 44-45]

2.3.3 Ontology Web Language

Eine weit verbreitete Modellierungssprache ist die Ontology Web Language (OWL, in der aktuellen Version OWL 2). OWL ist im Kontext des Semantic Web entstanden, einer Vision des WWW als Netz aus verlinkten und maschinenlesbaren Informationen, weshalb es web-freundliche Formate wie das Resource Description Framework (RDF) unterstützt. [37, S. 168-169], [38]

In OWL werden Wahrheitsaussagen, sogenannte Axiome, über Entitäten festgelegt. Diese gehören zu 3 Typen: *Classes*, *Properties* und *Individuals*. Ein *Individual* stellt ein konkretes Objekt dar. *Classes* stehen eine Abstraktionsebene darüber. Somit stellt jede *Class* eine Menge von *Individuals* dar. Zu den *Properties* gehören drei Unterkategorien: *Object Properties* beschreiben gerichtete Beziehungen zwischen Objekten (vgl. Abbildung 8). *Data Properties* geben gerichtete Beziehungen zwischen Objekten und Datenpunkten an. *Annotation Properties* ermöglichen eine zusätzliche Beschreibung von Entitäten ohne Einfluss auf den logischen Aufbau der Ontologie. [39]

Auch OWL nutzt URIs zur Identifikation und Bezeichnung von Entitäten, wobei jedoch keine Unique Name Assumption gilt [39]. So können zwei Entitäten mit verschiedenem Namen denselben Datenpunkt beschreiben. Gleichheit oder Disjunktheit können bei Bedarf explizit modelliert werden.

OWL Functional Style Syntax

Class Assertion

Zuordnung eines Individuals zu einer Class

	Class	Individual
`ClassAssertion`	`(:Person`	`:Max)`

Object Property Assertion

Herstellung einer gerichteten Beziehung zwischen zwei Individuals

	Property	Subject	Object
`ObjectPropertyAssertion`	`(:hat_Kind`	`:Max`	`:Paul)`

Data Property Assertion

Herstellung einer gerichteten Beziehung zwischen einem Individual und einem Datenpunkt

	Property	Individual	Wert mit Datentyp
`DataPropertyAssertion`	`(:ist_alt`	`:Paul`	`"12"^^xsd:integer)`

Abbildung 9 Überblick der OWL Functional Style Syntax

In Anlehnung an Vorlesungsunterlagen [31]

Ebenfalls werden zahlreiche Syntax-Stile unterstützt [40]. Abbildung 9 zeigt beispielhaft die Functional Style Syntax. Generell bietet OWL eine Vielzahl an Axiomen und Entwurfsmustern, welche bei Bedarf im Rahmen der Implementierung detailliert beschrieben werden.

2.3.4 KI in der Robotik

Ontologien finden breite Anwendung in der Praxis. In der Automatisierung bieten sie neben MRK ein hohes Potenzial, Industrieroboter zu verbessern und deren Einsatzgebiete zu erweitern. Insbesondere in Kombination mit subsymbolischer Technologie sind sie Gegenstand der Forschung.

Einen Ansatz für die Aufgabenplanung auf Basis eines KG stellen Miao et al. vor. Für Serviceroboter präsentieren sie ein Modell aus sechs Teilbereichen: Szenen, Objekte, Agenten, Aufgaben, Aktionen und Fähigkeiten. Für eine gegebene Aufgabe kann das System prüfen, ob diese bereits in der Wissensbasis modelliert ist. Ist dies nicht der Fall, wird das Vorgehen stattdessen auf Grundlage einer ähnlichen Aufgabe geplant. Zur Vorhersage der korrekten Reihenfolge an Einzelaktionen werden neuronale Netze eingesetzt. [41, S. 2, S. 13-15]

Auch eine Integration mit Computer Vision ist denkbar. So nutzen Bernardo et al. eine Ontologie zur Verbesserung der Bahnplanung. Durch Einsatz des Computer Vision Modells YOLO V3 werden Objekte in der Arbeitsumgebung eines Cobots erkannt. Auf Basis dieser Informationen wird anschließend eine OWL-Ontologie abgefragt. Diese identifiziert die Objekte und deren Eigenschaften. Durch die Kombination von Computer Vision und Domänenwissen ist das System in der Lage dynamisch eine kollisionsfreie Bahn zu planen. [42, S. 4]

Diese beiden Arbeiten stellen Ansätze für die Bahn- und Aufgabenplanung bei Industrierobotern mit Hilfe von Ontologien und subsymbolischer KI vor. Ein dritter Ansatz wird in dieser Arbeit diskutiert. Indem Arbeitsschritte, Bedingungen und notwendige Parameter direkt in der Wissensbasis modelliert sind, kann der korrekte Prozessablauf durch die logische Verkettung der Einzelaktionen bestimmt werden. Dadurch ist keine explizite Programmierung notwendig und der Roboter ist im Gegensatz zu statischen Programmen in der Lage, dynamisch auf veränderte Bedingungen zu reagieren.

2.4 Verwandte Arbeiten

Da heutzutage sowohl KI als auch Robotik Themen von enormer Bedeutung darstellen, gibt es zahlreiche Ansätze zur Fernsteuerung oder Funktionserweiterung von Robotern in Industrie und Forschung. Einige Beispiele für die Trennung von ausführender Hardware und Befehlslogik sollen kurz vorgestellt werden.

Bereits 1996 stellten Baginski und Kess für die Steuerung eines Systems aus mehreren Robotern, Sensoren und Manipulatoren eine mehrschichtige Struktur vor. Die Komponenten in der Hardwareschicht wurden an eine gemeinsame Schnittstelle angebunden. Dadurch erhalten Anwendungen aus der höheren Schicht Zugriff, welche die Form von komplexen Planungssystemen annehmen und in einer höheren Programmiersprache implementiert werden können. [43, S. 51]

Ein anderer Ansatz wurde von Sievers et al. verfolgt. Um einen humanoiden Serviceroboter mit KI-Funktionen auszustatten, werden diese als Online-Services auf einem Server bereitgestellt und können mittels Wi-Fi über eine API in das Roboterprogramm integriert werden [44, S. 431]. Der Zugriff auf Methoden über die Schnittstelle erfolgt mittels Uniform Resource Locators (URL). Dabei werden Daten im JSON-Format übertragen. [44, S. 433]

Auch Unternehmen zeigen Interesse an diesem Konzept, wie die Implementierung von Edge Computing über Ethernet durch das Telekommunikationsunternehmen NTT demonstriert. Durch die örtliche Entkopplung von steuernder Logik und Roboter, können Funktionen wie Bildanalyse mittels KI über zentralisierte Hardware bereitgestellt werden. Dadurch werden die notwendigen Investitionen im Vergleich zur individuellen Ausstattung jedes einzelnen Roboters mit zusätzlicher Rechenleistung gesenkt. Eine REST-API wird hierbei nicht für den Datenaustausch mit dem Roboter, sondern für die Kommunikation mit weiteren Softwaresystemen genutzt. [45]

Obwohl also bereits ähnliche Konzepte existieren, ist die Entwicklung einer quelloffenen Applikation sinnvoll, um neue Möglichkeiten für die Erforschung von KI-Anwendungen in der Robotik an der THI zu bieten. Insbesondere muss ein solches System clientseitig auf die vorhandene Hardware zugeschnitten sein.

3 Methodik

Neben der Betrachtung theoretischer Grundlagen ist auch ein erneuter Blick auf den Projektauftrag notwendig, um aus diesem eine sinnvolle Vorgehensweise abzuleiten. Dieser lautet wie folgt:

Ziel der Arbeit ist die Entwicklung eines erweiterbaren KI-Servers für einen Client, den Cobot *KUKA LBR iiwa 7 R800* im Produktionslabor der THI. Der Begriff Erweiterbarkeit beschreibt in diesem Fall primär eine einfache Integration zukünftiger KI-Funktionen. Als Ansatzpunkt fungiert eine vorangegangene Bachelorarbeit, in deren Rahmen eine REST-API definiert und bereits clientseitig auf der Robotersteuerung implementiert wurde. Außerdem soll serverseitig eine erste KI-Anwendung in Form einer Ontologie zur Steuerung des Knickarms implementiert werden. Abschließend wird die Funktionalität des Gesamtsystems im Labor geprüft.

Auf Basis dieser Zieldefinition lässt sich der Arbeitsumfang in zwei große Blöcke aufteilen: die Implementierung eines REST-Servers und einer ontologiebasierten KI-Anwendung. Ebenfalls zu beachten ist die Integration der beiden Softwarekomponenten mit dem Client. Da der Server als Bindeglied zwischen Roboter und KI-Applikation fungiert, ist dieser zunächst zu priorisieren (vgl. Abbildung 10).

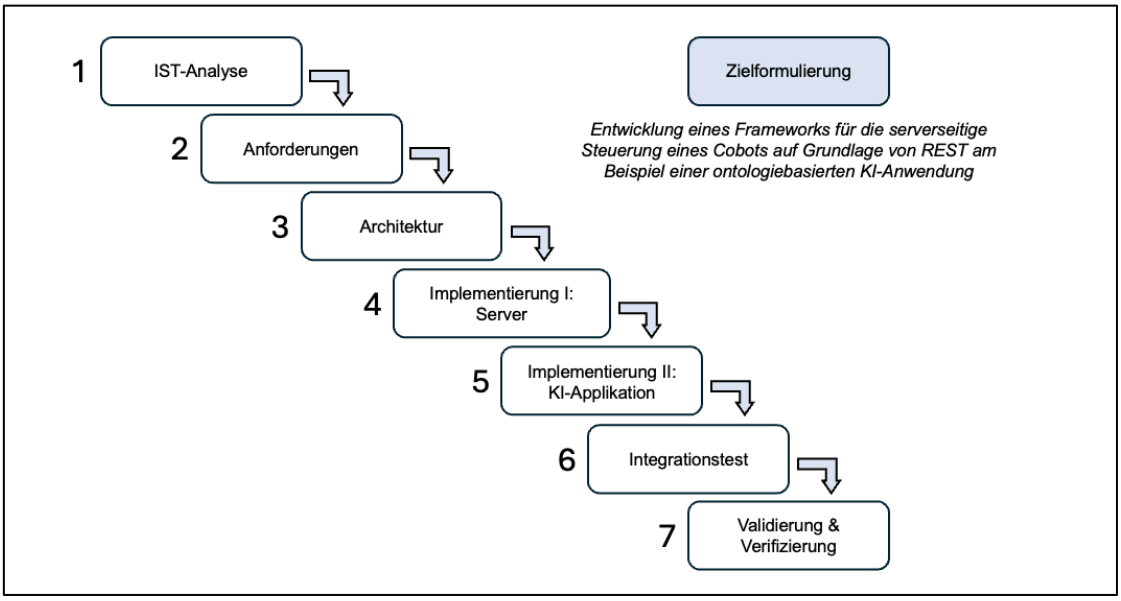

Abbildung 10 Entwicklungsprozess des Gesamtsystems

Zunächst ist eine Analyse der IST-Situation notwendig. Diese umfasst eine detaillierte Betrachtung der API-Endpunkte, welche aus den Ressourcen mit ihren zugehörigen URIs und HTTP-Verben bestehen. Ebenfalls ist es sinnvoll, die aktuelle Ausstattung des

Industrieroboters einzubeziehen, da diese den Rahmen für mögliche Testszenarien und somit für die Gestaltung der Ontologie vorgibt.

Im Anschluss an die IST-Analyse werden die Anforderungen an das System definiert. Zur Strukturierung dient die Unterscheidung zwischen funktionalen (FA) und nichtfunktionalen (NFA) Anforderungen (vgl. Kapitel 2.2.1), sowie zwischen Soll-Anforderungen (S) und Wünschen (W). Erstere stellen eine Priorität dar und müssen erfüllt werden, während letztere nur bei ausreichenden Ressourcen umgesetzt werden.

Die Anforderungsanalyse dient als Grundlage der Entwurfsphase. Aufgrund der geforderten Erweiterbarkeit kommt der in diesem Schritt entworfenen Architektur eine zentrale Rolle zu. Dieser Prozess ist iterativ zu verstehen, da bei der nachfolgenden Implementierung neue Impulse für die Architektur entstehen können. [12, S. 4]

Auf Basis der Anforderungen und der Architektur ist eine Entscheidung über die zu verwendenden Werkzeuge zu treffen. Dafür steht eine Vielzahl an Open-Source Software online zur Verfügung, um die verschiedenen Teilaufgaben zu lösen. Diese wird kurz vorgestellt. Im Anschluss erfolgt die Implementierung des Servers und somit der Abschluss des ersten Umfangs.

Den nächsten Arbeitsschritt stellt die Entwicklung der KI-Anwendung dar. Wesentlicher Bestandteil ist das Design der Ontologie (vgl. Abbildung 11). Hierfür ist eine eigene Anforderungsanalyse in Hinblick auf die Domäne und den Detaillierungsgrad notwendig [46, S. 4]. Nach ihrer Fertigstellung wird die Wissensbasis in eine Applikation integriert, welche den Server erweitert.

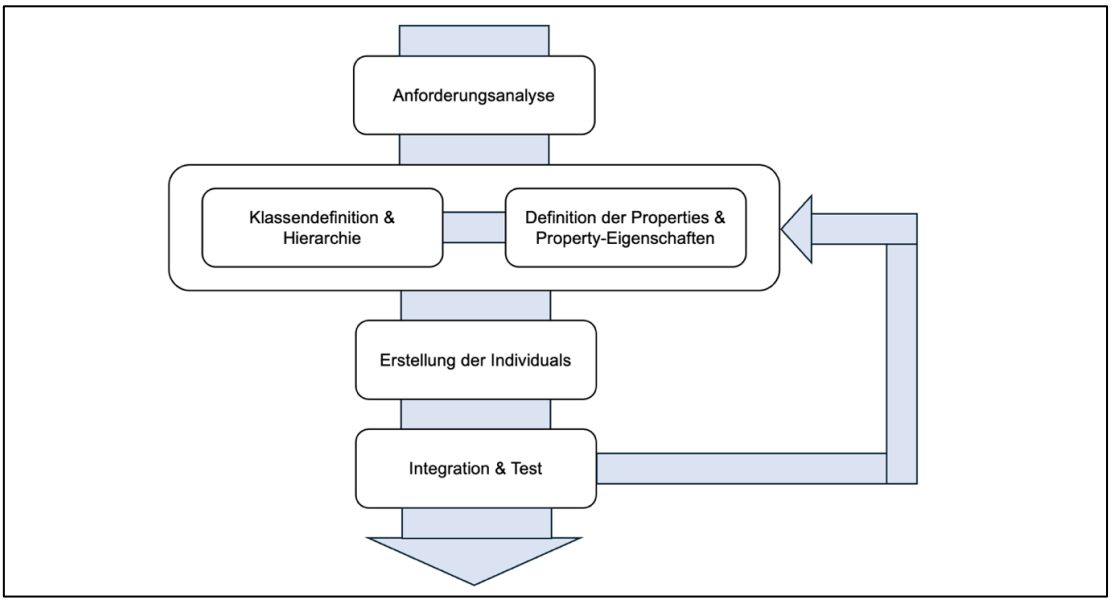

Abbildung 11 Entwicklungsprozess der Ontologie
In Anlehnung an [46, S. 4-11]

Zum Test des Gesamtsystems wird auf Grundlage der Ontologie ein Experiment entworfen, welches in Einklang mit den aus der IST-Analyse bekannten Einschränkungen steht. Dieses wird im Labor durchgeführt, wobei ein erfolgreicher Durchlauf den Abschluss der Implementierung darstellt. Bereits während der Implementierung werden Tests der einzelnen Komponenten durchgeführt, welche jedoch nicht gesondert betrachtet werden.

Für die abschließende Bewertung des Gesamtsystems auf Grundlage dieses Integrationstests können zwei Perspektiven betrachtet werden. Verifizierung beschreibt den objektiven Nachweis der Erfüllung festgelegter Anforderungen [47, S. 49]. Hierfür wird das Gesamtsystem mit den in der Planungsphase definierten Anforderungen abgeglichen.

Dagegen umfasst Validierung den Nachweis der Anforderungserfüllung für einen beabsichtigten Verwendungszweck [47, S. 50]. Um das Gesamtsystem zu validieren, wird dessen Eignung in Hinblick auf den ursprünglichen Projektauftrag und die geltenden Rahmenbedingungen diskutiert.

Im Anschluss an den Entwicklungsprozess werden die gewonnenen Erkenntnisse genutzt, um Erweiterungsmöglichkeiten, Einschränkungen und potenzielle Use Cases zu diskutieren. Diese sollen als Anregung für die Weiterentwicklung der existierenden Client-Server-Architektur dienen.

4 IST-Analyse

Wesentliche Vorarbeit wurde durch die Definition und clientseitige Implementierung der REST-API im Rahmen einer vorangegangen Bachelorarbeit geleistet [22]. Die Implementierung erfolgte in der Programmiersprache Java direkt auf der Robotersteuerung. Dafür ist die Nutzung von KUKAs proprietärer Entwicklungsumgebung Sunrise.Workbench notwendig.

Die Schnittstelle verfügt über fünf Endpunkte, auf welche clientseitig durch Requests und serverseitig durch Responses zugegriffen wird (vgl. Tabelle 3 und Tabelle 4). Als Format für die Datenübertragung zwischen Client und Server im *Message Body* wurde *application/json* festgelegt [22, S. 18].

Der Client sendet Requests gemäß einem festgelegten Programmablauf. Nach der erfolgreichen Anmeldung werden so lange Befehle angefragt, bestätigt und anschließend ausgeführt, bis der Server einen EXIT-Befehl schickt, woraufhin der Client diese Schleife beendet. [22, S. 16]

HTTP-Verb	Ressource	Message Body
POST	/login	{}
GET	/newcommand/{token}	{}
POST	/newcommand/{token}	{}

PUT	/saveinfo/{token}	{
		“msg“: Sting
		}

POST	/savelog/{token}	{
		”filename”: String,
		“data“: String
		}

Tabelle 3 Übersicht der Client-Responses

Quelle: [22, S. 17-18], teilweise mit Abweichungen im zugehörigen Quellcode [22, S. 33-36]

Die *Request POST /login* dient der Anmeldung eines Clients beim Server. Dieser liefert im Gegenzug ein Token, welches zur Identifizierung bei weiteren Anfragen genutzt wird (vgl. Tabelle 3). Dieses wird durch die Variable *{token}* dargestellt, wobei deren Sichtbarkeit innerhalb der URI in Bezug auf die Sicherheit des Systems kritisch zu bewerten ist [48]. Ebenfalls ist zu beachten, dass nicht das Token allein, sondern ein Text im JSON-Format innerhalb der URI übermittelt wird. [22, S. 19]

Der Befehl *GET /newcommand/{token}* wird genutzt, um einen neuen Befehl vom Server zu empfangen. *Die POST*-Request mit derselben URI bestätigt die Annahme und signalisiert dem Server, einen Folgebefehl zu generieren. [22, S. 19]

Die letzten beiden Requests dienen der Übertragung von Log-Dateien und weiteren Informationen an den Server. Insbesondere *PUT /saveinfo/{token}* könnte genutzt werden, um beispielsweise die aktuelle Roboterposition zu übermitteln. Beide Requests sind derzeit jedoch nicht vollständig implementiert. Die fehlende Möglichkeit, Informationen über den Clientzustand an den Server zu senden, erschwert die Einhaltung der Zustandslosigkeit nach REST. [22, S. 19, S. 35]

Verb der Request	**Ressource**	**Statuscode**	**Message Body**
POST	/login	200	{“token“: String}

GET	/newcommand/{token}	200	{
			“command“: String,
			“parameters“: Object
			}

POST	/newcommand/{token}	200	{}
PUT	/saveinfo/{token}	200	{}
POST	/savelog/{token}	200	{}

Tabelle 4 Übersicht der Server-Responses

Quelle: [22, S. 19-20], teilweise mit Abweichungen im zugehörigen Quellcode [22, S. 33-36]

Als Methode der Befehlsübertragung weist die Response zu *GET newcommand/{token}* den umfangreichsten *Message Body* auf. Die Variable *command* enthält die Bezeichnung eines Befehls und signalisiert dem Client die entsprechende Aktion. Neben der Übertragung von Logdateien (LOG) und sonstigen Informationen (SEND), umfasst die Liste an Befehlen die Beendigung des Programms (EXIT) sowie die drei Bewegungsarten PTP, LIN und CIRC. Die notwendigen Parameter richten sich nach der Befehlsart (vgl. Tabelle 5). Beispielsweise setzen Roboterbewegungen die Übertragung von Frames voraus.

Abschließend ist eine kurze Betrachtung des Industrieroboters sinnvoll. An diesem ist zu Projektbeginn noch kein Greifer installiert. Stattdessen befindet sich eine 3D-gedruckte Schaufel am Ende des Knickarms, welche die Aufnahme eines Kleinladungsträgers ermöglicht. Die Abwesenheit eines elektrischen oder hydraulischen Greifers sowie die fehlende Integration von Befehlen zur Greifersteuerung in der REST-API stellen derzeit

deutliche Einschränkungen für den Test der KI-Applikation dar. Zukünftig sollte diese Problematik zumindest hardwareseitig behoben sein, da die Inbetriebnahme eines Greifers Gegenstand einer parallelen Abschlussarbeit ist.

command	parameters	type	Aufbau der Frames
PTP	Frame, type	ParamPTP	{
LIN	Frame, type	ParamLIN	“x“: X,
CIRC	type, auxiliaryFrame, destination	ParamCIRC	“y“: Y, “z“: Z, “a“: A,
EXIT	type	ParamEXIT	“b“: B,
SEND	type	ParamSEND	“c“: C
LOG	type	ParamLOG	}

Tabelle 5 Übersicht der Roboterbefehle

Quelle: [22, S. 20], teilweise mit Abweichungen im zugehörigen Quellcode [22, S. 33-36]

5 Anforderungsmanagement

Auf Grundlage der bisherigen Analyse wurde eine Reihe von Anforderungen festgelegt. Diese werden in sieben Teilbereiche untergliedert, welche im Folgenden erläutert werden. Eine Klassifizierung erfolgt gemäß der beschriebenen Methodik nach Typ (FA/NFA) und Priorität (S/W).

5.1 Server

Nr.	Typ	Anforderung	Priorität
1	FA	Abdeckung der API-Endpunkte	S
2	FA	Bereitstellung eines gültigen Befehls	S
3	FA	Bereitstellung eines Log-In Tokens	S
4	FA	Speicherung des Tokens	S
5	FA	Prüfung des Tokens	S
6	FA	URI-Konformität des Tokens	S
7	FA	Persistente Speicherung von LOG-Dateien	S
8	FA	Persistente Speicherung von SEND-Dateien	S

Tabelle 6 Anforderungen an die Serverapplikation

Der Server stellt die Kernapplikation dar. Diese regelt den Datenaustausch über die REST-API und ermöglicht die weitere Datenverarbeitung mit Hilfe beliebiger KI-Applikationen. Dafür ist die Abdeckung der API-Endpunkte notwendig, wozu insbesondere die Übergabe eines durch den Client interpretierbaren Befehls gehört.

Die Kommunikationsstruktur gemäß API-Spezifikation erfordert die Generierung eines Tokens, woraus sich weitere Anforderungen ergeben. Dieses muss vom Server verarbeitet werden können und darf keine illegalen Zeichen enthalten, welche eine Übertragung in der URI verhindern.

Ebenfalls sinnvoll ist die Speicherung von Log- und Infodateien, welche vom Client übertragen werden. Dies ermöglicht etwa die systematische Analyse von Fehlern der Robotereinheit.

5.2 KI-Applikation

Nr.	Typ	Anforderung	Priorität
9	NFA	Integration einer Ontologie als Wissensmodell	S
10	FA	Zugriff der Applikation auf die Ontologie	S
11	FA	Befehlsgenerierung über Reasoning in der Ontologie	S
12	FA	Definition eines Experiments mit Abdeckung aller Befehle	S

Tabelle 7 Anforderungen an die KI-Applikation

Wesentliches Element der Zieldefinition ist die Bereitstellung von KI-Funktionen für den Roboter. Als konkrete Technologie wird hierbei eine Ontologie gewählt. Die KI-Applikation muss auf dieses Wissensmodell zugreifen können, um einen Befehl zu generieren, welcher gemäß der API-Spezifikation vom Roboter verarbeitet werden kann.

Zusätzlich ist das Design eines Experiments notwendig, um den Integrationstest am Roboter zu ermöglichen. Dies schließt die Erstellung geeigneter *Individuals* in der Ontologie ein, um alle Befehlsarten des Roboters im Rahmen des Tests abzudecken.

Diese grundlegenden Anforderungen an die KI-Anwendung werden im weiteren Projektverlauf um zusätzliche Aspekte ergänzt.

5.3 Systemarchitektur

Nr.	Typ	Anforderung	Priorität
13	NFA	Erweiterbarkeit in Hinblick auf zukünftige KI-Anwendungen	S
14	FA	Erweiterungsmanagement	S
15	NFA	Erweiterungen sollen sprachunabhängig sein	W

Tabelle 8 Anforderungen an die Systemarchitektur

Die Architektur stellt trotz einer geringen Anzahl an initialen Anforderungen ein wichtiges Element der Entwicklungsarbeit dar. Die entwickelte Softwareplattform soll wiederverwendbar sein, um in zukünftigen Projekten zusätzliche KI-Funktion integrieren zu können. Diese Erweiterbarkeit erfordert Managementfunktionen, welche den Zugriff durch den Server ermöglichen und zugleich regeln.

Die Festlegung des Industrieroboters als Client dient neben einer Reduzierung der Rechenoperationen auf der Steuerung auch der Sprachunabhängigkeit des Servers [22, S. 16]. Dieser kann theoretisch in einer beliebigen Programmiersprache entwickelt werden, insofern die REST-API korrekt implementiert wird. Die Möglichkeit Erweiterungen ebenfalls in einer favorisierten Sprache zu implementieren, kann die Attraktivität zukünftiger Studentenprojekte potenziell steigern. Diese Anforderung wird jedoch zunächst nicht priorisiert.

5.4 Testing

Nr.	Typ	Anforderung	Priorität
16	NFA	Prüfung der Datenübertragung über die API	S
17	NFA	Abschluss eines erfolgreichen Integrationstests	S

Tabelle 9 Anforderungen an Systemtests

Für einen erfolgreichen Projektabschluss bedarf es ausreichender Tests des Softwaresystems. Als Anforderungen werden zwei spezifische Testfälle definiert. Zunächst wird die Funktionalität der API bereits während der Entwicklung getestet. Somit kann Mehrarbeit im Labor ausgeschlossen werden.

Außerdem wird das Gesamtsystem bestehend aus dem Server-Kern, der KI-Applikation und dem Client mit Peripherie in Form des Industrieroboters im Labor erprobt. Grundlage für diesen Test bildet das Experiment, welches im Rahmen der Ontologie-Entwicklung definiert wird (vgl. 5.2 KI-Applikation).

5.5 Release

Nr.	Typ	Anforderung	Priorität
18	NFA	Kommentierter Code	S
19	NFA	Plattformunabhängigkeit: Docker Container	W
20	NFA	Bereitstellung über GitHub	S
21	NFA	Bereitstellung eines Produktionsservers	W

Tabelle 10 Anforderungen an den Release

Die Wiederverwendbarkeit und Erweiterbarkeit des Systems bilden zentrale Aspekte des Projekts. Dazu ist eine sinnvolle Release-Strategie sinnvoll. Die Software wird über ein

persönliches GitHub-Repository bereitgestellt. Von dort kann sie bei Bedarf innerhalb der THI verteilt werden. Neben der Masterarbeit dienen auch ausführliche Kommentare innerhalb des Quellcodes dazu, das Projekt zu dokumentieren und die Einarbeitung zu erleichtern.

Eine sinnvolle Ergänzung stellt die Erstellung eines Docker Containers dar. Dadurch wird der Quellcode mit allen Abhängigkeiten verpackt, so dass dieser plattformunabhängig eingesetzt werden kann [49]. Zusätzlich kann die Integration eines Produktionsservers geprüft werden. Diese weisen eine höhere Stabilität und Sicherheit auf als Entwicklungsserver, welche in einigen Web-Frameworks enthalten sind [50].

5.6 Rahmenbedingungen

Neben konkreten Anforderungen ergeben sich aus dem Projektziel, der Arbeitsumgebung und der REST-API gewisse Rahmenbedingungen, welche beachtet werden müssen.

Nr.	Typ	Anforderung	Priorität
22	FA	Serverseitige Nachverfolgung der Roboterposition	S
23	NFA	Festlegung von Python als Programmiersprache	S
24	NFA	Beachtung der Roboterausstattung im Experiment	S
25	NFA	Keine Anpassung des Clients	S
26	NFA	Nutzung freier Software	S
27	NFA	Datenübertragung im JSON-Format	S
28	NFA	Einhaltung der HTTP-Spezifikation	S
29	NFA	API muss RESTful sein	S
30	NFA	Einhaltung der API-Spezifikation	S

Tabelle 11 Rahmenbedingungen

Obwohl die API clientseitig voll funktionsfähig ist, tauscht dieser noch keine Informationen über den Zustand des Roboters aus. Da festgelegt wurde, den Client nicht weiter anzupassen, muss das System also in der Lage sein, die Roboterposition anderweitig zu verfolgen. Hierbei ist jedoch Vorsicht geboten, da der Server zustandslos sein muss, damit die API als *RESTful* bezeichnet werden kann.

Daneben sind die API- und HTTP-Spezifikationen zu beachten. Der Server muss den festgelegten Programmablauf des Clients spiegeln, um die Funktionalität des Gesamtsystems sicherzustellen. Ebenfalls ist das HTTP-Verb *GET* gemäß Spezifikation idempotent und sicher zu implementieren, während *PUT* lediglich idempotent sein muss.

6 Entwurf der Architektur

Vor dem Hintergrund der definierten Anforderungen wird eine geeignete Systemarchitektur entworfen. Gemäß Projektauftrag und Rahmenbedingungen kann diese auf der obersten Ebene als Drei-Schichten-Architektur beschrieben werden, welche aus dem Client, dem Server und einer Datenbank (DB) besteht (vgl. Abbildung 12). In diesem Fall laufen Server und DB jedoch auf demselben Rechner. [12, S. 327]

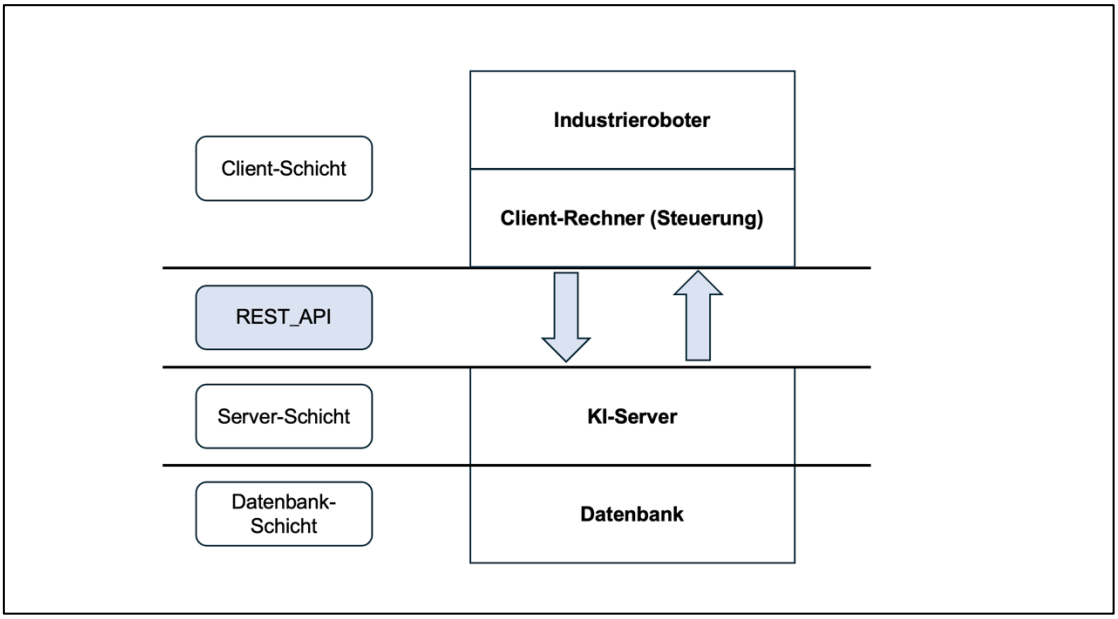

Abbildung 12 Modell der Drei-Schichten-Architektur

Da der Client bereits implementiert wurde und für die Datenbank-Schicht Fremdsoftware eingebettet werden kann, liegt der Fokus der Entwurfsphase auf der Server-Schicht. Aus den Anforderungskategorien Server und Systemarchitektur lassen sich vier Aufgabenbereiche ableiten: der Datenaustausch mit dem Client über die REST-API, die Speicherung von Daten in der DB, das Management von Token und von Erweiterungen.

Im Sinne der objektorientierten Programmierung können hieraus vier Klassen abgeleitet werden. Dadurch kann zum einen *Information Hiding* erreicht werden, zum anderen ermöglicht es die Unterteilung des Projekts in kleine Programmbausteine, welche einzeln entwickelt und unter Beachtung der Abhängigkeiten angepasst werden können.

Ein Architekturdiagramm zeigt die unterschiedlichen Komponenten und ihre Kardinalitäten (vgl. Abbildung 13). Der Client wird vereinfacht als einzelnes Element dargestellt, wobei dieser in der Realität selbst aus mehreren Einzelkomponenten besteht. Als zentrales Element verarbeitet der *Server* die Requests des Clients und generiert eine passende Response. Dazu nutzt dieser die Funktionalität der restlichen Klassen, welche in Anlehnung an das Konzept eines Plug-in-Managers ebenfalls nach

diesem Schema benannt werden. Diese übernehmen jeweils einen der restlichen drei Aufgabenbereiche.

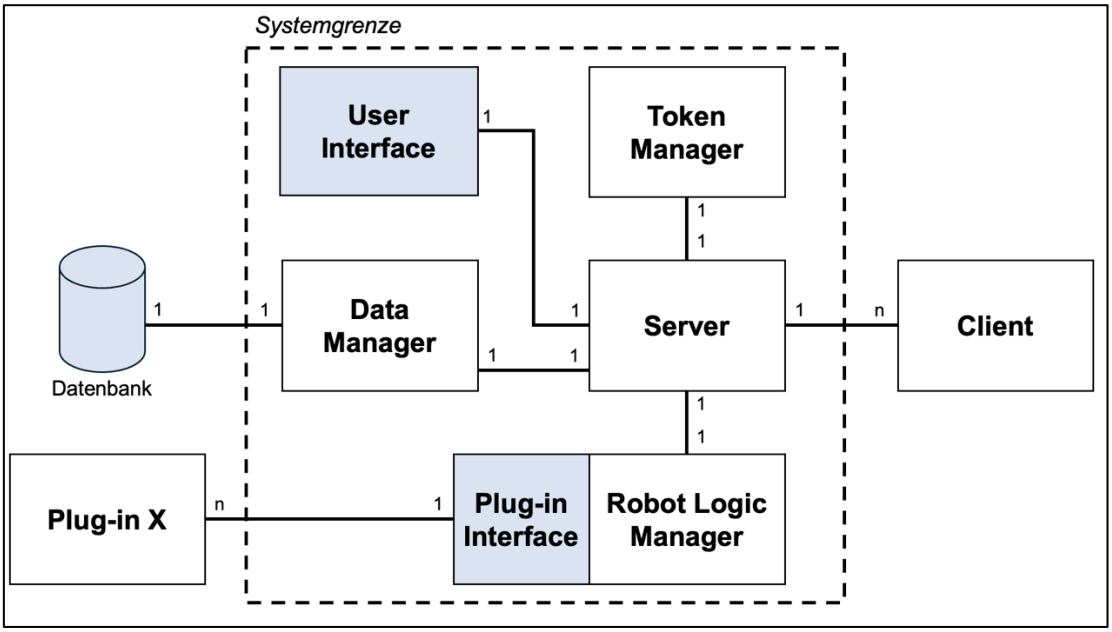

Abbildung 13 Architekturdiagramm mit Klassen und Kardinalitäten

Der *Token Manager* (TM) ist in der Lage ein Token aus einer Reihe an erlaubten ASCII-Zeichen zu generieren. Daraufhin wird dieses zur Authentifizierung bei Requests im weiteren Programmverlauf gespeichert.

Über den *Data Manager* (DM) wird ein DB-System angebunden. Bei Bedarf werden Daten vor der Speicherung durch den *DM* formatiert und ergänzt. Ebenfalls dient diese Klasse zur Datenabfrage.

Von wesentlicher Bedeutung ist der *Robot Logic Manager* (RLM), welcher in diesem System einem Plug-in-Manager entspricht (vgl. 2.2.1 Architektur). Obwohl andere modulare Architekturstile existieren, wird eine Plug-in-Architektur sowohl aufgrund ihrer Einfachheit als auch des kompakten Softwareumfangs umgesetzt. Der *RLM* entdeckt verfügbare Plug-ins, lädt diese dynamisch während des Programmablaufs und stellt diese dem System anschließend für die Befehlsgenerierung zu Verfügung.

Insbesondere diese letzte Aufgabe ist hervorzuheben. Keine der bisher beschriebenen Klassen ist in der Lage, einen Befehl zu generieren. Somit bildet die Kernapplikation bestehend aus *Server*, *TM*, *DM* und *RLM* lediglich eine Plattform oder ein Framework für die KI-basierte Steuerung des Clients, wobei die Mechanismen der Befehlserstellung in ein Plug-in ausgelagert werden. Dieser Umstand wird durch die Wahl der Bezeichnung RLM statt Plug-in-Manager betont, da es sich hierbei nicht um einfache Komfortfunktionen wie beispielsweise die Installation eines Werbeblockers in einem

Internetbrowser handelt, sondern um wesentliche Kernfunktionen für den Programmablauf. In dieser Hinsicht lässt sich dieser Entwurf mit der Architektur der integrierten Entwicklungsumgebung Eclipse vergleichen [51].

Zwei weitere Klassen sind notwendig, um die Plug-in-Architektur umzusetzen. Das *Plug-in Interface* stellt als abstrakte Klasse eine Blaupause für alle Plug-ins dar und definiert Methoden, welche durch diese implementiert werden müssen. Nicht in Abbildung 13 dargestellt sind die verschiedenen *Command*-Klassen, welche jeweils einen der Roboterbefehle (vgl. Tabelle 5) bereitstellen. Dadurch soll deren korrekte Formatierung im Rahmen der Client-Server-Kommunikation sichergestellt und die Entwicklung zukünftiger Erweiterungen erleichtert werden.

Zuletzt ist auch ein einfaches *User Interface* (UI) in dieser Architektur eingeplant. Da an dieses keine spezifischen Anforderungen gestellt werden, stellt es keinen Fokus der Arbeit dar. Dennoch ist eine einfache Benutzeroberfläche sinnvoll, um Funktionen zur Serversteuerung bereitzustellen oder dem Nutzer im laufenden Betrieb relevante Informationen anzuzeigen.

Durch den Entwurf einer anforderungsspezifischen Software-Architektur wird die anschließende Implementierung vereinfacht. Gemäß des Grundsatzes *Separation of Concerns* wird der Projektauftrag in kleinere Teilprobleme zerlegt, welche gemäß dem Paradigma der objektorientierten Programmierung in Form von Klassen gelöst werden. Diese Struktur ermöglich in Zukunft eine verbesserte Anpassbarkeit und Wartbarkeit des Systems. Außerdem können einige der Anforderungen aus Kapitel 5 durch die vorgeschlagene Architektur erfüllt werden, wozu insbesondere die Erweiterbarkeit mit zusätzlichen KI-Funktionen gehört. Beispielhafte Plug-ins werden im Rahmen der Arbeit vorgestellt. [12, S. 423]

7 Technology Stack

Im Anschluss an die Anforderungsanalyse und den Architekturentwurf erfolgt eine Entscheidung über die Werkzeuge, welche für die Implementierung der Software eingesetzt werden. Die Gesamtheit dieser Komponenten wird auch als Technology Stack oder verkürzt als Tech Stack bezeichnet [52]. Die Auswahl geeigneter Technologien ist wesentlich für den Projekterfolg, zumal einige der Entscheidungen wie die Auswahl der Programmiersprache nicht ohne erheblichen Aufwand rückgängig gemacht werden können.

Projekt Tech Stack

Container	Docker
Produktionsserver	Gunicorn
API-Testing	Postman
KI-Anwendung	OWL2, Protégé & OWLready2
User Interface	PyQT5
Datenbank	MongoDB & pymongo
Web-Framework	Flask
Programmiersprache	Python

Abbildung 14 Projekt Tech Stack

Der Tech Stack für dieses Projekt umfasst acht Bereiche (vgl. Abbildung 14). Der Einsatz der Programmiersprache Python wurde aufgrund persönlicher Erfahrung bereits als Rahmenbedingung definiert (vgl. Tabelle 11). Aufgrund ihrer Popularität bietet diese Zugriff auf zahlreiche Bibliotheken, welche nicht alle explizit im Tech Stack genannt werden. Insbesondere ist Python im Bereich Machine Learning etabliert, wodurch die Entwicklung zukünftiger KI-Funktionen erleichtert wird.

Für den Umgang mit HTTP gibt es im Python-Ökosystem mehrere Web-Frameworks. Da im Gegensatz zu den Bedürfnissen typischer Webentwickler jedoch nur grundlegende Funktionalität benötigt wird, eignet sich Flask als sogenanntes Micro-Framework. Dieses verfügt über einen integrierten Entwicklungsserver, um dem Client die Anwendung bereitzustellen [50].

Durch die Festlegung von JSON als Datenübertragungsformat der REST-API, empfiehlt sich das dokumentenbasierte MongoDB anstelle einer typischen SQL-DB. Für die Integration in den Code wird die zugehörige Bibliothek pymongo genutzt.

Die Grundlage für die Erstellung des UI bildet Qt 5. Dieses ist in Form von PyQt5 in Python implementiert und bietet eine Vielzahl an Bedienelementen und Anpassungsmöglichkeiten. Ebenfalls kann darüber Multithreading implementiert werden, wodurch eine gute Performance des UIs und der Kernapplikation sichergestellt wird [53].

Für die Modellierung der Ontologie in OWL2 wird der weitverbreitete Protégé-Editor eingesetzt. Obwohl der Aufbau der KI-Anwendung bisher noch nicht im Detail betrachtet wurde, wird das Plug-in die zugehörige Ontologie mittels der Bibliothek OWLready2 abfragen.

Abschließend kommen für die Erfüllung der Wunsch-Anforderungen Gunicorn als Produktionsserver, welcher den Entwicklungsserver des Flask-Frameworks ersetzt, sowie Docker für die Erstellung von Containern in Frage. Für das Testen der API eignet sich die proprietäre Software Postman. Dennoch bietet diese eine kostenlose Einstiegsversion und wird letztendlich nicht im laufenden Betrieb benötigt. Somit sind alle beschriebenen Werkzeuge entweder quelloffen oder zumindest als Student für nicht-kommerzielle Zwecke kostenfrei nutzbar. Dadurch wird die Rahmenbedingung Nr. 26 eingehalten (vgl. Tabelle 11).

8 Implementierung des Servers

Der Aufbau und die Funktionsweise der Serverapplikation werden im Folgenden detailliert betrachtet. Dazu werden zunächst die verschiedenen Module vorgestellt und die Plug-in-Architektur erläutert. Abschließend wird der Programmablauf dargestellt.

8.1 Übersicht der Module

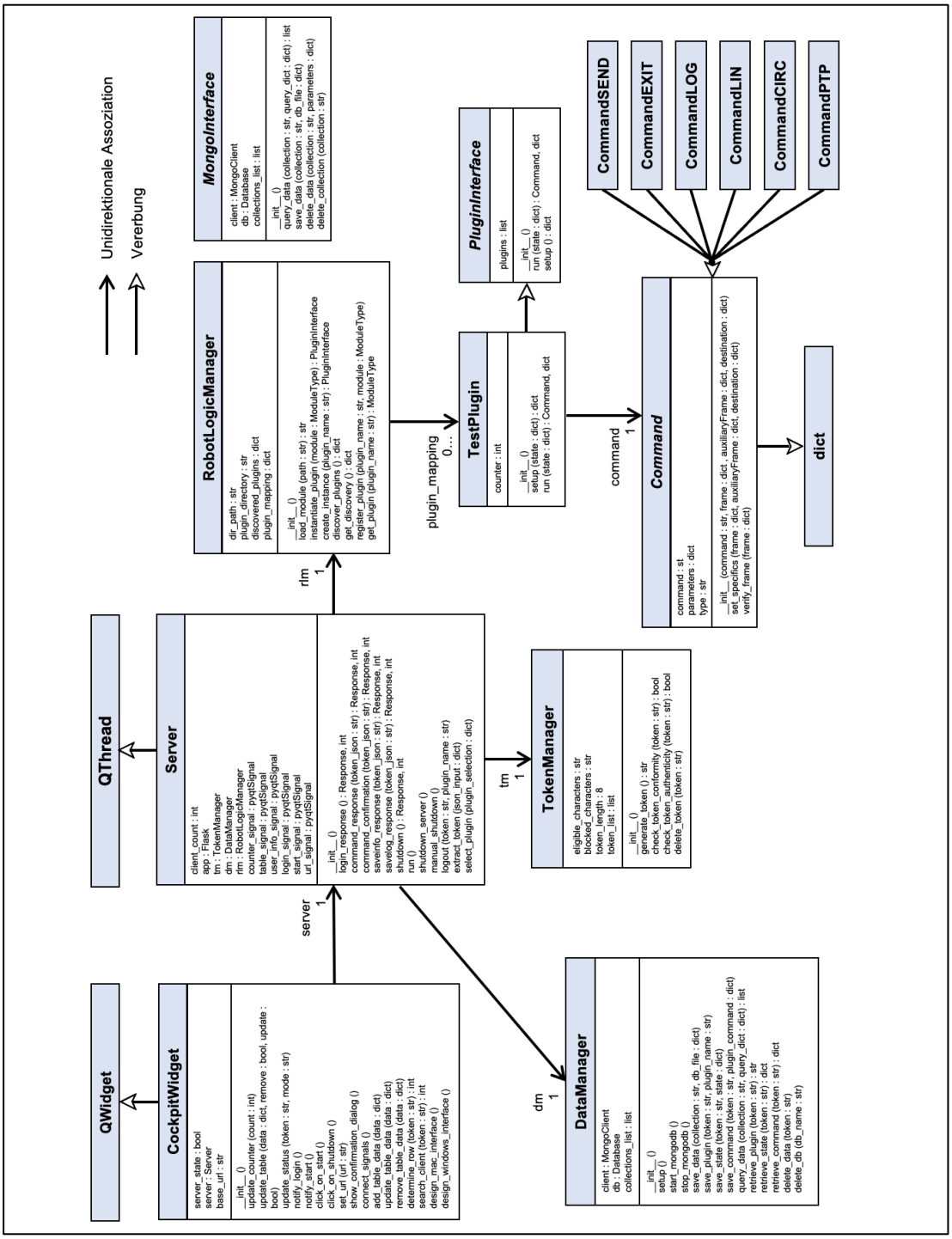

Abbildung 15 Klassendiagramm der Serverapplikation

8.1.1 Server

Die Serverklasse bildet das Herzstück des Programms. Zum einen sind in diesem Modul sämtliche Manager integriert, zum anderen wird an dieser Stelle die Flask-App konfiguriert und gesteuert. Dafür ist zunächst die Festlegung eines geeigneten Hosts und Ports notwendig. Hierbei sollte zwischen zwei Einsatzzwecken unterschieden werden: Experimente am Roboter im Labor oder die Plug-in-Entwicklung. Um ersteren Fall zu unterstützen, entspricht die Server-URL standardmäßig dem Wert, welcher auch auf der Robotersteuerung als Verbindungsziel hinterlegt ist. Bei der Ausführung des Codes ohne Ethernet-Verbindung zur Steuerung führt dies zu einem Fehler, weshalb diese Einstellung für die Weiterentwicklung des Servers außerhalb des Labors manuell angepasst werden muss.

Außerdem werden die sogenannten View-Funktionen definiert. Diese sind mit den Endpunkten der REST-API verknüpft und werden jedes Mal ausgeführt, wenn ein Client auf die zugrundeliegende Ressource zugreift [54]. Analog zu den fünf möglichen Client-Requests gibt es also fünf Methoden, welche eine Response gemäß der API-Spezifikation generieren und an den Client senden. Zusätzlich wurde ein sechster Endpunkt definiert, welcher das Abschalten des Servers ermöglicht. Dies erleichtert die Implementierung des Graphical User Interface (GUI). Eine detaillierte Betrachtung der verschiedenen View-Funktionen erfolgt während der Erläuterung des Programmablaufs.

8.1.2 TokenManager

Beim TM handelt es sich um ein vergleichsweises kleines Modul, welches die Authentifizierungstoken für Clients generiert, speichert und prüft. Da das Format der Token weder durch den Client noch die API-Spezifikation festgelegt ist, war es notwendig, einige Grundsatzentscheidungen zu treffen.

Die Tokenlänge wurde auf acht Zeichen festgelegt, welche zufällig aus einer Liste an erlaubten Zeichen gewählt werden. Darunter fallen Ziffern sowie das lateinische Alphabet in Groß- und Kleinschreibung. Nicht erlaubt sind Sonderzeichen und Leerstellen. Diese Limitierung dient der Einfachheit der Token, richtet sich aber auch nach den Empfehlungen der URI-Spezifikation [55, S. 11-14].

Ein generiertes Token wird in einer einfachen Liste gespeichert. Für die Authentifizierung im laufenden Betrieb, bietet diese Klasse eine geeignete Methode zum Abgleich mit dem Bestand.

8.1.3 DataManager

Diese Klasse ermöglicht die Integration von MongoDB und stellt unter Einsatz von pymongo Methoden zur Interaktion mit dem DB-System bereit. Um die Nutzererfahrung zu verbessern, wird die DB dabei direkt aus der Anwendung heraus gestartet, womit der manuelle Start über das Terminal vermieden wird. Dies stellt eine gewisse Herausforderung dar, da MongoDB ein je nach Plattform unterschiedliches Startverhalten aufweist. Während diese Herangehensweise auf MacOS problemlos funktioniert, führt der langsamere Start auf Windows zum Einfrieren des GUI. Nach einem Neustart der Anwendung ist die Nutzung des Servers jedoch problemlos möglich, da die DB in der Zwischenzeit gestartet wurde. Läuft der DB-Prozess jedoch erstmal im Hintergrund, kann dieser auf beiden Plattformen problemlos durch eine zweite Methode beendet werden.

MongoDB speichert Dokumente in sogenannten Kollektionen, welche eine gewisse Strukturierung innerhalb einer einzelnen DB ermöglichen. Die Serverapplikation nutzt fünf solcher Kollektionen: *app_state*, *logs*, *infos*, *plugins* und *commands*. Während *logs* und *infos* der Sicherung von Informationen dienen, welche über die entsprechenden Endpunkte vom Client an den Server übertragen werden, ist für die übrigen Kategorien ein Verständnis des Programmablaufs notwendig. Um Daten in diesen Kollektionen speichern oder aus diesen abfragen zu können, bietet der DM verschiedene Methoden. Ebenfalls können Daten gelöscht werden, was beispielsweise beim Ausloggen eines Clients geschieht.

8.1.4 RobotLogicManager

Als letzte Klasse der Kernapplikation ist der Plug-in-Manager entscheidend, um die Anforderungen an die Erweiterbarkeit des Servers zu erfüllen. Dieser ist in der Lage Plug-ins zu entdecken, zu registrieren und während des Programmablaufs dynamisch zu laden. Da die gesamte Befehlslogik in die Plug-ins ausgegliedert wird, wurde der Name *RobotLogicManager* gewählt, um die hohe Bedeutung dieses Moduls für die Architektur zu betonen.

Grundsätzlich kann dieser nur Plug-ins entdecken, welche sich im gleichnamigen Ordner befinden. Für eine erhöhte Übersichtlichkeit ist es jedoch möglich, ein Plug-in in Unterordnern zu strukturieren. Der Manager sucht darin nach einem Modul mit dem Präfix *telepath_*, welches die Server-Erweiterungen markiert. Gefundene Module werden in einem Python *Dictionary* festgehalten, einem Datentyp, welcher hinsichtlich

seiner Struktur mit JSON vergleichbar ist. Als Schlüssel dient der jeweilige Plug-in-Name, welchem der Dateipfad des Moduls als Wert zugeordnet wird.

Soll ein gefundenes Plug-in geladen werden, wird dieses zunächst in einem weiteren *Dictionary* registriert, wobei dem Namen diesmal das eigentliche Modul zugeordnet wird. Dadurch wird der Zugriff im laufenden Betrieb vereinfacht. Soll ein Befehl generiert werden, durchsucht der RLM das Modul aus dem *Dictionary* nach einer Klasse, welche das *PluginInterface* implementiert. Anschließend wird ein Objekt dieser Klasse erzeugt, welches nun in der Lage ist, den benötigten Roboterbefehl zu erstellen.

8.1.5 CockpitWidget

Ein GUI war nicht Bestandteil der ursprünglichen Anforderungen oder Analyse. Dennoch verbessert dieses die Benutzererfahrung signifikant, indem Informationen über die verbundenen Roboter angezeigt werden. Aufgrund des recht geringen Funktionsumfangs wird dieses jedoch als Widget bezeichnet, dessen technische Grundlage Qt 5 bildet.

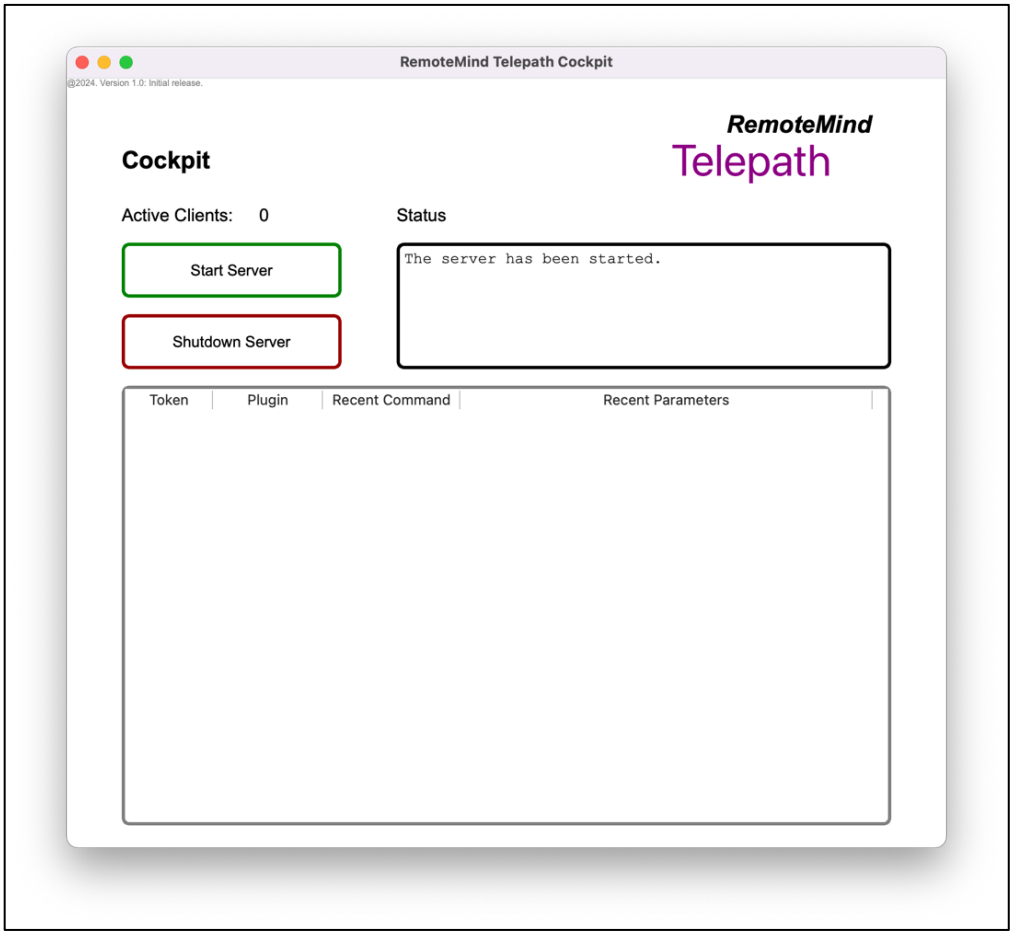

Abbildung 16 Server-Widget auf MacOS

Das *CockpitWidget* bietet einige Bedien- und Anzeigelemente (vgl. Abbildung 16). Jeweils eine Schaltfläche dient zum Starten und Abschalten des Servers, wobei letzteres über einen Dialog bestätigt werden muss. Der Zähler über den Schaltflächen gibt die Anzahl der eingeloggten Clients an. Der große Kasten am unteren Ende des Fensters beinhaltet eine Tabelle, welche jeweils das Token, das ausgewählte Plug-in sowie Informationen über den letzten Roboterbefehl aller aktiven Clients anzeigt. Die Status-Zeile rechts oben informiert den Nutzer über Ereignisse oder weist ihn darauf hin, das Terminal zu prüfen.

Tatsächlich ist das *CockpitWidget* nämlich nur ein Bestandteil eines hybriden Bedienkonzepts, welches ebenfalls das Terminal umfasst. Diese Entscheidung wurde getroffen, da nicht abgeschätzt werden kann, wann und wie oft ein Plug-in-Entwickler den Nutzer mit Eingaben oder Ausgaben einbeziehen möchte. Das Terminal als einheitliche Schnittstelle sorgt hierbei durch die Nutzung einfacher Funktionen wie *print()* und *input()* für die notwendige Flexibilität, während eine Anbindung über das GUI technisch komplex ist, ohne dabei einen großen Mehrwert zu bieten.

Die angesprochene technische Komplexität wird insbesondere hinsichtlich der Performance ersichtlich. Die Flask-App bzw. der Server muss in einem separaten Thread ausgeführt werden, während das Qt5-GUI im Haupt-Thread ablaufen muss, da die Anwendung ansonsten einfriert. Um dies zu ermöglichen, erbt die Server-Klasse von *QThread* (vgl. Abbildung 15). Ebenfalls werden sogenannte *pyqtSignals* genutzt, welche den Datenaustausch zwischen den Threads und somit die Aktualisierung der Anzeigeelemente ermöglichen. Eine andere Form der Kommunikation wird für das Abschalten des Servers genutzt. Hierbei wird eine Request an die neue, sechste View-Funktion des Servers geschickt.

8.2 Design der Plug-in-Architektur

Neben dem RLM bildet insbesondere das *PluginInterface* eine wesentliche Säule der modularen Applikation. Diese abstrakte Klasse kann nicht in Form von Objekten instanziiert werden. Sie verfügt über drei Methoden, welche von allen Plug-ins implementiert werden müssen: *__init__()*, welche in Python dem Konstruktor entspricht, *setup()* und *run(…)* (vgl. Abbildung 15). Hierbei sind weitere Besonderheiten der Programmiersprache zu beachten. Zunächst ist Python eine dynamisch typisierte Sprache. Durch einen Entwickler können zwar beispielsweise Datentypen für Funktionsparameter angegeben werden, welche jedoch nicht vom Python-Interpreter erzwungen werden. Außerdem ist es nicht möglich, die Anzahl oder den Typ von Rückgabewerten einer abstrakten Methode zu definieren, welche folglich von konkreten Implementierungen eingehalten werden müssten. Daher sind die folgenden Festlegungen unbedingt zu beachten.

Die erste Methode *setup()* wird vom Server aufgerufen, um das Plug-in einzurichten und gibt einen Startzustand *state* zurück. Die Notwendigkeit dieser Methode ergibt sich aus dem Programmablauf. Die zweite Methode *run(…)* wird aufgerufen, um das Plug-in auszuführen und einen gültigen Befehl für den Versand an den Client zu erhalten. Als Eingabe erhält diese vom Server eine Zustandsvariable *state*, als Ausgabe wird zuerst ein gültiger Roboterbefehl *command* und zusätzlich eine neue Zustandsvariable *state* erwartet. Somit kann *run(...)* als Hauptmethode jedes Plug-ins verstanden werden, da hier die Mechanismen der Befehlsgenerierung ausgelöst werden.

Die beiden Rückgabewerte müssen aufgrund ihrer enormen Bedeutung für die Funktionalität des Servers detailliert erläutert werden. Die Variable *command* entspricht dem Roboterbefehl und ist ein *Dictionary*, welches über die Flask-App im *Message Body* einer Response an den Client geschickt wird. Enthalten sind in diesem die Schlüssel-Wert-Paare gemäß API-Spezifikation (vgl. Tabelle 5). Um die korrekte Formatierung sicherzustellen, wurden *Command* und dessen Subklassen eingeführt. Der Import dieser Klassen in ein Plug-in erleichtert die Erzeugung der unterschiedlichen Befehle durch je eine Subklasse mit passendem Konstruktor. Eine technische Besonderheit stellt hierbei Pythons Umgang mit Datentypen dar, welche in dieser Programmiersprache jeweils durch eine eigene Klasse abgebildet werden. Da Roboterbefehle die Form eines *Dictionary* annehmen sollen, erben die *Command*-Klassen von deren Klasse *dict* und stellen somit jeweils ein spezialisiertes *Dictionary* dar (vgl. Abbildung 15).

Zur Sicherstellung der Funktionsfähigkeit eines Plug-ins benötigt dieses Kontext über vergangene Requests und Responses. Erhält ein Plug-in beispielsweise keine Information über bereits angefahrene Positionen, ist es nicht in der Lage die weiteren

Arbeitsschritte des Roboters zu planen. Hierbei stellt die Zustandsvariable *state* ein Resultat der architektonischen Einschränkung der Zustandslosigkeit durch die REST-API dar. Im Rahmen der Anforderungsanalyse wurde eingangs sowohl festgelegt REST mit den damit einhergehenden Einschränkungen vollständig zu implementieren, als auch die Clientanwendung auf der Robotersteuerung nicht anzupassen. Dies führt zu folgender Fragestellung: Wie ist der Server in der Lage dieses Prinzip einzuhalten, wenn Plug-ins Kontext benötigen und der Client gleichzeitig keine Möglichkeit hat seinen eigenen Zustand zu speichern und zu kommunizieren?

Dies führt zu folgender Lösung beziehungsweise Interpretation der REST-Prinzipien. Das Token, welches als einziges identifizierendes Merkmal vom Client übertragen wird, nimmt die Rolle einer Identifikationsnummer von Dokumenten in MongoDB ein. Dies gilt für alle in Kapitel 8.1.3 vorgestellten Kollektionen und somit auch für *app_state*. In dieser wird nämlich die Zustandsvariable *state* nach jedem Aufruf von *run(…)* gespeichert. Bei einer neuen Request des Clients wird das korrekte Dokument anschließend in MongoDB gesucht und an das soeben instanziierte Plug-in übergeben. Da weder die eigentliche Serverapplikation den Zustand über mehrere Abfragen speichert, noch dasselbe Plug-in-Objekt dauerhaft im Hintergrund existiert, wird die Zustandslosigkeit gemäß REST erfüllt, solange Server und DB als separate Entitäten im Sinne einer Drei-Schichten-Architektur betrachtet werden (vgl. Abbildung 12). Durch den Einsatz von MongoDB sollte die Applikation also entsprechend auf eine höhere Anzahl an Clients skalieren können, wobei dies weder Fokus der Arbeit ist, noch getestet wurde.

In diesem Rahmen kann *state* beliebig ausgestaltet werden, wobei zwei Bedingungen gelten. Bei der Variable muss es sich um ein *Dictionary* handeln, welches ausschließlich Datentypen enthält, die korrekt von MongoDB gespeichert werden können. Somit sind Zahlen, Strings sowie Pythons eigene Datenstrukturen wie *Dictionaries* und Listen erlaubt, Objekte dagegen nicht. Außerdem muss das Plug-in in der Lage sein, seine Befehlslogik so nahtlos fortzuführen, als würde eine einzige Plug-in-Instanz ihren Zustand über mehrere Requests hinweg speichern. Tatsächlich wird jedoch jedes Mal ein neues Objekt erzeugt.

Ein einfaches Beispiel hierfür ist das Plug-in *telepath_commands*, welches die Klasse *TestPlugin* enthält (vgl. Abbildung 15). Dieses nutzt einen Zähler zur Auswahl des korrekten Befehles, welcher mit jeder Request um eins erhöht wird. Theoretisch könnte eine Instanz einmal erzeugt werden und den Zähler dauerhaft im Hintergrund speichern. Da dieses Vorgehen jedoch die Zustandslosigkeit verletzen würde, wird stattdessen eine Zustandsvariable mit einem Schlüssel *“counter“* und dazugehörigem Zahlenwert erzeugt, welche anschließend mit einem Roboterbefehl an den Server zurückgegeben

wird. Dieser Vorgang wiederholt sich, bis der Zähler beim Stand von sechs einen EXIT-Befehl weiterleitet.

Obwohl also einige Rahmenbedingungen existieren, sind dem Design der Plug-ins in der Praxis kaum Grenzen gesetzt. Es muss lediglich eine korrekte und geeignete Implementierung der Zustandsvariable *state* umgesetzt werden. Diese Vielfalt spiegelt sich auch in den im Rahmen des Projekts entwickelten Plug-ins wider (vgl. Tabelle 12). Dennoch ist der Kontext durch diese einzelne Variable in manchen Fällen vielleicht nicht ausreichend. Dafür wurde das *MongoInterface* entworfen, welches mit einigen Sicherheitseinschränkungen direkten Zugriff auf die laufende DB-Instanz ermöglicht. Einen konkreten Anwendungsfall für diese Klasse bildet etwa *telepath_sync* ab. Hierbei muss für die Steuerung von zwei Clients durch dasselbe Plug-in, wohlgemerkt nicht dieselbe Instanz, auf den Zustand des jeweils anderen Roboters zugegriffen werden, um bereits erledigte Aufgaben zu prüfen.

Bezeichnung (ohne Präfix)	**Beschreibung**	**Einsatz des MongoInterface**
commands	Test-Plug-in: Einmalige Generierung jeder Befehlsart mit Hilfe eines Zählers	
skilltree	Test-Plug-in für die Ontologie-Anbindung. Ähnlicher Ablauf wie bei *commands*	
blueprints	Großes KI-Plug-in zur logischen Steuerung des Roboters mit einer Ontologie. Details in Kapitel 9	
sync	Demonstriert die synchronisierte Steuerung von zwei Clients durch eine einheitliche Codebasis	x

Tabelle 12 Übersicht der mitgelieferten Plug-ins

8.3 Betrachtung des Programmablaufs

Nachdem der statische Aufbau der Anwendung bereits beschrieben wurde, wird nun das Zusammenspiel der einzelnen Bestandteile während der Laufzeit betrachtet. Dabei dient der Programmablauf des Clients als Orientierung (vgl. Abbildung 17).

```
run_client():
    token = login()
    while(!Exit):
        command = get_newcommand(token)
        confirm_command(token)
        execute_command(command)
```

Abbildung 17 Pseudocode der Client-Anwendung
In Anlehnung an [22, S. 16]

8.3.1 Hochfahren des Servers

Vor der Verbindung mit dem Client muss der Server gestartet werden. Dazu dient das *CockpitWidget*, welches nach Ausführung des Skripts *main.py* auf dem Bildschirm erscheint. Durch einen Click auf die Schaltfläche *Start Server* wird die Anwendung vom Flask-Entwicklungsserver unter der im Code festgelegten Adresse zur Verfügung gestellt. Um den Nutzer über die erfolgreich abgeschlossene Startsequenz zu informieren, wird sowohl im Terminal als auch in der Statuszeile ein kurzer Text angezeigt.

8.3.2 Login

Ist der Server erst einmal aktiv, kann der Client eine *POST*-Request an */login* senden, um sich bei diesem anzumelden. Daraufhin führt die View-Funktion *login_response* eine Kette von Operationen aus, an deren Ende das Token in einer Response an den Client gesendet wird (vgl. Abbildung 18). Im Sequenzdiagramm wird die Kommunikation mit dem GUI aufgrund der besseren Übersichtlichkeit nicht betrachtet.

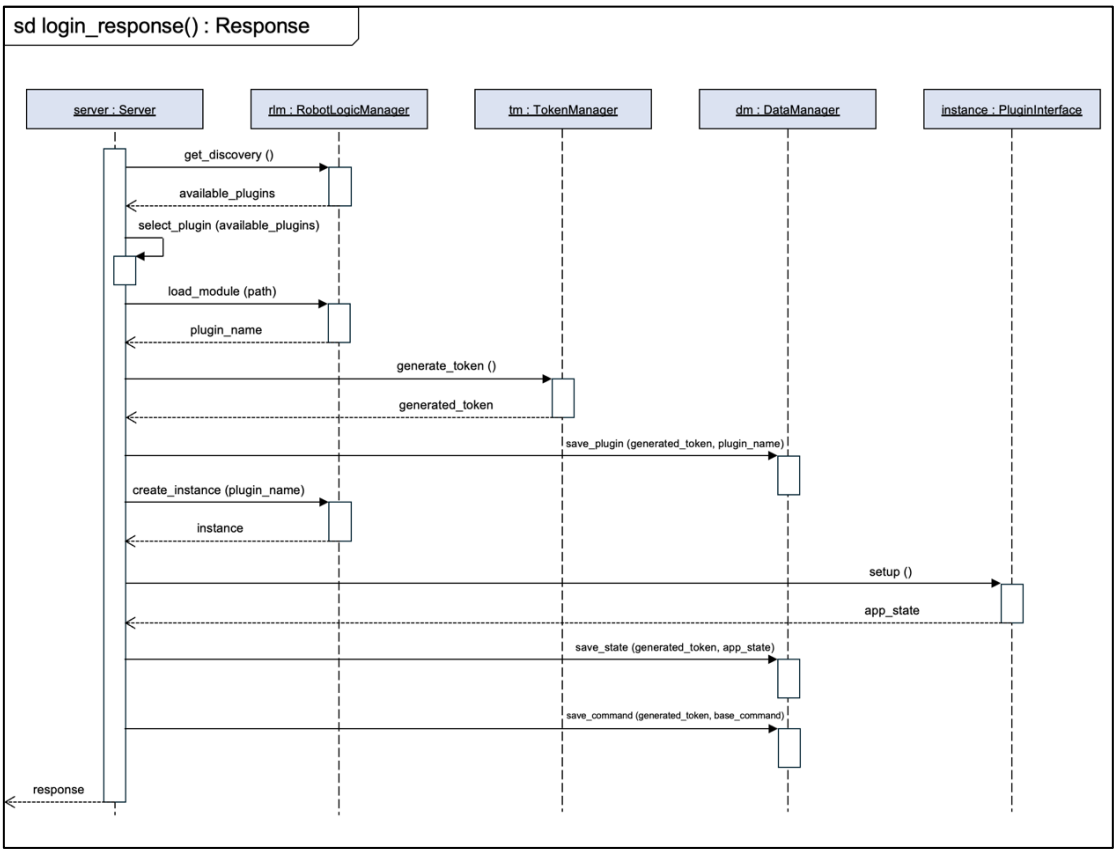

Abbildung 18 Sequenzdiagramm der Methode login_response

Zunächst ist es notwendig Informationen über die durch den RLM entdeckten Plug-ins zu erhalten. Der Server nutzt das *Dictionary available_plugins*, welches eine Zuordnung der Bezeichnungen zum entsprechenden Dateipfad abbildet, um dem Benutzer im Terminal eine Übersicht der verfügbaren Erweiterungen aufzulisten. Durch das Eintippen des gewünschten Plug-ins wird dieses vom RLM geladen und abgespeichert, um bei Bedarf ein Objekt der Hauptklasse zu instanziieren. Als Hauptklasse eines Plug-ins wird die Klasse innerhalb einer Datei mit dem Präfix *telepath_* im Plug-in-Ordner bezeichnet, welche *PluginInterface* implementiert.

Den nächsten Schritt im Login-Prozess stellt die Generierung des Authentifizierungstokens dar. Hierfür wird die Methode *generate_token()* des TM durch den Server aufgerufen, wodurch jener einen acht-stelligen String aus den erlaubten Zeichen erzeugt. Zusätzlich erfolgt eine Prüfung des Tokens. Sollte hierbei der unwahrscheinliche Fall eintreten, dass ein identisches Token bereits existiert oder dieses unerlaubte Zeichen enthält, wird ein Neues generiert.

Anschließend wird die Befehlsgenerierung bei weiteren Anfragen des Clients vorbereitet. Dafür wird zunächst der Plug-in-Name mit dem Token als Markierung vom DM in

MongoDB abgespeichert. Danach sind noch zwei weitere Variablen abzuspeichern: *base_command* und *app_state*.

Die *base_command* ist immer ein SEND-Befehl. Dieser ist notwendig, um den Client-Loop serverseitig korrekt zu spiegeln (vgl. Abbildung 17). Der Client fragt nach dem Login jeweils zuerst einen Befehl an, bestätigt diesen und führt ihn abschließend aus. Die Generierung eines neuen Roboterbefehls erfolgt jedoch erst im Rahmen der Befehlsbestätigung. Um die resultierende Lücke im ersten Loop zu schließen, wird also ein einheitlicher Startbefehl definiert. Ein Vorteil von SEND ist hierbei, dass durch eine zukünftige Anpassung des Clients gleich zu Beginn einige grundlegende Informationen wie die Roboterposition oder das Modell übergeben werden können, welche dann in den Plug-ins genutzt werden können.

Für die Generierung des *app_state* wird ein Objekt der Hauptklasse des Plug-ins durch den RLM instanziiert und an den Server zurückgegeben. Dieser ruft anschließen dessen Methode *setup()* auf, welche in Kapitel 8.2 nur oberflächlich vorgestellt wurde. Der Einsatz von Zustandsvariablen zur Wiederherstellung des benötigten Serverzustands bei jeder Request macht einen Ur-Zustand notwendig, welcher bereits beim ersten Aufruf der Hauptmethode *run(…)* eines Plug-ins an diese übergeben werden kann. Eine solche Zustandsvariable wird während des Logins einmalig durch *setup()* erzeugt.

Nach Abschluss dieser vorbereitenden Tätigkeiten wird ein Response-Objekt erzeugt, welches das Token im JSON-Format enthält, und an den Client gesendet. Dieser ist nun erfolgreich eingeloggt, wie durch die Tabelle im Widget ersichtlich ist.

8.3.3 Bereitstellung eines Befehls

Nach dem erfolgreichen Login wird der Client eine *GET*-Request an */newcommand/{token}* senden. Dies ruft automatisch die View-Funktion *command_response* auf, welche das Token aus der URI als einziges Argument bezieht.

Innerhalb der Methode wird das Token zunächst aus der Darstellung im JSON-Format innerhalb der URI extrahiert. Die nächsten Schritte sind von einer erfolgreichen Prüfung seiner Authentizität abhängig. Ist das Token bekannt, wird der zuletzt generierte Befehl aus MongoDB abgefragt. Direkt nach der Anmeldung handelt es sich hierbei immer um SEND. Dieser Befehl wird innerhalb der Response an den Client geschickt (vgl. Abbildung 19). Ist das Token dagegen unbekannt, wird eine Response mit dem Statuscode 401 erstellt, welche dem Client anzeigt, dass er keine Autorisierung für den Zugriff auf diese Ressource hat.

An die Methode *GET* werden besondere Anforderungen gestellt. Zunächst muss diese idempotent sein. Da bei einem Aufruf von *command*_response nur die Prüfung eines Wahrheitswerts erfolgt und gegebenenfalls Daten aus MongoDB abgefragt werden, kann dieser Prozess beliebig oft mit demselben Ergebnis wiederholt werden. Außerdem muss die Methode sicher sein, indem keine Zustandsänderung wie etwa das Löschen von Daten durch den Client angefordert wird. Durch die ausschließliche Abfrage der DB wird auch diese Anforderung erfüllt. Somit entspricht die Implementierung von *GET /newcommand/{token}* der HTTP-Spezifikation (vgl. Tabelle 1).

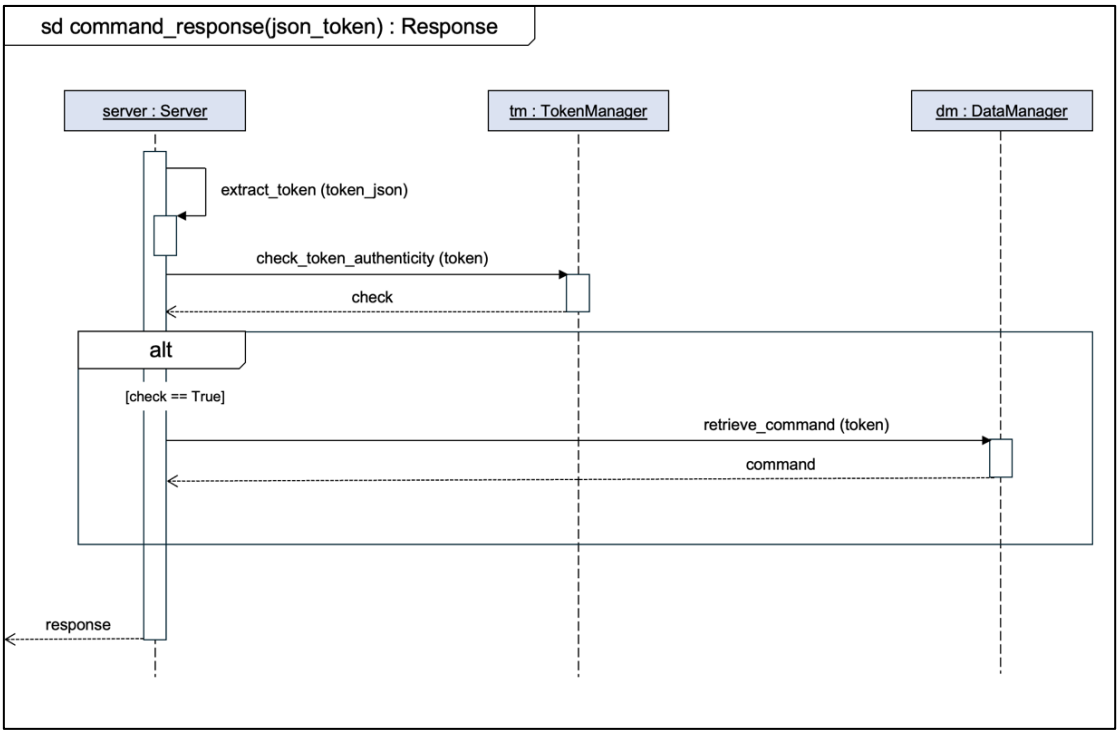

Abbildung 19 Sequenzdiagramm der Methode command_response

8.3.4 Bestätigung und Generierung eines Befehls

Der Erhalt jedes Befehls wird vom Client mit einer *POST*-Request an */newcommand/{token}* bestätigt. Dies dient gleichzeitig als Signal für die Generierung des nächsten Befehls durch die View-Funktion *command_confirmation* (vgl. Abbildung 20).

Wie bei allen Methoden außer *login_response* wird das Token zuerst aus der URI übernommen und aus dem Text in JSON-Format extrahiert. Ist dieses ungültig, erhält der Client wie zuvor den Statuscode 401. Bei Übermittlung eines korrekten Tokens werden zunächst die Zustandsvariable des Plug-ins, dessen Name sowie das aktuelle Kommando aus der DB abgefragt. Somit wird aus dem Kontext der Request in Form des Tokens der notwendige Zustand der Serveranwendung konstruiert.

Auf Basis des Befehlstyps erfolgt nun eine Entscheidung über das Verhalten des Servers. Handelt es sich beim aktuellen Befehl um ein EXIT-Kommando, bereitet der Server die bevorstehende Abmeldung des Clients vor. Hierbei werden mit dem Token assoziierte Daten in MongoDB gelöscht und das Widget aktualisiert. Die tatsächliche Abmeldung erfolgt anschließend auf der Client-Seite durch Ausführung des Kommandos.

Sollte es sich um eine andere Art von Kommando handeln, wird ein Folgebefehl generiert. Analog zu *login_response* wird mit Hilfe des *plugin_name* vom RLM eine neue Plug-in-Instanz erzeugt und an den Server übergeben. Diesmal wird jedoch die Methode *run(…)* statt *setup()* aufgerufen, welche den Roboterbefehl und eine neue Zustandsvariable zurückgibt. Anschließend werden beide in MongoDB abgespeichert.

Da es sich bei der Client-Anfrage um eine *POST*-Request handelt, ist diese weder sicher noch idempotent. Bei mehrfachen Anfragen wird somit stets der nächste Befehl erzeugt. Die korrekte Reihenfolge der Kommandos wird hierbei durch den Client-Loop sichergestellt, in welchem diese Anfrage niemals mehrfach hintereinander gesendet wird.

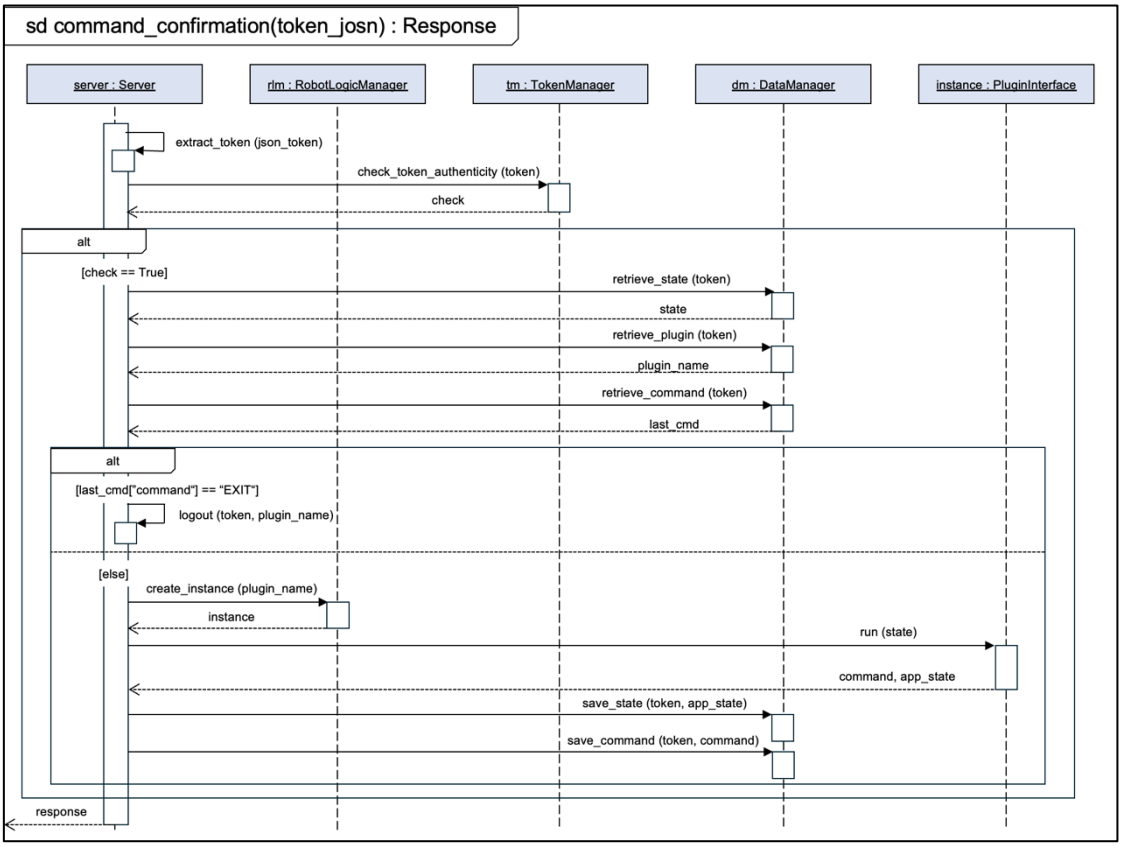

Abbildung 20 Sequenzdiagramm der Methode command_confirmation

8.3.5 Übermittlung von Roboterdaten

Für die Kommandoarten SEND und LOG existiert in Verbindung mit den Ressourcen */saveinfo/{token}* und */savelog/{token}* jeweils eine View-Funktion. An diese Endpunkte werden Requests gesendet, nachdem der entsprechende Befehl vom Server an den Client übermittelt wurde.

Die Funktionalität der zugehörigen Methoden *saveinfo_response* und *savelog_response* ist nahezu identisch. Abhängig von einer erfolgreichen Überprüfung des Tokens und des Medienformats werden die übermittelten Informationen in der jeweiligen Kollektion *infos* oder *logs* abgespeichert. Zu beachten ist hierbei lediglich, dass es sich im Fall von SEND um eine *PUT*-Request handelt, welche gemäß Spezifikation idempotent sein muss (vgl. Tabelle 1). Zur Einhaltung dieser Vorgabe ist im Gegensatz zu den Log-Dateien immer nur ein Dokument pro Token in der Kollektion *infos* zulässig. Bei aufeinanderfolgenden Anfragen verändert sich der Serverzustand also nicht, womit die Methode idempotent ist.

8.3.6 Abschalten des Servers

Ist die Aufgabe des Roboters abgeschlossen und wird der Server daher nicht weiter benötigt, kann dieser über die Schaltfläche *Shutdown Server* des GUI heruntergefahren werden. Dadurch wird zunächst der laufende MongoDB-Prozess beendet. Im Anschluss wird der Flask-Server gestoppt, wodurch ebenfalls das GUI geschlossen wird. Um zu vermeiden, dass die Abschaltsequenz aus Versehen aktiviert wird, muss die Entscheidung in einem Dialogfenster bestätigt werden.

9 Implementierung der KI-Applikation

Auf Grundlage des Servergerüsts können nun beliebige Plug-ins entwickelt werden. Dabei ist die Untersuchung von Ontologien als Steuerungsmechanismus für Roboter ein weiteres Ziel der Arbeit. Dafür ist zunächst eine eigenständige Analyse der fachlichen Domäne und der Anforderungen an die Anwendung notwendig. Anschließend wird das Design der Ontologie erläutert. Den Abschluss stellt eine Betrachtung des Plug-ins dar, welches das Wissensmodell zur Befehlsgenerierung heranzieht.

9.1 Analyse der Domäne und Plug-in-Anforderungen

Als Domäne der Ontologie kann die Schnittmenge der Bereiche Robotik, Produktion und Konstruktion betrachtet werden. Ein Produkt besteht aus mehreren Teilen, für die im Rahmen der Produktentwicklung Vorgaben für den korrekten Zusammenbau festgelegt werden. An einem Automobil muss beispielsweise eine Radnabe vorhanden sein, damit anschließend das Bremssystem und schließlich das Rad montiert werden können. Somit ergibt sich eine logische Verkettung von Aktionen, welche aber insbesondere bei komplexeren Produkten nicht notwendigerweise linear ist. In obigem Beispiel mag es zwar eine feste Reihenfolge der Bauteile an jedem Rad geben, jedoch ist nicht explizit festgelegt, mit welchem der vier Räder zu beginnen ist. Daraus resultiert eine gewisse Flexibilität bzw. Dynamik in der Entscheidungsfindung. In einer Robotikanwendung sind nun noch einige zusätzliche Informationen über die Roboterbewegung und das benötigte Werkzeug für jeden Arbeitsschritt notwendig.

Auf Grundlage der betrachteten Domäne sowie der Serverarchitektur lassen sich nun einige Anforderungen an die Applikation definieren (vgl. Tabelle 13). Diese ergänzen die Anforderungen aus Kapitel 5 und insbesondere 5.2. Dort wurden zunächst nur sehr allgemeine Anforderungen an die KI-Applikation gestellt.

Auf Seite der funktionalen Anforderungen muss das Plug-in einen Suchalgorithmus bieten, welcher auf Grundlage des modellierten Wissens eine korrekte Arbeitsreihenfolge bestimmen kann. Um diese sinnvoll mit dem Roboter zu nutzen, müssen die einzelnen Arbeitsschritte Informationen über die gewünschte Bewegung oder Aktion liefern.

Bezüglich der nicht-funktionalen Anforderungen ist zunächst die korrekte Implementierung des Plug-in-Interface zu nennen (vgl. Kapitel 8.2). Einen interessanten Ansatz für das Design der Ontologie könnte die Integration eines allgemeinen Wissensmodells für Roboter darstellen. Hierfür sind beispielsweise Ontologien wie die

Core Ontology for Robotics and Automation (CORA) verfügbar [56]. Die Wiederverwendung eines solchen Modells kann den Modellierungsaufwand reduzieren. Zuletzt sollen die Funktionen des Plug-ins den aktuellen Zustand des Roboters widerspiegeln. Es werden also beispielsweise keine Werkzeuge oder Hindernisse in der Ontologie modelliert, da derzeit keine Fähigkeiten für Werkzeugwechsel oder Hinderniserkennung implementiert sind.

Nr.	Typ	Anforderung	Priorität
31	FA	Modellierung von Arbeitsschritten	S
32	FA	Modellierung der Bedingungen jedes Arbeitsschritts	S
33	FA	Entwicklung eines geeigneten Suchalgorithmus zur Bestimmung einer korrekten Arbeitsfolge	S
34	FA	Generierung korrekt formatierter Roboterbefehle	S
35	FA	Support für alle Befehlsarten	S
36	NFA	Schnittstelle zwischen Plug-in und Ontologie	S
37	NFA	Speichern mehrerer Aufgaben in der Wissensbasis	S
38	NFA	Prüfung der Integration einer standardisierten Ontologie	W
39	NFA	Fokus auf Robotersteuerung auf Grundlage aktueller Fähigkeiten	S
40	NFA	Korrekte Implementierung des Plug-in Interface	S

Tabelle 13 Anforderungen an das KI-Plug-in

9.2 Entwurf der Ontologie

Gemäß dem vorgestellten Prozess der Ontologieentwicklung (vgl. Abbildung 11) werden im Anschluss an die Anforderungsanalyse Konzepte innerhalb des Wissensmodells definiert. Hierbei wurden vier Klassen mit jeweils eigenen Subklassen und *Properties* festgelegt: *Product*, *Part*, *Instruction* und *Action*.

Das *Product* stellt ein Gesamtprodukt dar, welches aus verschiedenen Einzelteilen besteht, die hier als *Part* bezeichnet werden. Jeder *Part* hat eine eigene Bauanleitung *Instruction*, welche jeden Einzelschritt für dessen Einbau als *Action* umfasst.

Für die Funktionalität des Plug-ins ist es besonders wichtig die Beziehungen zwischen verschiedenen *Actions* und *Parts* zu modellieren. Für Erstere wurde als Vereinfachung festgelegt, dass die korrekte Reihenfolge wie beispielsweise in einer realen Bauanleitung für ein Möbelstück festgelegt ist. Jede *Action* ist also maximal durch eine vorherige *Action* bedingt und hat maximal eine Folgeaktion. Für Letztere gilt diese Festlegung nicht, um das gewünschte dynamische Verhalten der Applikation zu ermöglichen. So kann es für einen *Part* mehrere vorangehende Teile, welche eine Bedingung darstellen, und mehrere nachfolgende Teile geben, deren Einbau erst durch das aktuelle Teil ermöglicht wird.

Die Modellierung dieser Beziehungen erfolgt jeweils über ein Paar aus inversen *Object Properties*, welche in der Übersicht hervorgehoben sind (vgl. Abbildung 21). Dabei empfiehlt es sich, immer nur die Bedingungen expliziert zu modellieren, wohingegen Nachfolger anschließend durch den *Reasoner* aus der Ontologie geschlussfolgert werden können. Dieses Vorgehen reduziert den Aufwand für die Modellierung eines konkreten *Products*.

Die Informationen zur tatsächlichen Steuerung des Roboters werden durch die *Data Properties* der Klasse *Action* abgebildet. Das Attribut *hasCommand* dient zur Speicherung des Befehlstyps wie beispielsweise CIRC. Für die Übergabe der Koordinaten dienen die *Data Properties hasFrame*, *hasAuxiliaryFrame* und *hasDestination*. Hierbei wird zwischen drei Subklassen von *Action* unterschieden. Ein *OnePointMovement* wird durch einen Punkt definiert und entspricht PTP oder LIN. Diese Befehle wurden zusammengefasst, da sie sich nur durch den Wert von *hasCommand* unterscheiden und somit vom *Reasoner* als äquivalent aufgefasst wurden. Ein *TwoPointMovement* wie CIRC wird dagegen durch zwei Punkte im Raum definiert und stellt somit eine separate Klasse dar. Besteht der Wunsch, die restlichen Roboterbefehle zu nutzen, ist die Subklasse *Communication* verfügbar, welche nur über das Attribut *hasCommand* verfügt.

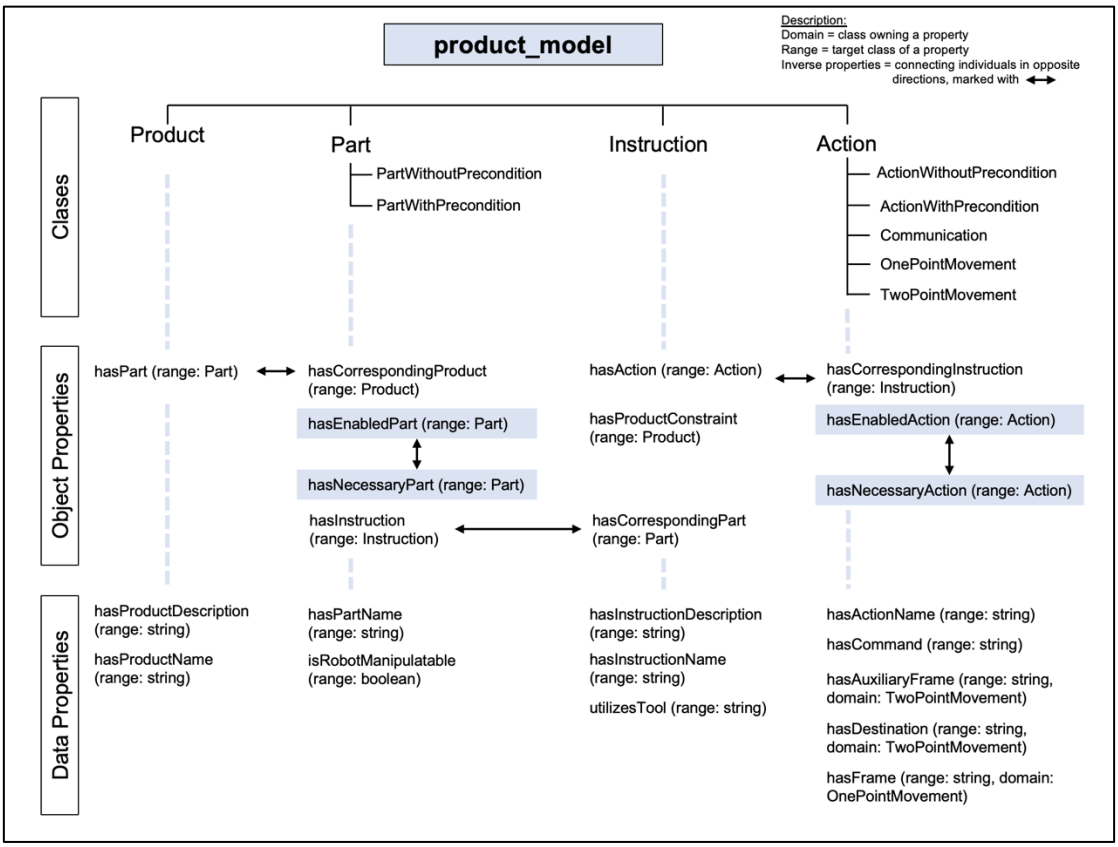

Abbildung 21 Übersicht wichtiger Konzepte in der Ontologie product_model

Mit Hilfe dieser *Classes* und *Properties* kann jedes beliebige Produkt ohne großes Vorwissen modelliert werden. Um die Einfachheit dieses Modells beizubehalten, wird es nicht mit einer allgemeinen Ontologie wie CORA kombiniert. Der Nutzen einer solchen Integration ist in diesem Fall begrenzt.

9.3 Integration der Ontologie in ein Server-Plug-in

Die Implementierung der KI-Applikation umfasst neben dem Entwurf der Ontologie zwei weitere Tätigkeiten: die korrekte Implementierung der Schnittstelle und die Anbindung des Wissensmodells durch einen geeigneten Suchalgorithmus. Für die Abfrage der Ontologie wird die Bibliothek OWLready2 verwendet (vgl. Abbildung 14). Ebenfalls erhält die Applikation wie in Tabelle 12 dargestellt den Namen *blueprints*.

Um als Plug-in vom Server genutzt werden zu können, sind neben der korrekten Datei-Bezeichnung mit Präfix *telepath_* einige weitere Punkte zu beachten. Zusätzlich erbt die einzige Klasse dieses Moduls die Methoden *__init__(), setup()* und *run(…)* von *PluginInterface*, welche nun so implementiert werden müssen, dass sie die erwarteten Rückgabewerte liefern (vgl. Abbildung 22).

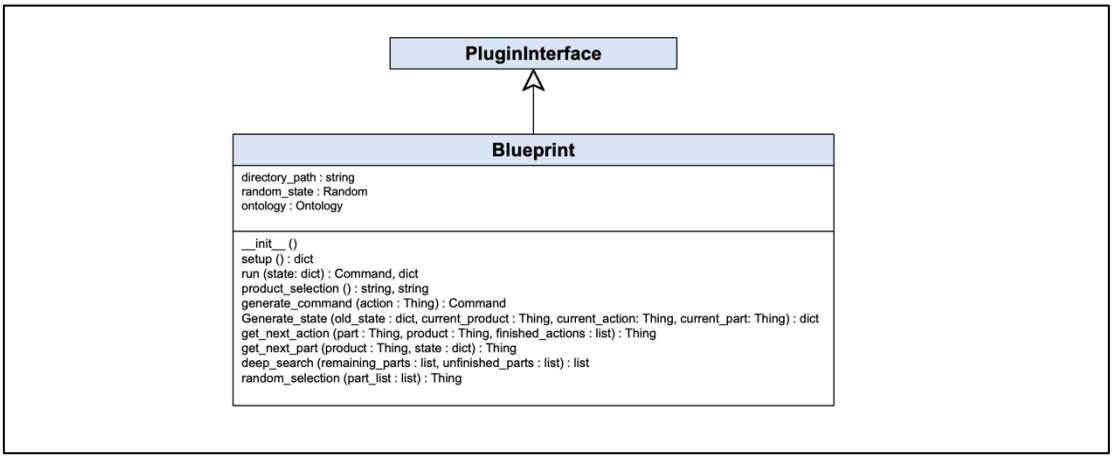

Abbildung 22 Klassendiagramm des Plug-ins telepath_blueprints

Der Konstruktor *__init__()* wird infolge der Instanziierung eines neuen Plug-in-Objekts bei jeder *POST*-Request an */newcommand/{token}* aufgerufen. Hierbei werden lediglich einige grundlegende Attribute wie der Dateipfad zu den Ontologien und der sogenannte *random_state* festgelegt. Dieser ermöglicht das Generieren von Zufallszahlen, wobei Ergebnisse unter Verwendung eines *Seeds* dennoch reproduzierbar sind.

Die Methode *setup()* erlaubt dem Nutzer zunächst per Texteingabe im Terminal die Ontologie und das Produkt, welches vom Roboter zusammengebaut werden soll, auszuwählen. Aus dem Produkt wird ein *Part* ohne Vorbedingungen in Form eines anderen Teils als Startpunkt ausgewählt. Die Namen des Produkts und des aktuellen Teils werden mit jeweils einer Liste für abgeschlossene Aktionen und verbaute Teile sowie dem Dateipfad der Ontologie in der Zustandsvariable *state* an den Server zurückgegeben. Diese wird im weiteren Verlauf kontinuierlich aktualisiert.

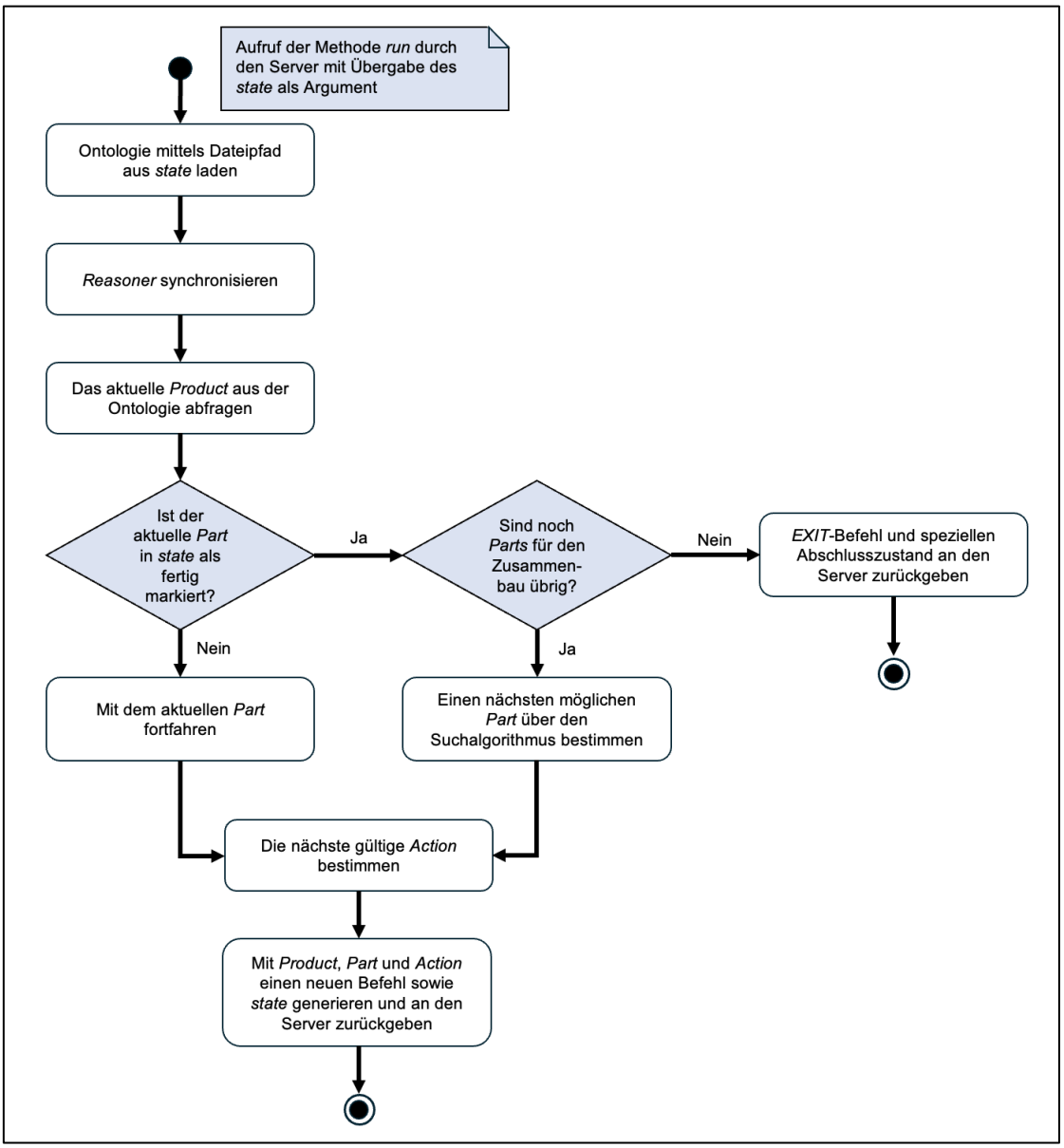

Abbildung 23 Ablaufdiagramm der Methode run(…) des Plug-ins telepath_blueprints

Um einen Befehl zu erhalten, wird der Server die Hauptmethode *run(…)* aufrufen (vgl. Abbildung 23). Innerhalb dieser Methode werden die korrekten *Individuals* für *Product*, *Part, Instruction* und *Action* aus der Ontologie abgefragt und zur Erstellung eines Roboterbefehls sowie der Aktualisierung der Zustandsvariable herangezogen.

Da die Kette der Einzelaktionen deterministisch modelliert ist, also eine feste Reihenfolge existiert, ist es relativ einfach die nächste Aktion zu bestimmen. Dafür werden in der Methode *get_next_action(…)* beginnend mit der einzigen Aktion ohne Bedingung solange die Folgeaktionen mittels der *Object Property hasEnabledAction* abgefragt, bis eine davon nicht als abgeschlossen in der Zustandsvariable gespeichert ist. Anschließend wird für diese *Action* über die Methode *generate_command(…)* der Roboterbefehl mit Parametern aus der Ontologie abgefragt.

Dagegen gestaltet sich die Suche nach möglichen *Parts* deutlich komplexer. Da diese nicht deterministisch modelliert sind, sondern verschiedene Optionen verfügbar sein können, besteht bei der zuvor beschriebenen Suchstrategie die Gefahr, Teile zu vergessen, wenn ausschließlich ein Pfad verketteter *Parts* betrachtet werden sollte.

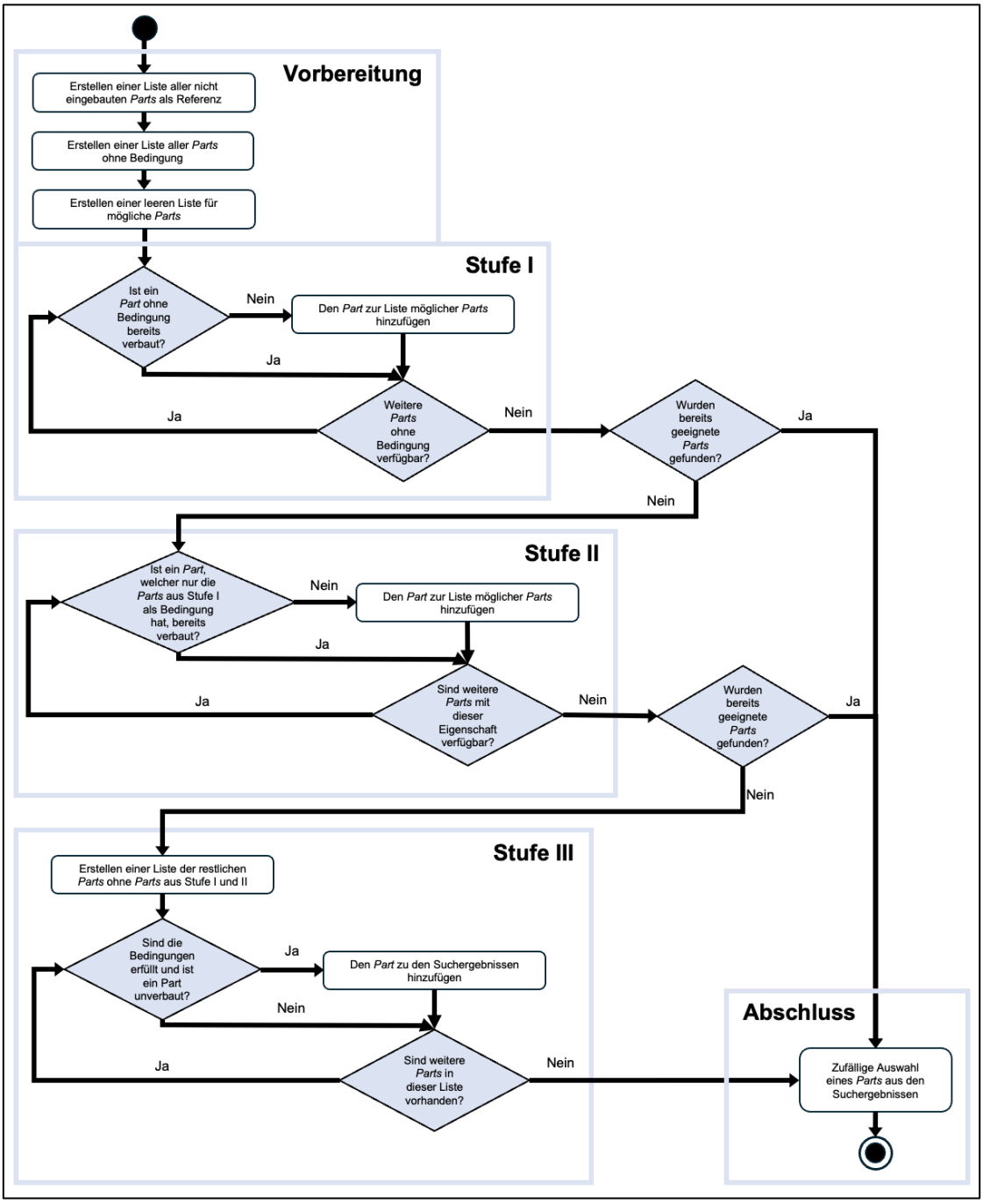

Abbildung 24 Ablaufdiagramm des Suchalgorithmus für Einzelteile im Plug-in telepath_blueprints

Stattdessen wurde hierfür ein dreistufiger Suchalgorithmus durch die Methoden *get_next_part(…)*, *deep_search(…)* und *random_selection(…)* definiert (vgl. Abbildung 24). Um das Probleme der verschiedenen Suchpfade zu lösen, beginnt dieser immer mit

allen Teilen ohne jegliche Bedingungen. Sollte hierbei kein passender *Part* gefunden werden, beginnt die nächste Stufe der Suche. Nun werden die Teile untersucht, welche ausschließlich diejenigen aus Stufe 1 als Bedingung aufweisen. Im Fall, dass weiterhin kein geeignetes Teil für den Einbau gefunden wurde, werden nun die übrigen Teile betrachtet. Um eine valide Option darzustellen, müssen die entsprechenden Bedingungen erfüllt sein und der *Part* selbst darf noch nicht verbaut sein. Unabhängig von der Phase, in der die Suche beendet wird, erfolgt die Auswahl aus den Suchergebnissen durch Zufall. Ist das Produkt bereits fertig zusammengebaut, werden keine Teile mehr durch den Algorithmus gefunden. Folglich wird gemäß Abbildung 23 ein EXIT-Befehl an den Server zurückgegeben.

10 Integrationstest im Labor

Bisher wurden sämtliche Tests der Anwendung entweder innerhalb der Entwicklungsumgebung VS Code oder mittels Postman durchgeführt. Da nun sowohl die Serverapplikation als auch das KI-Plug-in vollständig implementiert sind, können Experimente direkt am Roboter durchgeführt werden.

10.1 Planung des Experiments

Um die Funktionsfähigkeit des Plug-ins zu demonstrieren, muss die Basisontologie zunächst um geeignete *Individuals* ergänzt werden. Zwischenzeitlich ist ein Greifer am Cobot im Labor für Produktionstechnik installiert worden, welcher jedoch nicht über den Server angesteuert werden kann. Dies schränkt die Gestaltung eines Experiments signifikant ein.

Daher wurde die Entscheidung getroffen, Produkte mittels Zeichnungen zu abstrahieren. Hierbei ist lediglich notwendig, den Roboter zu Beginn mit einem Filzstift auszustatten. Einzelne Linien oder Linienzüge können hierbei als *Parts* betrachtet werden, welche in Summe das *Product* in Form der fertigen Zeichnung darstellen.

Als erstes Beispielprodukt wurde ein Haus gewählt (vgl. Abbildung 25). Dieses folgt dem logischen Aufbau eines realen Gebäudes dahingehend, dass zunächst das Fundament, dann die Wände, und abschließend das Fenster, die Tür und das Dach gezeichnet werden. Um die Fähigkeit des Plug-ins zu demonstrieren, flexibel auf veränderte Bedingungen zu reagieren, gibt es eine alternative Version dieser Ontologie. Darin muss das Dach stets vor dem Fenster und der Tür gezeichnet werden.

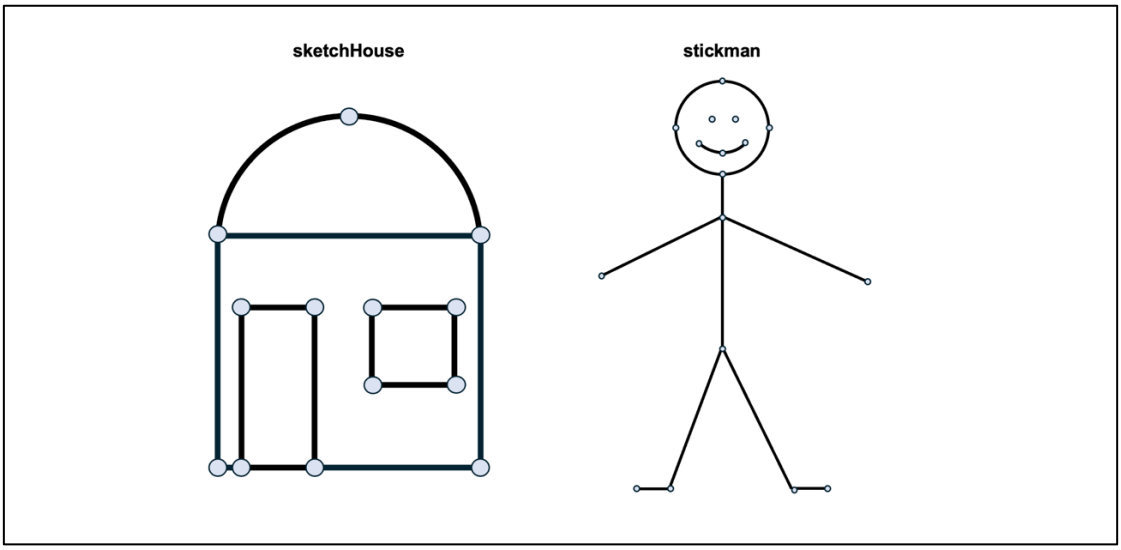

Abbildung 25 Die Produkte sketchHouse und stickman

Während das Haus also die Flexibilität des Suchalgorithmus demonstriert, soll ein Strichmännchen als zweites Produkt dessen Eignung für längere Ketten an Bedingungen verdeutlichen. Die Zeichnung der Figur beginnt bei den Füßen und endet mit dem Kopf und Gesicht (vgl. Abbildung 25). Jedes Körperteil ist durch einen individuellen *Part* abgebildet, sodass deren Gesamtzahl im Vergleich zum Haus höher ist.

Für jede einzelne Aktion in diesen beiden Ontologien müssen die Bewegungsart und die Zielkoordinaten bestimmt werden. Dafür wurde zunächst eine behelfsmäßige Halterung für einen Filzstift hergestellt. Als Vorlage wurden die beiden Produkte anschließend auf jeweils einem DIN A3 Zeichenkarton gezeichnet. Mit Hilfe der Halterung und dieser Vorlagen, wurde jeder benötigte Punkt manuell über das KUKA smartPAD angefahren, um dessen Koordinaten zu notieren (vgl. Abbildung 26). Diese wurden anschließend manuell in die verschiedenen Ontologien übertragen.

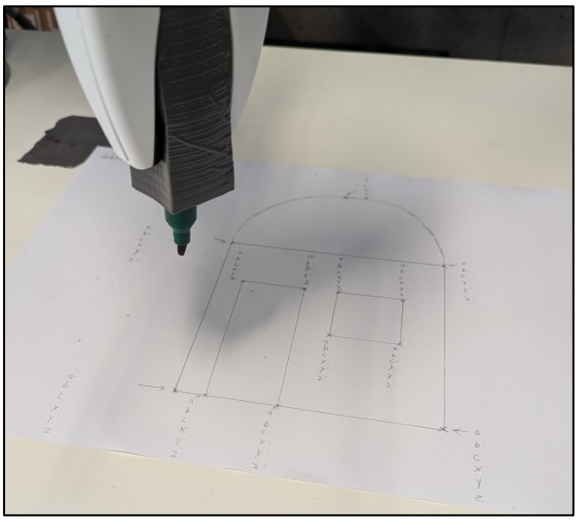

Abbildung 26 Durch den Robotergreifer fixierte Filzstift-Halterung

10.2 Durchführung und Ergebnisse des Tests

Nach der Vorbereitung der Ontologien konnte die Serverapplikation am Roboter getestet werden. Dafür wurde das MacBook Pro, auf dem die Anwendung in VS Code erstellt wurde, über ein Ethernet-Kabel an die Robotersteuerung angeschlossen. Die IP des Rechners wurde ebenso wie der Port und Host im Code angepasst, um eine Verbindung zwischen Client und Server zu ermöglichen. Ebenfalls wurde ein schwarzer Filzstift in der Halterung platziert und vom Greifer aufgenommen. Einen Überblick des Versuchsaufbaus zeigt Abbildung 27.

Zunächst wurde die grundlegende Funktionsfähigkeit des Systems über das Plug-in *telepath_commands* getestet. Nach einigen Anpassungen konnte der Roboter problemlos durch den Server gesteuert werden und dabei die verschiedenen Befehlsarten ausführen. Währenddessen stellten das GUI und das Terminal dem Nutzer laufend Informationen bereit. Somit wurde der erste Test des Servers erfolgreich abgeschlossen.

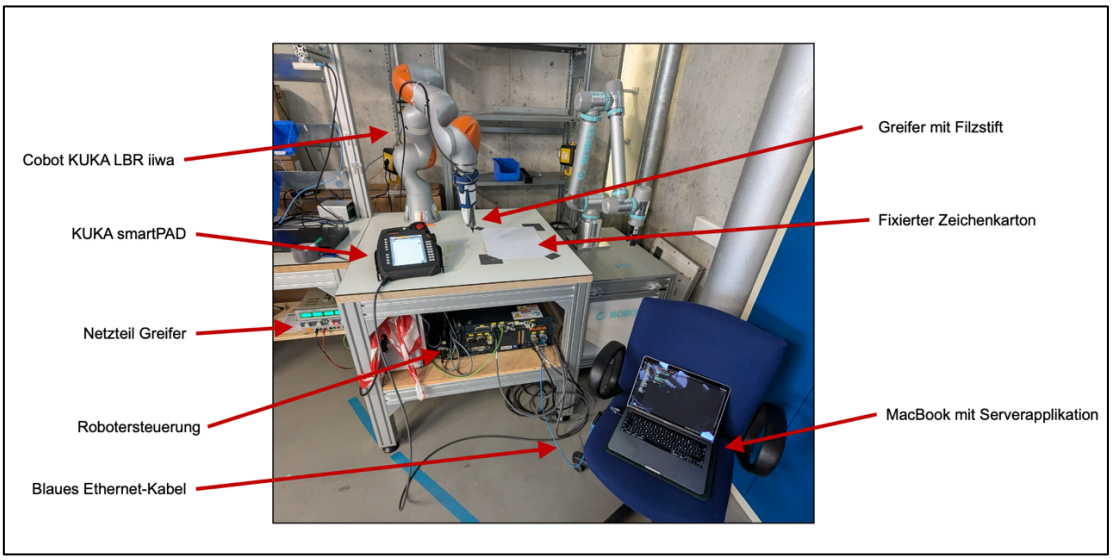

Abbildung 27 Versuchsaufbau im Labor

Daraufhin wurde das Plug-in *telepath_blueprints* getestet. Gemäß Sicherheitsunterweisung wurden zunächst einige Testläufe beider Produktarten mit reduzierter Verfahrgeschwindigkeit und manueller Steuerung des Programmablaufs unternommen. Da diese ohne Störungen abliefen, konnte der Roboter im weiteren Verlauf automatisch und mit voller Geschwindigkeit betrieben werden.

Zur Demonstration des Suchalgorithmus wurde das *sketchHouse* in originaler und alternativer Fassung herangezogen. Um die Wiederholbarkeit der Ergebnisse zu garantieren, wurden Seeds für den Zufallsgenerator innerhalb des Plug-ins genutzt (vgl.

Tabelle 14). Wie erwartet variiert die Reihenfolge der gezeichneten Parts in Einklang mit den modellierten Bedingungen bei verschiedenen Seeds. Gleichzeitig wird bei einem identischen Seed im Vergleich zwischen den beiden Ontologien klar, dass die neuen Bedingungen direkt vom Suchalgorithmus umgesetzt werden.

Reihenfolge der Teile	**Seed = 35**		**Seed = 42**		**Seed = 99**	
	original	*alternativ*	*original*	*alternativ*	*original*	*alternativ*
1	Fundament	Fundament	Fundament	Fundament	Fundament	Fundament
2	Wand rechts	Wand rechts	Wand links	Wand links	Wand rechts	Wand rechts
3	Wand links	Wand links	Wand rechts	Wand rechts	Wand links	Wand links
4	Fenster	Dach	Fenster	Dach	Tür	Dach
5	Dach	Fenster	Tür	Tür	Dach	Fenster
6	Tür	Tür	Dach	Fenster	Fenster	Tür

Tabelle 14 Übersicht der Versuche zu sketchHouse mit verschiedenen Seeds

Auch das zweite *Product stickman* konnte ohne Probleme durch den Roboter gezeichnet werden (vgl. Abbildung 28). Durch die erfolgreichen Testläufe wurde auch die Funktionalität der KI-Anwendung *telepath_blueprints* am Roboter demonstriert. Damit gelten der Integrationstest und die Implementierung des Systems als abgeschlossen.

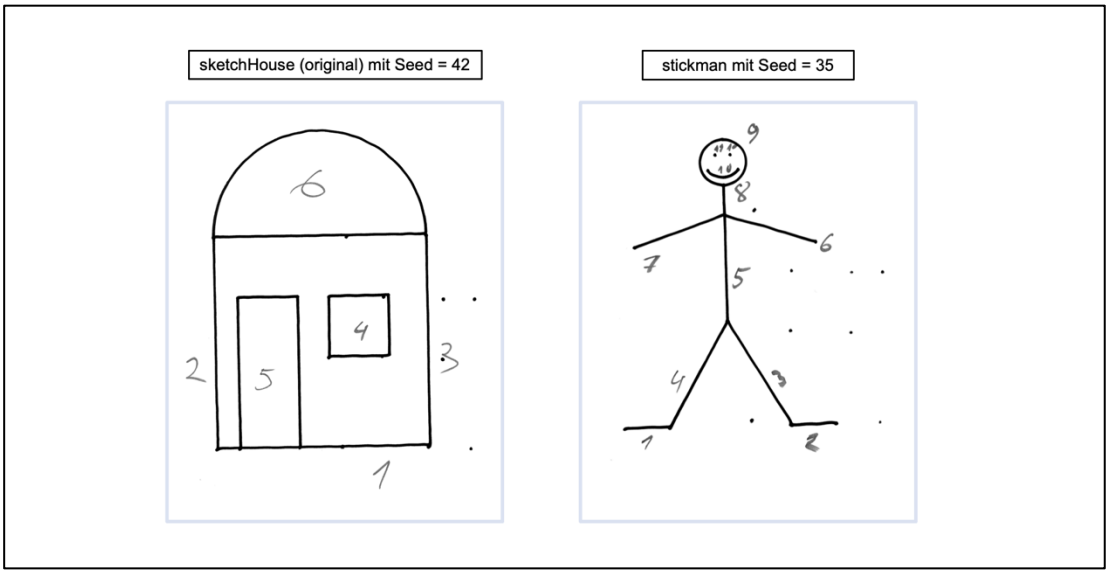

Abbildung 28 Durch den Roboter erstellte Skizzen von stickman und sketchHouse

11 Bewertung der Applikation

Die abschließende Bewertung der Anwendung erfolgt in zwei Teilen: durch Verifizierung und Validierung.

11.1 Verifizierung

Zunächst werden die verschiedenen Anforderungen erneut betrachtet, um zu ermitteln, inwieweit die Applikation diese erfüllt.

In Bezug auf die Funktionalität des Servers wurden diese vollständig erfüllt. Die verschiedenen Endpunkte der API werden ebenso abgedeckt, wie die Funktionen rund um das Token-Management sowie die Speicherung der Daten, welche über die Befehle LOG und SEND durch den Roboter versendet werden. Dabei wurden die zu Beginn diskutierten Rahmenbedingungen eingehalten, was beispielsweise an der Implementierung der View-Funktionen für *GET /newcommand* und *PUT /saveinfo* gemäß HTTP-Spezifikation ersichtlich ist (vgl. Kapitel 8.3.3 und 8.3.5).

Durch die implementierte Plug-in Architektur sind künftige Entwickler in der Lage, neue Steuerungssysteme für den Roboter mit geringem Aufwand in den Server zu integrieren. Anstatt auf die Kommunikationsmuster zwischen den Komponenten können sich diese nun auf das Design ihrer eigenen Anwendung fokussieren Die Schnittstelle ist minimalistisch und ist vor dem Hintergrund der REST-API entworfen worden, wodurch zahlreiche Plug-ins denkbar sind.

Dabei ist insbesondere das Prinzip der Zustandslosigkeit zu beachten. Wie bereits in Kapitel 8.2 erläutert kann der Server als zustandlos betrachtet werden, insofern die Datenbank, welche zur Wiederherstellung des notwendigen Serverzustands bei jeder Request herangezogen wird, als separate Entität innerhalb einer Drei-Schichten-Architektur betrachtet wird. Dennoch sollte das System in der Lage sein, den Kerngedanken hinter diesem Prinzip zu erfüllen, nämlich die Skalierbarkeit der Anwendung. Eine typische Interpretation des REST-Prinzips würde dagegen eine Anpassung des Clients erfordern, was zu Beginn des Projekts jedoch ausgeschlossen wurde. Dadurch könnten Information über Plug-ins und deren Zustände innerhalb von Responses an den Client geschickt werden, welcher diese Information verwaltet und bei Bedarf über Requests an den Server versendet. Die DB wäre somit nur noch für die Speicherung von Log-Dateien nötig.

Da der Server in Python implementiert wurde, ist es naheliegend, diese Programmiersprache auch für Plug-ins zu nutzen. Theoretisch kann jedoch eine eigene

Schnittstelle innerhalb eines Plug-ins implementiert werden, worüber dann die eigentliche Befehlslogik angebunden wird. Diese könnte dann in einer beliebigen Sprache programmiert werden. Dieses Konzept wurde bisher weder detailliert untersucht noch umgesetzt, führt jedoch zu einem deutlichen Mehraufwand. Außerdem profitiert Python von einer enormen Community und zahlreichen Bibliotheken im Bereich KI.

Ebenfalls werden durch den erfolgreichen Test im Labor die Anforderungen im Bereich Testing erfüllt. Im Zusammenhang mit dem Release der Applikation sollte hierbei die Plattformunabhängigkeit genauer untersucht werden. Die angestrebte Containerisierung mittels Docker wurde aufgrund von Problemen mit Qt5 aufgegeben. Da das GUI eine verbesserte Nutzerfahrung bei Experimenten oder Tests während der Entwicklung von Plug-ins bietet, hat dieses den Vorzug erhalten. Um die Wiederverwendung des Servers dennoch sicherzustellen, wurde die Applikation neben MacOS als Entwicklungsplattform auch auf Windows 10 getestet. Hierbei sorgen wie bereits in Kapitel 8.1.3 erläutert Unterschiede im MongoDB-Prozess zwischen den Plattformen für Probleme. Während die Anwendung unter MacOS reibungslos startet, friert das GUI auf Windows 10 zunächst ein. Wird die Ausführung gestoppt und die Anwendung über das Skript *main.py* erneut gestartet, kann diese ohne weitere Probleme genutzt werden. Bisher konnte hierfür noch keine dauerhafte Lösung gefunden werden. Dennoch sei angemerkt, dass ein schneller Neustart der Applikation nicht unbedingt langsamer oder komplexer ist als ein Start der DB über das Terminal oder ein separates Skript.

Neben Docker wurde auch die geplante Nutzung von Gunicorn verworfen. Der zusätzlichen Komplexität und erneuten Problemen im Zusammenhang mit dem GUI, standen ein geringer Mehrwert für die aktuelle Verwendung des Servers in einer Laborumgebung gegenüber. Dagegen wurden die ursprünglichen Anforderungen an die Dokumentation vollständig erfüllt. Neben dieser Arbeit enthält das GitHub-Repository eine ausführliche README-Datei. Auch der Code selbst ist kommentiert, um Anpassungen oder dessen Wiederverwendung zu erleichtern.

Neben dem Server wurden auch Anforderungen an die KI-Applikation definiert. Um diese zu erfüllen, wurde ein Steuerungssystem für den Roboter umgesetzt, welches Python-Code und eine Ontologie kombiniert, um gültige Roboterbefehle zu ermitteln. Das Grundmodell aus *Products*, *Instructions*, *Parts* und *Actions* sowie deren *Properties* unterstützt alle Befehlsarten und kann genutzt werden, um beliebige Produkte zu modellieren. Lediglich die Integration der verschiedenen Produkte in eine einzelne große Ontologie ist nicht erfolgt. Weder ist dies aus Gründen der Übersichtlichkeit sinnvoll, noch war dies durch technische Probleme mit dem Protégé-Editor möglich. Stattdessen

wurden die vorgestellten Produkte in separate Ontologien ausgegliedert, welche schrittweise über das Terminal durch den Nutzer ausgewählt werden. Protégé wurde ausschließlich auf MacOS verwendet und Unterschiede zur Performance auf Windows-Systemen können nicht ausgeschlossen werden.

Abschließend lässt sich festhalten, dass das Projekt in Hinblick auf die ursprünglichen Anforderungen erfolgreich abgeschlossen wurde. Insbesondere ist die volle Funktionalität des Servers gewährleistet, welcher mit einigen Plug-ins und sogar einem GUI ausgeliefert wird. Jedoch konnte eine Handvoll an Anforderungen nicht erfüllt werden, wozu jedoch keine kritischen Aspekte zählen.

11.2 Validierung

Eine andere Perspektive als die Verifizierung bietet die Frage, inwiefern das System in der Lage ist, den ursprünglichen Projektauftrag (vgl. Kapitel 3) zu erfüllen, und sich für den geplanten Verwendungszweck eignet.

Grundsätzlich muss festgehalten werden, dass der Server dazu fähig ist, den spezifizierten Cobot zu steuern. Dies wurde im Rahmen des Integrationstests sowohl durch einfache Plug-ins wie telepath_commands als auch durch die KI-Anwendung *telepath_blueprint*s demonstriert. Somit gilt der Auftrag als erfüllt. Dennoch sind einige Aspekte in Hinblick auf die Server-Architektur und die Einsatzmöglichkeiten des KI-Plug-ins zu beachten.

Der Server kann als zustandlos betrachtet werden, wofür insbesondere der Lademechanismus der Plug-ins sorgt. Die zugrundeliegenden Module werden zwar nur einmal beim Login des Clients geladen, jedoch wird bei jeder Anfrage eine neue Plug-in-Instanz erzeugt. Dadurch wird vermieden, dass der Server durch ein dauerhaft im Hintergrund existierendes Plug-in-Objekt Kontext über mehrere Anfragen hinweg speichert. Stattdessen wird eine Zustandsvariable in der DB gespeichert und für folgende Anfragen genutzt. Dieser Ansatz weist einen wesentlichen Nachteil auf, da die Instanziierung eines Plug-in-Objekts die Antwortzeiten des Servers signifikant erhöht. Zwar stellt Echtzeitfähigkeit weder eine Anforderung in diesem Projekt dar, noch ist diese in einer Laborumgebung unbedingt notwendig, sollte sich dieser Umstand jedoch ändern, könnte der derzeitige Lademechanismus für Plug-ins einen limitierenden Faktor darstellen. Dennoch ist bereits jetzt eine Vielzahl an Plug-ins denkbar, welche eher durch die einfache REST-API als durch die Server-Architektur limitiert werden.

Trotzdem ergibt sich daraus die Frage, ob der REST-Architekturstil die korrekte Wahl für das Gesamtsystem ist. Die korrekte Antwort hängt von den Plänen für dessen

Weiterentwicklung ab. Für die Steuerung eines einzelnen Cobots, wie dies aktuell der Fall ist, muss der Server nicht notwendigerweise zustandslos sein. Durch eine Auflockerung der architektonischen Einschränkungen könnten somit dessen Antwortzeiten reduziert werden. Würde die Anwendung stattdessen beispielsweise zu einer Open-Source Software weiterentwickelt werden, welche auch in einer industriellen Umgebung zur Steuerung mehrerer Roboter eingesetzt werden kann, würde der Aspekt der Skalierbarkeit an Bedeutung gewinnen. Die Notwendigkeit einer echtzeitfähigen Steuerung hängt hierbei vom konkreten Anwendungsfall ab. Abgesehen von der Zustandslosigkeit bieten die weiteren REST-Prinzipien jedoch große Vorteile für die Robotersteuerung. Insbesondere ermöglicht die einheitliche Schnittstelle eine Anbindung beliebiger Robotermodelle. Da ihre proprietäre Soft- und Hardware nämlich Einschränkungen durch die Hersteller unterliegt, wären ansonsten individuelle Lösungen notwendig. Deshalb wird empfohlen, den REST-Stil zunächst weiterzuverfolgen.

Zum jetzigen Zeitpunkt ist jeder Client außerdem auf ein Plug-in limitiert. Dieses kann durch eine erneute Anmeldung im Anschluss an einen EXIT-Befehl geändert werden, jedoch ist es nicht möglich verschiedene Plug-ins direkt am Server zu kombinieren. Diese Einschränkung ergibt sich durch das Design des Gesamtsystems, welches ausschließlich auf die Übertragung konkreter Befehle zur Steuerung des Clients ausgelegt ist. Stattdessen können unterstützende Plug-ins, welche etwa Objekte mittels Computer Vision identifizieren, jedoch keine Befehle generieren, über eine neue Schnittstelle an ein befehlsgenerierendes Plug-in angeschlossen werden. Dies ähnelt dem Architekturstil der Entwicklungsumgebung Eclipse [51].

Im Rahmen der Tests wurde die Flexibilität des KI-Plug-ins demonstriert. Dieses setzt Änderungen im Wissensmodel mit Hilfe des Suchalgorithmus direkt in eine korrekte Reihenfolge der Arbeitsschritte um. Ein statisches Roboterprogramm bietet keine solche Flexibilität. Zusätzlich können Änderungen des Programms außerhalb der Roboterzelle und ohne Kenntnisse einer proprietären Programmiersprache umgesetzt werden.

Dennoch handelt es sich im Moment um ein Konzept ohne konkreten Anwendungsfall. Um einen solchen zu finden, sollte zunächst der Aufwand bei der Erstellung eines Roboterprogramms zwischen üblichen Verfahren und dem ontologiebasierten Ansatz verglichen werden. Für den Integrationstest mussten gültige Koordinaten mit Hilfe des KUKA smartPADs ermittelt werden. Dieses Vorgehen ähnelt der Online-Programmierung mit Hilfe des Bediengeräts am Cobot. Für das Plug-in musste jedoch zusätzlich ein Produkt mit Hilfe der Basisontologie modelliert werden, wodurch der initiale Aufwand deutlich höher ist. Um das Modell *sketchHouse* später so anzupassen, dass das Dach immer vor dem Fenster und der Tür gezeichnet wird, waren jedoch

lediglich zwei neue *Object Properties* notwendig. Der Aufwand für die Anpassung eines traditionellen Roboterprogramms ist abhängig vom Programmierverfahren deutlich höher. Dies gilt insbesondere für die Online-Programmierung, welche einen produktiven Einsatz des Roboters verhindert. Zusätzlich fehlt diesen Verfahren die Flexibilität eines semantischen Modells mit Suchalgorithmus. Insgesamt erfordert das Plug-in also einen höheren initialen Aufwand, kann aber ab einer bestimmten Anzahl notwendiger Änderungen einen Vorteil gegenüber üblichen Programmierverfahren aufweisen (vgl. Abbildung 29). Der Fokus künftiger Forschung sollte also darauf liegen, den Aufwand zur Modellerstellung zu reduzieren, indem etwa Positionsdaten über Kamerabilder ermittelt werden und nicht mehr fester Bestandteil der Ontologie sind.

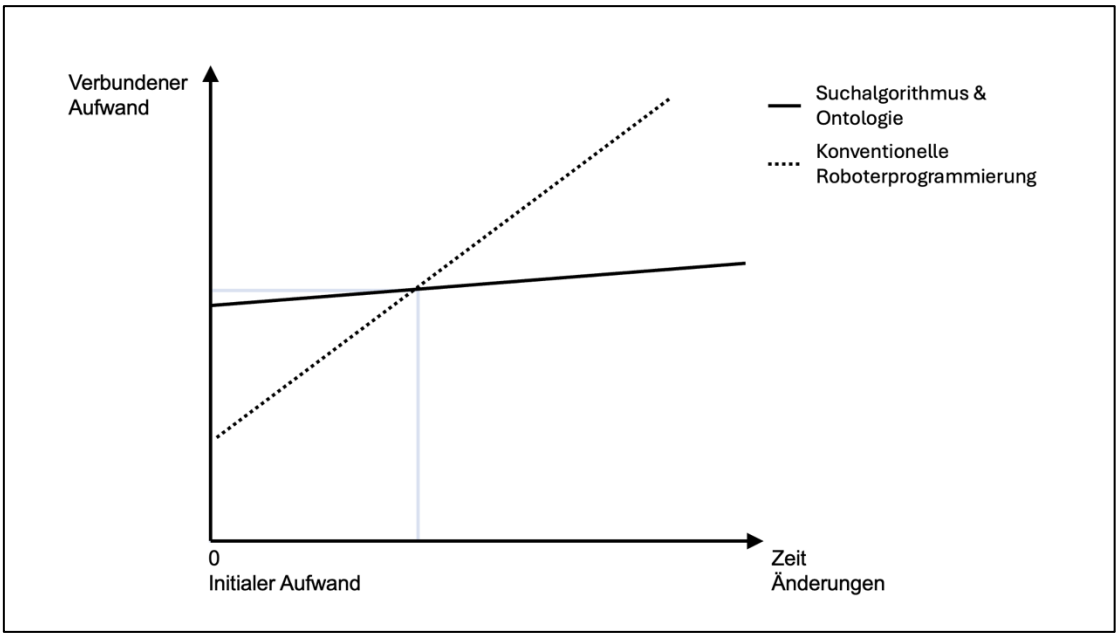

Abbildung 29 Skizzierter Aufwand bei Programmanpassungen für das KI-Plug-in und konventionelle Roboterprogrammierung

Folglich könnte MRK einen interessanten Anwendungsfall darstellen, da diese die Flexibilität der Software optimal ausnutzt. In einer Situation, in der ein Mensch und ein Cobot am selben Produkt arbeiten, könnte dieser dynamisch auf vom Menschen ausgeführte Tätigkeiten reagieren und stattdessen über einen Suchalgorithmus die nächste freie Aufgabe suchen. Zusätzlich könnte in der Ontologie modelliert werden, ob dem Cobot erlaubt ist, überhaupt mit gewissen Teilen zu interagieren. Eine Herausforderung in diesem Szenario bildet die Überwachung der menschlichen Arbeit, um fertiggestellte Tätigkeiten zu ermitteln und für den Cobot zu sperren. Eine Lösung könnte die Markierung der Teile durch Barcodes und das Einscannen mit Hilfe eines Handschuhscanners darstellen.

12 Fazit und Ausblick

Abschließend kann positiv auf den Projektverlauf und die entwickelte Software geblickt werden. Diese erfüllt die wesentlichen Anforderungen und bildet durch ihre Plug-in-Architektur eine Plattform für zukünftige Forschung und weitere Abschlussarbeiten an der THI. Während der Entwicklung konnte durch eine frühzeitige Analyse und Strukturierung der Anforderungen ein sinnvoller Rahmen für den Entwurf und die Implementierung der Software gesetzt werden, wodurch ein reibungsloser Projektablauf ermöglicht wurde. Da die wesentlichen fachlichen Aspekte zur Bewertung der Ergebnisse bereits im vorangegangenen Kapitel diskutiert wurden, soll an dieser Stelle nur die Qualität des Codes genannt werden. Dieser ist zwar ausführlich kommentiert, könnte jedoch schlanker gestaltet werden. Dieses sogenannte Refactoring ist jedoch auch in Zukunft noch möglich und ein Ausdruck der vergleichsweisen geringen Programmiererfahrung zu Beginn des Projekts.

An dieser Stelle sollen nun einige Vorschläge für die Weiterentwicklung der Client-Server-Architektur vorgestellt werden. Zunächst sollte die API erweitert werden, um neue Anwendungsfälle abzudecken. Diese ist im Moment ausschließlich auf die Steuerung eines Cobots beschränkt, jedoch ist es sinnvoll, auch zugehörige Sensoren wie Kameras zentral auf dem Server zu integrieren. Ebenfalls stellt die synchronisierte Steuerung mehrere Roboter einen vielversprechenden Ansatz dar. Hierbei kann die Server-Architektur eine Herausforderung darstellen, doch das Plug-in *telepath_sync* zeigt grundlegend auf, wie die Abstimmung verschiedener Clients mittels MongoDB als Schnittstelle für den Datenaustausch erfolgen kann.

Ebenfalls sollte der Client in Zukunft mehr Informationen speichern und verwalten als nur das Token. Dazu zählen beispielsweise der Name des genutzten Plug-ins und dessen Zustand. Dadurch kann das Prinzip der Zustandslosigkeit noch besser umgesetzt werden. Darüber hinaus kann es auch sinnvoll sein, Informationen über das Robotermodell oder dessen aktuelle Position an den Server zu übermitteln. Außerdem sollten Greifer direkt vom Client und folglich auch über die API vom Server angesteuert werden können. Hierfür sind zwei neue Befehlsarten für das Schließen und Öffnen denkbar.

Es existieren also zahlreiche Möglichkeiten für sinnvolle Anpassungen des Gesamtsystems. Diese ergeben sich teilweise aus dem bisherigen Entwicklungsansatz, in welchem die API inklusive clientseitiger Implementierung und der Server in zwei separaten Schritten ohne gemeinsames Leitbild entwickelt wurden. Daher ist zunächst eine ganzheitliche Analyse von Anforderungen für das Gesamtsystem zu empfehlen, welche insbesondere konkrete Anwendungsfälle identifiziert. Auf dieser Grundlage kann

anschließend die API überarbeitet werden, wobei folglich sowohl der Client als auch der Server angepasst werden müssten. Ein solches Vorgehen würde den Wert und die Wiederverwendbarkeit der Anwendung für zukünftige Forschungsprojekte sicherstellen.

Abbildungsverzeichnis

Abbildung 1 Mitarbeiter im Ford Werk Köln mit einem KUKA LBR iiwa Cobot 2
Abbildung 2 Bewegungskette eines Roboters mit eingezeichneten Koordinatensystemen 5
Abbildung 3 Bewegungsarten bei KUKA-Robotern 6
Abbildung 4 Schematische Darstellung der Plug-in-Architektur 8
Abbildung 5 Prinzipien des REST-Architekturstils 9
Abbildung 6 Aufbau von HTTP-Nachrichten 11
Abbildung 7 Ausgewählte KI-Technologien und Forschungsfelder nach Grundtyp und Anwendungsgebiet 13
Abbildung 8 Beispielhafter Knowledge Graph zur Darstellung von Familienverhältnissen 15
Abbildung 9 Überblick der OWL Functional Style Syntax 16
Abbildung 10 Entwicklungsprozess des Gesamtsystems 19
Abbildung 11 Entwicklungsprozess der Ontologie 20
Abbildung 12 Modell der Drei-Schichten-Architektur 30
Abbildung 13 Architekturdiagramm mit Klassen und Kardinalitäten 31
Abbildung 14 Projekt Tech Stack 33
Abbildung 15 Klassendiagramm der Serverapplikation 35
Abbildung 16 Server-Widget auf MacOS 38
Abbildung 17 Pseudocode der Client-Anwendung 43
Abbildung 18 Sequenzdiagramm der Methode login_response 44
Abbildung 19 Sequenzdiagramm der Methode command_response 46
Abbildung 20 Sequenzdiagramm der Methode command_confirmation 47
Abbildung 21 Übersicht wichtiger Konzepte in der Ontologie product_model 52
Abbildung 22 Klassendiagramm des Plug-ins telepath_blueprints 53
Abbildung 23 Ablaufdiagramm der Methode run(…) des Plug-ins telepath_blueprints 54
Abbildung 24 Ablaufdiagramm des Suchalgorithmus für Einzelteile im Plug-in telepath_blueprints 55
Abbildung 25 Die Produkte sketchHouse und stickman 57
Abbildung 26 Durch den Robotergreifer fixierte Filzstift-Halterung 58
Abbildung 27 Versuchsaufbau im Labor 59
Abbildung 28 Durch den Roboter erstellte Skizzen von stickman und sketchHouse ... 60
Abbildung 29 Skizzierter Aufwand bei Programmanpassungen für das KI-Plug-in und konventionelle Roboterprogrammierung 65

Tabellenverzeichnis

Tabelle 1 Übersicht der HTTP-Verben 10
Tabelle 2 Übersicht der HTTP-Statuscodes 12
Tabelle 3 Übersicht der Client-Responses 22
Tabelle 4 Übersicht der Server-Responses 23
Tabelle 5 Übersicht der Roboterbefehle 24
Tabelle 6 Anforderungen an die Serverapplikation 25
Tabelle 7 Anforderungen an die KI-Applikation 26
Tabelle 8 Anforderungen an die Systemarchitektur 26
Tabelle 9 Anforderungen an Systemtests 27
Tabelle 10 Anforderungen an den Release 27
Tabelle 11 Rahmenbedingungen 28
Tabelle 12 Übersicht der mitgelieferten Plug-ins 42
Tabelle 13 Anforderungen an das KI-Plug-in 50
Tabelle 14 Übersicht der Versuche zu sketchHouse mit verschiedenen Seeds 60

Literaturverzeichnis

[1] H. Maier, *Grundlagen der Robotik,* 3. Aufl. Berlin: VDE Verlag GmbH, 2022.

[2] K. Dinwiddie, *Industrial Robotics,* 1. Auflage., 2018.

[3] Ford Motor Company, *Robbie The Cobot Helps Disabled People And Those With Reduced Mobility Tackle Ford Assembly Work*, 2022. [Online]. Verfügbar unter: https://media.ford.com/content/fordmedia/feu/en/news/2022/05/25/robbie-the-cobot-helps-disabled-people-and-those-with-reduced-mo.html (Zugriff am: 16. Juli 2024).

[4] International Federation of Robotics, *Global Robotics Race: Korea, Singapore and Germany in the Lead*, 2024. [Online]. Verfügbar unter: https://ifr.org/ifr-press-releases/news/global-robotics-race-korea-singapore-and-germany-in-the-lead (Zugriff am: 24. Juli 2024).

[5] M. Bill, C. Müller, W. Kraus, und S. Bieller, „Presentation IFR Press Conference," 2023. Zugriff am: 24. Juli 2024. [Online]. Verfügbar unter: https://ifr.org/img/worldrobotics/2023_WR_extended_version.pdf

[6] KUKA AG, *Produkseite LBR iiwa.* [Online]. Verfügbar unter: https://www.kuka.com/en-de/products/robot-systems/industrial-robots/lbr-iiwa (Zugriff am: 24. Juli 2024).

[7] Wikipedia, *Denavit-Hartenberg-Transformation.* [Online]. Verfügbar unter: https://de.wikipedia.org/wiki/Denavit-Hartenberg-Transformation (Zugriff am: 24. Juli 2024).

[8] KUKA Deutschland GmbH, „KUKA Sunrise.OS 1.17: Bedien- und Programmieranleitung für Systemintegratoren," Augsburg, Juli 2020.

[9] KUKA Deutschland GmbH, *LBR iiwa 7 R800 Datenblatt*, 2022. [Online]. Verfügbar unter: https://www.kuka.com/-/media/kuka-downloads/imported/8350ff3ca11642998dbdc81dcc2ed44c/0000246832_en.pdf?rev=17b8a807a70745d4a16627fae9886121&hash=FBD6DB831EED428F307E6C5A6B79FA08 (Zugriff am: 25. Juli 2024).

[10] R. Müller, J. Franke, D. Henrich, B. Kuhlenkötter, A. Raatz, and A. Verl, Hg., *Handbuch Mensch-Roboter-Kollaboration,* 2. Aufl. München: Hanser, 2024.

[11] *IEEE1471-2000: Recommended Practice for Architectural Description of Software-Intensive Systems,* IEEE Computer Society, New York, Okt. 2000. [Online]. Verfügbar unter: https://standards.ieee.org/ieee/1471/2187/

[12] J. Goll, M. Koller, und M. Watzko, *Architektur- und Entwurfsmuster der Softwaretechnik: Mit lauffähigen Beispielen in Java,* 3. Aufl. Wiesbaden, Heidelberg: Springer Vieweg, 2023. [Online]. Verfügbar unter: https://link.springer.com/978-3-658-42383-4

[13] M. Richards, *Software architecture patterns: Understanding common architecture patterns and when to use them*. Sebastopol, CA: O'Reilly Media, 2015. [Online]. Verfügbar unter: https://permalink.obvsg.at/

[14] IBM, *What is an API?* [Online]. Verfügbar unter: https://www.ibm.com/topics/api (Zugriff am: 19. Juli 2024).

[15] R. T. Fielding, „Architectural Styles and the Design of Network-based Software Architectures," Dissertation, University of California, Irvine, 2000. Zugriff am: 19. Juli 2024. [Online]. Verfügbar unter: https://ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf

[16] Amazon Web Services, Inc., *Was ist eine API (Anwendungsprogrammierschnittstelle)?* [Online]. Verfügbar unter: https://aws.amazon.com/de/what-is/api/#:~:text=API%20stands%20for%20Application%20Programming,other%20using%20requests%20and%20responses. (Zugriff am: 19. Juli 2024).

[17] IBM, *Was ist eine REST-API?* [Online]. Verfügbar unter: https://www.ibm.com/de-de/topics/rest-apis (Zugriff am: 19. Juli 2024).

[18] Postman, Inc., *What Is a REST API? Examples, Uses, and Challenges.* [Online]. Verfügbar unter: https://blog.postman.com/rest-api-examples/ (Zugriff am: 22. Juli 2024).

[19] S. Tilkov, M. Eigenbrodt, S. Schreier, und O. Wolf, *REST und HTTP: Entwicklung und Integration nach dem Architekturstil des Web,* 3. Aufl. Heidelberg: dpunkt.Verlag, 2015. [Online]. Verfügbar unter: http://gbv.eblib.com/patron/FullRecord.aspx?p=2046366

[20] *RFC 9110: HTTP Semantics,* Internet Engineering Task Force, Fremont, 2022. [Online]. Verfügbar unter: https://datatracker.ietf.org/doc/html/rfc9110

[21] M. Topp, *OPC UA + MQTT = A Popular Combination for IoT Expansion.* [Online]. Verfügbar unter: https://opcconnect.opcfoundation.org/2019/09/opc-ua-mqtt-a-popular-combination-for-iot-expansion/ (Zugriff am: 18. September 2024).

[22] A. A. Ziegltrum, „Konzeption und Entwicklung einer REST API zur Server-basierten Steuerung eines kollaborierenden Roboters am Beispiel des KUKA LBR iiwa," Technische Hochschule Ingolstadt, Ingolstadt, 2023.

[23] *RFC 9112: HTTP/1.1,* Internet Engineering Task Force, Fremont, 2022. [Online]. Verfügbar unter: https://datatracker.ietf.org/doc/html/rfc9112

[24] *RFC 2046: Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types,* Internet Engineering Task Force, Fremont, 1996. [Online]. Verfügbar unter: https://datatracker.ietf.org/doc/html/rfc2046

[25] B. J. Copeland, *artificial intelligence.* [Online]. Verfügbar unter: https://www.britannica.com/technology/artificial-intelligence (Zugriff am: 25. Juli 2024).

[26] W. Zehentmeier und S. Schreiber, *KI-Hype – droht eine Aktienblase wie zu Beginn des Jahrtausends?,* 2024. [Online]. Verfügbar unter: https://www.br.de/nachrichten/netzwelt/ki-hype-droht-eine-aktienblase-wie-zu-beginn-des-jahrtausends,UIP656Y (Zugriff am: 25. Juli 2024).

[27] G. Oppy und D. Dowe, „The Turing Test," in *The Stanford Encyclopedia of Philosophy.* Zugriff am: 25. Juli 2024. [Online]. Verfügbar unter: https://plato.stanford.edu/entries/turing-test/

[28] G. Press, *A Very Short History Of Artificial Intelligence (AI)*, 2016. [Online]. Verfügbar unter: https://www.forbes.com/sites/gilpress/2016/12/30/a-very-short-history-of-artificial-intelligence-ai/ (Zugriff am: 25. Juli 2024).

[29] L. Heim, M. Anderljung, E. Bluemke, und R. Trager, *Computing Power and the Governance of AI*, 2024. [Online]. Verfügbar unter: https://www.governance.ai/post/computing-power-and-the-governance-of-ai#:~:text=Computing%20power%20–%20compute%20for%20short,have%20recently%20gained%20global%20attention. (Zugriff am: 25. Juli 2024).

[30] BBC, *Google AI defeats human Go champion*, 2017. [Online]. Verfügbar unter: https://www.bbc.com/news/technology-40042581 (Zugriff am: 25. Juli 2024).

[31] J. Bock, „Vorlesungsskript Business Analytics und Künstliche Intelligenz," 2023.

[32] E. Gillain, „A holistic view of AI techniques, their limitations and complementarities," in *De Gruyter STEM, Demystifying artificial intelligence: Symbolic, data-driven, statistical and ethical AI*, E. Gillain, Hg., Berlin: De Gruyter, 2024, S. 19–62.

[33] E. Ilkou und M. Koutraki, „Symbolic Vs Sub-symbolic AI Methods: Friends or Enemies?," in *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*, Virtual Event Ireland, 2020. Zugriff am: 26. Juli 2024. [Online]. Verfügbar unter: https://www.researchgate.net/publication/345243725_Symbolic_Vs_Sub-symbolic_AI_Methods_Friends_or_Enemies

[34] Lancaster University, *What is Statistical AI?* [Online]. Verfügbar unter: https://www.lancaster.ac.uk/maths/research/statistics/statistical-ai/ (Zugriff am: 26. Juli 2024).

[35] T. R. Gruber, „Toward principles for the design of ontologies used for knowledge sharing?," *International Journal of Human-Computer Studies*, vol. 43, 5-6, S. 907–928, 1995, doi: 10.1006/ijhc.1995.1081.

[36] G. Hüttenegger, *Open Source Knowledge Management: Mit 9 Tabellen*. Berlin, Heidelberg: Springer, 2006.

[37] I. Linden, „Knowledge representation and engineering with ontologies," in *De Gruyter STEM, Demystifying artificial intelligence: Symbolic, data-driven, statistical and ethical AI*, E. Gillain, Ed., Berlin: De Gruyter, 2024, S. 149–179.

[38] T. Berners-Lee, J. Hendler, und O. Lassila, „The Semantic Web," *Sci Am*, vol. 284, no. 5, S. 34–43, 2001, doi: 10.1038/scientificamerican0501-34.

[39] *OWL 2 Web Ontology Language: Structural Specification and Functional-Style Syntax (Second Edition),* World Wide Web Consortium, Wakefield, Dez. 2012. [Online]. Verfügbar unter: https://www.w3.org/TR/2012/REC-owl2-syntax-20121211/

[40] *OWL 2 Web Ontology Language OWL 2 Web Ontology Language Document Overview (Second Edition),* World Wide Web Consortium, Wakefield, Dez. 2012. [Online]. Verfügbar unter: https://www.w3.org/TR/2012/REC-owl2-overview-20121211/

[41] R. Miao, Q. Jia, F. Sun, G. Chen, H. Huang, und S. Miao, „Semantic Representation of Robot Manipulation with Knowledge Graph," *Entropy (Basel, Switzerland)*, Jg. 25, Nr. 4, 2023, doi: 10.3390/e25040657.

[42] R. Bernardo, J. M. Sousa, und P. J. Gonçalves, „A novel framework to improve motion planning of robotic systems through semantic knowledge-based reasoning," *Computers & Industrial Engineering*, Jg. 182, 2023, doi: 10.1016/j.cie.2023.109345.

[43] B. Baginski und G. Kess, „A client-server based system for programming and control of cooperating robots," in *Proc. of 5th International Workshop on Robotics in the Alpe-Adria-Danube Region RAAD'96*, S. 51–56. Zugriff am: 18. September 2024. [Online]. Verfügbar unter: https://mediatum.ub.tum.de/node?id=1292483

[44] T. Sievers, M. Bender, und R. Möller, „Connecting AI Technologies as Online Services to a Humanoid Service Robot," in *2023 15th International Conference on Computer and Automation Engineering (ICCAE)*, Sydney, Australia, 2023, S. 431–435.

[45] Nippon Telegraph and Telephone K.K., *Demonstration of remote robot control via server enhanced by softwarization of industrial Ethernet functions: Free choice of robots at factories and lower initial factory investment*, 2024. [Online]. Verfügbar unter: https://group.ntt/en/newsrelease/2024/05/16/240516a.html (Zugriff am: 18. September 2024).

[46] N. F. Noy und D. L. McGuinness, „Ontology Development 101: A Guide to Creating Your First Ontology," 2001. [Online]. Verfügbar unter: https://protege.stanford.edu/publications/ontology_development/ontology101-noy-mcguinness.html

[47] *Qualitätsmanagementsysteme – Grundlagen und Begriffe (ISO 9000:2015)*, DIN EN ISO 9000, DIN Deutsches Institut für Normung e. V., Berlin, Nov. 2015. [Online]. Verfügbar unter: https://www-nautos-de.thi.idm.oclc.org/VFZ/search/item-detail/DE30062915

[48] PortSwigger Ltd., *Session token in URL.* [Online]. Verfügbar unter: https://portswigger.net/kb/issues/00500700_session-token-in-url (Zugriff am: 31. Juli 2024).

[49] Docker Inc., *What is a Container?: Use containers to Build, Share and Run your applications.* [Online]. Verfügbar unter: https://www.docker.com/resources/what-container/ (Zugriff am: 1. August 2024).

[50] Pallets Projects, *Flask Documentation: Deploying to Production.* [Online]. Verfügbar unter: https://flask.palletsprojects.com/en/2.3.x/deploying/ (Zugriff am: 1. August 2024).

[51] Eclipse Foundation, *Eclipse documentation: Platform architecture.* [Online]. Verfügbar unter: https://help.eclipse.org/latest/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Fguide%2Farch.htm (Zugriff am: 18. September 2024).

[52] HubSpot, Inc., *Tech Stack: Definition, Aufbau und Auswahlkriterien*, 2023. [Online]. Verfügbar unter: https://blog.hubspot.de/website/tech-stack (Zugriff am: 7. August 2024).

[53] L. P. Ramos, *Use PyQt‘s QThread to Prevent Freezing GUIs.* [Online]. Verfügbar unter: https://realpython.com/python-pyqt-qthread/#author (Zugriff am: 8. September 2024).

[54] Pallets Projects, *Flask Documentation: Blueprints and Views.* [Online]. Verfügbar unter: https://flask.palletsprojects.com/en/2.3.x/tutorial/views/ (Zugriff am: 25. September 2024).

[55] *RFC 3986: Uniform Resource Identifier (URI): Generic Syntax,* Internet Engineering Task Force, Fremont, 2005. [Online]. Verfügbar unter: https://datatracker.ietf.org/doc/html/rfc3986

[56] *IEEE 1872-2015: Standard Ontologies for Robotics and Automation,* IEEE Robotics and Automation Society, New York, Apr. 2015. [Online]. Verfügbar unter: http://ieeexplore.ieee.org/servlet/opac?punumber=7084071

Quellcode

GitHub Repository	https://github.com/EngDePal/REST_AI_Server

Skripts	
main.py	XVI
Serverapplikation	
server.py	XVII
token_manager.py	XXV
data_manager.py	XXVII
robot_logic_manager.py	XXXIII
widget.py	XXXV
Plug-In-Architektur	
plugin_interface.py	L
commands.py	LII
mongo_interface.py	LV
Plug-Ins	
telepath_commands.py	LVII
telepath_sync.py	LIX
telepath_skills.py	LXVIII
telepath_blueprints.py	LXX

main.py

```
"""Main script to execute the application"""
#Importing modules
from packages.widget import CockpitWidget
from PyQt5.QtWidgets import QApplication
#Fixing imports
import sys

#Main function
def main():
    #Starting widget, which enables the user to start the server
    widget_app = QApplication(sys.argv)
    widget = CockpitWidget()
    widget.show()
    sys.exit(widget_app.exec_())

#Run app
if __name__ == "__main__":
    main()
```

server.py

```
1  """Handles the logic of the core application, especially HTTP requests and
       access to different manager classes."""
2  #Importing necessary modules and classes from the Flask web framework
3  from flask import Flask, jsonify, request, make_response
4  #Importing the various managers
5  from packages.token_manager import TokenManager
6  from packages.data_manager import DataManager
7  from packages.robot_logic_manager import RobotLogicManager
8  #Importing signal to handle events
9  import signal
10 #Importing command classes
11 from packages.plugins.utils.commands import *
12 #For Widget functionality
13 from PyQt5.QtCore import QThread, pyqtSignal
14 #Shutdown
15 import os
16 #Token extraction from client input
17 import json
18
19 #Server class
20 class Server(QThread):
21
22     #Creating signals for the widget
23
24     #For updating the client counter with the current count
25     counter_signal = pyqtSignal(int)
26
27     #For manipulating the client table
28     #Dict: Data like token, plugin, command and parameters
29     #Bool1: Adding(0) or removing data (1)
30     #Bool2: Creating a new line (0) or updating a row (1)
31     table_signal = pyqtSignal(dict, bool, bool)
32
33     #For creating messages about clients
34     #Str1: Token
35     #Str2: Mode (LOGIN, LOGOUT)
36     user_info_signal =pyqtSignal(str, str)
37
38     #For creating a login notification
39     login_signal = pyqtSignal()
40
41     #For server control
42     start_signal = pyqtSignal()
43
44     #Sending the URL to the widget
45     url_signal = pyqtSignal(str)
46
47     def __init__(self):
48
49         #For threading
50         super().__init__()
51
```

```
52          #Updating widget count
53          self.client_count = 0
54          self.counter_signal.emit(self.client_count)
55
56          #Create a Flask app
57          self.app = Flask(__name__)
58
59          #Creating instances of the managers
60          self.tm = TokenManager()
61          self.dm = DataManager()
62          self.rlm = RobotLogicManager()
63
64          #Register the shutdown function for SIGINT (Ctrl + C)
65          #Allows manual shutdown in the terminal
66          signal.signal(signal.SIGINT, self.manual_shutdown)
67
68          #The following sections defines the handling of incoming http requests
69          #Token_json is a json formatted string including the token, which
               serves as an identifier
70          #For more details on the individual commands please check GitHub or
               the accompanying thesis
71
72          #REST-API: POST /login 200 {}
73          #Login response: Generated token is sent to client
74          @self.app.route("/login", methods=["POST"])
75          def login_response():
76
77              #Login notification in GUI
78              self.login_signal.emit()
79
80              #Start of refined plugin loading proces
81              self.available_plugins = dict(self.rlm.get_discovery())
82              #Selecting plugin through user input in the terminal
83              path = self.select_plugin(self.available_plugins)
84
85              #Dynamically load and register the plugin
86              plugin_name = self.rlm.load_module(path)
87
88              #Generating and preparing the token for messaging
89              generated_token = self.tm.generate_token()
90              data = {
91                  "token" : generated_token
92              }
93
94              #Saving the combination of token and plugin name -> REST
                  statelessness
95              self.dm.save_plugin(generated_token, plugin_name)
96
97              #Getting starting state of the application
98              instance = self.rlm.create_instance(plugin_name)
99              app_state = instance.setup()
100             #Saving starting state of the application
101             self.dm.save_state(generated_token, app_state)
102
```

```
103             #Saving an SEND command as the starting command of the client
104             base_command = CommandSEND()
105             self.dm.save_command(generated_token, base_command)
106 
107             #UI updates
108 
109             #Update client count
110             self.client_count += 1
111             self.counter_signal.emit(self.client_count)
112 
113             #Updating client table
114             table_info = dict(data)
115             #Plugin name without telepath_ prefix and .py suffix
116             plugin_without_prefix = plugin_name[9:-3]
117             table_info["plugin_name"] = plugin_without_prefix
118             self.table_signal.emit(table_info, False, False)
119 
120             #Sending status update to GUI
121             self.user_info_signal.emit(generated_token, "LOGIN")
122 
123             #Sending the token to the client
124             response = make_response(jsonify(data))
125             response.headers['Cache-Control'] = 'no-cache'
126 
127             print("Login successful.")
128             return response, 200
129 
130 
131         #REST-API: GET /newcommand/<token> 200 {}
132         #Response to command request: parameters object must be sent back
133         @self.app.route("/newcommand/<token_json>", methods=["GET"])
134         def command_response(token_json: str):
135 
136             #Extract token from JSON
137             token = self.extract_token(token_json)
138 
139             #Check token authenticity
140             if self.tm.check_token_authenticity(token):
141 
142                 #Get generated commmand from MongoDB
143                 command = self.dm.retrieve_command(token)
144 
145                 #Update widget
146                 table_info = dict(command)
147                 table_info["token"] = token
148                 self.table_signal.emit(table_info, False, True)
149 
150                 #Response construction
151 
152                 #For larger data packages make_response is recommended
153                 #Since just returning jsonify(data package) might lead to data
                      loss during transmission
154                 response = make_response(jsonify(command))
155                 response.headers['Cache-Control'] = 'no-cache'
```

```
156
157                 print("Command sent.")
158                 return response, 200
159
160             else:
161                 print("Error: No valid token provided for command reception.")
162                 #Response creation
163                 response = make_response("")
164
165                 return response, 401 #Unauthorized
166
167         #REST-API: POST /newcommand/<token> 200 {}
168         #Response to command confirmation: no return object
169         #Generates a command and stores it in MongoDB
170         @self.app.route("/newcommand/<token_json>", methods=["POST"])
171         def command_confirmation(token_json: str):
172
173             #Extract token from JSON
174             token = self.extract_token(token_json)
175
176             #Check token authenticity
177             if self.tm.check_token_authenticity(token):
178
179                 #Gets the app state, plugin_name and last command from MongoDB
180                 #State is a dict
181                 state = self.dm.retrieve_state(token)
182                 plugin_name = self.dm.retrieve_plugin(token)
183                 last_cmd = self.dm.retrieve_command(token)
184
185                 #Checks, whether this was an EXIT command
186                 if last_cmd["command"] == "EXIT":
187                     #If it is an EXIT command, the logout sequence is
                        activated
188                     #No new command is generated
189                     self.logout(token, plugin_name)
190                     print("EXIT command confirmed. Client logged out.")
191
192                 #Otherwise a new command is generated by the system
193                 else:
194                     #Grab ID from MongoDB
195                     plugin_name = self.dm.retrieve_plugin(token)
196                     #Grab plugin instance
197                     instance = self.rlm.create_instance(plugin_name)
198
199                     #Run the plugin and return the output -> command object,
                            state dict
200                     #Pass application state as argument
201                     command, app_state = instance.run(state)
202
203                     #Save the app state and command
204                     #Should be function in DM
205                     self.dm.save_state(token, app_state)
206                     self.dm.save_command(token, command)
207
```

```
208                     print("Command confirmed. Next command generated.")
209
210                 #Response creation
211                 response = make_response("")
212
213                 return response, 200
214
215             else:
216                 print("Error: No valid token provided for command
                       confirmation.")
217                 #Response creation
218                 response = make_response("")
219
220                 return response, 401 #Unauthorized
221
222         #REST-API: PUT /safeinfo/<token> 200 {"msg" : String}
223         #Response to info file from client: file is saved in database
224         @self.app.route("/saveinfo/<token_json>", methods=["PUT"])
225         def saveinfo_response(token_json: str):
226
227             #Extract token from JSON
228             token = self.extract_token(token_json)
229
230             #Check token authenticity and media type
231             if self.tm.check_token_authenticity(token) and
                     request.content_type == "application/json":
232                 data = request.get_json()
233                 data["token"] = token
234                 self.dm.save_data("infos", data)
235
236                 print("Info saved.")
237
238                 #Response creation
239                 response = make_response("")
240
241                 return response, 200
242
243             else:
244                 print("Error: Check token validity and ensure content type
                       JSON.")
245                 #Response creation
246                 response = make_response("")
247
248                 return response, 401 #Unauthorized
249
250         #REST-API: POST /safelog/<token> 200 {"filename" : String, "data" :
                 String}
251         #Response to log file from client: file is saved in database
252         @self.app.route("/savelog/<token_json>", methods=["POST"])
253         def savelog_response(token_json: str):
254
255             #Extract token from JSON
256             token = self.extract_token(token_json)
257
```

```
258             #Check token authenticity and media type
259             if self.tm.check_token_authenticity(token) and
                     request.content_type == "application/json":
260                 data = request.get_json()
261                 data["token"] = token
262                 self.dm.save_data("logs", data)
263
264                 print("Log file saved.")
265
266                 #Response creation
267                 response = make_response("")
268
269                 return response, 200
270
271             else:
272                 print("Error: Check token validity and ensure content type
                      JSON.")
273                 #Response creation
274                 response = make_response("")
275
276                 return response, 401 #Unauthorized
277
278         #Not defined in the REST-API
279         #For server shutdown through GUI
280         @self.app.route("/shutdown", methods = ["POST"])
281         def shutdown():
282
283             self.shutdown_server()
284
285             response = make_response(jsonify({"succes": True}))
286
287             return response, 200
288
289     #Methods for server handling
290
291     #Starts the server and returns the base url
292     def run(self):
293
294         #GUI signal to notify user about start
295         self.start_signal.emit()
296
297         #Start the Flask server
298         #According to the client side API implementation
299
300         #The robot will attempt a connection to serverurl =
                "http://172.31.1.100:3000/"
301         my_host = "172.31.1.100"
302         my_port = 3000
303
304         #For development outside of the lab run:
305         # my_host = "127.0.0.1"
306         # my_port = 3000
307
308         #Return the base url to automatically setup the widget
```

```
309         base_url = "http://" + my_host + ":" + str(my_port) + "/"
310         self.url_signal.emit(base_url)
311 
312         #Start the server
313         self.app.run(host= my_host, port= my_port)
314 
315     #Shutting down server
316     def shutdown_server(self):
317 
318         #Stopping MongoDB
319         self.dm.stop_mongodb()
320 
321         #Kill python process
322         #This will end all threads
323         os.kill(os.getpid(), signal.SIGINT)
324 
325     #Defines the behaviour during server shutdown in the terminal
326     def manual_shutdown(self, sig, frame):
327         print("Shutting down server gracefully...")
328         #Stopping MongoDB
329         self.dm.stop_mongodb()
330         print("Server shut down")
331         exit(0)
332 
333     #Handling client logout
334     def logout(self, token: str, plugin_name: str):
335 
336         #Reduce client count
337         self.client_count -= 1
338         #Update widget
339         self.counter_signal.emit(self.client_count)
340 
341         #Removing token from TokenManager
342         self.tm.delete_token(token)
343 
344         #Removing relevant data from MongoDB: no errors, if an identical token
               is generated
345         self.dm.delete_data(token)
346 
347         #Update widget
348         table_info = {"token": token}
349         self.table_signal.emit(table_info, True, False)
350         self.user_info_signal.emit(token, "LOGOUT")
351 
352     #The token is send by the client in JSON format:
353     #e.g. {"token": "ABCD1234}
354     #This method helps to extract the token
355     def extract_token(self, json_input: dict):
356 
357         data = json.loads(json_input)
358         token = data["token"]
359 
360         return token
361 
```

```
362     #Allows plugin choice in the terminal
363     def select_plugin(self, plugin_selection: dict):
364         print("New client attempts to login!")
365         print("Available plugins for robot logic:")
366 
367         #Print a list of plugins
368         for element in plugin_selection.keys():
369             print("\t"+str(element))
370 
371         user_input = input("Please select a plugin by name: ")
372 
373         allowed_input = False
374         while allowed_input == False:
375             try:
376                 path = self.available_plugins[user_input]
377             except KeyError:
378                 user_input = input("Plugin not found. Please try again: ")
379             else:
380                 allowed_input = True
381                 print("Plugin selection succesful.")
382 
383         return path
```

token_manager.py

```python
"""TokenManager class is responsible for all operations involving
        authentication tokens."""
#Importing modules
import random

class TokenManager:

    def __init__(self):
        #There are additional eligible characters, but these should be enough
        #Allows for a total of 62^8 combinations
        self.eligible_characters =
            "0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"
        self.blocked_characters = """;/?:@=&$-_.+!*'(),"<>#%{ }|\^~[]`"""
        self.token_length = 8
        self.token_list = []

    #Following methods operate on tokens

    #Randomized creation of a token, saving it in the list and returning it
    def generate_token(self):
        token = ""
        for i in range(self.token_length):
            random_number = random.randint(0, len(self.eligible_characters)
                   - 1)
            token = "".join([token, self.eligible_characters[random_number]])

        #Checking conditions and generating a new token, if they are not met
        if self.check_token_authenticity(token) == False:
            if self.check_token_conformity(token) == True:
                self.token_list.append(token)
            else:
                token = self.generate_token()

        return token


    #Takes in a token as a string and checks for blocked characters, returning
        True for correct tokens
    def check_token_conformity(self, token: str):
        counter = 0
        if len(token) == self.token_length:
            for char in token:
                if char in self.blocked_characters:
                    counter += 1
            if counter == 0:
                return True
            else:
                return False
        else:
            return False


```

```
49      #Takes in a token as a string and compares it to existing tokens,
           verifying the input, returning True for an existing token
50      def check_token_authenticity(self, token: str):
51          if token in  self.token_list:
52              return True
53          else:
54              return False
55
56      #Remove a token during log out
57      def delete_token(self, token: str):
58          self.token_list.remove(token)
```

data_manager.py

```
1  """Handles operations involving the MongoDB database."""
2  #Pymongo is a Python driver for MongoDB
3  import pymongo
4  #Additional imports allow to start mongodb
5  import subprocess
6  import os
7  #For OS check
8  import platform
9  #Cross platform comptibility
10 import psutil
11
12 class DataManager:
13
14     #Constructor starts and set-ups database
15     def __init__(self):
16         self.start_mongodb()
17         self.setup()
18
19     #Initializes the database and collections with basic dictionaries
20     def setup(self):
21
22         self.client = pymongo.MongoClient("mongodb://localhost:27017")
23
24         #Loads or creates database if necessary
25         self.db = self.client["REST_AI_Server"]
26
27         self.collections_list = ["app_state", "logs", "infos", "plugins",
               "commands"]
28
29         #Remove temporary data first, in case there was no proper shutdown
30         for collection in ["app_state", "plugins", "commands"]:
31             target_collection = self.db[collection]
32             target_collection.delete_many({})
33
34         #Inserts collections with base entry, if necessary
35         #View these as examples or formats
36         #Except for app state, which depends on the plugin
37         for collection in self.collections_list:
38             if collection not in self.db.list_collection_names():
39                 new_col = self.db[collection]
40                 #State restores app state of plugins -> RESTful
41                 if collection == "app_state":
42                     #This is only an example of the robot coordinates as the
                           state
43                     #Properties of the state file are determined by the plugin
44                     db_file = {"token" : "",
45                             "command" : "",
46                             "parameters": {
47                                 "type": "",
48                                 "frame": {
49                                     "X": 0,
50                                     "Y": 0,
```

```
51                                      "Z": 0,
52                                      "A": 0,
53                                      "B": 0,
54                                      "Z": 0}
55                              }
56                          }
57                  #Log files of client
58                  elif collection == "logs":
59                      db_file = {"token" : "", "filename": "", "data": ""}
60                  #Info files of client
61                  elif collection == "infos":
62                      db_file = {"token" : "", "msg": ""}
63                  #Plugin selection of clients
64                  elif collection == "plugins":
65                      db_file = {"token": "", "plugin_name": ""}
66                  #Saving generated command
67                  elif collection == "commands":
68                       db_file = {"token" : "",
69                              "command" : "",
70                              "parameters": {
71                                  "type": "",
72                                  "frame": {
73                                      "X": 0,
74                                      "Y": 0,
75                                      "Z": 0,
76                                      "A": 0,
77                                      "B": 0,
78                                      "Z": 0}
79                              }
80                          }
81                  base_entry = new_col.insert_one(db_file)
82
83          print("Database is set-up!")
84
85      #Starts the MongoDB process accroding to OS expectation
86      def start_mongodb(self):
87
88          #Define paths
89          dir_path = os.path.dirname(os.path.realpath(__file__))
90
91          if platform.system() == "Darwin" or platform.system() == "Linux":
92              mongodb_bin_path = os.path.join(dir_path,
                    "../mongodb/bin/macos/mongod")
93          elif platform.system() == "Windows":
94              mongodb_bin_path = os.path.join(dir_path,
                    "../mongodb/bin/windows/mongod.exe")
95          else:
96              raise Exception("Unknown operating system. No MongoDB binaries
                    available.")
97          db_path = os.path.join(dir_path, "../mongodb/data")
98          log_path = os.path.join(dir_path, "../mongodb/logs/mongodb.log")
99
100         #Ensure paths exist
101         os.makedirs(db_path, exist_ok=True)
```

```
102         os.makedirs(os.path.dirname(log_path), exist_ok=True)
103
104         #Start MongoDB process
105         #Darwin is MacOS
106         if platform.system() == "Darwin" or platform.system() == "Linux":
107             process = subprocess.Popen(
108                 [mongodb_bin_path, "--dbpath", db_path, "--logpath", log_path,
                        "--fork"],
109                 stdout=subprocess.PIPE,
110                 stderr=subprocess.PIPE
111             )
112         #Windows takes longer to start MongoDB, leading to a freezing GUI
113         #Stop code execution and restart, which should allow the server to
                 work as intendede
114         elif platform.system() == "Windows":
115             process = subprocess.Popen(
116                 [mongodb_bin_path, "--dbpath", db_path, "--logpath",
                         log_path],
117                 stdout=subprocess.PIPE,
118                 stderr=subprocess.PIPE
119             )
120         else:
121             raise Exception("Unknown operating system. No MongoDB binaries
                     available.")
122
123         stdout, stderr = process.communicate()
124
125         if process.returncode == 0:
126             print("MongoDB started successfully.")
127         else:
128             print(f"Error starting MongoDB: {stderr.decode()}")
129
130     #Stops MongoDB
131     def stop_mongodb(self):
132
133         #Define paths
134         dir_path = os.path.dirname(os.path.realpath(__file__))
135         if platform.system() == "Darwin" or platform.system() == "Linux":
136             mongodb_bin_path = os.path.join(dir_path,
                     "../mongodb/bin/macos/mongod")
137         elif platform.system() == "Windows":
138             mongodb_bin_path = os.path.join(dir_path,
                     "../mongodb/bin/windows/mongod.exe")
139         else:
140             raise Exception("Unknown operating system. No MongoDB binaries
                     available.")
141
142         #Find MongoDB process ID and stop it
143         try:
144
145             if platform.system() == "Darwin" or platform.system() == "Linux":
146                 result = subprocess.run(["pgrep", "-f", mongodb_bin_path],
                            stdout=subprocess.PIPE)
147                 pid = int(result.stdout.strip())
```

```
148
149                 if pid:
150                     subprocess.run(['kill', str(pid)])
151                     print(f"MongoDB process {pid} stopped successfully.")
152                 else:
153                     print("MongoDB process not found.")
154
155             elif platform.system() == "Windows":
156                 #Using psutil on Windows to find and terminate the process
157                 for proc in psutil.process_iter(['pid', 'name']):
158                     if "mongod.exe" in proc.info['name']:
159                         proc.terminate()  #Send termination signal
160                         proc.wait()  #Wait until the process is terminated
161                         print(f"MongoDB process {proc.info['pid']} stopped
                                successfully.")
162                         break
163                 else:
164                     print("MongoDB process not found.")
165
166         except Exception as e:
167             print(f"Error stopping MongoDB: {str(e)}")
168
169     #Saves the dictionary in the specified collection
170     def save_data(self, collection: str, db_file: dict):
171         if collection in self.collections_list:
172             chosen_collection = self.db[collection]
173
174             #Regarding app_state: In order to avoid complex queries and
175             #Ensure the return of only the latest state, prior entries will be
                 deleted
176             if collection == "app_state":
177                 token = db_file["token"]
178                 doc = self.query_data("app_state", {"token": token})
179                 #Checks for previous entries, deletes them
180                 if len(doc) != 0:
181                     chosen_collection.delete_many({"token": token})
182
183             #Regarding commands: In order to avoid complex queries and
184             #Ensure the return of only the latest state, prior entries will be
                 deleted
185             if collection == "commands":
186                 token = db_file["token"]
187                 doc = self.query_data("commands", {"token": token})
188                 #Checks for previous entries, deletes them
189                 if len(doc) != 0:
190                     chosen_collection.delete_many({"token": token})
191
192             #Regarding infos: In order to keep the PUT methode idempotent,
                   prior entries will be deleted
193             if collection == "infos":
194                 token = db_file["token"]
195                 doc = self.query_data("infos", {"token": token})
196                 #Checks for previous entries, deletes them
197                 if len(doc) != 0:
```

```
198                     chosen_collection.delete_many({"token": token})
199
200             chosen_collection.insert_one(db_file)
201
202     #Save functions to shorten server code
203
204     def save_plugin(self, token: str, plugin_name:str):
205         db_file = {"token" : token, "plugin_name": plugin_name}
206         self.save_data("plugins", db_file)
207
208     def save_state(self, token: str, state: dict):
209         db_file = state
210         db_file["token"] = token
211         self.save_data("app_state", db_file)
212
213     def save_command(self, token: str, command:dict):
214         db_file = command
215         db_file["token"] = token
216         self.save_data("commands", db_file)
217
218     #Returns a list of the requested documents after taking in a query
            dictionary
219     #So far only used for database testing
220     #Empty dict will return the whole collection
221     def query_data(self, collection: str, query_dict: dict):
222         if collection in self.collections_list:
223             chosen_collection = self.db[collection]
224             #Returns a cursor object and turns it into a list of dictionaries
225             document = list(chosen_collection.find(query_dict))
226             return document
227
228     #Get plugin name from server
229     #Does not need a query dict as an argument
230     #Shortens server code
231     def retrieve_plugin(self, token: str):
232         #Get the fitting document from MongoDB
233         try:
234             query = {"token" : token}
235             result = self.query_data("plugins", query)
236             doc = result[0]
237             #Only returns the plugin name
238             id = doc["plugin_name"]
239             return id
240         except IndexError:
241             return 0
242
243     #Retrieves the app_state
244     def retrieve_state(self, token: str):
245         #Get the fitting document from MongoDB
246         try:
247             query = {"token" : token}
248             result = self.query_data("app_state", query)
249             doc = result[0]
250             doc.pop("_id")
```

```
251             return doc
252         except IndexError:
253             return {}
254 
255     #Retrieves recent command
256     def retrieve_command(self, token: str):
257         #Get the fitting document from MongoDB
258         try:
259             query = {"token" : token}
260             result = self.query_data("commands", query)
261             doc = result[0]
262             doc.pop("_id")
263             doc.pop("token")
264             return doc
265         except IndexError:
266             return {}
267 
268     #Deletes app state and plugins, if a client logs out
269     def delete_data(self, token: str):
270 
271         parameters = {"token" : token}
272 
273         for collection in ["app_state", "plugins", "commands"]:
274             target_collection = self.db[collection]
275             target_collection.delete_many(parameters)
276 
277     #Drops the database
278     #Allows for clean-up
279     def delete_db(self, db_name: str):
280         self.client.drop_database(db_name)
```

robot_logic_manager.py

```
1  """Plugin Manager for robot logic modules attaching to the core server
           application"""
2  #Importing necessary libraries
3  import importlib
4  import importlib.util
5  import os
6  import sys
7  from packages.plugins.utils.plugin_interface import PluginInterface
8  import inspect
9  import pkgutil
10 import types
11
12 #Plugin Manager handling robot logic
13 class RobotLogicManager:
14
15     #Initalizes the Plugin Manager
16     def __init__(self):
17
18         #Plugin directory
19         self.dir_path = os.path.dirname(os.path.realpath(__file__))
20         self.plugin_directory = os.path.join(self.dir_path, "plugins")
21
22         #Allows for plugin discovery
23         #Allows Python to search in the plugin directory
24         sys.path.append(self.plugin_directory)
25
26         #Allows Python to search in newly added plugins-subdirectories
27         for root, dirs, files in os.walk(self.plugin_directory):
28             if root not in sys.path:
29                 sys.path.append(root)
30
31         #Creates a dict of discovered plugins and their paths
32         self.dicscovered_plugins = self.dicscover_plugins()
33
34         #Mapping of loaded (cached) plugins by  names to the corresponding
                 module
35         self.plugin_mapping = dict()
36
37     #Source: https://gist.github.com/dorneanu/cce1cd6711969d581873a88e0257e312
38     #Allows for the dynamic loading of modules
39     def load_module(self, path: str):
40         #Gets filename from path
41         name = os.path.split(path)[-1]
42         #Creates a module spec
43         spec = importlib.util.spec_from_file_location(name, path)
44         #Creates a module from spec
45         module = importlib.util.module_from_spec(spec)
46         #Importing the Module
47         spec.loader.exec_module(module)
48
49         #Module management for further operations
50         self.register_plugin(name, module)
```

```
51
52          return name
53
54      #Creates and returns an instance of the main class of the plugin
55      def instantiate_plugin(self, module: types.ModuleType):
56          #There should be only one class inheriting from PluginInterface
57          for name, object in inspect.getmembers(module):
58          #Check if the object is a class and subclasses PluginInterface
59              if inspect.isclass(object) and issubclass(object, PluginInterface)
                    and object != PluginInterface:
60                  #Instantiate the class
61                  plugin_instance = object()
62
63          return plugin_instance
64
65      #Creates and returns a plugin instance
66      def create_instance(self, plugin_name: str):
67          module = self.get_plugin(plugin_name)
68          instance = self.instantiate_plugin(module)
69          return instance
70
71
72      #Discovers all plugins adhering to naming convention in the plugin
            directory
73      #Naming convention: telepath_plugin_name (Telepath is the software name)
74      def dicscover_plugins(self):
75          discovered_plugins = {
76              name: module_finder.find_spec(name).origin
77              for module_finder, name, ispkg in pkgutil.iter_modules()
78              if name.startswith('telepath_')
79          }
80
81          return discovered_plugins
82
83      #Getter Method for discovered plugins
84      def get_discovery(self):
85          return self.dicscovered_plugins
86
87      #Creates a mapping of plugin_names to loaded_modules
88      def register_plugin(self, plugin_name: str, module: types.ModuleType):
89
90          #Allows the server to cache modules, which are already in use
91          self.plugin_mapping[plugin_name] = module
92
93      #Returns the correct plugin module
94      #According to the provided name
95      def get_plugin(self, plugin_name: str):
96          module = self.plugin_mapping[plugin_name]
97          return module
```

widget.py

```
"""GUI for the REST AI Server"""
#Importing the PyQt5 GUI framework and other modules
from PyQt5.QtWidgets import QWidget, QLabel, QTableWidget, QTableWidgetItem,
         QPushButton, QTextEdit, QMessageBox
from PyQt5.QtGui import QFont
from PyQt5.QtCore import Qt
#Running server backend
from packages.server import Server
#For server shutdown
import requests
#For correct GUI layout
import platform

class CockpitWidget(QWidget):

    #Constructor:
    def __init__(self):

        #Setting up GUI
        super().__init__()

        #Switch between layouts depending on OS
        if platform.system() == "Darwin" or platform.system() == "Linux":
            self.design_mac_interface()
        elif platform.system() == "Windows":
            self.design_windows_interface()
        else:
             raise Exception("Unknown operating system. No Widget template
                     available.")

        #Server state: On -> True or Off -> False
        self.server_state = False

    #Methods handling logic and user inputs

    #Updates the amount of active clients
    def update_counter(self, count: int):
        self.client_counter.setText(f"{count}")

    #Updates the table with client infos
    def update_table(self, data: dict, remove: bool, update: bool):

        #False -> Add data
        if remove == False:

            #Check, whether new data is added or just updated

            #False -> Create a new line
            if update == False:

                self.add_table_data(data)

```

```
51              #True -> Update a line
52              if update == True:
53
54                  self.update_table_data(data)
55
56          #True -> Remove data
57          if remove == True:
58
59              self.remove_table_data(data)
60
61      #Updates the status line
62      def update_status(self, token: str, mode: str):
63
64          text = ""
65
66          if mode == "LOGIN":
67                text = f"New Client succesfully logged in.\nGenerated ID:
                    {token}."
68
69          elif mode == "LOGOUT":
70                text = f"A client has logged out.\nClient ID: {token}"
71
72          self.status_line.setText(text)
73
74      #Instructs the user to check the terminal
75      def notify_login(self):
76
77          text = "A new client attemps to login.\nPlease check the terminal for
                    further instructions."
78          self.status_line.setText(text)
79
80      #Info about server start
81      def notify_start(self):
82          text = "The server has been started."
83          self.status_line.setText(text)
84
85      #Starts the server
86      def click_on_start(self):
87
88          if self.server_state == False:
89
90              #Setting up server
91              self.server = Server()
92
93              #Connect signals
94              self.connect_signals()
95
96              #Start server in separate thread and get the url
97              self.server.start()
98
99              #Change state to on
100             self.server_state = True
101
102     #Stops the server
```

```
103     def click_on_shutdown(self):
104
105         if self.server_state == True:
106
107             #Notifying user
108             decision = self.show_confirmation_dialog()
109
110             self.server_state == False
111
112             if decision == True:
113                 #Calling shutdown method
114                 shutdown_url = self.base_url + "shutdown"
115                 response = requests.post(shutdown_url)
116
117     #Updating the URL with info from the server
118     def set_url(self, url: str):
119
120          self.base_url = url
121
122     #Confirmation dialog for server shutdown
123     def show_confirmation_dialog(self):
124
125         #Show the message box and capture the user response
126         reply = self.msg_box.exec_()
127
128         #Return True if 'Yes' is clicked, otherwise False
129         return reply == QMessageBox.Yes
130
131     def connect_signals(self):
132
133         #Connect server signals
134         self.server.counter_signal.connect(self.update_counter)
135         self.server.table_signal.connect(self.update_table)
136         self.server.user_info_signal.connect(self.update_status)
137         self.server.login_signal.connect(self.notify_login)
138         self.server.start_signal.connect(self.notify_start)
139         self.server.url_signal.connect(self.set_url)
140
141     #Adds the input data to the client table
142     def add_table_data(self, data: dict):
143
144         #Dict of column index to data
145             row_data = dict()
146             target_row = None
147
148             #Preparing data
149             #Token and plugin name are added during login
150             if "token" in data.keys():
151                     value = data["token"]
152                     target_row = self.determine_row(value)
153                     row_data[0] = str(value)
154             if "plugin_name" in data.keys():
155                     value = data["plugin_name"]
156                     row_data[1] = str(value)
```

```
157
158             #Filling up with empty strings to avoid Key Errors
159             row_data[2] = ""
160             row_data[3] = ""
161
162             #Adding data
163             for column in range(self.table_column_count):
164                 item = QTableWidgetItem(row_data[column])
165                 #Make data read only
166                 item.setFlags(item.flags() & ~Qt.ItemIsEditable)
167                 self.client_table.setItem(target_row, column, item)
168
169     #Updates the command and parameters columns of a row
170     def update_table_data(self, data: dict):
171
172         #Get the row of the existing token
173         if "token" in data.keys():
174             target_row = self.search_client(data["token"])
175         else:
176              raise Exception("Error in data transmission to widget: No token
                     included.")
177
178         #Update the fields if the row exists
179         if target_row is not None:
180             if "command" in data.keys():
181                     value = str(data["command"])
182                     item = QTableWidgetItem(value)
183                     self.client_table.setItem(target_row, 2, item)
184                     #Resize column depending on content
185                     if value in ["LIN", "PTP", "INFO", "LOG", "EXIT"]:
186                         self.client_table.setRowHeight(target_row, 50)
187                     elif value == "CIRC":
188                         self.client_table.setRowHeight(target_row, 100)
189             if "parameters" in data.keys():
190                     parameters = data["parameters"]
191                     parameters.pop("type")
192                     item = QTableWidgetItem(str(parameters))
193                     self.client_table.setItem(target_row, 3, item)
194
195     #Removes table rows
196     def remove_table_data(self, data: dict):
197
198         #Search for the token in the table
199         if "token" in data.keys():
200             target_row = self.search_client(data["token"])
201         else:
202             raise Exception("Error in data transmission to widget: No token
                    included.")
203
204         if target_row is not None:
205                 self.client_table.removeRow(target_row)
206
207         #Update row count to avoid empty rows
208         self.table_row_count -= 1
```

```
209
210     #Searches for a client in the table by token
211     #Returns the corresponding row
212     def determine_row(self, token: str):
213
214         target_row = self.search_client(token)
215
216         if target_row == None:
217             #If there are no rows yet, one will be created
218             if self.table_row_count == 0:
219                 target_row = 0
220                 self.table_row_count = 1
221                 self.client_table.setRowCount(self.table_row_count)
222             #Else a new row is added for the new data
223             else:
224                 target_row = self.table_row_count
225                 self.table_row_count += 1
226                 self.client_table.setRowCount(self.table_row_count)
227
228         #Check in case of errors
229         if target_row is None:
230             raise Exception("An issues with table data assignment has
                        occurred.")
231
232         return target_row
233
234     #Searches for an existing client in the table
235     #Returns None, if there is no match
236     def search_client(self, token: str):
237
238         target_row = None
239
240         #Checking all rows
241         for row in range(self.table_row_count):
242             #Token is always in the first column
243             item = self.client_table.item(row, 0)
244             if item is not None:
245                 if item.text() == token:
246                     target_row = row
247
248         return target_row
249
250
251     #Defines all interface elements for MacOS
252     def design_mac_interface(self):
253         #Application window setup
254             self.setWindowTitle("RemoteMind Telepath Cockpit")
255             self.setStyleSheet("background-color: white")
256             self.setGeometry(100, 100, 800, 700)
257
258             #Stylish App Name
259             prefix_font = QFont()
260             prefix_font.setPointSize(40)
261             prefix_font.setBold(True)
```

```
262
263             self.prefix_label = QLabel("Telepath", self)
264             self.prefix_label.setStyleSheet("color : purple; font : Lucida" )
265             self.prefix_label.setFont(prefix_font)
266             self.prefix_label.move(550,50)
267
268             suffix_font = QFont()
269             suffix_font.setPointSize(22)
270             suffix_font.setItalic(True)
271             suffix_font.setBold(True)
272             suffix_font.setFamily("Arial")
273
274             self.suffix_label = QLabel("RemoteMind", self)
275             self.suffix_label.setStyleSheet("color : dark grey;")
276             self.suffix_label.setFont(suffix_font)
277             self.suffix_label.move(600,30)
278
279             #Defining some general fonts
280             self.button_font = QFont()
281             self.button_font.setFamily("Arial")
282             self.button_font.setPointSize(14)
283
284             #Defining some general style sheets
285             generic_button_style = """QPushButton {
286                     border: 3px solid black;
287                     color: solid black;
288                     background-color: white;
289                     font-weight: normal;
290                     border-radius: 7px
291                 }
292                 QPushButton:hover {
293                     font-weight: bold;
294                 }
295             """
296
297             generic_textbox_style = """
298                 QTextEdit {
299                     border: 2px solid grey;
300                     border-radius: 7px;
301                     font-family: Courier New;
302                     font-size: 12px;
303                 }
304             """
305
306             #Setting font for the section header
307             self.section_font = QFont()
308             self.section_font.setBold(True)
309             self.section_font.setFamily("Arial")
310             self.section_font.setPointSize(22)
311
312             #Cockpit section: Check clients
313             self.cockpit_label = QLabel("Cockpit", self)
314             self.cockpit_label.setGeometry(50, 50, 100, 50)
315             self.cockpit_label.setFont(self.section_font)
```

```
316
317             #Server control
318
319             #Start server button
320             self.start_button = QPushButton("Start Server", self)
321             self.start_button.clicked.connect(self.click_on_start)
322             self.start_button.setFixedSize(200,50)
323             self.start_button.move(50, 150)
324             self.start_button.setFont(self.button_font)
325             self.start_button.setStyleSheet("""
326                 QPushButton {
327                     border: 3px solid green;
328                     color: black;
329                     background-color: white;
330                     border-radius : 7px;
331                 }
332                 QPushButton:hover {
333                     border: 3px solid green;
334                     color: green;
335                 }
336             """)
337
338             #Stop server button
339             self.shutdown_button = QPushButton("Shutdown Server", self)
340             self.shutdown_button.clicked.connect(self.click_on_shutdown)
341             self.shutdown_button.setFixedSize(200,50)
342             self.shutdown_button.move(50, 215)
343             self.shutdown_button.setFont(self.button_font)
344             self.shutdown_button.setStyleSheet("""
345                 QPushButton {
346                     border: 3px solid darkred;
347                     color: black;
348                     background-color: white;
349                     border-radius : 7px;
350                 }
351                 QPushButton:hover {
352                     border: 3px solid darkred;
353                     color: darkred;
354                 }
355             """)
356
357             #Status line
358
359             self.status_font = QFont()
360             self.status_font.setPointSize(16)
361             self.status_font.setFamily("Arial")
362
363             self.status_label = QLabel("Status", self)
364             self.status_label.setGeometry(300, 100, 50, 50)
365             self.status_label.setFont(self.status_font)
366
367             self.status_line = QTextEdit(self)
368             self.status_line.setGeometry(300, 150, 450, 115)
369             self.status_line.setStyleSheet("""
```

```
370                         QTextEdit {
371                             background-color: white;
372                             border: 3px solid black;
373                             border-radius: 7px;
374                             font-family: 'Courier New';
375                             font-size: 14px;
376                             color: black;
377                         }
378                     """
379                     )
380             self.status_line.setReadOnly(True)
381
382
383             #Number of active clients
384             self.counter_font = QFont()
385             self.counter_font.setPointSize(16)
386             self.counter_font.setFamily("Arial")
387
388             self.counter_label = QLabel("Active Clients: ", self)
389             self.counter_label.setGeometry(50, 100, 100, 50)
390             self.counter_label.setFont(self.counter_font)
391
392             self.counter_value = 0
393             self.client_counter = QLabel(f"{self.counter_value}", self)
394             self.client_counter.setGeometry(175, 100, 50, 50)
395             self.client_counter.setFont(self.counter_font)
396
397             #Table of registered clients
398             #Client is represented by the token
399             #Additionaly shows the plugin and the last command type plus its
                    confirmation status
400             self.client_table = QTableWidget(self)
401             self.column_headers = ["Token", "Plugin", "Recent Command",
                    "Recent Parameters"]
402             self.client_table.setColumnCount(len(self.column_headers))
403             self.client_table.setRowCount(0)
404             self.client_table.setHorizontalHeaderLabels(self.column_headers)
405             self.client_table.setGeometry(50, 280, 700, 400)
406
407             self.table_row_count = 0
408             self.table_column_count = len(self.column_headers)
409
410             #Setting column width
411             row_widths = [80, 100, 125, 375]
412             for i in range(self.table_column_count):
413                 self.client_table.setColumnWidth(i, row_widths[i])
414
415             self.client_table.setStyleSheet("""
416                         QTableWidget {
417                             border: 3px solid grey;
418                             border-radius: 7px;
419                             font-family: "Courier New";
420                             font-size: 14px;
421                         }
```

```
422                             QTableWidget::item {
423                                 border-bottom: 1px solid grey
424                             }
425                             QTableWidget::item:selected {
426                                 background-color: grey;
427                             }
428                             QHeaderView::section {
429                                 background-color: white;
430                                 color: black;
431                                 font-family: "Arial";
432                                 font-size: 14px
433                                 border-bottom: 1px solid grey
434                             }
435                             QTableWidget::horizontalHeader {
436                                 border-bottom: 1px solid grey
437                                 color: black
438                                 font-family: "Arial";
439                                 font-size: 14px;
440                             }
441                             QTableWidget::verticalHeader {
442                                 border-bottom: 1px solid grey
443                                 color: black
444                                 font-family: "Arial";
445                                 font-size: 14px;
446                             }
447                         """)
448
449             #Confirmation dialog
450
451             self.msg_box = QMessageBox()
452
453             #Dialog text
454             self.msg_box.setWindowTitle('Server Shutdown')
455             self.msg_box.setText('Do you really want to shut down the server?
                    This will close the entire application.')
456
457             #Dialog Functionality
458             self.msg_box.setStandardButtons(QMessageBox.Yes |
                    QMessageBox.Cancel)
459             self.msg_box.setDefaultButton(QMessageBox.Cancel)
460
461             #Dialog Styling
462             self.msg_box.setIcon(QMessageBox.Warning)
463
464             self.msg_box.setStyleSheet("""
465                 QMessageBox {
466                     background-color: white;
467                     color: black;
468                     font-size: 14px;
469                     font-family: "Arial"
470                 }
471                 QPushButton {
472                     background-color: white;
473                     color: black;
```

```
                font-family: "Arial"
                border-radius: 7px;
                font-size: 14px;
                border: 3px solid black

            }
            QPushButton:hover {
                font-weight: bold
            }
        """)

        #Version and License
        self.info_font = QFont()
        self.info_font.setPointSize(8)
        self.info_font.setFamily("Arial")

        info = "@2024. Version 1.0: Initial release."
        self.info_label = QLabel(info, self)
        self.info_label.move(0,0)
        self.info_label.setStyleSheet("color : grey")
        self.info_label.setFont(self.info_font)

    #Defines all interface elements for Microsoft Windows
    def design_windows_interface(self):
        #Application window setup
        self.setWindowTitle("RemoteMind Telepath Cockpit")
        self.setStyleSheet("background-color: white")
        self.setGeometry(100, 100, 800, 700)

        #Stylish App Name
        prefix_font = QFont()
        prefix_font.setPointSize(40)
        prefix_font.setBold(True)

        self.prefix_label = QLabel("Telepath", self)
        self.prefix_label.setStyleSheet("color : purple; font : Lucida" )
        self.prefix_label.setFont(prefix_font)
        self.prefix_label.move(550,50)

        suffix_font = QFont()
        suffix_font.setPointSize(22)
        suffix_font.setItalic(True)
        suffix_font.setBold(True)
        suffix_font.setFamily("Arial")

        self.suffix_label = QLabel("RemoteMind", self)
        self.suffix_label.setStyleSheet("color : dark grey;")
        self.suffix_label.setFont(suffix_font)
        self.suffix_label.move(600,30)

        #Defining some general fonts
        self.button_font = QFont()
        self.button_font.setFamily("Arial")
        self.button_font.setPointSize(14)
```

```
528
529             #Defining some general style sheets
530             generic_button_style = """QPushButton {
531                     border: 3px solid black;
532                     color: solid black;
533                     background-color: white;
534                     font-weight: normal;
535                     border-radius: 7px
536                 }
537                 QPushButton:hover {
538                     font-weight: bold;
539                 }
540             """
541
542             generic_textbox_style = """
543                 QTextEdit {
544                     border: 2px solid grey;
545                     border-radius: 7px;
546                     font-family: Courier New;
547                     font-size: 12px;
548                 }
549             """
550
551             #Setting font for the section header
552             self.section_font = QFont()
553             self.section_font.setBold(True)
554             self.section_font.setFamily("Arial")
555             self.section_font.setPointSize(22)
556
557             #Cockpit section: Check clients
558             self.cockpit_label = QLabel("Cockpit", self)
559             self.cockpit_label.setGeometry(50, 50, 150, 50)
560             self.cockpit_label.setFont(self.section_font)
561
562             #Server control
563
564             #Start server button
565             self.start_button = QPushButton("Start Server", self)
566             self.start_button.clicked.connect(self.click_on_start)
567             self.start_button.setFixedSize(200,50)
568             self.start_button.move(50, 150)
569             self.start_button.setFont(self.button_font)
570             self.start_button.setStyleSheet("""
571                 QPushButton {
572                     border: 3px solid green;
573                     color: black;
574                     background-color: white;
575                     border-radius : 7px;
576                 }
577                 QPushButton:hover {
578                     border: 3px solid green;
579                     color: green;
580                 }
581             """)
```

```
582
583             #Stop server button
584             self.shutdown_button = QPushButton("Shutdown Server", self)
585             self.shutdown_button.clicked.connect(self.click_on_shutdown)
586             self.shutdown_button.setFixedSize(200,50)
587             self.shutdown_button.move(50, 215)
588             self.shutdown_button.setFont(self.button_font)
589             self.shutdown_button.setStyleSheet("""
590                 QPushButton {
591                     border: 3px solid darkred;
592                     color: black;
593                     background-color: white;
594                     border-radius : 7px;
595                 }
596                 QPushButton:hover {
597                     border: 3px solid darkred;
598                     color: darkred;
599                 }
600             """)
601
602             #Status line
603
604             self.status_font = QFont()
605             self.status_font.setPointSize(16)
606             self.status_font.setFamily("Arial")
607
608             self.status_label = QLabel("Status", self)
609             self.status_label.setGeometry(300, 100, 100, 50)
610             self.status_label.setFont(self.status_font)
611
612             self.status_line = QTextEdit(self)
613             self.status_line.setGeometry(300, 150, 450, 115)
614             self.status_line.setStyleSheet("""
615                         QTextEdit {
616                             background-color: white;
617                             border: 3px solid black;
618                             border-radius: 7px;
619                             font-family: 'Courier New';
620                             font-size: 14px;
621                             color: black;
622                         }
623                     """
624                     )
625             self.status_line.setReadOnly(True)
626
627
628             #Number of active clients
629             self.counter_font = QFont()
630             self.counter_font.setPointSize(16)
631             self.counter_font.setFamily("Arial")
632
633             self.counter_label = QLabel("Active Clients: ", self)
634             self.counter_label.setGeometry(50, 100, 150, 50)
635             self.counter_label.setFont(self.counter_font)
```

```
636
637             self.counter_value = 0
638             self.client_counter = QLabel(f"{self.counter_value}", self)
639             self.client_counter.setGeometry(200, 100, 50, 50)
640             self.client_counter.setFont(self.counter_font)
641
642             #Table of registered clients
643             #Client is represented by the token
644             #Additionaly shows the plugin and the last command type plus its
                    confirmation status
645             self.client_table = QTableWidget(self)
646             self.column_headers = ["Token", "Plugin", "Recent Command",
                    "Recent Parameters"]
647             self.client_table.setColumnCount(len(self.column_headers))
648             self.client_table.setRowCount(0)
649             self.client_table.setHorizontalHeaderLabels(self.column_headers)
650             self.client_table.setGeometry(50, 280, 700, 400)
651
652             self.table_row_count = 0
653             self.table_column_count = len(self.column_headers)
654
655             #Setting column width
656             row_widths = [80, 100, 125, 375]
657             for i in range(self.table_column_count):
658                 self.client_table.setColumnWidth(i, row_widths[i])
659
660             self.client_table.setStyleSheet("""
661                             QTableWidget {
662                                 border: 3px solid grey;
663                                 border-radius: 7px;
664                                 font-family: "Courier New";
665                                 font-size: 14px;
666                             }
667                             QTableWidget::item {
668                                 border-bottom: 1px solid grey
669                             }
670                             QTableWidget::item:selected {
671                                 background-color: grey;
672                             }
673                             QHeaderView::section {
674                                 background-color: white;
675                                 color: black;
676                                 font-family: "Arial";
677                                 font-size: 14px
678                                 border-bottom: 1px solid grey
679                             }
680                             QTableWidget::horizontalHeader {
681                                 border-bottom: 1px solid grey
682                                 color: black
683                                 font-family: "Arial";
684                                 font-size: 14px;
685                             }
686                             QTableWidget::verticalHeader {
687                                 border-bottom: 1px solid grey
```

```
688                                 color: black
689                                 font-family: "Arial";
690                                 font-size: 14px;
691                             }
692                         """)
693
694             #Confirmation dialog
695
696             self.msg_box = QMessageBox()
697
698             #Dialog text
699             self.msg_box.setWindowTitle('Server Shutdown')
700             self.msg_box.setText('Do you really want to shut down the server?
                    This will close the entire application.')
701
702             #Dialog Functionality
703             self.msg_box.setStandardButtons(QMessageBox.Yes |
                    QMessageBox.Cancel)
704             self.msg_box.setDefaultButton(QMessageBox.Cancel)
705
706             #Dialog Styling
707             self.msg_box.setIcon(QMessageBox.Warning)
708
709             self.msg_box.setStyleSheet("""
710                 QMessageBox {
711                     background-color: white;
712                     color: black;
713                     font-size: 14px;
714                     font-family: "Arial"
715                 }
716                 QPushButton {
717                     background-color: white;
718                     color: black;
719                     font-family: "Arial"
720                     border-radius: 7px;
721                     font-size: 14px;
722                     border: 3px solid black
723
724                 }
725                 QPushButton:hover {
726                     font-weight: bold
727                 }
728             """)
729
730             #Version and License
731             self.info_font = QFont()
732             self.info_font.setPointSize(8)
733             self.info_font.setFamily("Arial")
734
735             info = "@2024. Version 1.0: Initial release."
736             self.info_label = QLabel(info, self)
737             self.info_label.move(0,0)
738             self.info_label.setStyleSheet("color : grey")
```

```
739             self.info_label.setFont(self.info_font)
```

plugin_interface.py

```python
"""This abstract class serves as an interface for all plugins"""
#Importing modules
from abc import ABC, abstractmethod
import os
import inspect

#File names must adhere to the naming convention telepath_"filename"
#There must only be one class in this file, which must implement the interface
#However, the plugin itself can consist of several other files as well
class PluginInterface(ABC):

    plugins = []

    #Can be used to setup the application
    @abstractmethod
    def __init__(self):
        pass

    #For every class that inherits from the current one,
    #the class name will be added to plugins
    #Source: https://gist.github.com/dorneanu/cce1cd6711969d581873a88e0257e312
    def __init_subclass__(cls, **kwargs):
        super().__init_subclass__(**kwargs)
        cls.plugins.append(cls)

    #Returns a list of all instantiated subclasses
    @classmethod
    def show_all_plugins(cls):
        return cls.plugins

    #Returns the path of the class
    @classmethod
    def show_path(cls):
        path = inspect.getfile(cls)
        return os.path.realpath(path)

    #The core method of every plug-in
    #Will be called by the server
    #Will always receive the last known application state as a dict
    #The form of state depends on the plugin design
    @abstractmethod
    def run(self, state: dict):
        #Must always return a tuple: command object (as specified by the API),
              application state (dict)
        #Command object can be imported as a seprate class inside 'plugins'
        #State must allow the plugin to resume functionality seemlessly
        #State will also be passed as a dict: plugin must access relevant keys
        pass

    #Must return a starting state during plugin selection
    #State must be a dict
    @abstractmethod
```

```
    def setup(self):
        pass
```

commands.py

```python
"""Implements command classes as output of every robot plugin as specified in the API"""
from abc import ABC, abstractmethod
import json

class Command(dict, ABC):

    #Initialize a custom dict with the keys command, parameters and type
    #If input is not given, frames default to {}
    #Should raise a error, due to faulty media type
    def __init__(self, command: str, frame: dict = {}, auxiliaryFrame: dict = {}, destination: dict = {}):

        #Checks input frames
        if command in ["LIN", "PTP"]:
            self.verify_frame(frame)
        elif command == "CIRC":
            self.verify_frame(auxiliaryFrame)
            self.verify_frame(destination)

        super().__init__()
        self.allowed_commands = ["EXIT", "LOG", "SEND", "PTP", "CIRC", "LIN"]

        if command not in self.allowed_commands:
            raise InvalidCommandError(command)
        else:
            self["command"] = command
            self["parameters"] = dict()
            self["parameters"]["type"] = "Param" + str(command)

        self.set_specifics(frame, auxiliaryFrame, destination)

    #This method is implemented by command subclasses to add unique elements
    @abstractmethod
    def set_specifics(self, frame: dict, auxiliaryFrame: dict, destination: dict):
        pass

    #Checks the frame for correct keys and datatypes
    def verify_frame(self, frame: dict):

        frame_elements = ["a", "b", "c", "x", "y", "z"]
        if not isinstance(frame, dict):
            raise TypeError(f"Expected frame to be a dict, but got {type(frame)}.")

        for element in frame_elements:
            try:
                #Must be int or float
                if type(frame[element]) not in [int, float]:
                    raise TypeError
            except KeyError as e:
```

```
                print(e)
                break

    #Returns the command itself
    def return_dict(self):
        # Convert instance to dictionary
        return dict(self)

    #Returns itself as a json object
    def return_json(self):
        return json.dumps(self.return_dict())

"Throws an error if a command is not EXIT, LOG, SEND, PTP, CIRC or LIN"
class InvalidCommandError(Exception):

    def __init__(self, command):
        self.allowed_commands = ["EXIT", "LOG", "SEND", "PTP", "CIRC", "LIN"]
        if command not in self.allowed_commands:
            super().__init__(f"Invalid command: {command}. Expected one of:
                  ['EXIT', 'LOG', 'SEND', 'PTP', 'CIRC', 'LIN']")

#Class for EXIT commands
class CommandEXIT(Command):

    def __init__(self):
        super().__init__(command = "EXIT", frame = {}, auxiliaryFrame = {},
             destination = {})

    #No specifics for EXIT
    def set_specifics(self, frame: dict, auxiliaryFrame: dict, destination:
             dict):
        pass

#Class for LOG commands
class CommandLOG(Command):

    def __init__(self):
        super().__init__(command = "LOG", frame = {}, auxiliaryFrame = {},
              destination = {})

    #No specifics for LOG
    def set_specifics(self, frame: dict, auxiliaryFrame: dict, destination:
              dict):
        pass

#Class for SEND commands
class CommandSEND(Command):

    def __init__(self):
        super().__init__(command = "SEND", frame = {}, auxiliaryFrame = {},
              destination = {})
```

```
97      #No specifics for SEND
98      def set_specifics(self, frame: dict, auxiliaryFrame: dict, destination:
                dict):
99          pass
100
101 #Class for LIN commands
102 class CommandLIN(Command):
103
104     def __init__(self, frame: dict):
105         super().__init__(command = "LIN", frame = frame, auxiliaryFrame = {},
                destination = {})
106
107     #Frame with 6 values
108     def set_specifics(self, frame: dict, auxiliaryFrame: dict, destination:
                dict):
109         self["parameters"]["Frame"] = frame
110
111 #Class for PTP commands
112 class CommandPTP(Command):
113
114     def __init__(self, frame: dict):
115         super().__init__(command = "PTP", frame = frame, auxiliaryFrame = {},
                destination = {})
116
117     #Frame with 6 values
118     def set_specifics(self, frame: dict, auxiliaryFrame: dict, destination:
                dict):
119         self["parameters"]["Frame"] = frame
120
121 #Class for CIRC commands
122 class CommandCIRC(Command):
123
124     def __init__(self, auxiliaryFrame: dict, destination: dict):
125         super().__init__(command = "CIRC", frame = {}, auxiliaryFrame =
                auxiliaryFrame, destination = destination)
126
127     #2 Frames with 6 values each
128     def set_specifics(self, frame: dict, auxiliaryFrame: dict, destination:
                 dict):
129         self["parameters"]["auxiliaryFrame"] = auxiliaryFrame
130         self["parameters"]["destination"] = destination
```

mongo_interface.py

```python
"""This serves as an interface to MongoDB for Plugins in need of additional data access beyond state"""
#Pymongo is a Python driver for MongoDB
import pymongo

class MongoInterface:

    #Connect to the running MongoDB instance
    def __init__(self):

        self.client = pymongo.MongoClient("mongodb://localhost:27017")

        self.db = self.client["REST_AI_Server"]

        #The standard server collections
        #Read only: Manipulation is not allowed and can impact the server
        self.collections_list = ["app_state", "logs", "infos", "plugins", "commands"]

    #Returns a list of the requested documents after taking in a query dictionary
    #So far only used for database testing
    #Empty dict will return the whole collection
    def query_data(self, collection: str, query_dict: dict):
        if collection in self.collections_list:
            chosen_collection = self.db[collection]
            #Returns a cursor object and turns it into a list of dictionaries
            document = list(chosen_collection.find(query_dict))
            return document

    #Saves the dictionary in the specified collection
    #If the collection does not exist yet, it will be created
    def save_data(self, collection: str, db_file: dict):

        if collection not in self.collections_list:
            chosen_collection = self.db[collection]

            chosen_collection.insert_one(db_file)

    #Deletes data based on the search parameters
    def delete_data(self, collection: str, parameters: dict):

        if collection not in self.collections_list:
            collection.delete_many(parameters)

    #Deletes the specified collection
    def delete_collection(self, collection: str):
        if collection in self.db.list_collection_names():
            if collection not in self.collections_list:
                self.db[collection].drop()
                print(f"Collection {collection} has been deleted.")
            else:
```

```
50                 print(f"Deletion of selected collection {collection} not
                       allowed.")
51         else:
52             print(f"Collection {collection} does not exist.")
```

telepath_commands.py

```
"""Test Plugin implementing the Plugin Interface to showcase all available commands"""
import sys
import os
sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), '..', '..')))

#Importing the interface
from packages.plugins.utils.plugin_interface import PluginInterface
#Importing command objects
from packages.plugins.utils.commands import *

#This is a test plugin
class TestPlugin(PluginInterface):

    #Constructor not necessary here
    def __init__(self):
        pass

    #Returns default state
    #Used during plug-in loading
    def setup(self):

        print("Setting up the Telepath Commands plug-in...")

        state = dict()
        state["counter"] = 1

        print("Telepath Commands active.")
        return state

    #Shows off all commands in a total of 6 request cycles
    def run(self, state: dict):

        #Application state is restored from DB info
        #State is passed in as a dict
        self.counter = state["counter"]

        #Command generation
        if self.counter == 1:
            frame = {
                    "x" : 498.7,
                    "y" : 436.5,
                    "z" : 478,
                    "a" : -1.6,
                    "b" : 0.03,
                    "c" : 3.0
                             }
            command = CommandLIN(frame=frame)
        elif self.counter == 2:
            frame = {
                    "x" : 520.7,
```

```
51                      "y" : 325.5,
52                      "z" : 400.0,
53                      "a" : -1.6,
54                      "b" : 0.03,
55                      "c" : 3.0
56                              }
57              command = CommandPTP(frame)
58
59          elif self.counter == 3:
60              auxiliaryFrame = {
61                              "x" : 510,
62                              "y" : 370.5,
63                              "z" : 400.0,
64                              "a" : -1.6,
65                              "b" : 0.03,
66                              "c" : 3.0
67                               }
68              destination =   {
69                              "x" : 498.7,
70                              "y" : 436.5,
71                              "z" : 478,
72                              "a" : -1.6,
73                              "b" : 0.03,
74                              "c" : 3.0
75                               }
76              command = CommandCIRC(auxiliaryFrame = auxiliaryFrame,
77  destination=destination)
78
79          elif self.counter == 4:
80              command = CommandLOG()
81          elif self.counter == 5:
82              command = CommandSEND()
83          elif self.counter == 6:
84              command = CommandEXIT()
85
86          #Generation of state dictionary
87          app_state = dict()
88          if self.counter < 6:
89              self.counter += 1
90          app_state["counter"] = self.counter
91
92          return command, app_state
```

telepath_sync.py

```
"""This demo serves as an example for control of two robots at once, which alternate in their requests"""
#Fixing imports
import sys
import os
sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), '..', '..')))

#Import plugin architecture utilities
from packages.plugins.utils.plugin_interface import PluginInterface
from packages.plugins.utils.commands import *
#Import the MongoDB interface
from packages.plugins.utils.mongo_interface import MongoInterface
#Calculating distance
import math

class SynchronizedControl(PluginInterface):

    #Constructor
    def __init__(self):

        #Create an instance of the data manager
        self.mongo = MongoInterface()

        #This plugin needs two robots to work
        self.max_robot_count = 2

        #Available positions: two opposite corners of a 14 cm x 14 cm square
        self.pos1 = {
                    "x" : 500,
                    "y" : 160,
                    "z" : 320,
                    "a" : -90.18,
                    "b" : 1.49,
                    "c" : 173.94
                            }
        self.pos2 = {
                    "x" : 640,
                    "y" : 300,
                    "z" : 320,
                    "a" : -90.18,
                    "b" : 1.49,
                    "c" : 173.94
                            }

        #List of all tasks
        self.all_tasks = dict(self.define_tasks())


    #Set-up
    def setup(self):

```

```
        #Creating the initial state
        state = dict()

        #If too many robots log in, they will be given an EXIT command
        exit_condition = self.check_exit_condition()
        state["Exit condition"] = exit_condition

        #Only create the correct state, if there are enough robots
        if exit_condition == False:

            #Here we will save a list of the synchronized robots
            state["Synced robots"] = self.get_clients()

            #If too many robots log in, they will be given an EXIT command
            exit_condition = self.check_exit_condition()

            #Saving the tasks done by the robot
            state["Finished tasks"] = list()

            #Saving the robot role/position
            #Since this is just a proof of concept, there will be two pre-
                  selected positions
            state["Position"] = self.select_position()

            #Check ability to start the operation
            state["Operational"] =
                  self.check_operational_capability(state["Synced robots"])

        print("Sync plugin sucessfully set-up.")

        return state

    #Main method for robot control
    def run(self, state: dict):

        #Immediately log out all unnecessary clients
        if state["Exit condition"]  == True:
            kick_state = {"Exit condition": True, "Logout state": "Succesful"}
            command = CommandEXIT()

            return command, kick_state

        updated_state = dict(state)

        #Update the client count first
        updated_state = dict(self.update_synced_clients(state))

        #Update the operational capability
        updated_state["Operational"] =
            self.check_operational_capability(updated_state["Synced robots"])

        #Check the operationl capability
        check = updated_state["Operational"]

```

```
        #If there are not enough clients logged in, the robot should not move
        if check == False:
            command = CommandSEND()

        #Else a new command is selected
        else:
            next_task = self.select_command(updated_state)

            #If tasks are still available
            if next_task != None:
                command = self.all_tasks[next_task]

                #Update state
                updated_state["Finished tasks"].append(next_task)

            #Otherwise Log-Out
            else:
                if "Wait" not in updated_state.keys():
                    updated_state["Wait"] = True

                #Wait before logging out, to allow the companion to catch this client's final tasks
                #This demo only works if the client requests alternate, otherwise a new MongoDB collection must be used to store the info about task completion
                if updated_state["Wait"] == True:
                    command = CommandSEND()
                    updated_state["Wait"] = False
                elif state["Wait"] == False:
                    command = CommandEXIT()

        print("Results")
        print(command)
        print(updated_state)

        return command, updated_state

    #Selects a new command based on the robot position and finished assignments
    def select_command(self, state: dict):

        #Stage I: Get all task keys, which are not yet listed as finished by this client
        available_tasks = list()
        for key in self.all_tasks.keys():
            if key not in state["Finished tasks"]:
                available_tasks.append(key)

        print(f"Stage I: available tasks: {available_tasks}")

        #Stage II: Check the other client's finished tasks and remove them too

        #Get the companion's token
        second_token = ""
```

```
151         for token in state["Synced robots"]:
152             if token != state["token"]:
153                 second_token = token
154 
155         #Query the companion's app state
156         collection = "app_state"
157         query = {"token": second_token}
158         results = self.mongo.query_data(collection, query)
159 
160         #Remove the finished tasks, if there are any
161         try:
162             result = results[0]
163             companion_tasks = result["Finished tasks"]
164 
165             print(f"Companions finished tasks: {companion_tasks}")
166 
167             for task_key in companion_tasks:
168                 if task_key in available_tasks:
169                     available_tasks.remove(task_key)
170 
171         except IndexError:
172             print("Error")
173 
174         finally:
175 
176             print(f"Stage II Available tasks: {available_tasks}")
177             if len(available_tasks) > 0:
178                 #Stage III: Select the task by priority
179                 position = state["Position"]
180                 recommended_task = self.prioritize_tasks(available_tasks,
                            position)
181 
182             else:
183                 recommended_task = None
184 
185             print(f"Recommended task: {recommended_task}")
186 
187             return recommended_task
188 
189     #Prioritizes tasks based on the distance between robot position and target
190     #Returns the recommended task
191     def prioritize_tasks(self, available_tasks: list, position: dict):
192 
193         #Defaults to recommending the first task
194         recommended_task = available_tasks[0]
195         command = self.all_tasks[recommended_task]
196         if isinstance(command, CommandLIN):
197             frame = command["parameters"]["Frame"]
198         elif isinstance(command, CommandPTP):
199             frame = command["parameters"]["Frame"]
200         elif isinstance(command, CommandCIRC):
201             frame = command["parameters"]["destination"]
202         base_distance = self.calculate_distance(frame, position)
203 
```

```
204         #Checking, whether other tasks are better
205         for task_key in available_tasks:
206             #Get the frame
207             command = self.all_tasks[task_key]
208             if isinstance(command, CommandLIN):
209                 frame = command["parameters"]["Frame"]
210             elif isinstance(command, CommandPTP):
211                 frame = command["parameters"]["Frame"]
212             elif isinstance(command, CommandCIRC):
213                 frame = command["parameters"]["destination"]
214             distance = self.calculate_distance(frame, position)
215             if distance < base_distance:
216                 base_distance = distance
217                 recommended_task = task_key
218 
219         return recommended_task
220 
221     #Calculates the euclidean distance between the target frame and the client
            positions
222     #This is used for the optimization of task seletion
223     def calculate_distance(self, target_frame: dict, client_position: dict):
224 
225         sum = 0
226 
227         for coordinate in ["x", "y", "z"]:
228             num = (target_frame[coordinate] - client_position[coordinate])**2
229             sum += num
230 
231         result = math.sqrt(sum)
232 
233         return result
234 
235 
236     #Allows the user to select one of two positions for the robot
237     def select_position(self):
238 
239         print("Robot selection for specific tasks will depend on their
                position.")
240 
241         pos1_status, pos2_status = self.verify_position()
242 
243         #Print the positions depending on the already registered robots
244         print("Available positions:")
245         if pos1_status == False and pos2_status == False:
246             print(f"Position 1:\n{self.pos1}")
247             print(f"Position 2:\n{self.pos2}")
248             available_positions = {"1": self.pos1, "2": self.pos2}
249         elif pos1_status == True and pos2_status == False:
250             print(f"Position 2:\n{self.pos2}")
251             available_positions = {"2": self.pos2}
252         elif pos1_status == False and pos2_status == True:
253             print(f"Position 1:\n{self.pos1}")
254             available_positions = {"1": self.pos1}
255 
```

```
256         user_input = input("Please select a position by index: ")
257
258         allowed_input = False
259         while allowed_input == False:
260             try:
261                 position = available_positions[user_input]
262             except KeyError:
263                 user_input = input("Position not found. Please try again: ")
264             else:
265                 allowed_input = True
266                 print("Position selection succesful.")
267
268         return position
269
270     #Checks, if any positions have already been selected
271     def verify_position(self):
272
273         #First look up, which robots are logged into this plugin as well
274         collection1 = "plugins"
275         #Plugin names end with .py
276         query1 = {"plugin_name": "telepath_sync.py"}
277         query_results1 = self.mongo.query_data(collection1, query1)
278
279         #Conditions
280         pos1_selected = False
281         pos2_selected = False
282
283         #Search for the app states of the clients using the plugin
284         for result in query_results1:
285             token = result["token"]
286             collection2 = "app_state"
287             query2 = {"token": token}
288             query_results2 = self.mongo.query_data(collection2, query2)
289
290             #Check the query results
291             for client in query_results2:
292                 try:
293                     if client["Position"] == self.pos1:
294                         pos1_selected = True
295                     elif client["Position"] == self.pos2:
296                         pos2_selected = True
297                 except KeyError:
298                     pass
299
300         return pos1_selected, pos2_selected
301
302     #Exit condition:
303     #If there are already two robots registered, this function will return
304 True
305     def check_exit_condition(self):
306
307         collection = "plugins"
308         #Plugin names end with .py
309         query = {"plugin_name": "telepath_sync.py"}
```

```
310         robot_list = self.mongo.query_data(collection, query)
311
312         if len(robot_list) > 2:
313             return True
314         elif len(robot_list) <= 2:
315             return False
316
317     #Checks, whether all necessary actors are logged in
318     def get_clients(self):
319
320         #Query all clients using the plugin
321         #Plugin names end with .py
322         collection1 = "plugins"
323         query1 = {"plugin_name": "telepath_sync.py"}
324         robot_list = self.mongo.query_data(collection1, query1)
325
326         #Check, whether any of these clients are about to be logged out
327         #Get a list of these tokens
328         forbidden_tokens = list()
329         for robot in robot_list:
330             collection2 = "app_state"
331             query2 = {"token": robot["token"]}
332             results = self.mongo.query_data(collection2, query2)
333             for result in results:
334                 try:
335                     if result["Exit condition"] == True:
336                         forbidden_tokens.append(result["token"])
337                 except KeyError:
338                     None
339
340         #Get and return the allowed tokens
341         tokens = list()
342         for robot in robot_list:
343             if robot["token"] not in forbidden_tokens:
344                 tokens.append(robot["token"])
345
346         return tokens
347
348     #Updates the state with the actual clients
349     def update_synced_clients(self, state: dict):
350
351         #First, add the clients own token to the state, if not already done
352         if state["token"] not in state["Synced robots"]:
353             state["Synced robots"].append(state["token"])
354
355         #Check the second clients token and append it
356         tokens = self.get_clients()
357         second_token = ""
358         for token in tokens:
359             if token != state["token"]:
360                 second_token = token
361         if second_token != "":
362             if second_token not in  state["Synced robots"]:
363                 state["Synced robots"].append(second_token)
```

```
364
365         return state
366
367     #Check the ability to start the operation
368     #Requires a list of clients as input like state["Synced robots"]
369     def check_operational_capability(self, active_client_list: list):
370
371         #Check ability to start the operation
372             if len(active_client_list) == 2:
373                 return True
374             else:
375                 return False
376
377     #Defines a total of six movement tasks
378     def define_tasks(self):
379
380         tasks = dict()
381
382         #First task
383         frame_one = {
384                     "x" : 540,
385                     "y" : 175,
386                     "z" : 320,
387                     "a" : -90.18,
388                     "b" : 1.49,
389                     "c" : 173.94
390                             }
391         command_one = CommandPTP(frame_one)
392         tasks[1] = command_one
393
394         #Second task
395         frame_two = {
396                     "x" : 500,
397                     "y" : 230,
398                     "z" : 320,
399                     "a" : -90.18,
400                     "b" : 1.49,
401                     "c" : 173.94
402                             }
403         command_two = CommandPTP(frame_two)
404         tasks[2] = command_two
405
406         #Third task
407         frame_three = {
408                     "x" : 430,
409                     "y" : 230,
410                     "z" : 320,
411                     "a" : -90.18,
412                     "b" : 1.49,
413                     "c" : 173.94
414                             }
415         command_three = CommandPTP(frame_three)
416         tasks[3] = command_three
417
```

```
418         #Fourth task
419         frame_four = {
420                     "x" : 640,
421                     "y" : 215,
422                     "z" : 320,
423                     "a" : -90.18,
424                     "b" : 1.49,
425                     "c" : 173.94
426                             }
427         command_four = CommandPTP(frame_four)
428         tasks[4] = command_four
429
430         #Fifth task
431         frame_five = {
432                     "x" : 580,
433                     "y" : 275,
434                     "z" : 320,
435                     "a" : -90.18,
436                     "b" : 1.49,
437                     "c" : 173.94
438                             }
439         command_five = CommandPTP(frame_five)
440         tasks[5] = command_five
441
442         #Sixth task
443         frame_six = {
444                     "x" : 500,
445                     "y" : 300,
446                     "z" : 320,
447                     "a" : -90.18,
448                     "b" : 1.49,
449                     "c" : 173.94
450                             }
451         command_six = CommandPTP(frame_six)
452         tasks[6] = command_six
453
454         return tasks
```

telepath_skilltree.py

```
"""This is a demo of an ontology featuring a video game skill tree of robot commands"""
#Fixing imports
import sys
import os
sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), '..', '..')))

#Importing owlready2
from owlready2 import get_ontology, sync_reasoner
#Importing the commands
from packages.plugins.utils.commands import *
#Importing interface
from packages.plugins.utils.plugin_interface import PluginInterface
#Parsing string and converting to json
import json

class Skilltree(PluginInterface):

    #Init loads the ontology
    def __init__(self):
        dir_path = os.path.dirname(os.path.realpath(__file__))
        path = os.path.join(dir_path, "skilltree_ontology.rdf")
        self.ontology = get_ontology(path).load()
        print("Ontology can be accessed.")

    #The skilltree will start at level one
    def setup(self):

        print("Setting up Telepath Skilltree plug-in...")

        initial_level = 1
        setup_file = dict()
        setup_file["level"] = initial_level

        print("Telepath Skilltree active.")

        return setup_file

    def run(self, state: dict):

        #Access the current level and query the corresponding individual
        level_individual = self.ontology.search_one(hasValue=state["level"])
        #Checks the corresponding skill and get associated data
        unlocked_skills = level_individual.unlocks
        skill = unlocked_skills
        name = skill.hasName

        #Generating a new command
        command = self.generate_command(skill)

        #Generating new state through reasoning
```

```
51          #Only a previous_level property is explicitely modeled
52          #The reverse property next_level must be identified by the reasoner
53
54          #Start the reasoner
55          with self.ontology:
56              sync_reasoner()
57
58          #An index error will occur on the last iteration
59          try:
60              next_level = list(level_individual.nextLevel)[0]
61
62          except IndexError as e:
63              print(e)
64              new_state = state
65          else:
66              new_state = dict()
67              new_state["level"] = next_level.hasValue
68
69          return command, new_state
70
71      #Creates the correct command based on ontology information
72      def generate_command(self, skill: object):
73          name = skill.hasName
74          if name == "LOG":
75              command = CommandLOG()
76              return command
77          elif name == "EXIT":
78              command = CommandEXIT()
79              return command
80          elif name == "SEND":
81              command = CommandSEND()
82              return command
83          elif name == "PTP":
84              json_string = skill.hasFrame
85              frame = json.loads(json_string)
86              command = CommandPTP(frame)
87              return command
88          elif name == "LIN":
89              json_string = skill.hasFrame
90              frame = json.loads(json_string)
91              command = CommandLIN(frame)
92              return command
93          elif name == "CIRC":
94              aux_string = skill.hasAuxiliaryFrame
95              dest_string = skill.hasDestination
96              auxiliaryFrame = json.loads(aux_string)
97              destination = json.loads(dest_string)
98              command = CommandCIRC(auxiliaryFrame, destination)
99              return command
```

telepath_blueprints.py

```
1  """This plugin generates commands by querying an ontology with a reasoner,
           which models the structure of a product part by part"""
2  #Fixing imports
3  import sys
4  import os
5  sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), '..',
           '..')))
6
7  #Importing owlready2
8  from owlready2 import get_ontology, sync_reasoner
9  #Importing the commands
10 from packages.plugins.utils.commands import *
11 #Importing interface
12 from packages.plugins.utils.plugin_interface import PluginInterface
13 #Parsing string and converting to json
14 import json
15 #Random part selection
16 import random
17
18 #Info for testing:
19 #The sketchHouse demo needs 31 iterations to finish -> range(1, 32)
20 #The stickman demo needs 48 iterations to finish -> range(1, 49)
21
22 class Blueprint(PluginInterface):
23
24     #Init loads the ontology
25     def __init__(self):
26         #Three ontologies modeling drawings of objects are bundled with the
                   application: two variants of a house and a stickman
27         current_dir_path = os.path.dirname(os.path.realpath(__file__))
28         self.directory_path = os.path.join(current_dir_path, "products")
29
30         #Create a random state with a specific seed
31         #For reproducing results enter a seed
32         self.random_state = random.Random()
33
34     #The initial state of the plugin is determined by the parts of the
               product,
35     #which can be assembled without any preconditions
36     #In the examples it will be the foundation of the house or the feet of the
               stickman
37     def setup(self):
38
39         print("Setting up Telepath Blueprints plug-in...")
40
41         #Allowing the user to decide, which product to built
42         product, path = self.product_selection()
43
44         #Querying the ontology for all parts without preconditions
45         parts_list = product.hasPart
46         #So far the PartWithoutPrecondition class does not seem to properly
                   work with the reasoner
```

```
47          #Instead we will search for the parts with conditions and remove them
                 from the parts list
48          parts_with_precondition =
                 self.ontology.PartWithPrecondition.instances()
49          parts_without_precondition = list()
50          for part in parts_list:
51              if part not in parts_with_precondition:
52                  parts_without_precondition.append(part)
53
54          #Even if there are several elements in the list,
55          #the initial setup will always select the first part to generate the
                initial state
56          first_part = parts_without_precondition[0]
57          state = dict()
58          #State information
59          state["Product"] = product.name #Product name for later checking
60          state["Current part"] = first_part.name #Name of current part
61          state["Finished actions"] = list() #Only instructions of the current
                 part
62          state["Finished parts"] = list() #Finished part to avoid repeating
                 steps
63          state["Ontology path"] = path
64
65          print("Telepath Blueprints active.")
66
67          return state
68
69      #This method selects the next building step by querying the ontology
70      #It then returns a robot command and the updated application state
71      def run(self, state: dict):
72
73          #Loading ontology
74          path = state["Ontology path"]
75          self.ontology = get_ontology(path).load()
76          print("Ontology can be accessed.")
77
78          #Syncing reasoner
79          with self.ontology:
80              sync_reasoner()
81          print("Reasoner active.")
82
83          current_product = None
84          #Get the product instance from the ontology
85          for product in self.ontology.Product.instances():
86              if product.name == state["Product"]:
87                  current_product = product
88
89          #Raise error if no product is found
90          if current_product == None:
91              raise Exception("No matching product found in the ontology!")
92
93          #Decision about the active part
94
95          #If the current part has been marked as finished in a previous step
```

```
96          #We will get the next one from the ontology
97          current_part = None
98          if state["Current part"] in state["Finished parts"]:
99              current_part = self.get_next_part(current_product, state)
100             #Returns None if all parts have been assembled
101             #The plugin then returns an exit command and a final state
102             if current_part is None:
103
104                 #Creating final state
105                 final_state = dict()
106                 final_state["Product"] = current_product.name
107                 final_state["Status"] = "Finished"
108                 final_state["Parts"] = state["Finished parts"]
109
110                 exit_command = CommandEXIT()
111                 return exit_command, final_state
112
113         #Else we will directly grab the part instance
114         else:
115             #Get the current part
116             for part in current_product.hasPart:
117                 if  part.name == state["Current part"]:
118                     current_part = part
119
120         #Raise error, if no part is found
121         if current_part == None:
122             raise Exception("No matching part found in the ontology!")
123
124         #Get the correct action
125         current_action = self.get_next_action(current_part, current_product,
                      state["Finished actions"])
126
127         #Extracting the command
128         command = self.generate_command(current_action)
129
130         #Creating a new state variable
131         new_state = self.generate_state(state, current_product,
                      current_action, current_part)
132
133         return command, new_state
134
135     #Returns the instance of the product to built after user selection
136     def product_selection(self):
137
138         #Getting all of the ontology rdf files
139         files = dict()
140         for dirpath, dirnames, filenames in os.walk(self.directory_path):
141             for filename in filenames:
142                 if filename.endswith('.rdf'):
143                     full_path = os.path.join(dirpath, filename)
144                     files[filename] = full_path
145
146         #Prints the filenames
147         print("The following ontologies are available: ")
```

```
148         for key in files.keys():
149             #Removing the file ending
150             print(key[:-4])
151 
152         #Allowing ontology selection
153         user_input = input("Please select an ontology: ")
154         allowed_input = False
155         while allowed_input == False:
156             try:
157                 ontology_path = files[user_input + ".rdf"]
158             except KeyError:
159                 user_input = input("Wrong input. Please try again: ")
160             else:
161                 allowed_input = True
162                 print("Ontology selection succesful.")
163 
164         #Loading ontology
165         self.ontology = get_ontology(ontology_path).load()
166         print("Ontology can be accessed.")
167 
168         #Syncing reasoner
169         with self.ontology:
170             sync_reasoner()
171         print("Reasoner active.")
172 
173         #Access the product class and its instances
174         product_class = self.ontology.Product
175         product_instances = product_class.instances()
176         available_products = dict()
177 
178         #Prints the names of the instances
179         print("The following blueprints are available: ")
180         for instance in product_instances:
181             print(instance.name)
182             available_products[instance.name] = instance
183 
184         #Allowing and checking user input
185         user_input = input("Please select the product to be built: ")
186         allowed_input = False
187         while allowed_input == False:
188             try:
189                 product_to_build = available_products[user_input]
190             except KeyError:
191                 user_input = input("Wrong input. Please try again: ")
192             else:
193                 allowed_input = True
194                 print("Product selection succesful.")
195 
196         return product_to_build, ontology_path
197 
198 
199 
200     #Extracts ontology information like frame and command type and returns a
            formatted command
```

```
201     def generate_command(self, action: object):
202         type = action.hasCommand
203         command = dict()
204 
205         if type == "LOG":
206             command = CommandLOG()
207 
208         elif type == "EXIT":
209             command = CommandEXIT()
210 
211         elif type == "SEND":
212             command = CommandSEND()
213 
214         elif type == "PTP":
215             #HasFrame is a string, which must be formatted
216             json_string = action.hasFrame
217             frame = json.loads(json_string)
218             command = CommandPTP(frame)
219 
220         elif type == "LIN":
221             json_string = action.hasFrame
222             frame = json.loads(json_string)
223             command = CommandLIN(frame)
224 
225         elif type == "CIRC":
226             aux_string = action.hasAuxiliaryFrame
227             dest_string = action.hasDestination
228             auxiliaryFrame = json.loads(aux_string)
229             destination = json.loads(dest_string)
230             command = CommandCIRC(auxiliaryFrame, destination)
231 
232         return command
233 
234     #Generates a new state
235     def generate_state(self, old_state: dict, current_product: object,
            current_action: object, current_part: object):
236 
237         #Assigning new state variable
238         new_state = dict(old_state)
239 
240         #First step: Overwrite the current part and delete the action list
            (for readability)
241         if current_part.name != new_state["Current part"]:
242             new_state["Current part"] = current_part.name
243             new_state["Finished actions"] = list()
244 
245         #Second step: Add the current action to the list of finished actions
246         if current_action.name not in new_state["Finished actions"]:
247             new_state["Finished actions"].append(current_action.name)
248         #Raises an error, in case the current action is already part of the
            state variable
249         else:
250             raise Exception(f"An error has occurred. Step
                    {current_action.name} might have been repeated!")
```

```
251
252         #Third step: Get the correct instructions for the action
253         current_instruction = None
254         for instruction in current_part.hasInstruction:
255             if instruction.hasProductConstraint == current_product:
256                 current_instruction = instruction
257         if instruction is None:
258             raise Exception(f"Modelling error: Part {current_part.name} has no
                    instructions for the product {current_product.name}!")
259
260         #Fourth step: Check wether all actions of this part are completed
261         counter = 0
262         for action in current_instruction.hasAction:
263             if action.name not in new_state["Finished actions"]:
264                 counter += 1
265
266         #If this is true the current part will be added as well
267         if counter == 0:
268             new_state["Finished parts"].append(current_part.name)
269         #In the next run the system will then automatically select the next
                part
270
271         return new_state
272
273
274     #Based on the provided part and a list of finished actions, the correct
            next action will be returned
275     def get_next_action(self, part: object, product: object, finished_actions:
            list):
276
277         #Get the correct instructions for this product
278         for instruction in part.hasInstruction:
279             if instruction.hasProductConstraint == product:
280                 current_instruction = instruction
281
282         #Get a list of the instruction actions
283         list_of_actions = current_instruction.hasAction
284         #Check for the first action: the one with no precondition
285         #Actions must be modeled deterministically in the ontology: One action
                only leads to one other action
286         #So there is a clear order with only one starting action
287         actions_with_precondition =
                self.ontology.ActionWithPrecondition.instances()
288         actions_without_precondition = list()
289         for action in list_of_actions:
290             if action not in actions_with_precondition:
291                 actions_without_precondition.append(action)
292         #Raise Exception if several actions would be available
293         if len(actions_without_precondition) != 1:
294             raise Exception("Modelling Error: There can only be one action
                    without conditions per instruction!")
295
296         #Determine the current action
297         first_action = actions_without_precondition[0]
```

```
298
299         #If it has not been finished yet, the current action will always be
                the one without conditions
300         current_action = first_action
301
302         #Checking the action chain modeled in the ontology,
303         #Until an unfinished action is found to overwrite the variable
304
305         while current_action.hasEnabledAction is not None:
306             if current_action.name in finished_actions:
307                 next_action = current_action.hasEnabledAction
308                 current_action = next_action
309             else:
310                 break
311
312         return current_action
313
314     #Returns the next possible part from the ontology
315     def get_next_part(self, product: object, state: dict):
316
317         #Preparation
318
319         #Get list of all product parts
320         part_list = product.hasPart
321
322         #Creates a list of unfinished parts for reference
323         unfinished_parts = list()
324         for part in part_list:
325             if part.name not in state["Finished parts"]:
                    unfinished_parts.append(part)
326
327         #Searches for all parts without preconditions
328         parts_with_precondition =
329              self.ontology.PartWithPrecondition.instances()
330         parts_without_precondition = list()
331         for part in part_list:
332             if part not in parts_with_precondition:
333                 parts_without_precondition.append(part)
334
335         #Three stage search algorithm
336         possible_parts = list()
337         stage_two_list = list()
338
339         #Stage I
340         #Parts without any condition are checked first
341         #If one has not been finished yet, it will be added to the part list
342         for part_without_precondition in parts_without_precondition:
343             if part_without_precondition in unfinished_parts:
344                 possible_parts.append(part_without_precondition)
345
346         #Stage II
347         #If the list remins empty, we will check the parts,
348         #Which only have elements from parts_without_conditions marked as
                 necessary parts
```

```
349         if len(possible_parts) == 0:
350             #Part in parts
351             for part_without_precondition in parts_without_precondition:
352                 enabled_part_list = part_without_precondition.hasEnabledPart
353                 for enabled_part in enabled_part_list:
354                     #Checks every enabled part for its requirements
355                     condition_check = 0
356                     condition_list = enabled_part.hasNecessaryPart
357                     for condition in condition_list:
358                         if condition not in parts_without_precondition:
359                             condition_check += 1
360                     #If the requirements are exclusively parts from Stage I,
                            the part is taken into consideration
361                     if condition_check == 0 and enabled_part in
                            unfinished_parts:
362                         possible_parts.append(enabled_part)
363                     #If it is unfinished, but does not meet this requirement,
                            it will be checked in Stage 3
364                     if condition_check == 0:
365                         stage_two_list.append(enabled_part)
366
367         #Stage III
368         #If the list still remains empty, we will check the rest of the parts
369         if len(possible_parts) == 0:
370             remaining_parts_list = part_list
371             #Remove parts of stage I
372             for part in parts_without_precondition:
373                 remaining_parts_list.remove(part)
374             #Remove results of stage II
375             for part in stage_two_list:
376                 remaining_parts_list.remove(part)
377             #Calling separate method to find possible parts
378             search_results = self.deep_search(remaining_parts_list,
                        unfinished_parts)
379             for result in search_results:
380                 possible_parts.append(result)
381
382         #If no part has been found, no action will be returned
383         #This will lead to an Exit Command
384         current_part = None
385         #If parts have been found, one will be randomly selected
386         #Randomness will help to demonstrate the non-scripted nature of the
                application
387         if len(possible_parts) != 0:
388             #Random selection
389             current_part = self.random_selection(possible_parts)
390
391         return current_part
392
393     #Part of stage III of the search algorithm
394     #Check whether remaining parts have fulfilled conditions and are not
            finished yet
395     #If true, returns them as search results
```

```
396     def deep_search(self, remaining_parts: list, unfinished_parts: list):
397
398         search_results = list()
399
400         #Checking if conditions of these parts are met
401         for part in remaining_parts:
402             unfulfilled_conditions = 0
403             for condition in part.hasNecessaryPart:
404                 if condition in unfinished_parts:
405                     unfulfilled_conditions += 1
406             #If all of the part's conditions are met and it is not finished,
407             #Add it to the search results
408             if unfulfilled_conditions == 0:
409                 if part in unfinished_parts:
410                     search_results.append(part)
411
412         return search_results
413
414     #Returns a random element from a list
415     def random_selection(self, part_list: list):
416
417         random_element = self.random_state.choice(part_list)
418
419         return random_element
```