Hochschule für Technik und Wirtschaft Dresden

Fakultät Informatik/Mathematik

Diplomarbeit

im Studiengang Medieninformatik

Thema: Integration eines Sensorfußbodens und eines smarten Pflegebetts in openHAB 3

eingereicht von: Yonatan Pepper

eingereicht am: 15.05.2024

Erster Betreuer: Prof. PD Dr.-Ing. habil. Hans-Joachim Böhme
Zweiter Betreuer: Dipl.-Inf. (FH) Mathias Klingner

Zusammenfassung

Die Diplomarbeit befasst sich mit der Integration von Technologien zur Unterstützung einer Ambient Assisted Living Umgebung, speziell durch die Anbindung eines Sensorfußbodens und eines smarten Pflegebetts in den open Home Automation Bus (openHAB) 3.

Basierend auf dieser Integration ist es das Ziel der Arbeit ein automatisiertes Beleuchtungssystem zu realisieren, das insbesondere nachts die Sicherheit und Komfort der pflegebedürftigen Person verbessert.

Die Arbeit basiert auf dem Konzept des Ambient Assisted Living, das darauf abzielt älteren-, pflegebedürftigen Personen ein selbstständiges Leben zu ermöglichen, indem technologische Unterstützung in deren häuslichen Umgebung integriert wird.

Praktische Tests des Systems evaluieren dessen Zuverlässigkeit und unterstreichen Schwachstellen der genutzten Technologien. Die Arbeit bietet einen Grundbaustein für die Integration externer Geräte in den *openHAB* und demonstriert die Möglichkeit einzelne Ambient Assisted Living Technologien zu komplexen cyber-physikalischen Systemen zu kombinieren. Zukünftige Forschungen könnten sich darauf konzentrieren, die Interoperabilität zwischen verschiedenen Geräten und Systemen zu verbessern und die Benutzerfreundlichkeit zu erhöhen, um eine breitere Akzeptanz und effektivere Nutzung in alltäglichen Situationen zu ermöglichen.

Inhaltsverzeichnis

1 Einleitung 1

2 Theoretischer Hintergrund 4
2.1 AAL-Living Lab Cultus Bühlau 4
2.2 Ableitung der Forschungsfrage 5
2.3 Definition von relevanten Begriffen 6
2.3.1 Ambient Assisted Living 6
2.3.2 Smart Home 8
2.4 openHAB 8
2.4.1 Beschreibung der Architektur 8
2.4.2 Weitere Komponenten 11
2.4.3 Detaillierte Untersuchung der Kommunikationen mit Things und Items 12
2.5 Future-Shape - SensFloor 13
2.5.1 Aufbau des SensFloor 13
2.5.2 Alternative Arten der Bewegungsermittlung 14
2.5.3 Schnittstellen und Datenübertragung 14
2.5.4 Event-Typen 15
2.5.5 Verbindungsaufbau zu der SensFloor-API 17
2.5.6 Potentielle Fehlerquellen 18
2.5.7 Alternative Systeme zum SensFloor 19
2.6 PhysioNova - RotoFlex 19
2.6.1 Integrierte Wiegetechnik von A.S.T. 19
2.6.2 Verbindungsaufbau zu der RotoFlex-API 20

3 Methodik 21
3.1 Vorhandene Architektur im AAL-Living Lab Cultus Bühlau 21
3.2 Untersuchung möglicher Vorgehensweisen für die Anbindung von SensFloor an openHAB 22
3.2.1 Event-Typen der SensFloor-API 22
3.3 Analyse der Arbeitsschritte 25
3.4 Integrationsansätze 26
3.4.1 Integrationsansatz als Thing mit Exec Binding 28
3.4.2 Integrationsansatz als Thing mit Custom Binding 29
3.4.3 Integrationsansatz als Item über REST-API 32
3.4.4 Schlussfolgerung zu den Integrationsmethoden 35
3.5 Detaillierte Analyse der Integration über REST-API 36
3.5.1 Aufteilung in Arbeitsschritte 36

3.5.2 Struktur der SensFloor-Daten . . . 36
3.5.3 Struktur der RotoFlex-Daten . . . 37
3.6 Praktische Analysen . . . 38
3.6.1 Analyse der Datenintegrität von SensFloor . . . 38
3.6.2 Testing der openHAB Rules . . . 40
3.7 Nächtliches Beleuchtungssystem . . . 43
3.7.1 Grundkonzept . . . 43
3.7.2 Ähnliche Systeme . . . 44
3.7.3 Feststellung von Nachteintritt . . . 45
3.7.4 Steuerung der LED-Streifen . . . 45
3.8 Programmbeschreibung . . . 46
3.8.1 Item-Erstellung . . . 46
3.8.2 Main-Funktion . . . 47
3.8.3 Logger . . . 50
3.8.4 RotoFlex-Verbindung . . . 50
3.8.5 SensFloor-Verbindung . . . 53
3.8.6 OpenHABConnectionHandler . . . 56
3.8.7 Sturzerkennung . . . 61
3.8.8 Config . . . 62
3.8.9 Zusammenfassung der Programmbeschreibung . . . 64
3.8.10 Gestaltung der Benutzeroberfläche . . . 65
3.8.11 Langzeittests . . . 66
3.8.12 Korrektur Ghost-Erkennung . . . 68
3.8.13 Konfiguration der Ghost-Erkennung . . . 76
3.8.14 Inbetriebnahme . . . 76
3.8.15 Einrichtung des Systems in anderen Umgebungen . . . 77
3.9 Ergebnisse . . . 78
3.9.1 Skalierbarkeit . . . 78
3.9.2 Bewertung der RotoFlex-Anbindung . . . 78
3.9.3 Bewertung der SensFloor-Anbindung . . . 79

4 Schlussbemerkung 81
4.1 Diskussion . . . 81
4.1.1 Auswertung des SensFloor-Systems . . . 81
4.1.2 Auswertung der openHAB-Anbindung . . . 82
4.2 Fazit . . . 83

A Erweiterte Codeausschnitte 90

Abkürzungsverzeichnis

AAL Ambient Assited Living

AES Advanced Encryption Standard

API Application Programming Interface

A.S.T. Angewandte System Technik

ASYNC Asynchron

cURL client Uniform Resource Locator

EvAAL Evaluating Ambient Assited Living

HSV Hue, Saturation, Value

HTTP Hypertext Transfer Protocol

ID Identifikation

JSON JavaScript Object Notation

LED Light Emitting Diodes

OAuth Open Authorization

openHAB open Home Automation Bus

OSGi Open Services Gateway initiative

PIR Passiv-Infrarot-Sensor

PVC Polyvinylchlorid

REST Representational State Transfer

RGB Red, Green, Blue

SCCA Smart Co-Care Apartment

SSH Secure Shell

SYNC Synchron

TCP Transmission Control Protocol

UI User Interface

URL Uniform Resource Locator

WLAN Wireless Local Area Network

XML Extensible Markup Language

Abbildungsverzeichnis

2.1 openHAB Architektur 9

3.1 Zustandsdiagramm am Beispiel von drei Räumen 23
3.2 Darstellung der Beziehungen zwischen den Komponenten bei der Integration als *Thing* 26
3.3 Darstellung der Beziehungen zwischen den Komponenten bei der Integration als *Item Group* 27
3.4 Darstellung der Beziehungen zwischen den Komponenten bei der Integration als einzelnes *Item* 27
3.5 Datenübermittlung von Empfang bis Weiterleitung 37
3.6 Grundriss der Laborwohnung auf ein Koordinatensystem projiziert 40
3.7 openHAB *Rule* Ansicht 40
3.8 Änderung der Identifikation (ID) von Objekten bei kurzzeitigem Verschwinden 67
3.9 Distanzberechnung zwischen zwei Punkten (Objekten) mittels Hypotenusenbestimmmung 74
3.10 Verzeichnisbaum des Projekts 76

Tabellenverzeichnis

2.1 Übersicht einiger von *openHAB* 3 unterstützen *Item*-Typen und Befehle . . 12

3.1 *Item*s der Light Emitting Diodes (LED)-Streifen im *openHAB* 43

1 Einleitung

Um die Relevanz der Forschungsarbeit zu erkennen, ist es wichtig, den Status Quo zu erläutern. Die deutsche Bevölkerung erlebt einen Wandel ihrer demografischen Struktur, der die zunehmend alternde Gesellschaft vor neue Herausforderungen stellt. Die aus medizinischem Fortschritt und verbesserten Lebensbedingungen steigende Lebenserwartung ist maßgeblich verantwortlich für eine deutliche Zunahme der Zahl der älteren- und häufig pflegebedürftigen Bevölkerung (Destatis (2023b); Radtke (2023)).

Mit zunehmendem Alter steigt die Anzahl an pflegebedürftigen Menschen aufgrund von gesundheitlichen Einschränkungen im Alltag, die das Sehen, Hören, den Bewegungsapparat und die kognitiven Fähigkeiten betreffen (Walter et al. (2006)). Dabei steigt ab einem Alter von 75 Jahren der Anteil an schwer-behinderten Personen stark an (Destatis (2013)), die ihren Alltag nicht mehr alleine bewältigen können und auf Hilfe angewiesen sind. Andererseits tendieren Frauen heutzutage zur bedeutend späteren Geburt und insgesamt ist die Zahl der Geburten pro Frau niedriger, was langfristig zu einem potenziell Arbeitskräftemangel und wirtschaftlichen Auswirkungen führen kann (Destatis (2023a)).

Dieser Wandel der Demografie sorgt für eine Verdopplung der Zahl an pflegebedürftigen Personen bis 2050 und mehr Ein-Personen Haushalte ü70 (Destatis (2010)). In den Medien wird jetzt schon vermehrt von einer 'Pflegekrise' gesprochen, da sich die Anzahl der Pflegebedürftigen kontinuierlich erhöht, während die Anzahl von Pflegekräften sinkt (Vogler et al. (2023)).

Menschen, die in ihrem Alltag auf Hilfe angewiesen sind, können sowohl in einer betreuten Wohneinrichtung, als auch in ihren eigenen vier Wänden versorgt werden. Laut Angaben von Destatis (2010) werden bereits 70% der Pflegebedürftigen Zuhause versorgt. Diese Art der Versorgung basiert jedoch auf der Annahme eines angemessen eingerichteten und ggf. barrierefreien Haushaltes und schließt eine dauerhafte Überwachung und Sicherstellung von Hifsmöglichkeiten aus. Dennoch äußern pflegebedürftige Personen höheren Alters vermehrt den Wunsch, in ihrem eigenen Heim wohnhaft zu bleiben. (Hedtke-Becker et al. (2012); Strube (2011))

Eine mögliche Lösung für die unvermeidlich bevorstehenden Herausforderungen ist Ambient Assited Living (AAL), ein Thema, das im letzten Jahrzehnt durch den Zuwachs an intelligenter Technologie in Heimautomatisierung, Bildung, Rehabilitation und Unterstützung immer präsenter geworden ist (Ramkumar et al. (2019)). Unter der Bezeichnung 'Ambient Assisted Living' werden auf nationaler und internationaler Ebene verschiedene Forschungs- und Entwicklungsprojekte gefördert, mit dem Ziel älteren und hilfsbedürftigen Menschen ein langes und selbstbestimmtes Leben zu ermöglichen (BMBF (2008)).

Eine Befragung zu digitalen Kompetenzen von Älteren ermittelt die selbst geschätzte Sicherheit im Internet beim Umgang mit Geräten wie Smartphone, Tablet und PC, nach Altersgruppe. Die Umfrage ergab dabei, dass lediglich 39% der befragten Personen über 60 Jahre ihre Fähigkeiten mit 'sehr sicher' oder 'eher sicher' beurteilen, während 55% sich 'eher unsicher' oder 'völlig unsicher' fühlen (Bertelsmann Stiftung (2020)). Aus einer Umfrage zur Nutzung des Internets bei Personen ab 60 Jahren in Deutschland im Jahr 2023 kommt hervor, dass fast ein Drittel aller Umfrageteilnehmer keine Internetnutzer sind. (Statista Research Department (2023))

Diese statistischen Daten werfen die Frage auf, inwieweit ältere Menschen AAL als akzeptable Technologie betrachten würden. Aufgrund der hohen Unsicherheitsquoten mit dem Umgang von modernen Technologien, ist es von Bedeutung zu untersuchen, ob die positiven Aspekte von AAL, wie die Erhaltung der Selbstständigkeit und die Verbesserung der Lebensqualität, von älteren Menschen tatsächlich begrüßt werden.

Eine Befragung von Senior:innen im Landkreis Görlitz hat eine hohe Akzeptanz von AAL ergeben. *'Es zeigte sich, dass besonders Technologien, die die Sicherheit erhöhen, von den Befragten gewünscht werden. Aber auch der Komfort spielt eine nicht unerhebliche Rolle.'* (Preißler et al. (2016)). Die Umfrage ergab ebenfalls, dass Hilfssysteme, wie u.a. automatische Flurbeleuchtung, eine akzeptierte Technologie darstellen. Als Gründe für eine Ablehnung von AAL-Technologien wurden hauptsächlich hohe Anschaffungs- und Einrichtungskosten sowie eine schlichte Nicht-Notwendigkeit genannt. *'[Probanden antworteten], dass hierin kein Bedarf bestehe, z. B. weil bei nachlassender Gesundheit ein Umzug in eine betreute Wohneinrichtung oder in ein Pflegeheim geplant sei.'* (Preißler et al. (2016))

Insgesamt lässt sich aus der Befragung jedoch deutlich entnehmen, dass AAL bereits jetzt eine hohe Akzeptanzquote hat und somit potentiell in vielen Wohnungen zum Einsatz kommen wird. Die Befragung bestätigt die Annahme, dass Senior:innen ihre Selbstständigkeit langfristig erhalten möchten, indem sie den Wohnkomfort und die Sicherheit in ihrem eigenen Heim erhöhen.

Unter den begründeten Aspekten beschäftigt sich auch das Smart Co-Care Apartment (SCCA) mit der Einrichtung und Entwicklung einer automatisierten Wohnung für pflegebedürftige Personen, mit dem Ziel, ihr autonomes Leben zu verlängern und gleichzeitig die Altenpflege zu entlasten. Prototypisch dafür ist eine Wohnung in Dresden Bühlau eingerichtet worden. Diese soll den Bewohner unter anderem bei Alltagsaufgaben unterstützen, die Steuerung von Klima und Beleuchtung automatisieren und sich dabei an das Verhalten und den individuellen Tagesablauf anpassen. Zugleich werden Verhaltensdaten gesammelt, deren Auswertung die Wohnung maßgeschneidert an die Bedürfnisse der verschiedenen Benutzer anpassen soll. Außerdem werden medizinische Messwerte genutzt, um eine häusliche medizinische Überwachung zu ermöglichen.

Ein essentieller Teil der zu integrierenden Technologie ist ein Sensor-Fußboden, der in der Lage ist, Bewegungsdaten und Sturzevents in der Wohnung zu erfassen und weiter zu kommunizieren. Zusätzlich existiert in der Wohnung ein Pflegebett mit integrierter Wiegemesstechnik. welches dazu genutzt werden kann, Gewicht, Position und Bewegung einer Person im Bett zu erkennen. Besonders relevant wird das Projekt durch die geplante Anbindung der einzelnen Komponenten an das *openHAB* 3 System. Ziel dieser Integration ist eine zentrale Steuerung und Überwachung unterschiedlicher Technologien, die durch eigene, teilweise inkompatiblen, Schnittstellen ansonsten nicht interoperabel wären. Dies spielt eine entscheidende Rolle, da eine reibungslose Kommunikation zwischen den diversen Geräten eine erhebliche Verbesserung der Gesamtleistung des cyber-physischen Systems SCCA bewirken kann.

2 Theoretischer Hintergrund

2.1 AAL-Living Lab Cultus Bühlau

Die Initiative 'AAL-Living Lab Cultus Bühlau' ist eine Kooperation der Hochschule für Technik und Wirtschaft Dresden und der Cultus gGmbH Dresden, eine Organisation, die sich mit der Betreuung, Pflege und Rehabilitation pflegebedürftiger Menschen befasst. Ziel des Projektes ist es, die Lebensqualität und Autonomie älterer Menschen durch den Einsatz verschiedener Technologien aufzuwerten. Das Projekt widmet sich der Herausforderung, cyber-physische Systeme in bestehende betreute Wohnkomplexe zu integrieren (Klingner et al. (2022)).

Das Projekt fand seinen Ursprung im Jahr 2019, während die Wohnanlage umfangreich saniert wurde. Die Zusammenarbeit mit der Cultus gGmbH Dresden ermöglichte die Integration von AAL-Technologien im Rahmen der Sanierung, was sowohl Herausforderungen als auch Chancen für die Umsetzung mit sich brachte (Klinger et al. (2020a)). Beteiligt am Projektstart waren unter anderem Vertreter der Cultus gGmbH Dresden, die Unternehmen Cibek und Future-Shape sowie verschiedene Gewerke (Klinger et al. (2020b)). Finanziert wird die Forschung der digitalen Transformation der Pflege Umgebungen durch KI und Robotik unter anderem durch European Regional Development Funds und Steuermittel des Sächsischen Landtags (Klingner et al. (2022)).

Bei dem betrachteten Objekt des Projektes handelt es sich um eine Wohnanlage in Dresden Bühlau, die seit 1929 ein Stiftungsheim für pensionierte Staatsbeamte ist. Insgesamt umfasst die Wohnanlage 7 Häuser und beherbergt 58 nicht- bis geringfügig pflegebedürftige Bewohner in 2-Zimmerwohnungen. Das AAL-Labor wurde in einer 37 m^2 großen 2-Zimmer-Wohnung im Erdgeschoss des Haus C eingerichtet, wobei fast vollständige Barrierefreiheit durch verschiedene bauliche Maßnahmen gewährleistet wurde (Klinger et al. (2020a); Klinger et al. (2020b)).

Der Einbau des *Future-Shape SensFloor* und weiterer AAL-Technologien wie einer barrierefreien Küche und eines Kleiderschranks gestalteten sich aufgrund der zugleich stattfindenden Sanierungsmaßnahmen anspruchsvoll. Insbesondere die koordinierte Installation des *SensFloor* von *Future-Shape* und des *Paul-Systems* von *Cibek* sowie die Anpassungen für die barrierefreie Möblierung erforderten eine präzise Planung (Klinger et al. (2020b)).

Bei der Fortsetzung des Projektes im Jahr 2020 lag der Fokus auf dem technologischen Ausbau des AAL-Labors. Unter anderem wurde eine umfassende IT-Infrastruktur geschaffen, welche aus einem zentralen Server und Raspberry Pi Clients besteht, um die Informationsverarbeitung, Machine Learning Prozesse und Steuerungsaufgaben zu ermöglichen (Klinger et al. (2020b); Klinger et al. (2021)). Außerdem vervollständigen Sensoren für

verschiedene Lebensbereiche, ein flexibles, digitales Beleuchtungssystem, sowie eine barrierefreie Einrichtung, das Portfolio der Technologien. Unter anderem wurden Sensoren für Türen, Fenster, den Müll, zur Temperatur- und Licht-Erkennung, für die TV-Steuerung verbaut (Klinger et al. (2020b); Klinger et al. (2021)).

Für das Beleuchtungssystem wurden *Philips Hue* Leuchtmittel mit individueller Konfiguration und LED Nachtlichter für die indirekte Beleuchtung eingebaut. Die Barrierefreie Einrichtung wird durch höhenverstellbare Einrichtungen in Küche, Schlafzimmer und Bad, einem *Rotoflex*-Pflegebett und *VIANDOpflege* Pflegesessel, sowie Möbel von *Mormor Aktivmöbel* für Sicherheit und Belastbarkeit gewährleistet (Klinger et al. (2020b); Klinger et al. (2021)). Auch eine virtuelle Tour für eine anschauliche Präsentation der Musterwohnung wurde ermöglicht (Klinger et al. (2020b); Klinger et al. (2021)). Der technologische Ausbau des AAL-Labors in Dresden Bühlau kann demzufolge als erfolgreich gewertet werden und es wurde eine einzigartige Forschungsumgebung für Ambient Assisted Living in Deutschland geschaffen.

Das AAL-Living Lab Cultus Bühlau zeichnet sich durch einen hohen Grad an Barrierefreiheit und Autonomie aus, der durch die installierten Technologien erreicht wurde. Mit dem Projekt wurde erreicht, die Technologie der Zielgruppe näher zu bringen. Die positive Resonanz der Bewohner, erweckt Interesse an der Möglichkeit des Probewohnens (Klinger et al. (2021)). Die zukunftsrelevanten Perspektiven des Projektes werden durch die geplante Erweiterung des Labors mit weiteren ortsveränderlichen Technologien und der Antrag für ein Folgeprojekt zur umfassenden Ausstattung der Wohnung verdeutlicht.

2.2 Ableitung der Forschungsfrage

Die Forschungsfrage dieser Arbeit zielt darauf ab zu untersuchen, wie ein Sensorfußboden und ein smartes Pflegebett in die bestehende Smart Home Umgebung integriert werden können, um in Echtzeit die Aktivität der Bewohner darzustellen. Anschließend soll ein intelligentes Beleuchtungssystem aufgebaut werden, welches auf die Aktivität der Bewohner reagiert. Dabei müssen die einzelnen Systeme untersucht werden und eine geeignete Anbindungsmethode gewählt werden.

Es gibt einige Parameter, die bei der Forschung beachtet werden müssen. Ziel ist es nicht nur eine Kommunikation zwischen den Geräten und *openHAB* herzustellen, sondern auch einen Lösung zu entwickeln, die auf zukünftige Integrationen anwendbar ist. Außerdem ist es wichtig zu klären, wie die Benutzeroberfläche von *openHAB* gestaltet werden kann, um den Bewohnenden eine einfache Interaktion mit dem System zu ermöglichen.

2.3 Definition von relevanten Begriffen

2.3.1 Ambient Assisted Living

AAL bezeichnet jegliche Methoden, Konzepte, Produkte und Dienstleistungen, Informations- und Kommunikationstechnologie um die Lebensqualität von hilfsbedürftigen Menschen, zu erhöhen und zu sichern (A. Braun et al. (2016); Georgieff (2008)). Die Forschung an AAL Lösungen ist die Reaktion auf den demografischen Wandel. Deutschland erlebt seit Jahren eine sinkende Geburtenrate, aber deutlich steigende Lebenserwartung, was zu einem drastisch erhöhten Anteil älterer Personen führt (Destatis (2009)).

Die Bezeichnung Ambient Assisted Living entstand 2004 im Rahmen europäischer Forschungsrahmenprogramme, an denen Deutschland von Anfang an beteiligt war (A. Braun et al. (2016)). Dass der Verband der Elektrotechnik Elektronik Informationstechnik e.V. jährlich einen Kongress dazu veranstaltet, bestätigt das wachsende Interesse der Bevölkerung an zukunftsorientierten AAL Lösungen.

Becks et al. (2007) beschreiben folgendes: *„Ambient Assisted Living bedeutet Leben in einer durch 'Intelligente' Technik unterstützen Umgebung, die sensibel und anpassungsfähig auf die Anwesenheit von Menschen und Objekten reagiert und dabei dem Menschen vielfältige Dienste bietet. Ziel ist es, die persönliche Freiheit und Autonomie über die Förderung und Unterstützung der Selbstständigkeit zu erhalten, zu vergrößern und zu verlängern. Der Mensch in allen Lebenssituationen von Arbeit und Freizeit, insbesondere der allein lebende Mensch und/oder Mensch mit Behinderung ist Adressat."*

Das übergeordnete Ziel von AAL kann demnach definiert werden, als die Gewährleistung von smarten Remote- Produkten als Hilfestellung für ältere Menschen im Alltag, um diesen einen möglichst langen Aufenthalt im eigenen Heim zu ermöglichen. Dazu zählt auch die automatisierte Sammlung und Bereitstellung relevanter medizinischer- bzw. pflegerelevanter Daten an entsprechende Fachkräfte (A. Braun et al. (2016)).

Ambient Intelligence

Ambient Intelligence bezieht sich auf elektronische Systeme, die Dienstleistungen anbieten, welche sensibel und reaktionsschnell auf die Anwesenheit von Menschen reagieren und sich unauffällig in unser tägliches Umfeld integrieren (Aarts et al. (2001); Emile Aarts (2006)). Oft werden hierfür neue Technologien entwickelt und genutzt, die auf Ambient Intelligence beruhen, um die verschiedenen Anwendungen zu integrieren (Sun et al. (2009)).

Ambient Intelligence eröffnet faszinierende Perspektiven für die Mensch-Technologie-Interaktion. In einer Welt, in der unsere Umgebung immer stärker von Elektronik geprägt

ist, kann die Vorstellung einer intelligenten Umgebung, die auf unsere Anwesenheit reagiert, unser Leben erheblich bereichern. Dieser technologische Fortschritt hat das Potenzial, die Lebensqualität zu verbessern, erfordert jedoch auch eine sorgfältige Abwägung von Fragen zur Privatsphäre und Sicherheit.

Anwendungen von AAL

Der Anwendungsbereich von AAL konzentriert sich gezielt auf die Steigerung des Wohnkomforts und der Sicherheit des Bewohners. Die Unterstützung bei Alltagsaufgaben ist zweifellos der prominenteste Aspekt von AAL. Das Hauptziel besteht darin, möglichst viele Prozesse des alltäglichen Lebens zu automatisieren. Dies umfasst auch Funktionen zur Energieeinsparung, wie die Steuerung der Heizung und das Abschalten von Strom und Licht. AAL wird außerdem eingesetzt, um lebensbedrohliche Situationen, wie Unfälle oder Stürze zu erkennen, oder die Sicherheit der Person gewährleisten, indem Einbrüche erkannt und gemeldet werden (A. Braun et al. (2016)). Beispielsweise können Stürze durch proaktive Haussteuerung und die Integration von Sensoren und Sicherheitsmaßnahmen erkannt oder verhindert werden. In gefährlichen Situationen werden automatisch Notfalldienste kontaktiert (Rashidi et al. (2013)).

AAL-Systeme können Lebensgewohnheiten und Vitalwerte überwachen und Änderungen des Gesundheitszustand präventiv erkennen, was eine frühzeitige Intervention ermöglicht und das Gesundheitssystem entlastet (Ni et al. (2015); A. Braun et al. (2016)). Außerdem wird dadurch eine deutlich effektivere Telemedizin und Fernüberwachung ermöglicht, die dem Gesundheitspersonal der Senior:innen eine verbesserte Beurteilung des Gesundheitszustandes gestattet. Unter anderem erspart es der betreffenden Person unnötige Arztbesuche, da Fernkonsultationen, Medikamentenmanagement und die Fernüberwachung durch die Integration von Sensoren und Kommunikationsmöglichkeiten ersetzt werden. Dies ist insbesondere für Personen, die in abgelegenen Gebieten leben, eine große Erleichterung (Memon et al. (2014)).

Darüber hinaus bieten AAL-Systeme Menschen mit kognitiven Beeinträchtigungen wie Demenz oder Alzheimer Unterstützung in unterschiedlichen Formen. Durch Medikamentenerinnerungen, Gedächnisaufgaben anderen automatisierten Routinen kann die Unabhängigkeit der kognitiv eingeschränkten Personen gefördert werden. In Kombination mit künstlicher Intelligenz können AAL-Systeme sich individuell an die Bedürfnisse der Person anpassen und personalisierte, kognitive Unterstützung bieten (Rashidi et al. (2013)).

Zusammengefasst lässt sich schlussfolgern, dass AAL-Systeme das Potenzial haben, die Lebensqualität älterer Menschen erheblich zu verbessern. Durch eine breite Palette von personalisierbaren Anwendungsmöglichkeiten, können Senior:innen bei vielen Hindernissen des Alterns mit Hilfe der Integration von AAL unterstützt werden. Vor allem durch medizinische Überwachung, Möglichkeiten zur Telemedizin und kognitive Beihilfe kann das Altern an Ort und Stelle ermöglicht und Pflegepersonal entlastet werden.

2.3.2 Smart Home

Smart Home beschreibt das Konzept eines Wohnortes, der sich an seine Bewohner anpasst und ihnen Arbeit abnimmt. Ein wichtiger Grund dafür ist der gesteigerte Wohnkomfort. Ein automatisiertes Haus ermöglicht aber auch Energieeinsparung und unterstützt die Sicherheit. Meistens hat ein Smart Home ein zentrales Schaltsystem, einen Server, der den Ankerpunkt zwischen allen Geräten des Smart Homes darstellt. Die Aufgabe des Schaltsystems ist es, die Geräte zu steuern, deren Status zu überwachen und gegebenenfalls den Nutzer über *Events* zu informieren. Der Erfolg eines Smart Homes ist geprägt durch die minimale Eigenleistung des Bewohners. Ein Beispiel für ein erfolgreich implementiertes Smart Home ist, wenn das System selbstständig erkennen kann, wo Personen sich befinden und entsprechend das Licht schaltet (Heinle (2015)).

2.4 openHAB

Ein häufiges Problem von modernen Smart Home Lösungen sind die starken Einschränkungen, die die Ökosysteme vieler Hersteller mitbringen. Untereinander sind diese Lösungen oftmals nicht kompatibel und bieten keine Integrationsschnittstellen. Die Kopplung unterschiedlicher Systeme miteinander ist die Aufgabe des open Home Automation Bus - *openHAB*. Der *openHAB* ist ein Open Source System, entwickelt von Kai Kreuzer, der das Projekt erstmals auf Google Code am 21. Februar 2010 veröffentlichte (Kreuzer (2010)).

Dieser Abschnitt widmet sich einer umfassenden Untersuchung der Funktionsweise von *openHAB* in der für die Forschung genutzten Version 3. Der *openHAB* ist eine Plattform für die Heimautomatisierung, die eine Vielzahl von Systemen und Technologien in eine Lösung integriert. Die Informationen der Erläuterung beziehen sich auf die offizielle Dokumenation von *openHAB* (*Welcome to openHAB* (o. D.)).

Die Plattform ermöglicht eine einheitliche Steuerungsoberfläche für eine Vielzahl von smarten Geräten, unabhängig von Hersteller und Technologiestandards. *OpenHAB* stützt sich auf ein modulares Konzept, das es den Nutzern erlaubt, verschiedene Komponenten eines Smart Homes flexibel zu verbinden. Eine solche Flexibilität und Offenheit unterscheiden *openHAB* von der Konkurrenz und die Vielzahl an verfügbaren Erweiterungen machen *openHAB* zu einer zentralen Säule moderner Smart Home Systeme (Tsakalidis et al. (2023)).

2.4.1 Beschreibung der Architektur

Als zentrales Schaltsystem bietet *openHAB* individualisierbare Möglichkeiten zur Darstellung von physischen Geräten. Um die Architektur von *openHAB* zu veranschaulichen, kann diese in drei Ebenen unterteilt werden. An erster Stelle steht die physische Abstraktionsebene. Diese besitzt eine direkte Repräsentation des physischen Geräts im *openHAB* über eine direkte Anbindung (*Binding*). Mittels Kommunikationskanäle (*Channels*), leitet

die Abstraktionsebene Informationen an die Steuerungsebene. Diese stellt den Hauptteil der *openHAB*-Funktionalität dar. Die Steuerungsebene widmet sich der Konfiguration für das Verhalten des Systems und die Implementierung von Steuerungsregeln (*Rules*). Abschließend kommt die Nutzerebene, also die Ebene, von welcher der Nutzer mit dem System interagieren kann. Abbildung 2.1 veranschaulicht die Architektur.

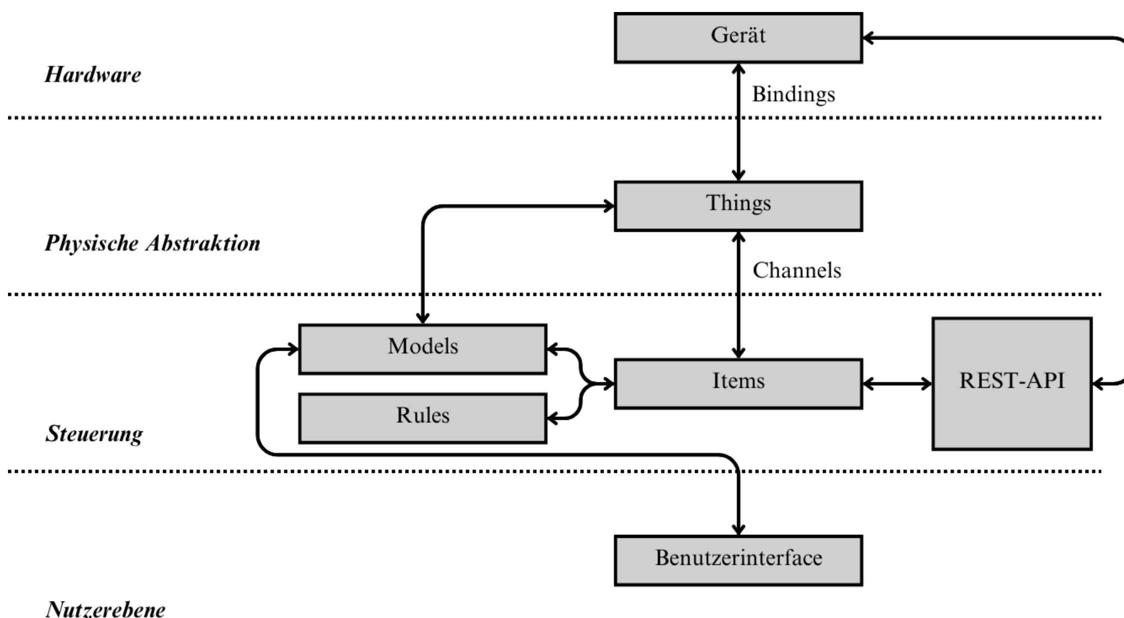

Abbildung 2.1: openHAB Architektur

Bindings

Jedes *Thing* im *openHAB* benötigt eine Anbindung an die Schnittstelle des physischen Objekts. Diese Brücke zwischen der virtuellen Darstellung des Objekts im *openHAB* (*Thing*) und des externen Systems, bzw. des Geräts, erfolgt über *Bindings*. Die Kommunikation zwischen *Thing* und externen System ist bidirektional, ermöglicht also sowohl das Empfangen von Daten und Statusänderungen von dem externen System, sowie eine Benachrichtigung an das System über mögliche Änderungen. Viele Hersteller bieten bereits eigene *Bindings* für ihre Geräte an, bzw. bietet *openHAB* für Geräte ohne vorhandenem *Binding* die Möglichkeit individuelle Datentransfers zu erstellen über eine Vielzahl von Kommunikationsprotokollen (*Developing a Binding* (o. D.)).

Things und Channels

In *openHAB* repräsentieren sogenannte *Things* physische Entitäten eines Smart Homes. Jedes *Thing* stellt unterschiedliche Funktionalitäten über *Channels* zur Verfügung, die wiederum mit *Items* verbunden werden können. Diese Verbindung ermöglicht die Steuerung der *Things* und den Zugriff auf ihre Informationen über die Benutzeroberfläche und das Regelsystem von *openHAB* (*Things* (o. D.)).

Items

Items sind Elemente in *openHAB*, die der Darstellung und Kontrolle von Informationen und Daten dienen. Es gibt unterschiedliche Typen, wie z. B. Schalter (Switch), Dimmer oder Temperaturwerte, die *Items* annehmen können. *Items* können so konfiguriert werden, dass sie Befehle akzeptieren und damit ihren Zustand verändern. Man kann *Items* in *openHAB* also auch als Variablen betrachten, die Daten kapseln und unterschiedliche Zustände annehmen können. Ein *Item* kann zu jedem Zeitpunkt nur einen Zustand haben. *openHAB* bietet die Möglichkeit, *Items* unabhängig anzulegen, gleichzeitig ist es aber auch möglich, ein *Item* als Eigenschaft eines *Things* zu nutzen. Beispielsweise kann ein *Item* die Farbtemperatur einer Glühbirne speichern, während ein anderes *Item* die Helligkeit kapselt (*Items* (o. D.)).

Rules

Rules ermöglichen die Definition von benutzerdefinierten Regeln im *openHAB*. *Rules* werden von personalisierbaren Triggern ausgelöst und führen basierend darauf einen Satz von Aktionen aus. Trigger können von *Item*- oder *Group-Events*, Zeitpunkten, System- oder *Thing*-Statusänderungen ausgelöst werden. *Rules* ermöglichen eine flexible Erstellung von Abläufen, die basierend auf Änderungen im System automatisiert ausgeführt werden (*Rules* (o. D.)).

REST

Representational State Transfer (REST) ist ein Softwarearchitekturstil, zur Gestaltung der Architektur im World Wide Web. REST definiert einheitliche Regeln zum Datentransfer. Die Grundidee von REST ist es eine einheitlichen Art des Informationszugriff zu definieren (Fielding (2000)).

RESTful Application Programming Interface (API)s sind Schnittstellen, die den Prinzipien der REST-Architektur folgen. Sie nutzen gängige Hypertext Transfer Protocol (HTTP)-Methoden, wie *GET*, *POST*, *PUT* und *DELETE*, um mit Ressourcen zu interagieren. Jede Ressource ist über eine eindeutige Uniform Resource Locator (URL) identifizierbar. Insbesondere JavaScript Object Notation (JSON) oder Extensible Markup Language (XML) werden als Datenformat der Kommunikation genutzt (Biehl (2016)).

Der *openHAB* bietet neben *Bindings* eine weitere Schnittstelle über ein REST-Interface. Prinzipiell lassen sich alle Elemente des *openHAB* über die API ansteuern, da der Großteil der Elemente aber nur lesend zugänglich ist, ist in Abbildung 2.1 nur die lesend-/schreibende Anbindung dargestellt. Die REST-API bietet somit eine Möglichkeit, externe Systeme mit eigener Steuerung an den *openHAB* anzubinden (*REST API* (o. D.)).

Models

Models definieren die Struktur des Smart Homes. Ein *Model* folgt einer Baumstruktur. Ein Beispiel für ein *Model* könnte sein, eine Wohnung in Räume aufzuteilen und jedem Raum die *Things* und *Items* zuzuordnen, die sich dort befinden. Damit bekommen die einzelnen Elemente des *openHAB* eine semantische Struktur (*Models* (o. D.)).

Main UI

Dem Nutzer wird die Möglichkeit geboten über das lokale Netzwerk ein Webinterface aufzurufen, welches neben ausführlichen Konfigurationen des Systems auch alle Statusinformationen darstellt. Dabei ist die Darstellung frei anpassbar. Es können von *Label*-Kacheln, die Textinformationen anzeigen, bis hin zu Schaltern und Webanwendungen die Seiten nach Belieben eingerichtet werden, um visuelle Rückmeldung und einfache Steuerungsmöglichkeiten zu gewährleisten, ohne, dass der Nutzer sich mit der technischen Umsetzung auskennen muss (*Main UI* (o. D.[a])).

2.4.2 Weitere Komponenten

Events

Events sind innerhalb von *openHAB* generierte Ereignisse, die über Statusänderungen und Aktionen informieren. *Events* können von *Things*, *Items* oder dem System selbst ausgelöst werden. Es gibt eine Vielzahl von unterschiedlichen *Event*-Typen in *openHAB*. Beispielsweise signalisiert ein *ItemCommandEvent*, dass ein *Item* einen neuen *Command* gesendet bekommen hat und das *ThingStatusInfoChangedEvent* informiert darüber, dass sich der Status eines *Things* geändert hat (*Events* (o. D.)).

Groups

Um *Items* zu gruppieren bietet *openHAB* die Möglichkeit *Groups* zu erstellen. Damit können mehrere *Items* in logische Einheiten kombiniert werden, welche gemeinsam überwacht und gesteuert werden können. Beispielsweise kann man alle Lampen eines Raumes gruppieren, um diese mit einem einzigen *Command* ein- und auszuschalten (*Items* (o. D.)).

Scripts

Scripts sind Unterkategorien von *Rules* und deshalb nicht in Abbildung 2.1 dargestellt. Sie kapseln kleine Programme. Im Vergleich zu *Rules* haben *Scripts* jedoch keinen Trigger und werden deshalb nicht ständig überwacht. Ein Script kann also entweder manuell ausgeführt werden, oder aus einer *Rule* heraus (*Scripts* (o. D.)).

2.4.3 Detaillierte Untersuchung der Kommunikationen mit Things und Items

Things repräsentieren in der Regel physische Geräte im *openHAB*. Die Eigenschaften der Geräte sind über *Channels* in *Items* abgebildet. Ein *Item* kann unabhängig von einem *Thing* angelegt werden und als eigene Datenkapsel dienen. Somit ist es prinzipiell möglich, für einzelne Informationen *Items* zu nutzen, ohne *Things* anzulegen. Einige Hersteller bieten *Bindings* an, die die Anbindung ihrer Geräte an das eigene *openHAB* erleichtern. Philips bietet bspw. zu den Hue-Lampen eine Hue Bridge an, welche als *Thing* im *openHAB* angelegt werden kann und über das bereitgestellte *Hue Binding* mit den Lampen kommuniziert (*Philips Binding* (o. D.)). Anschließend kann das *Thing* über *Channels* mit den passenden *Items* verbunden werden. In Tabelle 2.1 eine Auswahl der in *openHAB* 3 unterstützten *Item*-Typen aufgeführt (*Items* (o. D.)).

Tabelle 2.1: Übersicht einiger von *openHAB* 3 unterstützen *Item*-Typen und Befehle (verändert nach *Items* o. D.)

Item	**Beschreibung**	**Befehle**
Color	Farbwert	ON/OFF, Increase/Decrease, Percent, HSB, Refresh
DateTime	Zeit- und Datumswert	DateTime
Dimmer	Prozentwert	ON/OFF, Increase/Decrease, Percent
Group	Gruppierung mehrerer Items	-
Number	Dezimalzahl	Dezimalzahl, Refresh
String	Text	Text, Refresh
Switch	Binärwert für An und Aus in der Regel für Lichtschalter verwendet	ON/OFF, Refresh

Aus Tabelle 2.1 wird ersichtlich, dass *Item*-Typen im *openHAB* als Datentypen dienen und *Items* deshalb auch als Variablen betrachtet werden können.

2.5 Future-Shape - SensFloor

SensFloor ist ein Assistenzsystem des deutschen Unternehmens Future-Shape GmbH (GmbH (2024)). Grundlegend soll es mithilfe von unsichtbar unter dem Boden platzierten Sensoren Bewegungen und Positionen im Raum erfassen. Vom Bundesministerium für Bildung und Forschung als Verbundprojekt gefördert, soll das System dem demografischen Wandel und dem damit verbundenen wachsenden Pflegemangel entgegenwirken, sowie den Wunsch nach verlängerter Eigenständigkeit ermöglichen (BMBF (2008)).

Folgendes Kapitel widmet sich der Entwicklung, Funktionsweise und Anwendung des *SensFloor*. Der *SensFloor* ist das im Projekt genutzte Bodensensorsystem, welches zur Erfassung und Analyse von Bewegungen und Aktivitäten im Raum genutzt wird. Insbesondere wird der *SensFloor* im Kontext von AAL für die Gewährleistung der Sicherheit und Gesundheit und der Erleichterung im Alltag eingesetzt (Lauterbach et al. (2013)). Durch seine in diesem Kapitel beschriebenen Eigenschaften wird verdeutlicht, wieso der *SensFloor* sich für seine Anwendung in der AAL-Umgebung eignet.

2.5.1 Aufbau des SensFloor

Der *SensFloor* ist ein Bodensensorsystem, welches zur Erfassung von Bewegungen im Raum genutzt wird. Das System basiert auf kapazitiven Näherungssensoren, die als *SensFloor* Underlay bezeichnet werden. Dieses wird unter den Bodenbelag eingebettet und ist somit unsichtbar. Future-Shape gibt an, dass das Underlay unter allen gängigen Bodenbelägen funktioniert, wie z. B. Teppichboden, Laminat und Polyvinylchlorid (PVC)-Beläge (Lauterbach et al. (2013)).

Kapazitive Sensoren werden aufgrund ihrer einzigartigen Eigenschaften und Vorteile weitläufig in unterschiedlichsten Anwendungen genutzt. Sie funktionieren nach dem Prinzip der Kapazität, d.h. der Fähigkeit eines Systems, elektrische Ladung zu speichern. Kapazitive Sensoren bestehen aus zwei konduktiven Platten, die durch ein dielektrisches Material separiert werden. Wenn Spannung anliegt, formt sich zwischen den Platten ein elektrisches Feld. Die Kapazität des Sensors verändert sich basierend auf externen Faktoren, wie Druck oder Nähe eines Objektes und ermöglicht die Erkennung und Messung dieser Parameter (Hu et al. (2010)). Kapazitive Sensoren bieten einige Vorteile gegenüber anderen Sensortechnologien. So verfügen diese über eine sehr hohe Sensibilität und verbrauchen weniger Strom (J. Yang et al. (2020)). Sie können nicht-invasiv eingesetzt werden (sie erfordern keinen direkten Kontakt mit der Person) und können somit auch unter Fußbodenbelägen versteckt werden, was sie optisch ansprechend macht.

Allerdings beschreibt (Zuk et al. (2018)) die hohe Empfindlichkeit kapazitiver Sensoren gegenüber Umgebungsänderungen, da sie bspw. temperatur- und feuchtigkeitsabhängig sind und sogar andere elektrische Geräte in der Nähe Interferenzen auslösen können, die die Genauigkeit der Messwerte beeinflusst.

Der *SensFloor* ist rund um die Uhr aktiv, wodurch es essentiell ist, dass die verbauten Sensoren energiesparend sind, da es sonst zu sehr hohen Betriebskosten führt. Das kapazitive Sensorsystem, welches die Grundlage für das *SensFloor* Underlay bildet, besteht aus elektronischen Modulen, die in ein Verbundtextil in einem regelmäßigen Raster eingebettet sind. Diese Module ermöglichen die Messung der elektrischen Kapazität in räumlich diskreten definierten Bereichen auf dem Boden (Hoffmann et al. (2015)).

2.5.2 Alternative Arten der Bewegungsermittlung

Die kapazitiven Sensoren des *SensFloor* spielen eine entscheidende Rolle für die Bewegungserkennung einer oder mehrerer Personen in einem Raum. Es gibt jedoch auch andere erwähnenswerte Technologien, die für Bewegungserkennung in Frage kommen.

Eine relevante Methode zur Bewegungsermittlung beschreibt Piccardi (2004) in seinem Übersichtsartikel, in dem er das Verfahren der Hintergrundsubtraktion zur Erkennung von Bewegungsänderungen mithilfe einer statischen Kamera beschreibt. Obwohl es sich dabei um einen kamerabasierten Aufbau handelt, der von dem AAL-Living Lab Cultus Bühlau bewusst aus Datenschutzgründen abgelehnt wurde, kann die Hintergrundsubtraktion ebenfalls auf den *SensFloor* angewendet werden. Bei der Hintergrundsubtraktion wird das aktuelle Bild mit einem Referenzhintergrundbild verglichen, um Bereiche zu identifizieren, in denen Bewegungen stattfinden. Diese Technik kann angepasst werden, um die von den *SensFloor* Sensoren erfassten Kapazitätsdaten zu analysieren und Änderungen des Drucks oder der Bewegung zu erkennen.

Eine weitere Referenz ist der Übersichtsartikel von (C. Yang et al. (2010)), der einen Überblick über auf Beschleunigungsmessung basierende tragbare Bewegungsdetektoren für die Überwachung körperlicher Aktivität bietet. Obwohl sich diese Referenz auf tragbare Geräte konzentriert, wird die Verwendung von Beschleunigungssensoren zur Bewegungserkennung und -klassifizierung hervorgehoben. Beschleunigungssensoren können Änderungen der Beschleunigung messen, aus denen sich Bewegungsmuster ableiten lassen.

Darüber hinaus wird in dem Artikel von Wang et al. (2018) ein hochgradig dehnbarer, transparenter, selbstversorgender, triboelektrischer Tastsensor zur Erkennung und räumlichen Abbildung von Bewegungsprofilen vorgestellt. Triboelektrische Sensoren können Änderungen in der elektrischen Ladung erkennen, die durch Kontakt und Bewegung entstehen. Wang et al. (2018) erläutert eine Alternative zu kapazitiven Sensoren und hebt die Nutzung triboelektrischer Effekte für die Bewegungserkennung hervor.

2.5.3 Schnittstellen und Datenübertragung

Das *SensFloor Live API* Dokument (Future-Shape GmbH (2022b)) beschreibt die Schnittstelle zur Abfrage der Daten des *SensFloor*, unter dessen Nutzung der *SensFloor* in eigene Systeme integriert und individuelle Anwendungsfälle entwickelt werden können. Die API liefert Informationen über Sensoren, Aktivitäten auf dem Fußboden und

aktive Alarme. Über die *SensFloor Config App* lassen sich Alarme individuell einrichten (Future-Shape GmbH (2022a)). Dem Nutzer ist es möglich hier das Verhalten der API unter anderem bei Personenpräsenz in bestimmten Regionen (z. B. Bettein und -ausstieg, Zimmerwechsel, Toilettengang, etc.), oder bei der Erkennung eines Sturzes festzulegen.

Die Daten werden in Form von *Events* geliefert. Diese werden entweder Asynchron (ASYNC) oder Synchron (SYNC) generiert. Synchrone *Events* bei APIs sind solche, bei denen die Daten in regelmäßigen Intervallen gesendet oder abgerufen werden, unabhängig davon, ob Änderungen stattgefunden haben oder nicht. Dem gegenüber werden asynchrone *Events* nur ausgelöst, wenn es tatsächlich eine Änderung der Daten gibt. Die API nutzt WebSockets, speziell Socket.IO und ist kompatibel mit Node.js oder JavaScript, wobei auch weitere Sprachen unterstützt werden. Das Dokument liefert eine Anleitung dafür, wie der Zugriff auf die API erfolgt, sowie Beispiele für das Aufsetzen von *Event-Listeners*, zur Verarbeitung von unterschiedlichen *Event*-Typen.

2.5.4 Event-Typen

Die *SensFloor Live API* bietet eine Vielzahl unterschiedlicher Event-Typen, die bei Interaktionen mit den Sensoren gesendet werden. Jeder Event-Typ stellt unterschiedliche Daten zur Verfügung. Im Folgenden werden diese Event-Typen erläutert. Die Beschreibungen folgender Event-Typen basiert ebenfalls auf Future-Shape GmbH (2022b).

messages-raw

Dieses *Event* sendet alle *SensFloor*-Daten ohne Vorverarbeitung aus. Die Daten werden in Form eines Arrays mit Zeitstempel versendet und enthalten detaillierte Informationen über jeden einzelnen kapazitiven Zustand aller Sensorfelder. Dieses *Event* ist asynchron.

new-activities-on-field

Das *Event* sendet nur dann eine Nachricht, wenn wenigstens einer der Sensoren eine Änderung seines kapazitiven Wertes hat, was das *Event* asynchron macht. Dieses *Event* eignet sich besonders für Anwendungen, die auf sensible Änderungen reagieren, zum Beispiel Änderungen der Bewegungsmuster oder Unterscheidung von Aktivitäten. Die Daten werden ähnlich dem *messages-raw-Event* in einem Array aus mit den neuen Sensorwerten und Zeitstempel versendet.

step

Das *step-Event* wird immer dann ausgelöst, wenn der *SensFloor* einen neuen Schritt erkannt hat, weshalb auch dieses *Event* asynchron ist. Das *Event* liefert dann Informationen zu den Koordinaten, an denen der Schritt registriert wurde. Die Qualität und Zuverlässigkeit von *step* ist maßgeblich von der Gangart und der Auflösung der Sensoren abhängig. Im Gegensatz zu den vorherigen *Events* wird bei *step* kein Array versendet, sondern ein zeitstempelloses Tupel aus x- und y-Koordinate.

activity-update

Im Gegensatz zu den Wertänderungen der einzelnen Sensoren, vereint activity-update die einzelnen Sensoren zu weniger hoch auflösenden Sensorfeldern. Das *Event* sendet alle 100 ms ein Objekt mit Informationen zu allen Punkten des *SensFloor*, an denen es seit dem letzten *Event* Änderungen des kapazitiven Wertes und Zustandes gab. Das *Event* ist also synchron. Punkte sind in dem Fall immer die Mittelpunkte der Sensorfelder. Im Gegensatz zu dem kapazitiven Wert ist der Zustand eines Punktes binär. Er kann entweder aktiv oder inaktiv sein.

cluster-update

Dieses *Event* vereinfacht die Auflösung der Daten weiter, indem die einzelnen Punkte zu Clustern zusammengefasst werden. Cluster bestehen immer aus mindestens einem aktiven Punkt und besitzen eine eigene ID. Sind mehrere Punkte nebeneinander aktiv, so formen sie zusammen ein Cluster. Sind zwei Punkte weit voneinander entfernt, dann bilden sie zwei verschiedene Cluster mit unterschiedlichen IDs. Auch dieses *Event* ist synchron. Neue Nachrichten werden in einem festen Zeitintervall versendet und enthalten ein Array aus den aktuell geformten Clustern.

objects-update

Bei diesem *Event* werden Informationen zu allen aktuell erkannten Objekten auf dem *SensFloor* synchron versendet. Objekte sind Gruppen von Clustern, die in der Regel Personen darstellen. So kann die Position einer Person im Raum grob über die Position der Cluster bestimmt werden. Auch *objects-update* sendet eine synchrone Nachricht, mit einem Array, welches alle aktuell aktiven Objekte beinhaltet.

alarms-detected & alarms-active

Das *Event alarms-detected* generiert ein Array mit allen aktuell erkannten Alarmen in einem festen Zeitintervall (also ebenfalls synchron). Alle Alarme, die über die *SensFloor*

Config App vom Nutzer eingerichtet wurden, werden ebenfalls berücksichtigt, außer der Alarm wird spezifisch deaktiviert. Da dieses *Event* alle Alarme verarbeitet, wird die API in dem Array aus erkannten Alarmen auch Alarme mit inaktivem Zustand versenden. Ähnlich dazu versendet *alarms-active* ein Array mit allen aktuell erkannten Alarmen, mit dem Unterschied, dass nur aktive Alarme berücksichtigt werden. Die Unterscheidung kann an dem Beispiel des Sturzalarms verdeutlicht werden:

Wird von dem Algorithmus des *SensFloor* ein Sturz erkannt, so senden sowohl *alarms-detected*, als auch *alarms-active* ein Datenpaket mit Informationen darüber, dass der Alarm *Fall* aktiv ist. Sobald die Person vom Fußboden aufsteht und der *SensFloor* keinen Sturz mehr erkennen kann, wird *alarms-detected* in der nächsten Nachricht den Alarm *Fall* wieder senden, dieses Mal mit inaktivem Zustand. *Alarms-active* wird den inaktiven Zustand ignorieren und keinen Sturzalarm mehr senden.

alarms-changed

Als letztes wird *alarms-changed* betrachtet. Im Gegensatz zu den beiden anderen Alarm-*Events*, ist *alarms-changed* asynchron und versendet nur dann ein Array mit Alarmen, wenn ein oder mehrere Alarme ihren Zustand geändert haben. Die Nachricht kann sowohl aktive, als auch inaktive Alarme beinhalten, sendet diese aber immer nur genau einmal bei der Änderung des Zustandes von *aktiv* auf *inaktiv* oder andersherum.

2.5.5 Verbindungsaufbau zu der SensFloor-API

Listing 1 demonstriert eine Variante der Anbindung zu der API des *SensFloor*. Das Verbindungsskript für den *SensFloor* nutzt grundlegend die *socketio*-Bibliothek für das Empfangen der Daten.

```
1  @sio.event
2  def connect():
3      print('Connection to SensFloor established.')
4
5  @sio.event
6  def disconnect():
7      print('Disconnected from SensFloor server.')
8
9  sio = socketio.Client()
10 sio.connect('http://192.168.172.22:8000')
11 sio.wait()
```

Listing 1: Auszug aus *SensFloor*-Verbindungsskript

Listing 1 stellt das vereinfache Client-Skript dar, welches eine dauerhafte Verbindung zum *SensFloor* herstellt, der unter der in Future-Shape GmbH (2022a) angegebenen Adresse[1] zugänglich ist.

Socket.IO ist eine JavaScript-Bibliothek, die die Kommunikation zwischen Webclients und Servern ermöglicht. Socket.IO ist sowohl auf der Client- als auch auf der Serverseite in JavaScript geschrieben. Die Kommunikation von Socket.IO basiert auf WebSockets, bietet jedoch zusätzlich Fallback-Optionen, um eine bessere Kompatibilität zu gewährleisten. Ein entscheidendes Merkmal von Socket.IO ist die *Event*-basierte Kommunikation, die es Entwicklern erlaubt, eigene *Events* zu erstellen, auf diese zu hören und bei *Event*-Eintritt darauf zu reagieren (Socket.IO Team (2024)).

Das Skript bietet einige beispielhafte Definitionen von *Eventhandler*-Funktionen. Diese Funktionen werden bei Empfangen eines bestimmten *Events* ausgeführt. In Listing 1 sind *connect*- und *disconnect*-*Eventhandler* definiert. Die *connect*-Funktion wird aufgerufen, wenn die Verbindung zum *SensFloor* erfolgreich war, die *disconnect*-Funktion wird hingegen ausgeführt, wenn die Verbindung zum Server getrennt wurde.

Eine Instanz des *socketio.Client* wird der globalen Variable *sio* zugewiesen, die Verbindung zum Server wird hergestellt und anschließend wartet das Programm durch die *sio.wait()*-Methode auf weitere *Events*. Die Verbindung wird nur durch eine Unterbrechung der Verbindung oder durch explizite Beendigung des Programms geschlossen.

2.5.6 Potentielle Fehlerquellen

Eine zentrale Herausforderung bei der Entwicklung kapazitiver Sensorfußböden ist die Genauigkeit der Sensordaten. Insbesondere treten so genannte *Ghost*-Signale auf, bei denen der Sensor eine Anwesenheit registriert, obwohl weder physische Objekte noch Personen vorhanden sind. Elektromagnetische Störungen aus benachbarten elektronischen Geräten können solche Fehlsignale induzieren, die fälschlicherweise von kapazitiven Sensoren als physische Anwesenheit interpretiert werden (Osoinach (2008)).

Eine Studie von Arshad et al. (2017) untersucht die Sensibilität von kapazitiven Sensoren gegenüber Umgebungsstörungen. Sie betont die Notwendigkeit, Sensoren gegenüber möglichen Interferenzen abzuschirmen und hochwertige Materialien zu verwenden. In 3.8.11 folgen praktische Untersuchungen, welche ermitteln werden, wie groß der Einfluss solcher Interferenzen auf die Zuverlässigkeit des *SensFloor* ist.

[1] http://192.168.172.22:8000

2.5.7 Alternative Systeme zum SensFloor

Das *CapFloor-System*, vorgestellt auf der Evaluating Ambient Assited Living (EvAAL) 2011 (Andreas Braun et al. (2012)) ist ein weiterer Ansatz zur Ortung von Personen mittels kapazitiver Sensortechnologie. Auch dieser Sensorfußboden wurde speziell im Kontext von AAL konzipiert und nutzt flexible Sensormatten, die kostengünstig hergestellt werden. Ähnlich dem *SensFloor* bietet der *CapFloor* ebenfalls die Erkennung von Stürzen, was ihn besonders für die Unterstützung älterer Menschen eignet. Der *CapFloor* besitzt eine zentrale Plattform, die drahtlos mit den Sensormatten verbunden ist. Die Plattform sammelt und verarbeitet die Daten der Sensoren. Im Rahmen des EvAAL-Wettbewerbs stellte der *CapFloor* 2011 seine Fähigkeit unter Beweis, nahtlos in das häusliche Umfeld integriert werden zu können.

2.6 PhysioNova - RotoFlex

RotoFlex ist ein Aufstehbett des Herstellers PhysioNova. Aufstehbetten sollen die Lebensqualität von Personen verbessern, die in ihrer Mobilität eingeschränkt sind, indem es ihnen ermöglicht, ohne fremde Hilfe aufzustehen. Aufstehbetten fördern die Unabhängigkeit und gehen Hand in Hand mit den Kernkonzepten von AAL. *RotoFlex* ist nach europäischer Norm zertifiziert und in verschiedenen Ausführungen erhältlich. PhysioNova bietet verschiedene Varianten an, um individuellen Bedürfnissen gerecht zu werden. So ist das Bett unter anderem in unterschiedlichen Größen erhältlich oder kann wahlweise (je nach Raumeinrichtung) in rechts- oder linksdrehend bestellt werden. PhysioNova bietet weiteres Zubehör an, um das Aufstehbett zu erweitern, wie Aufstehgriffe und Infusionshalter. (PhysioNova GmbH (2024))

2.6.1 Integrierte Wiegetechnik von A.S.T.

Angewandte System Technik (A.S.T.) GmbH ist ein Unternehmen in Dresden, beschäftigt sich mit der Entwicklung und Fertigung von Kraftmesstechnik und Sensorik. Unter anderem entwickelt A.S.T. Integrationen von intelligenten Geräten zur Unterstützung der Selbstständigkeit im Alter, darunter das intelligente Bett. Das Bett ist mit mehreren Kraftsensoren ausgestattet, die Verlagerungen des Schwerpunkts des Patienten aufzeichnen und überwachen. Das Bett sendet Signale aus, wenn das Bett verlassen wird oder wenn über einen längeren Zeitraum keine Bewegung festgestellt wird, zur Vermeidung von Wundliegen oder Druckgeschwüren (A.S.T. GmbH (2024); A.S.T. – Angewandte System Technik GmbH, Mess- und Regeltechnik (2024)).

Das Pflegebett im AAL-Living Lab Cultus Bühlau kombiniert die Barrierefreiheit des *RotoFlex* mit der intelligenten Sensorik von A.S.T. Durch die Erweiterung des *RotoFlex* ist es möglich, Schwerpunktverlagerungen über den A.S.T. *CANopen Logger* oder eine API abzurufen und basierend darauf eigene Anwendungen zu entwickeln (A.S.T. – Angewandte System Technik GmbH, Mess- und Regeltechnik (2024)).

2.6.2 Verbindungsaufbau zu der RotoFlex-API

Um eine Verbindung zu der *RotoFlex*-API herzustellen wird ebenfalls ein Python-Skript geschrieben, welches die Anbindung eines Clients zu der API übernimmt.

```python
def on_message(wsapp, message):
    print(message)

wsapp = websocket.WebSocketApp('ws://192.168.200.99/api/v2/live/all/ws',
                               on_message=on_message)
wsapp.run_forever()
```

Listing 2: Auszug aus RotoFlex-Verbindungsskript

Listing 2 ist eine verkürzte Version des Python-Skripts. Das für die Verbindung genutzte *WebSocket* ist ein Kommunikationsprotokoll, welches mit einer einzelnen Transmission Control Protocol (TCP)-Verbindung eine dauerhafte Kommunikation zwischen Client und Server aufbauen kann. Mithilfe von *WebSockets* können Daten kontinuierlich übertragen werden, ohne dass die Notwendigkeit besteht, bei jeder Übertragung eine neue HTTP-Anfrage zu senden (Fette et al. (2011)).

Bei Initialisierung des *WebSocketApp-Objektes* (Client) wird die Adresse des *RotoFlex*, sowie eine *Eventhandler*-Funktion definiert. Der *Eventhandler on_message* wird jedes mal ausgeführt, wenn der Client eine neue Nachricht von der angegebenen Adresse erhält. Die *run_forever*-Methode sorgt dafür, dass die Verbindung offen bleibt und kontinuierlich läuft, damit das Programm weiterhin Nachrichten empfangen kann.

3 Methodik

3.1 Vorhandene Architektur im AAL-Living Lab Cultus Bühlau

Die Integration des *SensFloor* und *Rotoflex*-Pflegebetts geschieht in ein bereits aufgesetztes *openHAB*-System. Die Laborwohnung ist zum Beginn der Diplomarbeit mit unterschiedlichen AAL-Technologien ausgestattet. Unter anderem befinden sich in der Wohnung Beleuchtungssysteme von *Philips*, welche über unterschiedliche Schnittstellen bedient werden können (Klinger et al. (2021)), Monitore zur Darstellung verschiedener Parameter, höhenverstellbare Küchen- und Badmöbel, Thermostate und Hygrometer zur Bestimmung des Raumklimas, sowie Maßnahmen zur Überwachung der Sicherheit mithilfe von Sensoren an allen Fenstern.

Außerdem verfügt die Wohnung bereits über das *RotoFlex*-Pflegebett, mit eingebauter A.S.T. Wiegezelle. Der *SensFloor*-Sensorfußboden ist ebenfalls verbaut. Des Weiteren sind beide Geräte in das lokale Netzwerk eingebunden.

Die Wohnung verfügt über ein eigenes Netzwerk von mehreren Servern, die unterschiedliche Aufgaben übernehmen. Als erste Instanz agiert ein *Raspberry Pi*, der als Jump Server dafür zuständig ist, Zugriffe auf weitere interne Systeme zu verwalten und zu kontrollieren. Über einen angelegten User mit individuellem Schlüsselpaar kann sichergestellt werden, dass nur authentifizierte Nutzer Zugriff auf die internen Systeme erhalten. Dies erhöht die Sicherheit der Daten von den zu entwickelnden Anbindungen. Der *RotoFlex* und der *SensFloor* sind direkt mit einem internen Router über Wireless Local Area Network (WLAN) verbunden. Außerdem gibt es ein Linux-System, welches zur Ausführung unterschiedlicher Automatisierungen zuständig ist. Der *openHAB* ist auf einem Raspberry Pi 4 mit Raspian OS installiert.

Das eingerichtete *openHAB*-System besitzt für die Navigation und Verwaltung eine Main User Interface (UI) sowie eine Konsole (*Console* (o. D.), *MainUI* (o. D.[b])). Beide Varianten setzen voraus, dass der Nutzer angemeldet ist. Danach kann die Konsole in einem Terminalfenster über den Befehl *openhab-cli* gestartet werden (*Logs* (o. D.)).

Um die Main UI aufzurufen, wird in einem Web-Browser die *Loopback*-Adresse (localhost) auf dem Port 8080 aufgerufen (*Einrichtung* (o. D.)). Aus dem Grund, dass die Main UI gleichzeitig für die Darstellung der einzelnen Wohnungsparameter für Endnutzer genutzt wird, ist es für die Anbindung der Geräte an den *openHAB* essentiell, dass auch diese eine grafische Darstellung der aktuellen Zustände anbieten. Im Rahmen der Entwicklung kann die Main UI zur Erstellung aller *openHAB*-relevanten Komponenten genutzt werden (inklusive der Benutzeroberfläche), sowie die Konsole für Backend-Entwicklungen und Log-Einsichten.

Zusätzlich ist auf der Entwicklungsmaschine Python[1] installiert. Python zeichnet sich durch einfache und intuitive Syntax aus. Es unterstützt vielfältige Programmierstile und bietet eine große Auswahl an kompatiblen Bibliotheken für verschiedenste Anwendungen. Dank der Plattformunabhängigkeit ist Python sehr vielfältig einsetzbar und eignet sich deswegen für die Entwicklung der Anbindungen (Python (2021)).

3.2 Untersuchung möglicher Vorgehensweisen für die Anbindung von SensFloor an openHAB

Bei der Anbindung des *SensFloor* an den *openHAB*, sowie für die Entwicklung eines automatisierten Beleuchtungssystems, bei dem der *SensFloor* für die Erkennung der Präsenz von Personen verantwortlich ist, müssen einige Faktoren beachtet werden. Es folgen Analysen zu allen, für die Entwicklung entscheidenden, Parametern.

3.2.1 Event-Typen der SensFloor-API

Die *SensFloor*-API bietet eine Vielzahl von *Event*-Typen und personalisierbaren Alarmen, die für die Erfassung von Bewegung, Präsenz und Aktivitäten auf dem Fußboden genutzt werden können (siehe 2.5.4). Die *Events* lassen sich in zwei Kategorien einteilen: Rohdaten-*Events* und Alarm-*Events*.

Rohdaten-*Events* sind all die *Events*, die dem Nutzer detaillierte Informationen zu kapazitiven Wertänderungen der einzelnen Sensoren oder Sensorgruppen bieten. Diese *Events* beinhalten eine deutlich höhere Informationsdichte und werden nicht vom *SensFloor* vorverarbeitet. Basierend auf diesen Informationen können sehr genaue Aussagen über den Zustand von jedem einzelnen Sensor des *SensFloor* getroffen werden. Gleichzeitig ist es aber in der Verantwortung des Datenempfängers, die Daten korrekt auszuwerten. Weil diese *Event*-Typen keine Datenverarbeitung anbieten, ist es erforderlich, eigene Algorithmen zu entwickeln, die Situationen, wie bspw. Anwesenheit von Personen, Stürze und Schritte erkennen. Zu dieser Kategorie gehören z. B. *messages-raw* und *new-activity-on-field.*

Alarm-*Events* sind all die *Events*, die keine detaillierten Informationen zu den einzelnen Sensoren versenden, sondern stattdessen die verarbeiteten Daten in Form von Alarmen bereitstellen. Auslösende Bereiche für bestimmte Alarme (z. B. *room in*) können zwar in der *SensFloor Config App* individualisiert werden, letztendlich verlässt sich der Datenempfänger dabei aber auf eine korrekte Datenauswertung durch den *SensFloor*. Beispielsweise können über Alarm-*Events* Präsenzen von Personen in bestimmten Räumen festgestellt werden, ohne zu wissen, welche Sensorzustände und -werte vorliegen. Zur Vereinfachung wird das *step-Event* ebenfalls als Alarm-*Event* betrachtet. Zwar existiert im *SensFloor*-System kein direkter *step-Alarm*, jedoch erfordert die Erkennung eines Schritts

[1] Version 3.9.2 (laut Ausführung des Befehls *python –version* in der Konsole)

eine algorithmische Verarbeitung der Daten seitens des *SensFloor*, weshalb das *step-Event* die gleichen Eigenschaften annimmt, wie andere Alarm-*Events* auch.

Die Entscheidung zwischen der Nutzung von Rohdaten-*Events* und Alarm-*Events* ist selbstverständlich abhängig von der Anwendung. Rohdaten-*Events* liefern präzise Daten und ermöglichen dadurch die Entwicklung von maßgeschneiderten Algorithmen zur Verarbeitung der Daten. Alarm-*Events* bieten auf der anderen Seite vorgefertigte und kalibrierte Lösungen, die alle Funktionalitäten bereitstellen, wie die Erkennung von Präsenz, Sturz und Raumwechsel, die für die Anbindung an den *openHAB*, sowie die Entwicklung des Beleuchtungssystems erforderlich sind.

Praktische Tests (siehe 3.6) zeigen, ob Alarm-*Events* für eine zuverlässige Standortermittlung ausreichen. Vorerst besteht die Annahme, dass sowohl die Anbindungen in den *openHAB*, als auch die Aktivierung der LED-Streifen Raumabhängig und Koordinatenunabhängig ist (es ist wichtig zu wissen, in welchem Raum eine Person ist, aber nicht wo genau innerhalb des Raumes), weshalb es möglicherweise ausreichend sein wird, sich auf Alarm-*Events* zu verlassen. Damit ist gemeint, dass wenn eine Person einen Raum betritt das Licht in dem gesamten Raum eingeschaltet werden soll und spezifische Koordinaten keine Rolle spielen.

Außerdem ist es essentiell, die Unterschiede der Datenübertragung zu untersuchen, da manche *Events* synchron, andere aber asynchron versendet werden. Wie bereits beschrieben unterscheiden sich die zwei Arten der Datenübertragung darin, ob die API Daten in festen Zeitabständen versendet oder aber die Daten nur versendet werden, wenn bestimmte Voraussetzungen erfüllt sind. Im Fall des *SensFloor* basieren asynchrone *Events* auf der Änderung von Werten und Zuständen. Hier ist also wichtig zu entscheiden, ob es bei der Anbindung des *SensFloor* an den *openHAB* reicht, nur Zustandsänderungen darzustellen.

Weil für das automatisierte Beleuchtungssystem auf Änderungen von Zuständen auf dem Fußboden reagiert werden muss und keine konstante Zustandsüberwachung benötigt wird, ist hypothetisch davon auszugehen, dass asynchrone *Event*-Typen für diese Anwendung ausreichen. Auch die Darstellung des *SensFloor* im *openHAB* muss nicht zwingend kontinuierlich überprüft werden. Es ist anzunehmen, dass die Zustände des *SensFloor* in einem Zustandsdiagramm dargestellt werden können.

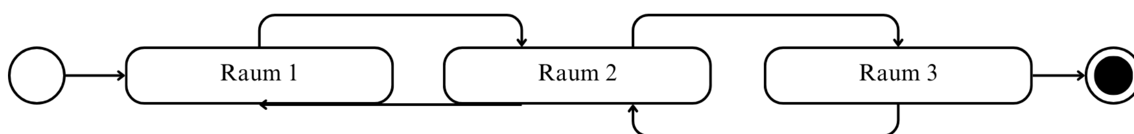

Abbildung 3.1: Zustandsdiagramm am Beispiel von drei Räumen

In dem Beispiel in Abbildung 3.1 sind drei Räume dargestellt. Zu jedem Zeitpunkt kann eine Person die Wohnung betreten und sich in *Raum 1* befinden oder sich in einem der zwei anderen Räume aufhalten. Für dieses Modell würde die Information über die Änderungen im Raum ausreichen, um bestimmen zu können, ob sich eine Person im Raum befindet oder nicht. Die Annahme kann an folgendem Beispiel demonstriert werden:

Eine Person startet in *Raum 1*, wo sie sich fünf Sekunden aufhält. Sie geht von dort in *Raum 2*, bleibt hier ebenfalls fünf Sekunden und wechselt weiter in *Raum 3*, wo das Beispiel nach weiteren fünf Sekunden endet. Nutzt man die synchrone Datenübertragung, so werden alle 100 ms die Zustände der Räume übermittelt. Weil der gesamte Prozess 15 s dauert, werden also insgesamt 150 mal Daten generiert, von denen jeweils 50 gleich sind. Betrachtet man nur asynchrone Übermittlungen, so werden unabhängig davon, wie viel Zeit die Person sich in einem Raum aufhält, nur drei Nachrichten gesendet, mit Informationen über die Zustandsänderung im Raum.

Da sowohl für die Anbindung des *SensFloor* an den *openHAB*, als auch für die Entwicklung des Beleuchtungssytems nur relevant ist, ob Zustandsänderungen geschehen sind, ist es deutlich ressourcensparender, wenn asynchrone *Event*-Typen genutzt werden. Da dieses Beispiel sich aber auf eine einzelne Person mit langsamen und strukturierten Zustandsübergängen bezieht, ist es essentiell im späteren Verlauf zu untersuchen, was passiert, wenn sich mehrere Personen in der Wohnung befinden.

3.3 Analyse der Arbeitsschritte

Die Arbeitsschritte lauten wie folgt: Der *SensFloor* und der *RotoFlex* müssen an den *openHAB* angebunden werden. Eine erfolgreiche Anbindung ist modular und bildet ein replizierbares Grundgerüst für die Anbindung weiterer Systeme in der Zukunft. *SensFloor* und *RotoFlex* sollen eine einheitliche Darstellung im Gesamtsystem haben. Das bedeutet, dass die Systeme eine variable Anwendung haben sollen, die in alle vorhandenen und kommenden Entwicklungen im *openHAB* nahtlos integriert werden können. Die Objekte sollen eine Repräsentation des aktuellen Zustandes haben, sowie nutzerfreundliche Darstellungen anbieten. Die Anbindung wird folglich zwei Aufgaben übernehmen: Bereitstellung und Verarbeitung der aktuellen Zustandsinformationen im Backend und eine vereinfachte, visuelle Darstellung der Daten für den Benutzer im Frontend.

Basierend auf den verarbeiteten Daten von *SensFloor* und *RotoFlex* wird prototypisch das nächtliche Beleuchtungssystem entwickelt. Die genaue Architektur des Beleuchtungssystems kann erst entschieden werden, wenn die Anbindung der Geräte an den *openHAB* geschehen ist, da dies die Struktur der Daten bestimmen wird. Nichtsdestotrotz muss bei der Anbindung der Geräte an den *openHAB* bedacht werden, dass die Daten eine einheitliche und eindeutige Form besitzen müssen, da die Anwendung für die Steuerung der Beleuchtung auf der Anbindung aufbauen wird. Hierfür ist eine Untersuchung der Datentypen erforderlich. Des weiteren muss die Anbindung des *SensFloor* eine Verarbeitung der Sturzerkennung umfassen, da dies ein wichtiger Bestandteil der Sensorfußbodens ist. Auch das ist bei der Ausarbeitung der Datenstruktur zu beachten.

Aus diesen Erkenntnissen geht hervor, dass als erstes eine Anbindung der Geräte im Backend zu entwickeln ist, da diese Anbindung das Grundgerüst für alle weiteren Arbeitsschritte darstellt. Darauf aufbauend erfolgt die Entscheidung der Datenstruktur, die Entwicklung der Datenverarbeitung, das Gestalten der Benutzeroberfläche und schlussendlich die Verbindung aller Arbeitsschritte für die Ausarbeitung des automatisierten Beleuchtungssystems.

3.4 Integrationsansätze

Aus der Architektur des *openHAB* (siehe 2.1) geht hervor, dass entweder ein *Thing* oder ein *Item* angelegt werden muss, welches die von den Geräten gesendeten Daten empfängt und für die weitere Verarbeitung kapselt. Die Hauptunterschiede bestehen darin, dass die Integration über ein *Thing* eine einheitliche Konfiguration bietet und somit eine konsistente Weise ermöglicht Geräte hinzuzufügen und zu konfigurieren. Die bidirektionale Kommunikation zwischen *Thing* und Gerät bietet den großen Vorteil, dass nicht nur Daten von dem Gerät empfangen werden können, sondern es auch vereinfacht Steuerungsbefehle von dem *openHAB* an das Gerät zurückzusenden. Die Integration des *Philip Hue* als *Thing* demonstriert, wie einerseits die Glühbirnen Informationen über den aktuellen Zustand liefern, andererseits über das *Thing* Befehle an die Glühbirne gesendet werden können, die Eigenschaften, wie die Farbe und Helligkeit steuern. Abbildung 3.2 verallgemeinert die Integration eines Gerätes in den *openHAB* unter Anlegung eines *Things*.

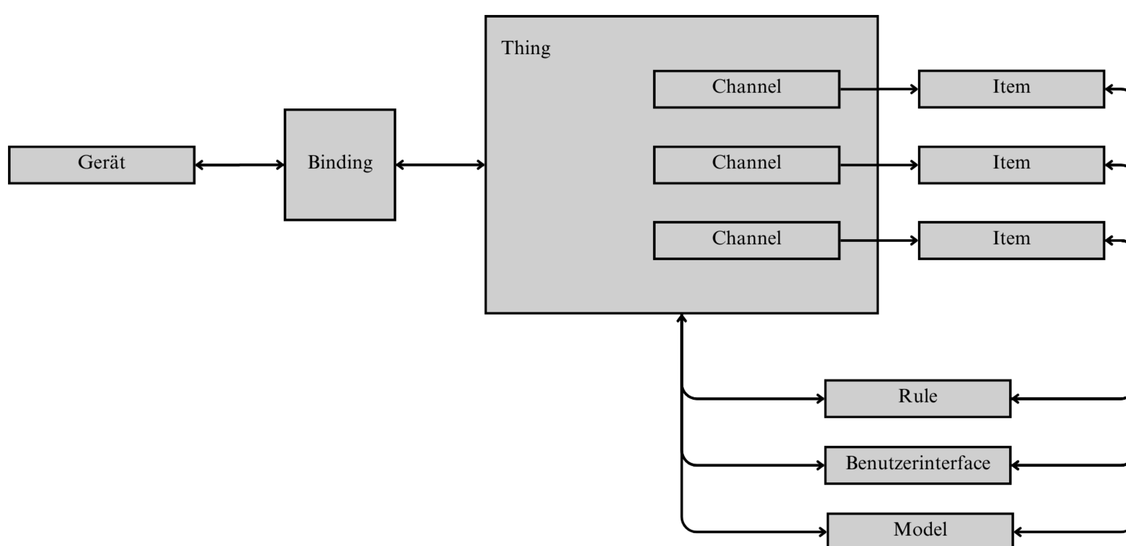

Abbildung 3.2: Darstellung der Beziehungen zwischen den Komponenten bei der Integration als *Thing*

In Abbildung 3.2 wird deutlich, dass sowohl *Rules*, als auch die Benutzeroberfläche und die *Models* von *openHAB* gleichzeitig mit den *Things* und den *Items* kommunizieren können. Es ist bspw. möglich *Rule-Trigger* basierend auf einer Zustandsänderung eines *Items* oder eines *Things* festzulegen. Genauso können auch *Items* als *Equipment* in ein *Model* übernommen werden. Es ist gleichermaßen möglich Statusinformationen von *Items* oder *Things* über die Benutzeroberfläche darstellen zu lassen. Tatsächlich basiert letzteres sogar häufiger auf *Items*, da es in der Regel sinnvolle Funktionalitäten bieten kann, wie zum Beispiel das Anlegen eines Farbkreises, der direkt mit dem *Color-Item* einer Glühbirne verbunden ist.

Komplizierter wird es, wenn es kein bereitgestelltes *Binding* eines Herstellers gibt. In dem Fall bleibt die Möglichkeit eines der standardisierten *Bindings* zu nutzen. Speziell das *Exec Binding* und das *HTTP Binding* bieten eine universelle Schnittstelle zu einem

Thing. Außerdem ist es möglich ein eigenes *Binding* (*Custom Binding*) zu erstellen und dieses mit einem *Thing* zu verknüpfen. *Custom Bindings* sind besonders nützlich, um spezifische Protokolle zu unterstützen, die in der Standardbibliothek von *openHAB* fehlen. Die Erstellung eines *Custom Binding* erfordert jedoch mehrere Arbeitsschritte und ist in der Umsetzung deutlich aufwendiger.

Schlussendlich besteht die Möglichkeit auf *Bindings* und *Things* zu verzichten, da die meisten Funktionalitäten eines *Things* auch über *Items* realisiert werden können. In Abbildung 3.2 kann man erkennen, dass *Things* mehrere *Items* über *Channels* verknüpfen. Aus diesem Grund können *Things* als eine Art Container betrachtet werden. Abbildung 3.3 zeigt, dass sich diese Container-Eigenschaft auch über eine *Group* replizieren lässt.

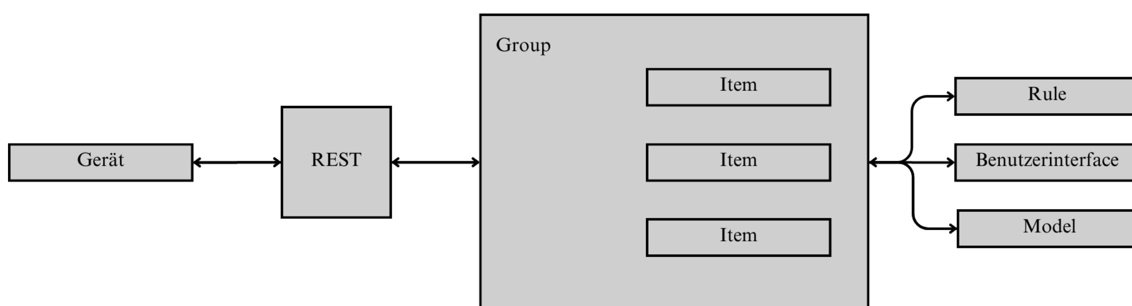

Abbildung 3.3: Darstellung der Beziehungen zwischen den Komponenten bei der Integration als *Item Group*

Das Gerät kann statt einem *Binding* die REST-API nutzen, um mit den *Items* zu kommunizieren. Die *Rules*, *Models* und das Benutzerinterface können nach wie vor mit den *Items* interagieren, ohne, dass diese an ein *Thing* gekoppelt sind. Auch die Verwendung einer *Group* ist nicht immer notwendig. Wenn es Geräte gibt, die einen einzigen Kommunikationskanal besitzen, kann die Integration auch über ein einzelnes *Item* gelöst werden (Abb. 3.4).

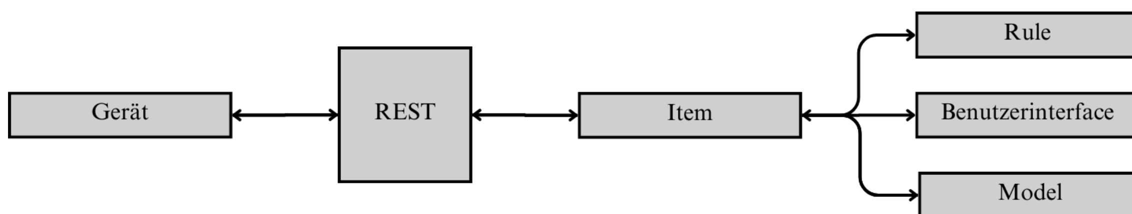

Abbildung 3.4: Darstellung der Beziehungen zwischen den Komponenten bei der Integration als einzelnes *Item*

Die Frage nach der besten Integrationsmethode bleibt durch praktische Entwicklungsversuche zu klären. Folgende Abschnitte widmen sich der Erläuterung dieser Entwicklungsansätze der Anbindung.

3.4.1 Integrationsansatz als Thing mit Exec Binding

Bevor die Konstruktion eines *Custom Binding* in Erwägung gezogen werden kann, ist es sinnvoll die bereits existierenden *Bindings* der *openHAB*-Bibliothek zu untersuchen. Da der *openHAB* auf dem gleichen Server gehostet wird, wie die Verbindungen zu *RotoFlex* und *SensFloor* starten (die Daten der Verbindungsskripte sind also direkt auf dem selben System verfügbar), gibt es die Möglichkeit die Skripte über ein *Exec Binding* mit einem *Thing* zu verbinden. *Exec Bindings* in *openHAB* ermöglichen es beliebige Shell-Befehle auszuführen, um externe Programme oder Skripte innerhalb des *openHAB* zu steuern (*Exec Binding* (o. D.)). Das bedeutet, dass es möglich ist, ein Python-Skript über das *Exec Binding* über einen Shell-Befehl zu starten und den Rückgabewert des Skriptes als Datengrundlage für das *Thing* zu nutzen. Die *openHAB* Dokumentation empfiehlt alle Befehle die über das *Exec Binding* ausgeführt werden in die Whitelist zu schreiben. Diese befindet sich unter misc/exec.whitelist. Um die Ausführung eines Python-Skriptes zu erlauben muss also der gesamte Befehl in eine neue Zeile in die Datei geschrieben werden. Listing 3 zeigt eine Verallgemeinerung, wie ein Python Befehl in der Whitelist hinterlegt werden muss.

```
1 /usr/bin/python3 /path/to/script.py %2$s
```

Listing 3: Beispiel für *openHAB* Whitelist

In der Konfiguration des *Exec Binding* bezieht sich *%2$s* auf eine Formatierungsspezifizierung. Dadurch wird festgelegt, dass Argumente im Shell-Befehl verwendet werden können. Danach kann das *Thing* mit den dazugehörigen *Items* konfiguriert werden. Das *Thing* wird in der **.things*-Datei definiert und mit den *Items* der **.items*-Datei, wie in Listing 4 dargestellt, verknüpft.

```
1 [*.things]
2 Thing exec:command:pythonScript
3 [command='/usr/bin/python3 /path/to/script.py %2$s', interval=1, timeout=5,
4 autorun=false]
5
6 [*.items]
7 String PythonScriptInput {channel='exec:command:pythonScript:input'}
8 String PythonScriptOutput '[%s]' {channel='exec:command:pythonScript:output'}
```

Listing 4: Beispiel für **.things*- und **.items* Definition

Die Konfigurationsparameter *Interval*, *Timeout* und *Autorun* legen fest, wie *openHAB* den Befehl ausführt (*Exec Binding* (o. D.)). *Interval* definiert, wie häufig das *Exec Binding* den Befehl starten wird. *interval=1* bedeutet, dass das Skript jede Sekunde gestartet wird. *Timeout* bestimmt (in Sekunden), wie lange es dauert, bis die Ausführung des

Befehls zwangsläufig beendet wird. Mithilfe des Parameters *Autorun* wird festgelegt, ob der Befehl jedes mal automatisch ausgeführt wird, wenn der *Input-Channel* einen neuen Wert annimmt, oder ob der Start manuell erfolgt, bspw. durch die Festlegung von Intervallen oder durch andere manuelle Trigger. Hierfür könnte man ein Switch *Item* anlegen, welches dem Nutzer erlaubt über die Benutzeroberfläche den Befehl zu starten. Da für die Integration des *SensFloor* und *RotoFlex* kein Input vorgesehen ist und der Befehl ohne Parameter gestartet wird, kann das *PythonScriptInput Item* für den spezifischen Anwendungsfall weggelassen werden. *PythonScriptOutput* ist ein String *Item* und mit dem Output *Channel* des *Exec Binding* verbunden. Es speichert die Ausgabe des zuletzt ausgeführten Befehls. Wenn das Python-Skript Daten ausgibt, werden diese im *PythonScriptOutput* gespeichert und können innerhalb von *openHAB* für weitere Verarbeitung über *Rules* genutzt werden. Die Syntax *[%s]* gibt an, dass das *Item* in der Benutzeroberfläche als reiner Text formatiert werden soll.

Obwohl der Ansatz für viele Integrationen relevant ist, bietet er bei dem Versuch *SensFloor* und *RotoFlex* an das *openHAB* anzubinden keine Lösung für das Problem. Beide Geräte senden kontinuierlich Daten und die Verbindungen zu den APIs ist persistent. Das *Exec Binding* bietet keine Möglichkeit an, Verbindungen aufrechtzuerhalten, weshalb, je nach Intervallfestlegung, alle X-Sekunden die Verbindungen neu aufgebaut und wieder getrennt werden müssen. Wenn der Verbindungsaufbau schnell genug geschieht, ist es trotzdem möglich über diesen Ansatz die Geräte in den *openHAB* anzubinden, jedoch stellt es eine unnötige Belastung für das Netzwerk und den Server dar, ständig Verbindungen auf- und abzubauen. Auch wenn die API des *RotoFlex* synchron arbeitet und es prinzipiell möglich wäre ein Intervall festzulegen, was möglichst gleichzeitig zu der API läuft, sind viele Alarme des *SensFloor* asynchron. Würde man diese über das zwangsläufig synchrone *Exec Binding* abrufen, besteht das Risiko, dass Daten verloren gehen, die zwischen zwei Ausführungen des Befehls gesendet wurden.

Fazit Integration über Exec Binding

Das *Exec Binding* von *openHAB* bietet eine flexible Methode zur Integration externer Skripte. Durch regelmäßige Ausführung können Daten von Python-Skripten unter anderem für die Definition von *Rules* oder das Anlegen von Elementen in der Benutzeroberfläche genutzt werden. Das *Exec Binding* stößt bei der Integration von Geräten, die eine kontinierliche Datenübertragung oder persistente Verbindungen erfordern aber an seine Grenzen. Insbesondere der unnötige Verbrauch von Server- und Netzwerkressourcen, sowie der mögliche Datenverlust machen das *Exec Binding* für die Integration des *SensFloor* und *RotoFlex* an den *openHAB* ungeeignet.

3.4.2 Integrationsansatz als Thing mit Custom Binding

Alternativ zu vorgefertigten *Bindings* bietet *openHAB* die Möglichkeit eigene *Bindings* zu erstellen. Die folgende Erläuterung bezieht sich auf die Dokumentation *Developing a Binding* (o. D.). Das Erstellen eigener *Bindings* umfängt mehrere Arbeitsschritte und ist

eine verhältnismäßig aufwändige Art der Integration, da man sich hierbei selten auf vorgefertigte Lösungen verlassen kann, die *openHAB* anbietet. Die Grundlegenden Schritte zur Erstellung eines *Custom Bindings* sind die Definition eines *Thing*-Typen, die Implementierung eines *ThingHandlers* und die Erstellung der *HandlerFactory*. *Thing*-Typen sind über XML-Dateien beschriebene Konfigurationen für das *Binding*. Diese definieren den Kommunikationsablauf zwischen Python-Skript und *Thing*. *ThingHandler* sind Java-Klassen, die die Logik für die Steuerung und Überwachung der Geräte beinhaltet. Die *HandlerFactory* instanziiert den *ThingHandler*. Die *Factory* prüft, ob die Definitionen des *Thing*-Typen unterstützt werden und erstellt den entsprechenden *Handler*.

Schritte zur Erstellung eines Custom Bindings

Als erstes empfiehlt die Dokumentation des *openHAB* die Definition des *Thing*-Typs. Dafür wird in dem selbst angelegten *Binding*-Verzeichnis unter *ESH-INF/thing* eine XML-Datei erstellt. Das *Thing* benötigt für die Kommunikation mit einem Python-Skript einen String *Channel* für das Speichern des Outputs.

```
<channel-type id='ChannelID'>
    <item-type>String</item-type>
    <label>GeräteItem</label>
</channel-type>
<thing-type id='ThingID'>
    <label>GeräteThing</label>
    <channels>
        <channel id='ChannelID' typeId='string-channel'/>
    </channels>
</thing-type>
```

Listing 5: Beispiel für eine *Thing*-Typ Definition

Listing 5 zeigt ein einfaches und verkürztes *Thing*-Typ. Es können weitere Eigenschaften definiert werden, wie Tags, Beschreibungen, etc., die für die vereinfachte Veranschaulichung der Funktionalität aber keine Rolle spielen. In Listing 5 wird sowohl ein *Thing*, als auch ein *Channel*-Typ im XML-Format beschrieben. Wie jedes Element im *openHAB* müssen *Channels* und *Things* mit einer eindeutigen ID versehen werden. Der *Channel*-Typ besitzt außerdem eine Defintion seines Datentyps (String). Dem *Thing*-Typ wird der *Channel*-Typ untergeordnet, was bedeutet, dass dieses *Thing* später in der Erstellung den *Channel* besitzen wird. Es ist wichtig zu betonen, dass hier lediglich die Definition von den Typen erfolgt. Die XML-Datei beschreibt nur den Aufbau eines *Channels* und eines *Things*, instanziiert diese jedoch nicht.

Als nächstes wird der *ThingHandler* erstellt. *OpenHAB* bietet eine abstrakte Klasse namens *BaseThingHandler*, die bereits viele Grundfunktionalitäten eines *ThingHandlers* bereitstellt. Durch Vererbung kann eine eigene Java-Klasse erstellt werden, die die Kommunikation zwischen dem Gerät und des *Things* beschreibt. Die *ThingHandler*-Klasse hat in

der Grundausführung einen Konstruktor, welcher das *Thing*-Objekt an den *ThingHandler* übergibt, eine Initialisierungsmethode, die bspw. den Status des *Things* auf *Online* setzen kann, oder andere Aufgaben bei der Initialisierung ausführt und eine *dispose*-Methode, die aufgerufen wird, wenn das *Thing* vom System nicht mehr benötigt wird, um Ressourcen freizugeben und laufende Aufgaben abzuschließen. Die beispielhafte Implementierung eines *ThingHandlers* in Java ist in Listing 6 dargestellt.

```
1  public class ThingHandler extends BaseThingHandler {
2      private ScheduledFuture<?> pollingJob;
3
4      public ThingHandler(Thing thing) {
5          super(thing);
6      }
7
8      @Override
9      public void initialize() {
10         updateStatus(ThingStatus.ONLINE);
11     }
12
13     @Override
14     public void dispose() {
15         if (pollingJob != null && !pollingJob.isCancelled()) {
16             pollingJob.cancel(true);
17             pollingJob = null;
18         }
19     }
20 }
```

Listing 6: Grundlage für einen *ThingHandler*

Die Klasse besitzt eine private Variable *pollingJob* vom Datentypen *ScheduledFuture*. *ScheduledFuture* ist eine Java-Schnittstelle, die speziell für die Verwaltung periodisch auszuführender Aufgaben genutzt wird. Im Kontext von *Custom Bindings* kann eine *ScheduledFuture* verwendet werden, um die Datenabfrage zu planen. Es bietet mehrere Methoden zu Aufgabenverwaltung, wie *cancel*, *isCancelled*, *isDone*, etc., die es ermöglichen Aufgaben einem gewählten Ablauf folgend zu starten und Ressourcen bei Nichtbedarf wieder freizugeben. Am Beispiel der *dispose*-Methode ist die Notwendigkeit am besten zu erkennen. Bevor der Aufgabenablauf beendet wird, prüft die Methode, dass er überhaupt existiert und nicht bereits abgebrochen wurde. Der *ThingHandler* übernimmt somit die Aufgabe des *Exec Commands* im *Exec Binding*, mit dem Unterschied, dass die Implementierung der Kommunikation freigestellt ist.

Als letztes wird eine *ThingHandlerFactory* registriert, die den *ThingHandler* initialisiert. Diese ist ebenfalls eine Java-Klasse, die zum Erstellen von *ThingHandler*-Instanzen dient

und ebenfalls von einer abstrakten Klasse *BaseThingHandlerFactory* von *openHAB* erbt. Jedes *Custom Binding* benötigt eine eigene Factory, die die *ThingHandler*-Objekte koordiniert. Die Implementierung der Methode *supportsThingType* prüft dabei, ob der *Thing*-Typ unterstützt wird und erstellt in *createHandler* die dafür vorgesehenen *ThingHandler*.

Fazit Integration über Custom Binding

Die Integration eines Gerätes in den *openHAB* durch die Erstellung eines *Custom Bindings* ist eine valide Option, die es erlauben würde Daten aus persistenten Verbindungen an ein *Thing* im *openHAB* zu leiten, erzwingt aber auch eine Verarbeitung der Daten innerhalb des *openHAB*-Systems. Der Ablauf unter Nutzung von *Custom Bindings* wäre, zwei *Things* anzulegen für *RotoFlex* und *SensFloor* und für diese jeweils eigene *Bindings* entwicklen. Die Daten die letztendlich im *openHAB* ankommen können dann weiterverarbeitet werden. Die Datenverarbeitung erfolgt nach Dateneingang über *Rules*, welche sogar anbieten eigene *Scripts* auszuführen. Nichtsdestotrotz ist der *openHAB* keine effektive Entwicklungsumgebung und nicht für die Erstellung von großen Projekten gedacht. Es bietet keine freie Wahl der Programmiersprache beim Schreiben von *Rules*, sondern erfordert die Nutzung einer eigenen Regelsprache, die auf *Xtend* basiert, oder wahlweise einige auf JavaScript basierende Optionen, wie *GraalVM* JavaScript für *openHAB* 3. Es beschränkt außerdem die Verwendung von Frameworks und Bibliotheken auf Open Services Gateway initiative (OSGi)-kompatible Module. Ein weiterer kritischer Nachteil dieses Vorgehens ist die Bindung an ein einziges System. Der Server, auf dem der *openHAB* aufgesetzt ist muss gleichzeitig auch die komplette Datenpipeline unterstützen. Es wäre zwar prinzipiell möglich die Verbindung zu den Geräte-APIs auf einem externen System geschehen zu lassen und die Daten über die REST-API an das *Binding* zu senden, dies erhöht aber die ohnehin schon sehr hohe Komplexität der Integration.

3.4.3 Integrationsansatz als Item über REST-API

Eine weitere Integrationsmethode ist die Anbindung des Geräts über die REST-API mit einem *Item*. Das *Item* ist dabei nicht an ein *Thing* gekoppelt, sondern existiert eigenständig im *openHAB*. Die REST-API von *openHAB* ermöglicht Zugriff auf die meisten Aspekte des Systems, einschließlich der Möglichkeit Daten und Befehle an *Items*, *Things* und *Bindings* zu senden. Eine HTTP *POST*-Anfrage kann zum Beispiel genutzt werden, um einem *Item* den Befehl zum Ausschalten zu senden (Listing 7).

```
curl -X POST \
  --header 'Content-Type: text/plain' \
  --header 'Accept: application/json' \
  -d 'OFF' \
  'http://{openHAB_IP}:8080/rest/items/ItemName'
```

Listing 7: Änderung des *Item*-Status über eine HTTP *POST*-Anfrage

Listing 7 zeigt einen client Uniform Resource Locator (cURL)-Befehl, der genutzt wird, um mit einer Schnittstelle zu kommunizieren. Der Befehl sendet eine *POST*-Anfrage an den *openHAB*. Der erste Header legt fest, dass der Inhaltstyp der Anfrage *text/plain* ist, was bedeutet, dass die übertragenen Daten als reiner Text (String) interpretiert werden sollen. Der zweite Header übermittelt dem Server, dass die Antwort im JSON-Format erwartet wird. Die letzten Teile des Befehls beinhalten die eigentlichen Daten, die übermittelt werden (*OFF*) und die Adresse des *Items*.

Für den Zugriff auf den *openHAB* über die API wird *Basic Access Authentication* und Open Authorization (OAuth) genutzt. *Basic Access Authentication* ist eine Methode für die Authentifizierung von HTTP-Transaktionen. Benutzername und Passwort werden hierbei über einen HTTP-Header übermittelt. OAuth ist ein offener Standard für Zugriffsrechte. Es werden Zugriffsberechtigungen, in der Regel über Zugriffstoken vergeben, die von dem Server ausgegeben werden. Anwender nutzen diese Zugriffstoken, um auf die geschützten Ressourcen zuzugreifen (OAuth Community (2024)).

Ein interner API-Explorer ist erreichbar über http://{openHAB_IP}:8080/developer/api-explorer. Hier ist die gesamte Auswahl der API-Fähigkeiten dokumentiert. Aus der Dokumentation wird ersichtlich, dass die API eine Schnittstelle zu allen möglichen Elementen des *openHAB* bietet und vor allem *Items* eine große Auswahl an Anfragearten unterstützen. Unter anderem lassen sich über einen *GET /items* alle verfügbaren *Items* des *openHAB* in einer JSON-Datei auflisten, über *GET /items/itemname/state* lässt sich bspw. für ein spezifisches *Item* der aktuelle Status als String wiedergeben. Essentiell für die Entwicklung der Anbindung ist jedoch die *PUT*- und *POST*-Anfrage, die die API für *Items* unterstützt. *POST /items/itemname* erlaubt es einen Befehl an ein *Item* zu senden (z. B. *ON*, *OFF*). Ähnlich dazu wird *PUT /items/itemname/state* dazu genutzt, um den Status des *Items* festzulegen. Im Gegensatz zu dem *POST*-Anfrage hat der *PUT*-Anfrage keine direkte Aktionsaufforderung. Der Unterschied kann an folgendem Beispiel verdeutlicht werden: Ist das Ziel der Anfrage eine Glühbirne auszuschalten, bietet es sich an dieser einen direkten *OFF*-Befehl über eine *POST*-Anfrage zu senden, da dieser die Glühbirne direkt auffordert sich abzuschalten. Andererseits kann die *PUT*-Anfrage eingesetzt werden, um den Status eines Pflegebetts auf ’Person ist im Bett’ oder ’Bett ist leer’ zu setzen. Dieser Status kann direkt in der Benutzeroberfläche visualisiert werden, er kann gleichzeitig aber auch als Trigger für eine *Rule* genutzt werden (z. B.: *Wenn Status von Item ’Bett’ sich ändert zu: ’Bett ist leer’*).

Diese beiden Anfragen dienen als Grundlage für die Integration. Die Verarbeitung der Daten ist flexibel. Es besteht entweder die Möglichkeit die Gerätedaten außerhalb des *openHAB* zu parsen und zu verarbeiten und lediglich fertige Befehle und Statusänderungen an die zugehörigen *Items* zu senden, oder die Rohdaten an den *openHAB* zu senden, welche mit *Rules* verknüpft werden können, die letztendlich die Verarbeitung und Steuerung übernehmen. Ein möglicher Nachteil der Integration ist die Skalierbarkeit. Obwohl REST-APIs in der Regel gut skalierbar sind, können bei einem sehr großen System mit vielen Geräten und häufigen Anfragen Leistungsprobleme auftreten, da jede Interaktion eine neue HTTP-Anfrage erfordert.

Stress Testing

Um die Skalierbarkeit des Integrationsansatzes als *Item* über REST-API zu testen dient ein Stress Test. Beim Stress Test wird die Belastung einer Anwendung unter extrembedingungen getestet. Dabei wird überprüft, wie die Anwendung einen überdurchschnittlichen Betrieb bewältigt und welche Antwortzeiten erwartet werden können. Stress Tests sind besonders aussagekräftig hinsichtlich der Skalierbarkeit und Stabilität einer API.

```
1  def stress_test(url: str, num_requests: int):
2      start_time = time.time()
3      results = []
4
5      with ThreadPoolExecutor(max_workers=5) as executor:
6          future_to_req = {executor.submit(test_api, url):
7                           i for i in range(num_requests)}
8          for future in as_completed(future_to_req):
9              result = future.result()
10             results.append(result)
11
12     duration = time.time() - start_time
13     success_count = sum(1 for result in results if result == 200)
14     failure_count = num_requests - success_count
15
16     return {
17         'total_requests': num_requests,
18         'success_count': success_count,
19         'failure_count': failure_count,
20         'duration': duration
21     }
```

Listing 8: Stress Testing der *openHAB* REST-API

Der Python-Code in Listing 8 zeigt eine Stress Test-Funktion auf einer Webressource.

Die Funktion akzeptiert zwei Parameter: die URL der zu testenden API und die Anzahl der parallel ablaufenden Anfragen. Der *ThreadPoolExecuter* aus dem Framework *concurrent.futures* wird eingesetzt, um die API-Anfragen zu parallelisieren. Für jede der Anfragen wird geprüft, ob diese erfolgreich war (Statuscode *200* als Antwort), oder ob sie fehlgeschlagen ist. Die Funktion gibt ein Dictionary zurück, mit Informationen zu den durchgeführten Anfragen, wie viele davon erfolgreich waren und wie lange die gesamte Ausführung gedauert hat. Mithilfe des Programms kann die Zuverlässigkeit der *openHAB*-API getestet werden. Die Ausführung mit 100 parallelen Anfragen ergab im Test, dass alle 100 Anfrage erfolgreich verarbeitet wurden mit einer gesamten Verarbeitungsdauer von unter fünf Sekunden.

Fazit Integration über REST-API

Das Ergebnis verdeutlicht, dass die API sehr stabil ist und sich für die Verarbeitung von großen Anfragemengen eignet. Die Verarbeitungsdauer von rund fünf Sekunden kann jedoch für manche Anwendungen zu langsam sein. So eine hohe Menge an parallelen Anfragen ist jedoch nicht zu erwarten, weshalb sich schlussfolgern lässt, dass die Datenübertragung mithilfe der REST-API von *openHAB* sich für die Anbindung von Geräten eignet.

3.4.4 Schlussfolgerung zu den Integrationsmethoden

Die Diskussion um die effektivste Integrationsmethode von Geräten in den *openHAB* zeigt, dass die Auswahl stark von den spezifischen Anforderungen des Projekts anhängig ist. Das *Exec Binding* ist besonders geeignet für regelmäßige, einfache Arbeiten, stößt bei kontinuierlicher Datenübertragung und persistenten Verbindungen jedoch an seine Grenzen und verursacht folglich potentiellen Datenverlust.

Die Erstellung von *Custom Bindings* stellt eine weitere Möglichkeit dar. *Custom Bindings* bieten tiefe Integrationen in das *openHAB*-System, erfordern jedoch umfangreiche Entwicklungsarbeit besonders in komplexeren Geräteintegrationen. Die Bindung an den *openHAB* limitiert die Auswahl der Entwicklungswerkzeuge und bietet keine einfache Möglichkeit der Trennung von Datenverarbeitung und *openHAB*-spezifischen Steuerung.

Die Wahl, Daten extern zu verarbeiten und nur Endbefehle und Statusupdates zu übermitteln ist eine mögliche Lösung, um eine effiziente Datenverarbeitung und Übermittlung zu gewährleisten. Der flexible Aufbau bietet völlige Entwicklungs- und Werkzeugfreiheit. Der erfolgreiche Stress Test der REST-API beweist die Zuverlässigkeit und Leistungsfähigkeit unter hoher Last.

Für die Entwicklung der Integration von *RotoFlex* und *SensFloor* an den *openHAB* fällt aus den oben genannten Gründen die Entscheidung auf die REST-API-Methode. Die hohe Datenkomplexität und die asynchrone Kommunikation erfordern eine durchdachte und

aufwendige Vorverarbeitung der Daten, die außerhalb des *openHAB* effektiver realisierbar ist. Die Entscheidung ist jedoch anwendungsspezifisch getroffen und ist auf keinen Fall universell auf andere Projekte übertragbar. Die ausführliche Analyse der einzelnen Integrationsmethoden geht detailliert auf die Vor- und Nachteile ein und dient als Entscheidungshilfe für kommende Projekte.

3.5 Detaillierte Analyse der Integration über REST-API

3.5.1 Aufteilung in Arbeitsschritte

Prinzipiell lässt sich die Anbindung eines externen Systems an den *openHAB* unter Nutzung der REST-API in vier Schritte unterteilen, deren Umsetzung ausgearbeitet werden muss:

1. Empfangen von Daten des externen Systems
2. Parsen der Daten in von *openHAB* unterstützte Formate
3. Übermittlung der Daten an *openHAB*
4. Darstellung des Status in der Benutzeroberfläche

Diese vier Schritte bilden die Grundlage für die Entwicklung jeder Anbindung im *openHAB*. Da es eine Vielzahl von verschiedenen Voraussetzungen und Umsetzungsmöglichkeiten gibt, ist es von großer Wichtigkeit, einen Weg zu finden, der möglichst universell anwendbar ist. Allein am Beispiel von *RotoFlex* und *SensFloor* erkennt man, dass sich nicht nur die Kommunikation mit den Geräten selbst unterscheidet (Socket.IO und *WebSocket*), sondern auch das Format der Daten, die von den APIs geliefert werden. Das bedeutet, dass Schritt 1 und Schritt 2 der Entwicklung individuelle Lösungen erfordern werden. Die Priorität liegt aber nicht nur darin, eine funktionierende Lösung zu entwickeln, sondern vor allem eine Grundlage zu schaffen für die Anbindung weiterer Systeme an den *openHAB*.

3.5.2 Struktur der SensFloor-Daten

Im ersten Schritt müssen die aus der API empfangenen Daten nach relevanten Informationen gefiltert werden. Diese können, je nach *Event*, bspw. aktive Alarme, Raumzustände, etc. sein. Weil die *Items* des *openHAB* keine komplexen Datentypen unterstützen, müssen die API Informationen in den meisten Fällen vor-verarbeitet werden, da diese bei den meisten *Events* Daten als Array von JSON-artigen Informationen liefern.

Um den *SensFloor* abbilden zu können, benötigt das System mehrere Informationen. Für jede Zustandsänderung muss der betroffene Raum sowie der neue Zustand vorliegen.

Eine initiale Untersuchung der Daten lässt annehmen, dass Strings die einzige Möglichkeit sein werden, die Daten im *openHAB* zu speichern, da Tupel, Arrays oder ähnliche Datentypen nicht unterstützt werden und Strings somit die einzige Möglichkeit bieten mehrere Informationen auf einmal zu versenden. Die einmalige Datenübermittlung ist in Ablaufdiagramm 3.5 demonstriert.

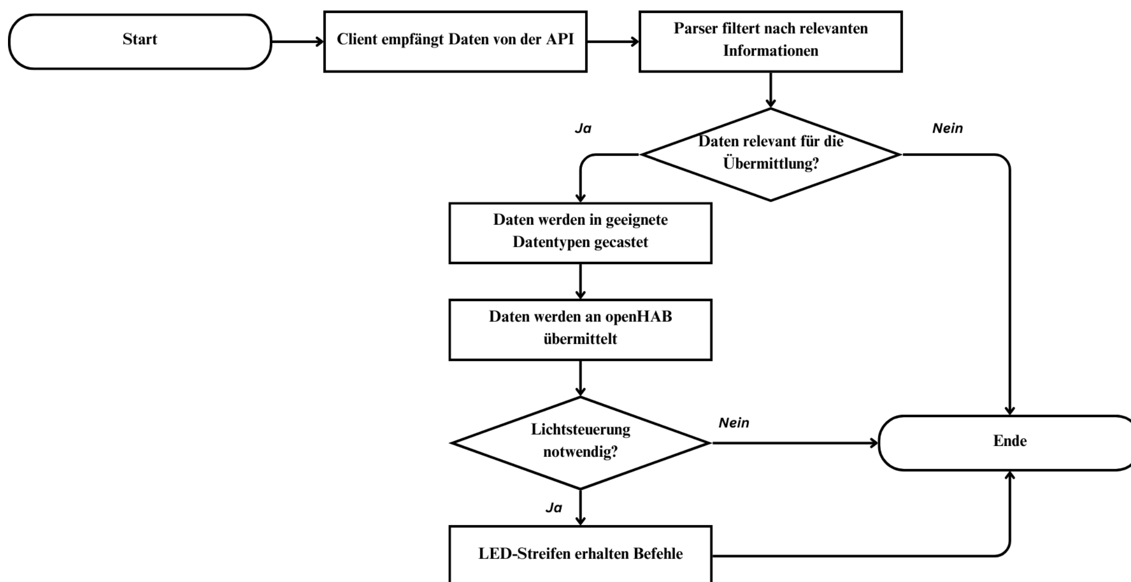

Abbildung 3.5: Datenübermittlung von Empfang bis Weiterleitung

Abbildung 3.5 stellt keinesfalls den gesamten Programmablauf dar, sondern ist explizit dafür gedacht einen einzelnen Zyklus beispielhaft zu veranschaulichen. Ob die *Rules* im *openHAB* letztendlich das Licht steuern und die Daten verarbeiten werden ist bisher nicht festgelegt. Dieser Schritt kann je nach Anforderung auch vor der Datenübermittlung geschehen.

3.5.3 Struktur der RotoFlex-Daten

Prinzipiell ist für eine Repräsentation des Pflegebetts im *openHAB* nur die Information relevant, ob sich jemand im Bett befindet oder nicht. Dies kann entweder in Form eines binären Wertes geschehen (*im Bett* und *nicht im Bett*) oder aber als Zahlenwert, der das aktuelle Gewicht anzeigt, welches vom Bett erfasst wurde. Beide Varianten erfordern lediglich einen Wert. Da *openHAB* keinen Boolean unterstützt, fällt die Entscheidung auf String für die binäre Darstellung, bzw. auf Number für die Anzeige des Gewichtes.

3.6 Praktische Analysen

Vor der Entwicklung der Anbindungen von *RotoFlex*, *SensFloor* und der Entwicklung des automatisierten Beleuchtungssystems, ist es wichtig einige praktische Tests durchzuführen. Die Schlussfolgerungen der Tests spielt eine maßgebliche Rolle in der Entwicklung.

3.6.1 Analyse der Datenintegrität von SensFloor

Die *SensFloor*-API bietet eine Vielzahl unterschiedlicher *Event*-Typen (siehe 3.2.1). Abgesehen davon, dass nicht alle *Event*-Typen für die Entwicklung relevante Daten liefern, muss getestet werden, wie zuverlässig die Daten gesendet werden. Der Test soll dazu dienen festzustellen, ob es ausreichend ist, sich auf einen *Event*-Typen zu verlassen, oder ob mehrere *Event*-Typen einbezogen werden müssen, um eine genaue Aussage zu treffen. Besonders bei den Alarmen ist es wichtig zu prüfen, ob diese akkurat ausgelöst werden. Das bereitgestellte Python-Skript für die Verbindung mit der *SensFloor*-API (siehe 2.5.5) wird für den Test um einige Funktionen erweitert. Am Beispiel des Alarms *alarms-active* wird verdeutlicht, wie die Tests für die einzelnen *Event*-Typen aussieht (Listing 9).

```
1  sio = socketio.Client()
2
3  @sio.on('alarms-active')
4  def on_alarms_active(data):
5      for obj in data:
6          print(obj)
7
8  sio.on('alarms-active', on_alarms_active)
9  sio.connect('http://192.168.172.22:8000')
10 sio.wait()
```

Listing 9: SensFloor-API Datentest Beispiel

Der Dekorator @*sio.on('alarms-active')* wird verwendet, um die Funktion *on_alarms_active* als *Eventhandler* für das *Event alarms-active* zu registrieren. Das bedeutet, dass jedes mal, wenn das *Event* von der *SensFloor*-API gesendet wird, darauf folgend die Funktion mit dem Parameter *data* - also den Daten des *Events* - aufgerufen wird. Die Funktion iteriert über alle Objekte der Liste *data*. Für den Test werden vorerst die Daten in die Konsole ausgegeben. Mit Start des Programms bewegt sich ein Proband durch die Wohnung. Die ausgelösten *Events* werden aufgezeichnet und ausgewertet.

Der Test aus Listing 9 wird für weitere *Event*-Typen wiederholt. Insbesondere interessant sind die *Events alarms-active* und *alarms-changed* für die Nutzung der Alarme des

SensFloor, sowie *clusters-update* und *objects-update* für eine Aussage über die Sensorzustände ohne Vorverarbeitung (Rohdaten).

Schlussfolgerung über die Event-Typen

Die Tests haben folgendes ergeben: Die API reagiert außerordentlich schnell auf Aktionen auf dem *SensFloor*. Die Frequenz der synchronen *Events* ist hoch genug, dass Aktionen, wie Raumwechsel, schnell von der API gesendet werden.

Während *alarms-active* synchrone Nachrichten sendet mit allen Alarmen, die zum aktuellen Zeitpunkt aktiv sind, liefert *alarms-changed* nur Alarmänderungen (aktiv zu inaktiv, oder inaktiv zu aktiv). Obwohl sie fast identische Daten liefern, ist im Test aufgefallen, dass die zwei *Event*-Typen nicht austauschbar sind. Während *alarms-active* kontinuierlich alle aktiven Alarme in festen Zeitintervallen sendet, reagiert *alarms-changed* nur auf Änderungen, ist also deutlich ressourcensparender. Prinzipiell lässt sich vermuten, dass die Änderungen der Alarme ausreichend ist, um alle Zustände in der Wohnung abbilden zu können. Das *Item* im *openHAB* wechselt den Zustand auch nur, wenn es Änderungen am Sensorzustand gab. Es gibt keinen Grund Anfragen an das *Item* zu senden, wenn der Zustand gleich bleibt. Jedoch hat der Test gezeigt, dass *alarms-changed* bei vielen, schnell aufeinander folgenden Alarmwechseln nicht alle Alarme sendet. Dies kann zu Problemen führen, wie die nicht-Erkennung von Raumwechseln oder Stürzen. Außerdem wird deutlich gezeigt, dass nicht alle Alarme zuverlässig erkannt werden. Während Präsenzalarme und Stürze in allen Testversuchen korrekt interpretiert wurden, kam es bei der Erkennung von *Room In/-Out*, *Bed In/-Out*, *Toilet In/-Out*, etc. sehr häufig zu Fehlern. All die Alarme, die nur von bestimmten Clustern ausgelöst werden, eignen sich nicht für eine zuverlässige Zustandsaussage. Im Idealfall verlässt eine Person einen Raum immer gleich und die Bereiche für die Alarmaktivierungen von bspw. *Room In/-Out* lassen sich genau kalibrieren. Dies ist aber in der Praxis nicht der Fall. Schon allein die Geschwindigkeit, mit der die Person den Raum verlässt, verändert die Schrittlänge wesentlich. Insgesamt haben die Tests der Alarm-*Events* gezeigt, dass Alarme mit großflächigem Auslöser (Raumpräsenz, Sturz) mit hoher Sicherheit für die Auswertung des *SensFloor* genutzt werden können, wohingegen Alarme mit kleinen Auslösungsflächen (*Room in*, *Bed in*, etc.) keine valide Option für die Abbildung des *SensFloor* darstellen.

Als nächstes werden die Testergebnisse von *clusters-update* und *objects-update* betrachtet. Diese sind im Gegensatz zu den Alarm-*Events* nicht verarbeitet und es liegt kein Interpretationsalgorithmus dahinter, der fehlerhafte Ergebnisse senden kann. Dementsprechend zeigen die Testergebnisse keine Fehler. Was bei den *Events* aber nicht geboten wird ist eine Raumzuordnung. Es ist also erforderlich die Räume der Wohnung auf ein Koordinatensystem zu projizieren (mappen), um festzustellen, in welchem Raum das *Event* ausgelöst wurde (Abb. 3.6).

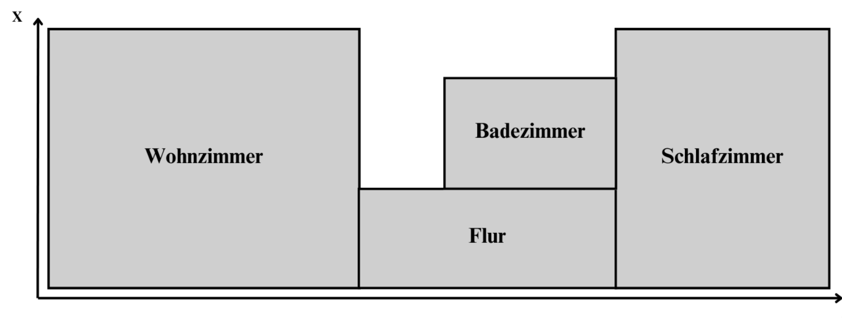

Abbildung 3.6: Grundriss der Laborwohnung auf ein Koordinatensystem projiziert

Aus den Tests schlussfolgernd, scheint *alarms-active* der zuverlässigste und unkomplizierteste Weg zu sein Raumpräsenzen und Stürze festzustellen. Die Datenintegrität von *alarms-changed* reicht nicht aus, um den ressourcensparenden, asynchronen *Event*-Typ zu nutzen. Die nicht-verarbeiteten Typen *clusters-update* und *objects-update* liefern zwar fehlerfreie und zuverlässige Ergebnisse, bieten gegenüber *alarms-active* aber für die spezifische Anwendung (Zustandsabbildung im *openHAB*) aber keinen aus dem Test hervorgehenden Vorteil. Wird bei der Entwicklung festgestellt, dass eine detailliertere Analyse der Sensordaten erforderlich ist, wird auf diese beiden *Event*-Typen zurückgegriffen werden.

3.6.2 Testing der openHAB Rules

Thread Test

Um die Entscheidung treffen zu können, wie viel der Datenverarbeitung über *Rules* im *openHAB* oder extern in einem Python Skript ablaufen wird, muss die Leistung von *Rules* getestet werden. Eine wichtiger Test ist dabei die Überprüfung, ob *openHAB* in der Lage ist parallel mehrere *Rules* zu starten, oder ob *Rules* sich gegenseitig blockieren. Dafür werden mehrere *Rules* erstellt, die den gleichen Trigger besitzen. Als Trigger wird eine bestimmte Systemzeit genutzt. Alle *Rules* sollten demnach zum genau gleichen Zeitpunkt ausgelöst werden und parallel arbeiten.

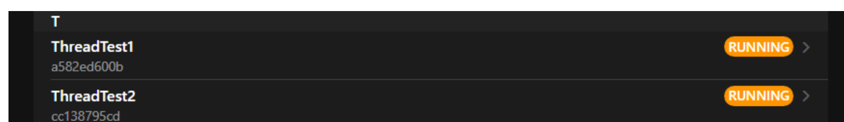

Abbildung 3.7: openHAB *Rule* Ansicht
openHAB Foundation 2024a

Abbildung 3.7 zeigt eine Bildschirmaufnahme aus dem *openHAB Rule-Interface*. Da beide *Rules* gleichzeitig den Status *RUNNING* besitzen zeigt dies, dass beide *Rules* gleichzeitig gestartet und ausgeführt werden können. Die *Rules* loggen zu Beginn die Info-Nachricht

'ThreadTestX started', warten 1000 ms und loggen abschließend die Nachricht 'ThreadTestX finished'.

```
==> /var/log/openhab/openhab.log <==
2024-01-23 12:24:01.321 [INFO] [org.openhab.rule.cc138795cd] -
ThreadTest2 started
2024-01-23 12:24:01.865 [INFO] [org.openhab.rule.a582ed600b] -
ThreadTest1 started
2024-01-23 12:24:02.393 [INFO] [org.openhab.rule.cc138795cd] -
ThreadTest2 finished
2024-01-23 12:24:02.916 [INFO] [org.openhab.rule.a582ed600b] -
ThreadTest1 finished
```

Listing 10: Ausgabe des *openhab-cli showlogs*-Befehls des Thread Tests

Aus der Log-Ausgabe 10 wird ersichtlich, dass beide *Rules* mit einem Versatz von 544 ms die erste Log-Meldung ausgeben, danach beide parallel laufen und mit einem Versatz von 523 ms mit der zweiten Log-Meldung abschließen. Damit ist bewiesen, dass *Rules* im *openHAB* parallel ausgeführt werden und damit eine valide Option für die Datenauswertung bieten, da diese während der Ausführung keine anderen *Rules* blockieren.

Performance Test

Für einen Vergleich der Leistung zwischen einem Python Skript und einem *openHAB* Skript wird eine einfache Berechnung in eine Schleife gesetzt und die Laufzeit für die *Rule* und das Skript gemessen. Beide Programme initialisieren eine Variable *sum* mit 1 und starten eine Schleife, die 1000 mal *sum* mit 1 addiert (siehe Anhang A, Listing 43 & Listing 44). Die Log-Ausgaben (Listing 11) zeigen eine Ausführungsdauer von ca. 34 ms für die *Rule* und ca. 0,35 ms für das Python Skript.

```
[openhab rule]
2024-01-26 19:45:32.240 [INFO] [org.openhab.rule.521d605e2c] -
PerformanceTest started
2024-01-26 19:45:32.274 [INFO] [org.openhab.rule.521d605e2c] -
PerformanceTest finished

[python script]
1713870472.660718 - PerformanceTest started
1713870472.661067 - PerformanceTest finished
```

Listing 11: Ausgabe des *openhab-cli showlogs*-Befehls des Performance Tests

Setzt man die Programme in eine übergeordnete Schleife, die diese mehrmals hintereinander startet und am Ende einen Durchschnittswert der Ausführungszeiten bildet, so wird deutlich, dass dieses Ergebnis kein Ausnahmefall ist, sondern, dass der *openHAB* Berechnungen weitaus langsamer ausführt. Aus diesem Grund ist es wichtig, große und rechenintensive Arbeiten außerhalb des *openHAB* durchführen zu lassen.

3.7 Nächtliches Beleuchtungssystem

Der folgende Abschnitt befasst sich mit der Vorbereitung der Entwicklung des nächtlichen Beleuchtungssystems. Ziel des Systems wird es sein basierend auf den *SensFloor*- und optional den *RotoFlex*-Daten ein automatisiertes Beleuchtungssystems für die Nacht zu schaffen. Dabei stehen bereits an den *openHAB* angebundene LED-Streifen zur Verfügung, die auf Kniehöhe entlang der Wände platziert sind. Dabei gibt es vier LED-Streifen im Wohnzimmer, zwei im Flur und einen im Schlafzimmer. Die LED-Streifen sind nicht adressierbar, das bedeutet, dass es nicht möglich ist eine Gruppe von LEDs innerhalb eines Streifens individuell zu kontrollieren. Es kann nur der gesamte LED-Streifen an- und ausgeschaltet werden. Weil es sich um Red, Green, Blue (RGB) LEDs handelt, ist es möglich die Helligkeit und Farbe der LEDs einzustellen. Jeder LED-Streifen ist als *Thing* im *openHAB* hinterlegt und ist mit fünf *Channels* an *Items* verknüpft (Tab. 3.1).

Tabelle 3.1: *Items* der LED-Streifen im *openHAB*
openHAB Foundation 2024b

Item	Itemtyp	Beschreibung
Farbe	Color	Steuert die Farbe, die Helligkeit und schaltet das Licht ein und aus.
Farbtemperatur	Number	Steuert die Farbtemperatur des Lichts (in Kelvin).
Dauer	Number:Time	Steuert die Dauer, welche die Lampe eingeschaltet bleibt, bevor sie automatisch ausgeschaltet wird (0=für immer an).
Alert	String	Befehl zum An- und Ausschalten
Farbeffekt	String	Von der Lampe unterstützer Farbeffekt (bspw. *colorloop*) (*deCONZ REST API* (o. D.))

Diese *Items* können sowohl über *Rules*, als auch über die REST-API angesprochen werden. Die gesamte Steuerung der LED-Streifen kann also intern oder extern erfolgen.

3.7.1 Grundkonzept

Das Beleuchtungssystem ist in separate Aufgaben aufteilbar. Initial muss es in der Lage sein selbstständig Informationen zu existierenden LED-Streifen zu erhalten. Dies ist besonders wichtig, um die Modularität zu gewährleisten. Sollten bspw. in Zukunft neue LED-Streifen hinzugefügt werden, soll das System mit möglichst geringem Aufwand anpassbar sein. Des weiteren muss das System in der Lage sein die Helligkeit zu bestimmen und zu entscheiden, ob es dunkel genug ist, um das Licht einzuschalten. Tagsüber soll das Beleuchtungssystem

inaktiv sein, um Energie zu sparen. Als letztes soll das System auf eingehende Daten reagieren und die richtigen LED-Streifen einschalten. Es sollen nur die Streifen in dem Raum eingeschaltet werden, in dem sich auch eine Person befindet.

Aus den Tests (siehe 3.4.3 und 3.6.2) geht hervor, dass während die REST-API von *openHAB* hohe Anfragefrequenzen verarbeiten kann, die *Rules* schlecht für rechenintensive Aufgaben geeignet sind. Deshalb ist es effektiver die LED-Streifen über die REST-API zu steuern. Selbst bei sehr schnellen Raumänderungen rechnet das System mit einer Anfragefrequenz von weit unter 100 Anfragen gleichzeitig. Dennoch ist es wichtig Netzwerkressourcen zu sparen und unnötige Anfragen zu reduzieren. Das macht das System nicht nur performanter, sondern sichert auch die Skalierbarkeit. Aus diesem Grund muss das System unterscheiden können, ob es notwendig ist Anfragen an die LED-Streifen zu senden, oder nicht. Wenn eine Person sich eine längere Zeit in einem Raum aufhält, sendet die API des *SensFloor* kontinuierlich Informationen darüber, dass eine Präsenz in dem Raum erkannt wird. In diesem Fall müssen Anfragen an die LED-Streifen aber nicht mit der gleichen Frequenz gesendet werden, wie die Daten des *SensFloor* eingehen. Es reicht eine Anfrage zu Beginn zu senden, die den Streifen einschaltet und zu warten, bis der Zustand sich wieder ändert.

Abschließend ist es wichtig, dass die LED-Streifen sich bei Verlassen des Raumes nicht sofort abschalten, sondern der Person noch eine gewisse Zeit lassen. Das fördert die Sicherheit, indem der vorherige Raum auch bei Verlassen beleuchtet bleibt. Es ist dabei jedoch nicht nötig die LED-Streifen bei voller Helligkeit zu lassen. Es reicht aus, die LED-Streifen des verlassenen Raumes 5-10 s lang bei niedriger Helligkeit anzulassen, bevor das System sie abschaltet. Weil dies ein paralleler Vorgang ist, der bei Wieder-betreten des Raumes abgebrochen werden kann, erfordert die Dimm-Logik einen eigenen Thread. Aus folgenden zwei Gründen ist es möglich diesen Teil des Systems in eine *openHAB Rule* auszulagern: Zum einen ist die Berechnung der Dimm-Logik relativ einfach. Zum anderen ist eine Verzögerung der *Rule*-ausführung unproblematisch, sollte es doch zu Ressourcenknappheit kommen. Ob das Licht 5 s oder 6 s zum Abschalten benötigt verursacht letztendlich einen kaum wahrnehmbaren Unterschied. Die Dimm-Logik auf eine *Rule* auszulagern ist außerdem vorteilhaft, weil dieser Programmteil keinen direkten Zugriff auf *SensFloor*- und *RotoFlex* Daten benötigt und durch die *openHAB*-interne Verarbeitung weitere API-Anfragen spart.

3.7.2 Ähnliche Systeme

Eine Studie von Cheng et al. (2020) beschreibt die Entwicklung eines intelligenten Beleuchtungssystems, welches auf einem drahtlosen Sensornetzwerk aufbaut, um den Energieverbrauch in Gebäuden zu reduzieren. Die Nutzung von passiven Infrarot- und Mikrowellen-Doppler-Sensoren, sowie Umgebungslichtsensoren ermöglichen es dem System eine Aussage über das benötigte Beleuchtungsmaß zu treffen und passen die Dimmlevel der Leuchten automatisch an. Das System reagiert dynamisch auf die Änderung der Raumbelegung und Helligkeit. Auch, wenn diese Arbeit nicht im Zusammenhang mit AAL entwickelt wurde, unterstreicht sie doch die Signifikanz von automatisierten Be-

leuchtungssystemen zum Einsparen von Energie, was schlussendlich auch eines der Ziele von AAL ist.

3.7.3 Feststellung von Nachteintritt

Die Laborwohnung verfügt über keine Lichtsensoren, die dafür genutzt werden können, die Helligkeit in der Wohnung festzustellen. Deshalb muss eine andere Methode genutzt werden, die für einen Standort aktuelle Helligkeitswerte angeben kann. Dafür muss die Methode für eine beliebige Orts- und Zeitangabe den Sonnenstand ermitteln können, aus dem errechnet werden kann, ob die aktuelle Uhrzeit zwischen Abenddämmerung und Morgengrauen liegt.

3.7.4 Steuerung der LED-Streifen

Das ein- und ausschalten der LED-Streifen erfolgt über die REST-API. Das Dimmen des Lichtes für einige Sekunden, bevor es ausgeht geschieht über *Rules* im *openHAB*, um weitere API-Anfragen zu vermeiden. Eine Alternative dazu wäre es eine API-Anfrage an das Dauer-*Item* (Tab. 3.1) zu senden. Um das Licht gleichzeitig einzuschalten und ihm einen Farbwert zu geben kann eine *POST*-Anfrage an das *Item* gesendet werden (Listing 12).

```
1 curl -X POST \
2   --header 'Content-Type: text/plain' \
3   --header 'Accept: application/json' \
4   -d '20,100,100' \
5   'http://{openHAB_IP}:8080/rest/items/FarbitemName'
```

Listing 12: *POST*-Anfrage an das Farbe-*Item* eines LED-Streifens

Philips Hues arbeiten mit Hue, Saturation, Value (HSV) Farbraum (Ruettgers (2022)). Der gesendete Farbwert *20,100,100* ist demnach ein HSV-Farbwert für die Farbe rot. HSV-Farbwerte beinhalten die drei Parameter für Hue, Saturation und Value (Farbton, Sättigung, Helligkeit) (Wikipedia (2024)). Es bietet sich deshalb an das Farbe-*Item* für die Steuerung der LED-Streifen zu nutzen, da man gleichzeitig Helligkeit und Farbe bestimmen kann.

Eine Studie von Plitnick et al. (2010) zeigt, dass blaues Licht die Müdigkeit der Probanden in der Nacht reduzierte. Die Ergebnisse der Studie haben zwar nur einen leichten Einfluss der Lichtfarbe gezeigt, dennoch wurde die Farbe Rot gewählt, um den Schlaf der Personen möglichst wenig einzuschränken. Abschließend benötigt die Dimmer-*Rule* einen Auslöser. Ein möglicher Auslöser für das Dimmen der LED-Streifen per *Rule* könnte die Verminderung der Helligkeitswerts sein (z. B.: *Wenn 'LED-Streifen_Farbe' den Befehl '20,100,10' erhält*). Es ist dabei irrelevant wie viele LED-Streifen der Raum enthält, da jeder einzelne den selben Befehl erhält.

3.8 Programmbeschreibung

3.8.1 Item-Erstellung

Zu Beginn werden die *Items* im *openHAB* angelegt, die später die Sensordaten erhalten werden. Die Anleitung zu Erstellung eines *Items* in *openHAB* ist unter *Items* (o. D.) dokumentiert. Der folgende Abschnitt bezieht sich auf Informationen aus der *openHAB*-Dokumentation.

Die Erstellung der *Items* kann entweder über die Main UI oder über die Konfiguration einer Textdatei erfolgen. Soll eine Textdatei zur Konfiguration verwendet werden, wird eine *.items*-Datei im Verzeichnis *conf/items* in der *openHAB*-Installation erstellt oder die bestehende Datei geöffnet. Dort wird ein *Item* in Textform definiert. Über die Main UI wird unter dem Reiter *Einstellungen* das Menü *Items* ausgewählt. Über die Schaltfläche + (*Add*), wird ein neues *Item* erstellt.

Für beide Varianten sind die Konfigurationsfelder gleich. Der Name ist ein eindeutiger Bezeichner für das *Item*. Während der Name bspw. keine Leerzeichen enthalten soll, kann optional ein *Label* erstellt werden, welches eine leserlichere *Item*-Bezeichnung trägt. Das Feld *Typ* wählt den Datentyp des *Items* (siehe 2.4.3). Tags können optional hinzugefügt werden, um das *Item* einfacher im System auffindbar zu machen. Die *Group-Membership* definiert die Zugehörigkeit zu einer *Group* und als letztes kann über Semantic Class dem *Item* eine *openHAB*-interne Funktion zugeordnet werden. Die Erstellung der *SensFloor-Items* und des *RotoFlex-Items* ist ähnlich. Alle *Items* sind vom Typ String und sind semantisch im *openHAB* als *Equipment_Sensor* angelegt. Außerdem haben alle benötigten *Items* den Raum in dem sie sich befinden als übergeordnete *Group*. Die Unterscheidungen liegen lediglich in der Bezeichnung.

Der *SensFloor*, obwohl es sich um ein Gerät handelt, benötigt separate *Items* für jeden Raum. Die Ursache dafür ist, dass *openHAB* keine Möglichkeit anbietet Listen-Typen zu erstellen, die mehrere Daten kapseln können (siehe 2.4.3). Aus diesem Grund muss jeder Raum eine eigene *SensFloor*-Repräsentation im *openHAB* besitzen. Vor allem, wenn sich mehrere Personen in einer Wohnung aufhalten, ist es essentiell jeden Raum einzeln abbilden zu können. Kann man garantieren, dass die Wohnung ausschließlich von einer Person genutzt wird, reicht ein *Item*. Das System würde dann so konfiguriert werden,

dass das *Item* Statusmeldungen anzeigt, wie ’Präsenz im Schlafzimmer’. Stattdessen, um die Anwendung skalierbarer zu gestalten, werden mehrere *Items* für den *SensFloor* erstellt, welche jeweils nur Statusmeldungen für den zugehörigen Raum anzeigen (z. B.: ’SensFloorBedRoom: Person im Raum; SensFloorLivingRoom: Person im Raum; SensFloorHall: Raum leer’).

3.8.2 Main-Funktion

Im Mittelpunkt der Steuerlogik steht die *main.py*, eine Python-Datei, welche ausgeführt wird, um das Programm zu starten. Diese Datei beinhaltet eine *main*-Funktion, die vom Python-Interpreter bei Programmstart automatisch ausgeführt wird. Die *main*-Funktion hat nur eine Aufgabe: Sie nutzt die *Asyncio*-Bibliothek von Python, um die *run_tasks*-Funktion auszuführen. *Asyncio* ist eine Python-Bibliothek, die verwendet wird um asynchrone Aufgaben zu starten. Es verwendet die Schlüsselwörter *await* und *async*, um asynchrone Funktionen zu definieren. Diese Aufgaben werden in sogenannten *Coroutinen* ausgeführt, weshalb sie nicht-blockierend für den Haupt-*Event-Loop* sind (Python Software Foundation (2024a)).

Die *main.py*-Datei beinhaltet zwei wichtige Definitionen. Zum einen wird eine Datenklasse *Equipment* festgelegt, welche alle für ein neues *Equipment* relevanten Abhängigkeiten speichert. (Listing 13)

```
1  @dataclass
2  class Equipment:
3      queue: Queue
4      connect_method: callable
5      data: Any = None
6      prev_data: Any = None
7      name: str = ''
8      async def start_connection(self):
9          try:
10             await self.connect_method(self.queue, self.name)
11         except Exception as e:
12             logger.error(f'Error in {self.name} connection: {e}', e)
```

Listing 13: *Equipment*-Klasse

Wie in Python Software Foundation (2024b) beschrieben, wird der Dekorator @*dataclass* aus dem *dataclasses*-Modul verwendet, um die Lesbarkeit der Klasse zu verbessern. Methoden, wie die Konstruktormethode __*init*__, werden automatisch hinzugefügt, ohne manuell definiert zu werden und basieren auf den angegebenen Feldern der Klasse. Die Felder sind Bestandteile einer Datenklasse.

Das *queue*-Feld bekommt bei Initialisierung des Objektes eine Instanz von Queue. Queues werden in Python verwendet, um Daten zwischen verschiedenen Teilen der Anwendung asynchron auszutauschen. Das Feld *connect_method* ist eine aufrufbare Methode. Das schafft eine universelle Einsetzbarkeit der *Equipment*-Klasse für jedes Gerät, welches an den *openHAB* angebunden werden muss, weil es den Verbindungsweg zum Gerät selbst von der *Equipment*-Klasse trennt. Dieser kann in einer separaten Verbindungsfunktion für jedes *Equipment* individuell definiert werden. Die Felder *data* und *prev_data* sind optionale Datenhalter, die mit *None* initialisiert wird. Der Typehint *Any* erlaubt es dem Datenhalter jede beliebige Art von Daten zu speichern, je nachdem, was für die Verarbeitung der Gerätedaten am besten geeignet ist.

Während *data* die aktuellen Daten des Gerätes speichert, beinhaltet *prev_data* die vorhergehenden Daten. Dies ist notwendig, um Zustandsänderungen über Iterationen hinweg festzustellen. Das letzte Feld *name* ist lediglich für die eindeutige Identifizierung des Gerätes zuständig. Es ist wichtig eindeutige Namen zu vergeben, um die Lesbarkeit und Skalierbarkeit des Programms beizubehalten.

Außerdem besitzt die Datenklasse eine asnychrone Methode *start_connection*. Es wird eine Methode innerhalb der Klasse definiert, die die *Equipment*-spezifische Verbindungsmethode aufruft und mögliche Fehler abfängt und diese loggt. Die Funktionsweise des logger wird in 3.8.3 erläutert. Ebenfalls wird der Aufbau der Verbindungsmethoden von *SensFloor* und *RotoFlex* in 3.8.4 und 3.8.5 detailliert erklärt. Der in Listing 14 aufgeführte Programmcode ist ein Ausschnitt aus der gesamten Funktion und dient der Verdeutlichung der Funktionsweise.

Die *run_tasks*-Funktion (Listing 14) ist der zweite wichtige Bestandteil der *main.py*.

```
async def run_tasks():
    openhab_connection_handler = OpenHABConnectionHandler()
    sensfloor_monitor_livingroom = SensFloorMonitor()

    sensors: dict[str, Equipment] = {
        'LivingRoom': Equipment(
            queue=Queue(),
            connect_method=
            sensfloor_monitor_livingroom.connect_and_receive_sensfloor_data,
            data=('Bootup SensFloor', False),
            prev_data=('', False),
            name='LivingRoom'),
        'RotoFlex': Equipment(
            queue=Queue(),
            connect_method=connect_and_receive_rotoflex_data,
            data='Bootup RotoFlex',
            prev_data='',
            name='RotoFlex')
    }
    tasks = [sensor.start_connection() for sensor in sensors.values()]
    await asyncio.gather(*tasks)
    await night_light.evaluate_data(sensors)
```

Listing 14: Ausschnitt der run_tasks-Funktion

Die gesamte Funktion befindet sich in Anhang A, Listing 45. Zuerst wird eine Instanz der *OpenHABConnectionHandler*-Klasse erstellt, also die Klasse, die das Beleuchtungssystem und die
openHAB-Anbindung übernimmt. Es wird außerdem ein Objekt der Klasse *SensFloor-Monitor* erstellt. Das Objekt ist verantwortlich für die Verbindung zur *SensFloor*-API und die Auswertung der Daten. Die Funktion beinhaltet identisch zu der Erstellung des Wohnzimmer-*SensFloor* auch die Initialisierung von Schlafzimmer und Flur. Weil diese sich - außer vom Namen - nicht vom Wohnzimmer unterscheiden, wurden sie für die Vereinfachung in Listing 14 ausgelassen. Die Notwendigkeit der Erstellung individueller *Sens-Floor*-Verbindungen für jeden Raum wird ebenfalls in 3.8.5 erläutert. Für jedes Gerät wird eine asynchrone Aufgabe (*task*) erzeugt, die den Verbindungsaufbau (*start_connection*) initiiert. Mithilfe von *asyncio.gather*, werden diese Funktionen parallel ausgeführt. Die parallele Ausführung ist deshalb wichtig, weil die einzelnen Verbindungen unabhängig voneinander Sensordaten senden und diese gleichzeitig empfangen und verarbeiten müssen. Bspw. wenn eine Person sich im Schlaf- und eine sich im Wohnzimmer befindet, ist es notwendig diese parallel verarbeiten zu können. Die *evaluate_data*-Methode erhält alle *Equipments* und übernimmt die *openHAB*-Anbindung.

3.8.3 Logger

Das Logger-Modul (siehe Anhang A, Listing 46) beinhaltet eine zentralisierte Protokollierungslösung. Der Logger wird als Singleton initialisiert, was gewährleistet, dass in der gesamten Anwendung nur eine Instanz des Logger-Objektes verwendet wird. Der Logger ermöglicht das Protokollieren von Info-, Warnungs- und Fehler-Meldungen in die Konsole und eine dedizierte Datei. Während Info-Meldungen in der Regel eine Übersicht über den aktuellen Programmablauf bieten, sind Warnungs- und Fehlermeldungen dafür da, um nachträglich über fehlgeschlagene Aktionen zu informieren. Das Speichern der Log-Meldungen in eine Datei hilft außerdem auch nach Programmabschluss noch den Programmdurchlauf nachzuvollziehen und ggf. eine Fehleranalyse zu betreiben.

Verwendung des Loggers

Für die Verwendung des Logger wird eine Instanz der Logger-Klasse erstellt. Diese erhält einen Dateinamen als Parameter, in welche später die Nachrichten gespeichert werden. Info- und Warnungs-Meldungen erhalten einen String als Parameter. Die Error-Meldung hingegen erhält zusätzlich eine Exception, damit der von dem Python-Interpreter aufgerufene Fehler ebenfalls gespeichert werden kann. (Listing 15)

```
1  logger = Logger(log_file_path='log.txt')
2  logger.info('Loggt eine Informationsnachricht')
3  logger.warning('Loggt eine Warnmeldung')
4  logger.error('Loggt eine Fehlermeldung', Exception('Ein Fehler ist aufgetreten'))
```

Listing 15: Verwendungsbeispiel der Logger-Klasse

3.8.4 RotoFlex-Verbindung

Das *RotoFlex*-Modul übernimmt die gesamte Kommunikation mit dem *RotoFlex*. Es stellt eine Websocket-Verbindung zur API des Pflegebetts her, empfängt die Sensordaten und verarbeitet diese. Die gesamte Logik zur Verarbeitung der *RotoFlex*-Daten befindet sich hier. Außerhalb des *RotoFlex*-Moduls hat das System keine Kenntnisse über die Verbindung, sondern erhält nur noch die final verarbeiteten Daten. Es besitzt eine Verbindungsfunktion, welche mit dem *Equipment* verbunden wird (Listing 14). Die Verbindungsfunktion *connect_and_receive_rotoflex_data*, welche asynchron in einem eigenen Thread läuft, enthält intern verwendete Funktionen, die im gleichen Thread ausgeführt werden. Es verwendet zudem eine Queue, um die Daten thread-sicher zu verarbeiten. Basierend auf dem Inhalt der Daten werden diese geparsed und in die Queue zur Weiterverarbeitung durch das *OpenHABConnectionHandler*-Modul eingefügt.

Ein *WebSocketApp*-Objekt wird mit der URL der *RotoFlex*-API und den spezifischen Callback-Funktionen[2] für verschiedene *Events* erstellt (siehe 2.6.2). Zu den Callback-Funktionen gehören *on_message*, *on_error*, *on_open*, und *on_close*. Die Initialisierung des *WebSocketApp*-Objekts ist in Listing 16 dargestellt.

```python
ws = websocket.WebSocketApp(
    'ws://192.168.172.26/api/v2/live/all/ws',
    on_message=on_message,
    on_error=on_error,
    on_close=on_close,
    on_open=on_open
)

ws_thread = threading.Thread(target=ws.run_forever)
ws_thread.daemon = True
ws_thread.start()
```

Listing 16: WebSocketApp-Initialisierung

Außerdem wird ein neuer Thread gestartet, der die *run_forever*-Methode des *WebSocketApp*-Objekts ausführt, um eine dauerhafte Verbindung aufrechtzuerhalten. Der Thread läuft im Hintergrund, als sogenannter *deamon*-Prozess.

Die Callback-Funktionen werden bei bestimmten *Events* des *WebSocketApp*-Objekts ausgeführt. Das macht sie zu *Eventhandlern*. Listing 17 zeigt am Beispiel der *on_message*- und *on_error*-Funktionen die Aufgabe der Callback.

[2] Funktionen, die an andere Funktionen als Argumente übergeben werden und zu einem späteren Zeitpunkt ausgeführt werden (Python Software Foundation (2024c))

```
def on_message(wsapp: websocket.WebSocketApp, message: str):
    try:
        parsed = json.loads(message)
    except json.JSONDecodeError as e:
        logger.error(f'Error decoding JSON from message: {e}', e)

    act_mass = parsed.get('values', {}).get('mass', -1)
    if act_mass > 10:
        queue.put('in bed')
    else:
        queue.put('out of bed')

def on_error(wsapp: websocket.WebSocketApp, error: str):
    logger.error(f'Connection to RotoFlex error', error)
```

Listing 17: Auszug der Callback-Funktionen der *RotoFlex*-Verbindungsfunktion

Ähnlich zu den *on_open*- und *on_closed*-Funktionen hat auch die *on_error*-Funktion lediglich die Aufgabe die Aktivität zu loggen. Entsprechend der Funktionsnamen loggt *on_open* den erfolgreichen Verbindungsaufbau, *on_closed* den Verbindungsabbau und *on_error* einen Fehler, der in der Verbindung aufgetreten ist.

Die *on_message*-Funktion übernimmt die Datenverarbeitung der *RotoFlex*-Daten. Jedes mal, wenn das *WebSocketApp*-Objekt neue Daten erhält wird *on_message* ausgeführt. Die Nachricht wird als JSON-artiger String erwartet. Mit *json.loads(message)* wird versucht, den String in ein Python-Dictionary (*parsed*) umzuwandeln. Aus dem geparsted JSON-Objekt wird der Wert *mass* extrahiert. Dies geschieht durch verschachteltes Zugreifen auf Schlüssel *value* und *mass*. Statt die Schlüssel direkt zu indexieren, wird die *get*-Methode genutzt, da diese die Definition eines Standardrückgabewerts im Falle eines fehlgeschlagenen Zugriffs erlaubt. Der Unterschied ist, dass der direkte Zugriff auf einen falschen Schlüssel in einem Python-Dictionary eine *KeyError*-Exception hervorruft. Die Nutzung der *get*-Methode ermöglicht es über den zweiten Parameter zu definieren, was zurückgegeben wird, falls der Schlüssel nicht existiert. Dabei wird vom Interpreter kein Fehler aufgerufen. Basierend auf dem Wert von *mass* - welcher für das gemessene Gewicht im Bett steht - werden unterschiedliche Nachrichten in die Queue eingefügt. Wenn *mass > 10*, also das Gewicht im Bett größer als 10kg ist, wird angenommen, dass eine Person im Bett ist. In diesem Fall wird der String *in bed* in die Queue eingefügt. Andernfalls wird *out of bed* eingefügt, was bedeutet, dass keine Person im Bett ist. Die Entscheidung einen String anstatt eines Booleans zu nutzen basiert auf der Erkenntnis, dass das *openHAB Item* später ebenfalls einen String erhalten wird. Statt den Boolean *True* nachträglich in einen String *in bed* zu casten, wird direkt ein String verwendet.

3.8.5 SensFloor-Verbindung

Das *SensFloor*-Modul übernimmt die gleichen Aufgaben, wie das *RotoFlex*-Modul (siehe 3.8.4). Es ist ebenfalls für die Verbindung zur API des Gerätes, der Verarbeitung der Daten und der Weiterleitung von Informationen über eine Queue zuständig. Das *SensFloor*-Modul unterscheidet sich in zwei Hinsichten: Zum einen hat es, basierend auf dem Format der Daten, eine andere Verarbeitungslogik als das *RotoFlex*-Modul und zum anderen besitzt es eine andere Verbindungsart, und somit einen etwas anderen Aufbau.

Für die Verbindung zur *SensFloor*-API wird die *socketio*-Bibliothek genutzt (siehe 2.5.5). Ähnlich zu der *WebSocketApp*-Verbindung wird auch hier ein Client initialisiert, welcher sich mit einer URL und Callback-Funktionen verbindet (Listing 18). Gleichermaßen sind auch hier die *Eventhandler*-Funktionen *on_connect* und *on_disconnect* hauptsächlich für Log-Meldungen zuständig.

```
1  sio = socketio.AsyncClient()
2
3  sio.on('connect', on_connect)
4  sio.on('disconnect', on_disconnect)
5  sio.on('alarms-changed', on_alarms_changed)
6
7  await sio.connect('http://192.168.172.22:8000')
```

Listing 18: SocketIO-Initialisierung

Ursprünglich übernahm das *SensFloor*-Modul die Kommunikation mit dem gesamten *SensFloor*. Später wird das Modul erweitert, um mehrere Instanzen zu ermöglichen, damit der *SensFloor* in die einzelnen Räume zur parallelen Verarbeitung aufgeteilt werden kann. Mithilfe der vordefinierten Alarme konnten bei Nachrichteneingang, die Nachrichten in die einzelnen Räume eingeordnet werden.

```
room_dict = {
    1: 'LivingRoom',
    5: 'Hall',
    ...
}

@sio.on('alarms-changed')
def alarms_changed(data):
    for obj in data:
        type = obj['type']
        if type == 'presence':
            alarm_nr = obj['alarmNumber']
            state = obj['state']
            room = room_dict.get(alarm_nr)
            queue.put((room, state))
```

Listing 19: *Eventhandler*-Funktion am Beispiel von *alarms-changed* mit einer *SensFloor*-Verbindung für alle Räume

Weil die Alarmnummern als Integer in der Nachricht unter dem Schlüssel *alarmNumber* vorliegen, wird diese Nummer einer Raumbezeichnung über ein Dictionary zugeordnet. Ist der Typ des Alarms ein Präsenzalarm, so wird ein Tuple mit der Raumbezeichnung und dem Status des Alarme in die Queue zur Weiterleitung eingefügt. Obwohl diese Variante funktioniert, um eine Person erfolgreich in der Wohnung zu überwachen, stößt die Vorgehensweise bei mehreren Personen in einer Wohnung aufgrund der mangelnden Parallelisierung auf Probleme.

Erweiterung: Personenerkennung bei mehreren Personen durch Aufteilung der Räume in eigene Equipments

Läuft die gesamte *SensFloor*-Instanz auf einem Thread, dann ist die Verarbeitung nicht schnell genug und Alarme werden teilweise überschrieben, wenn zeitgleich mehrere Alarme ausgelöst werden. Aus diesem Grund werden für jeden Raum eigene *Equipment*-Instanzen erstellt, die jeweils eine eigene *SensFloor*-Verbindung besitzen. Die Räume werden demnach unabhängig voneinander auf eigenen Threads verarbeitet. Die *Eventhandler*-Funktion wird entsprechend angepasst (Listing 20)

```python
@sio.on('alarms-changed')
def on_alarms_changed(data):
    alarms_for_room = room_dict[name]
    for obj in data:
        if obj['alarmNumber'] in alarms_for_room:
            if obj['type'] == 'presence':
                state = obj['state']
                queue.put((name, state))
```

Listing 20: Angepasste *Eventhandler*-Funktion am Beispiel von *alarms-changed* mit mehreren *SensFloor*-Verbindung für alle Räume

Weil jedes *Equipment* einen Namen besitzt, kann jede Instanz der *SensFloor*-Moduls über diesen Namen zuordnen, welche Alarme zum jeweiligen Raum gehören. Das *room_dict* wird entsprechend umgeschrieben, sodass es jetzt für die Raumbezeichnung die passenden Alarme zurück gibt. An dieser Stelle wird ersichtlich, wieso in 3.8.2 ein eigenes *SensFloor*-Modul für jeden Raum erstellt wird.

Erweiterung: Sturzerkennung

Das *SensFloor*-Modul wird durch eine Sturzerkennung erweitert. Der Sturz muss anders behandelt werden als eine normale Raumpräsenz und ist im *openHAB* auch klar differenzierbar anzuzeigen. Weil unter den Alarm-*Events* der *SensFloor*-API auch ein Alarmtyp für Stürze angeboten wird, ist die Erweiterung direkt integrierbar. Die Funktion wird lediglich durch eine weitere Prüfung des Alarmtypen erweitert (siehe Listing 21).

```python
...
state = obj['state']
if obj['type'] == 'fall':
    queue.put((name + 'fall', state))
elif obj['type'] == 'presence':
    queue.put((name, state))
```

Listing 21: Erweiterung durch Weiterleitung des erkannten Sturzes

Weil jeder Sturz gleichzeitig auch eine Raumpräsenz bedeutet ist es essentiell eine *elif*-Abfrage zu nutzen, damit im Falle eines Sturzes die Queue nicht direkt mit einer regulären Präsenz überschrieben wird. An die Bezeichnung des Alarms in der Queue wird außerdem das Schlüsselwort *fall* angehangen. Weil das restliche Programm keine Kenntnisse über die *SensFloor*-Daten hat, sondern nur die weitergeleiteten Daten der Queue verarbeitet,

ist das angehangene Schlüsselwort entscheidend, um zwischen den beiden Alarmtypen zu differenzieren.

3.8.6 OpenHABConnectionHandler

Der *OpenHABConnectionHandler* ist eine umfangreiche Klasse, die den gesamten Prozess von dem Eingang neuer Daten der Sensoren, bis hin zur Übermittlung von Daten und Befehlen an den *openHAB* übernimmt. Sie steuert die Interaktion mit der *openHAB*-API und befasst sich insbesondere mit der Verwaltung des Beleuchtungssystems.
Die Aufgaben der Klasse können in folgende zwei Schritte unterteilt werden:

1. Abrufen und Kategorisieren von LED-Streifen in der Wohnung

Dies geschieht im Konstruktor der Klasse, welcher bei Initialisierung eines Objektes die Methode *get_all_lights* (Listing 22) aufruft.

```python
def get_all_lights(self) -> Dict[str, list]:
    response = requests.get('http://localhost:8080/rest/items')
    if response.status_code != 200:
        return {}
    items = response.json()

    lights = self.categorize_lights(items)
    return lights

def categorize_lights(self, items: dict) -> dict:
    lights = {}
    room_mapping = {'Wohnzimmer': 'LivingRoom', 'Schlafzimmer': 'BedRoom', ...}
    for item in items:
        if 'RGB_LED_Strip' in item.get('label', ''):
            room = room_mapping.get(item.get('groupNames', [None])[0], None)
            if room:
                lights.setdefault(room, []).append(item)
    return lights
```

Listing 22: Verkürzte Version von der *get_all_lights*-Methode

Dabei wird ein *GET*-Anfrage an die übergeordnete *Item*-URL gesendet, ohne ein spezifisches *Item* anzugeben, um alle *Items* in JSON-Format zu erhalten. Wird kein Erfolgscode *200* zurückgegeben, bedeutet es, dass ein Fehler bei der Übertragung stattgefunden hat.

Weitere Fehlerbehandlung und Log-Nachrichten sind für die Veranschaulichtung in Listing 22 weggelassen worden.

Der Rückgabewert *response* kann in ein Python-Dictionary gecastet werden. Dieses Dictionary enthält alle *Items* inklusive aller dazugehörigen Informationen, wie z. B. die Gruppenzugehörigkeit unter dem Schlüssel *groupNames*. Diese Eigenschaft macht sich die *categorize_lights*-Methode zu Nutze, indem Sie alle *Items*, mit der Bezeichnung *RGB_LED_Strip* im Namen darauf prüft, welcher *Group* sie zugehörig sind. Im *openHAB* sind die *Items* der LED-Streifen immer einem Raum zugeordnet. Es gibt weitere Wege, wie man die LED-Streifen in dem Dictionary identifizieren kann (bspw. über Tags). Das *room_mapping* ist dafür da, um die Bezeichnung der Gruppen im *openHAB* in die systeminterne Bezeichnung zu konvertieren. Die Namen *LivingRoom*, *BedRoom*, etc. sind gleichzeitig auch die Namen der *Equipments* und sorgen damit für eine Einheitlichkeit innerhalb der Codebase.

2. asynchrone Verarbeitung von Sensordaten

Die asynchrone Methode *evaluate_data* ist die Verarbeitungsschleife aller *Equipment*-Daten (*sensors*, siehe 3.8.2).

```
1 async def evaluate_data(self, sensors: Dict[str, Queue]):
2     while True:
3         await self.process_equipment_data(sensors)
4         await self.check_sensfloor_and_control_lights(sensors)
5         await self.process_rotoflex_data(sensors)
6         await asyncio.sleep(0.1)
```

Listing 23: *evaluate_data*-Methode

Die Endlosschleife läuft so lange, bis das Programm beendet wird. In jedem Schleifendurchlauf werden vier weitere asynchrone Funktionen ausgeführt. Zunächst durchläuft die *process_equipment_data*-Methode jedes *Equipment* prüft, ob es neue Daten in den Queue gibt, und legt die Daten aus der Queue in das *data*-Feld des jeweiligen *Equipment*-Objektes (Listing 24).

```
1 async def process_equipment_data(self, sensors: Dict[str, Queue]):
2     for sensor in sensors.values():
3         while not sensor.queue.empty():
4             sensor.data = await self.get_data(sensor.queue)
```

Listing 24: *process_equipment_data*-Methode

Als nächstes bekommt die *check_and_control_lights*-Methode (Listing 25 das Sensor-Dictionary und prüft dieses auf Änderungen.

```
1  async def check_sensfloor_and_control_lights(self, sensors: dict):
2      for room_name, sensor in sensors.items():
3          sensfloor_data = sensor.data
4          if sensfloor_data and sensfloor_data[0] != 'Bootup SensFloor' and \
5                  sensfloor_data != sensor.prev_data:
6              sensor.prev_data = sensfloor_data
7              room, state = sensfloor_data
8              await self.openhab_communication(room, state)
```

Listing 25: Verkürzte Version der *check_sensfloor_and_control_lights*-Methode

Es wird wieder über alle *Equipment*s iteriert und verglichen, ob die vorherigen Daten des jeweiligen *Equipment*s nicht gleich den neuen Daten sind, und ob das *Equipment* nicht erst initialisiert wurde. Sind beide Kriterien erfüllt, dann werden die neuen Sensordaten auf *prev_data* abgelegt, um im nächsten Durchlauf wieder auf Datenänderung prüfen zu können. Das von dem *SensFloor*-Modul in die Queue gelegte Tuple wird in Raum und Zustand aufgeteilt und über die *openhab_communication*-Methode (Listing 26) an den *openHAB* weitergeleitet.

Weiterleitung der Sensordaten und Steuerung der LED-Streifen basierend auf Sensorik, Tageszeit und Raumbelegung

Zuletzt ist die *openhab_communication*-Methode zuständig für die eigentliche *openHAB*-Anbindung, also die Kommunikation der finalen Daten mit den *SensFloor-Items*.

```
async def openhab_communication(self, room: str, state: bool):
    if state:
        data = 'Person im Raum'
        light_state = '20,100,100'
    else:
        data = 'Raum leer'
        light_state = '20,100,10'

    await self.make_api_call(f'http://localhost:8080/rest/items/' \
                             f'SensFloor{room}/state', data=data,
                             request_type='put')
    if state is False:
        for light in self.lights.get(room):
            await self.make_api_call(f'{light.get('link')}_Farbe',
                                     data=light_state, request_type='post')

    elif await self.is_dark(latitude=LAT, longitude=LONG) and \
            self.bed_state == BED_OUT:
        for light in self.lights.get(room):
            await self.make_api_call(f'{light.get('link')}_Farbe',
                                         data=light_state, request_type='post')
```

Listing 26: Verkürzte Version der *openhab_communication*-Methode

Die Methode übernimmt daher zwei Aufgaben. Sie bekommt die zwei Parameter *room* und *state* und legt basierend darauf, ob *state* == *True* ist die an den *openHAB* zu übermittelnden Daten, sowie die Lichtwerte (HSV), fest. Die Methode *make_api_call* sendet die erste API-Anfrage an das jeweilige *SensFloor-Item*, des jeweiligen Raums, in dem sich die Daten geändert haben. Weil die Raumbezeichnung systemweit gleich ist, kann jedes *SensFloor-Item* über einen Platzhalter *room* angesprochen werden. Ist der Raumzustand *False*, also hat eine Person den Raum verlassen, wird ein Befehl als *POST*-Anfrage an das Farbe-*Item* jedes LED-Streifens des Raumes mit der neuen Lichtfarbinformation gesendet. Zusätzlich wird anhand des Längen- und Breitegrads der Wohnung geprüft, ob es dunkel ist. Außerdem wird sichergestellt, dass keine Person sich im Bett befindet (dafür werden die aktuellen Sensordaten des *RotoFlex* genutzt). Die *is_dark*-Methode nutzt die Python-Bibliothek *astral*, um den Sonnenstand bestimmter geografischer Standorte zu ermitteln (*Astral Documentation* (2022)). Die Boolean-Variable *bed_state* speichert über Iterationen hinweg den Status des Pflegebetts. Nur wenn alle Bedingungen erfüllt sind, wird das

Licht in dem jeweiligen Raum geschaltet. Dass die Farbinformation für das Ausschalten der LED-Streifen die gleiche ist, wie die für das Anschalten, nur mit einer verminderten Helligkeit, hat die Ursache, dass das eigentliche Abschalten über eine *Rule* im *openHAB* gesteuert wird (siehe 3.7.4).

Dimmer-Rule

Ähnlich zu der *Item*-Erstellung (siehe 3.8.1) können auch *Rules* im *openHAB* über die Main UI angelegt werden. Über die Main UI wird dafür über den Reiter Einstellungen das Menü *Rules* geöffnet und über das +-Symbol wird eine neue *Rule* angelegt. Jede *Rule* bekommt einen eindeutigen Namen, sowie optionale Beschreibung und Tags. Der Aufbau der *Rule* besteht aus drei Teilen: *When*, *Then* und *But only if*. *When* entspricht dem Auslöser (Trigger) der Rule. Auf das hier festgelegte *Event* wird ständig gehört. Tritt es ein, so wird der *Then*-Teil der *Rule* ausgeführt. Optional kann eine weitere Kondition über *But only if* definiert werden, die vor der Ausführung von *Then* zusätzlich geprüft wird. Der Unterschied zwischen *When* und *But only if* ist, dass auf letzteres nicht gehört wird, diese aber trotzdem zur Prüfung der *Rule*-Ausführung genutzt wird. Die Kondition *But only if* kann demnach selbstständig keine *Rule* auslösen. Folgendes Beispiel demonstriert den Unterschied: Wenn Samstag eintritt geht man Spazieren, aber nur, wenn schönes Wetter ist. Die *Rule* würde demnach lauten '*When*: Heute ist Samstag', '*Then*: Geh spazieren', '*But only if*: Es ist schönes Wetter'. *Rule-Trigger* können alle möglichen Veränderungen des *openHAB*-Systems sein. Von der Änderung eines *Item*-Zustandes, bis hin zu bestimmten Fehlermeldungen oder der Systemuhrzeit. Der *Then*-Teil bietet bereits vorgefertigte Lösungen an (z. B. direktes Setzen von *Item*-Zuständen), erlaubt aber auch die Ausführung von *Scripts*.

Die Rule, die dafür zuständig ist die LED-Streifen zu dimmen, bevor sie abgeschaltet werden, ist in Listing 27 dargestellt.

```
1 rule '<SchlafzimmerLEDStripDimmer>'
2 when
3     Item RGBLEDStripSchlafzimmer_Farbe received command 20,100,10
4 then
5     thread.sleep(6000);
6     if (itemRegistry.getItem('SensFloorBedRoom').getState() == 'Raum leer') {
7       events.sendCommand('RGBLEDStripSchlafzimmer_Farbe', 'OFF');
8     }
9 end
```

Listing 27: Beispiel der Definition der Dimmer-*Rule* für das Schlafzimmer

Der Trigger der *Rule* ist das Erhalten des neuen *Item*-Zustandes *20,100,10* (Listing 26). Wenn also ein LED-Streifen dem abgedunkelten Rot-Ton erhält, wird der *Rule-Thread*

für sechs Sekunden pausiert. Anschließend wird geprüft, ob der Zustand des *SensFloor-Items* immer noch *Raum leer* ist. Dies ist notwendig, um abzufangen, dass in den sechs Sekunden keine Person den Raum erneut betreten hat. In dem Fall darf das Licht nicht ausgeschalten werden. Nur wenn auch nach der vergangenen Zeit niemand im Raum ist, wird der Befehl *OFF* an den LED-Streifen gesendet, der diesen ausschaltet. Um auch das Dimm-Verhalten zu parallelisieren ist es vorteilhaft einzelne *Rules* für jeden Raum (für jedes *SensFloor-Item*) anzulegen, damit alle Dimmer-*Rules* auf eigenen Threads laufen.

3.8.7 Sturzerkennung

Das Programm beinhaltet neben den bereits beschriebenen Funktionen eine Sturzerkennung. Ein Sturz wird durch das System registriert, indem die *alarms-changed*-Funktion durch den Code in Listing 28 erweitert wird.

```
1 if obj.get('type') == 'fall':
2     queue.put((name + 'fall', obj.get('state')))
```

Listing 28: Erweiterung der *Eventhandler*-Funktion um die Sturzerkennung

Wird also ein Sturzalarm vom *SensFloor* gesendet, dann wird dieser ebenfalls in die Queue gelegt. Die Raumbezeichnung bekommt den String *fall* angehangen. Dies ist für die spätere Identifikation des Sturzes im weiteren Programmverlauf notwendig.

```
1  async def check_sensfloor_and_control_lights(self, sensors: dict):
2      ...
3      if 'fall' in room:
4          if not self.fall:
5              self.fall = True
6              room = room.replace('fall', '')
7              await self.openhab_communication(room, state=True, fall=True)
8      elif self.fall:
9          self.fall = False
10         await
11         await self.openhab_communication(room, state=False, deactivate_fall=True)
12     else:
13         await self.openhab_communication(room, state)
```

Listing 29: Erweiterung der *check_sensfloor_and_control_lights*-Funktion um die Sturzerkennung

Die Erweiterung der *check_sensfloor_and_control_lights*-Funktion ist dargestellt in Listing 29. Die Klasse besitzt nun eine Variable *fall*, die einen Boolean-Wert annimmt, je nachdem, ob ein Sturz erkannt wurde oder nicht. Wenn das Schlüsselwort *fall* in der Raumbezeichnung erkannt wird, wird zusätzlich geprüft, ob in der vorhergehenden kein Sturz erkannt wurde. Diese Prüfung sorgt dafür, dass selbst wenn ein Sturz lange anhält (über mehrere Iterationen hinweg) die Übermittlung an den *openHAB* nur einmal geschieht. Wird ein neuer Sturz erkannt, so wird ein *PUT*-Anfrage an das dem Raum zugehörigen *Item* gesendet, mit dem neuen Status *Sturz erkannt*. Die *openhab_communication*-Methode (um den Parameter *fall* erweitert) wird ebenfalls aufgerufen.

```
1  async def openhab_communication(self, room: str, state: bool, fall: bool = False,
2                                  deactivate_fall: bool = False):
3      if fall:
4          await self.make_api_call(f'{OHI.URL}/SensFloor{room}/state',
5                  data='Sturz erkannt', request_type='put')
6          await self.make_api_call(f'{OHI.URL}/AlleLampen_Betrieb',
7                  data='ON', request_type='post')
8          return
9      elif deactivate_fall:
10         await self.make_api_call(f'{OHI.URL}/SensFloor{room}/state',
11                 data='Sturz vorbei', request_type='put')
12         await self.make_api_call(f'{OHI.URL}/AlleLampen_Betrieb',
13                 data='OFF', request_type='post')
14         return
15     ...
```

Listing 30: Erweiterung der *openhab_communication*-Funktion um die Sturzerkennung

Ist der Parameter *fall* == *True* gesetzt, dann sendet die *openhab_communication*-Methode einen *ON*-Befehl an das *AlleLampen_Betrieb Item*, um das gesamte Licht in der Wohnung einzuschalten.

Ist der Sturz-Alarm vorbei - also die Person ist wieder aufgestanden - wird der Befehl *OFF* gesendet, um alle Lampen wieder zu deaktivieren. Danach läuft die Beleuchtung wie gehabt weiter. Für die Differenzierung innerhalb der Methode, ob der Sturz-Alarm vorbei ist, wird der optionale Parameter *deactivate_fall* genutzt.

3.8.8 Config

Damit die Steuerung des Systems von außen erfolgen kann, besitzt das Programm ein Konfigurationsverzeichnis (*config*). In diesem Verzeichnis werden wichtige Programmelemente gespeichert, um das System von außen steuerbar zu gestalten. Die Hauptkonfigurationsdatei (*config.py*) enthält unter anderem Parameter, wie die Lichtfarbwerte der

LED-Streifen, die an den *openHAB* zu übermittelnden Daten, Längen- und Breitengrade des Standortes, sowie ein Gewichtsschwellenwert für den *RotoFlex*. Die Datei enthält außerdem einen *MAX_RUNS*-Integer, über welchen festlegt wird, nach wie vielen Programmdurchläufen die log-Datei geleert wird und ein *ROOM_MAPPING*-Dictionary, wo beliebig viele Räume für die Verarbeitung hinzugefügt werden können.

Die Konfigurationsdatei enthält zwar auch eine *OpenHABInfo*-Datenklasse (Listing 31), welche die Informationen über die *openHAB*-Verbindung kapselt, die jedoch keine direkte Angabe über den Zugriffstoken oder die URL besitzt. Das *headers*-Feld der Datenklasse wird durch die *default_headers*-Funktion initialisiert. Die *default_headers*-Funktion ist eine Factory-Funktion, die jedes mal, wenn sie aufgerufen wird, ein neues Dictionary erzeugt. Der Vorteil der Verwendung einer Factory-Funktion ist, dass sichergestellt werden kann, dass der HTTP-Header nicht durch Modifikation verändert werden kann.

```
1  def default_headers():
2      return {
3          'accept': '*/*',
4          'Content-Type': 'text/plain',
5          'Authorization': f'Bearer {os.getenv('AccessToken')}'}
6
7  @dataclass(frozen=True)
8  class OpenHABInfo:
9      AccessToken: str = os.getenv('AccessToken')
10     URL: str = os.getenv('URL')
11     headers: dict = field(default_factory=default_headers)
```

Listing 31: *OpenHABInfo*-Datenklasse

Da dies sensible Daten sind, auf die nur verschlüsselt zugegriffen werden soll, besitzt das Konfigurationsverzeichnis eine *.env.gpg*-Datei, in der Zugriffstoken und URL gespeichert sind. *.env*-Dateien sind einfache Textdateien, die dazu verwendet werden, Umgebungsvariablen zu definieren. Die Struktur solcher Dateien basiert in der Regel auf einem Schlüssel-Wert-Paar-System. Um die Datei in eine *.gpg*-Datei zu verschlüsseln nutzt man folgenden Befehl:

```
1  gpg --symmetric --cipher-algo AES256 --output config/.env.gpg config/.env
```

Der Befehl verschlüsselt die Datei *.env* im Verzeichnis *config* mit dem Advanced Encryption Standard (AES)256-Algorithmus und speichert das Ergebnis als *config/.env.gpg*. Um der Konfigurationsdatei die Informationen aus der verschlüsselten Umgebungsdatei bereitzustellen, wird nach Prüfung der Dateiexistenz, der Befehl zur Entschlüsselung ausgeführt, welcher die *.env.gpg*-Datei nach Passwortaufforderung zurück in eine *.env*-Datei konvertiert (Listing 32).

```
if not isfile('config/.env'):
    os.system('gpg --output config/.env --decrypt config/.env.gpg')
load_dotenv(find_dotenv())
```

Listing 32: Entschlüsselung von *.env.gpg*

Der Befehl *load_dotenv(find_dotenv())* aus dem *dotenv*-Paket wird verwendet, um die *.env*-Datei zu lokalisieren und die Umgebungsvariablen zu laden. (Kumar et al. (2023))

Anschließend ist es möglich innerhalb jeder beliebigen Datei des Systems über den Wildcard-Import *from config.config import ** auf die Umgebungsvariablen zuzugreifen.

3.8.9 Zusammenfassung der Programmbeschreibung

Zusammengefasst ist das Programm folgendermaßen aufgebaut: Es gibt ein Hauptmodul (*main*), welches übergeordnet die einzelnen, zu dem Programm gehörenden Module initialisiert und auf einzelnen Threads startet.

Das *RotoFlex*-Modul stellt eine Verbindung zur *RotoFlex*-API her und empfängt die Daten synchron, also in festen Zeitintervallen. Die Daten beinhalten Informationen zu dem Gewicht, welches von der A.S.T. Wiegezelle auf dem Bett registriert wurde. Anhand der Gewichtsangabe kann das *RotoFlex*-Modul entscheiden, ob eine Person im Bett liegt oder nicht. Zur Weiterverarbeitung wird diese Information in eine Queue gelegt. Das *SensFloor*-Modul übernimmt die gleiche Aufgabe, wie das *RotoFlex*-Modul, mit dem Unterschied, dass das *SensFloor*-Modul für jeden Raum einzeln initialisiert wird, um eine parallele Verarbeitung aller Räume zu ermöglichen. Jede Instanz des *SensFloor*-Moduls erhält die gleichen Daten, prüft aber jedes mal anhand der Alarm-Nummern, ob die Daten zu dem jeweiligen Raum der Instanz gehören. Wenn ja, werden die Informationen, ob eine Person im Raum ist, ein Sturz erkannt wurde, oder niemand im Raum ist in die Queue zur Weiterverarbeitung gelegt. Jede Instanz des *SensFloor*-Moduls besitzt eine eigene Queue, im Sinne der Parallelisierung.

Auf einem eigenen Thread laufend, prüft die Klasse *OpenHABConnectionHandler* alle 100 ms die Daten aller Queues von allen Verbindungsmodulen und entscheidet, ob neue Statusmeldungen an die *Items* im *openHAB* gesendet werden müssen, bzw. ob das Licht ein- oder ausgeschaltet werden muss.

Die Konfigurationsdatei und der Logger übernehmen Aufgaben zur Verbesserung der Bedienbarkeit des Programms, indem sie die Konfiguration wichtiger Parameter von außen ermöglichen und den Programmablauf protokollieren.

Die meiste Steuerungslogik geschieht in dem externen Python-Programm. Im *openHAB* werden die Daten nur final empfangen und benutzerfreundlich dargestellt. Die einzige Logik, die der *openHAB* übernimmt ist das Dimm-Verhalten der LED-Streifen über *Rules*.

3.8.10 Gestaltung der Benutzeroberfläche

Benutzeroberfläche der SensFloor-Daten

Die Gestaltung der Benutzeroberfläche des *openHAB* für die Anzeige der *SensFloor*-Daten geschieht über die Main UI. Über den Reiter *Einstellungen* kann das Menü *Pages* ausgewählt werden, welches erlaubt benutzerdefinierte Ansichten zu erstellen. Die Ansichten besitzen einen Namen und eine Einstellung der Sichtbarkeit. Zudem werden Ansichten tabellenförmig angelegt. Eine *Page* kann mehrere Tabellen (*Blocks*) enthalten. Dabei kann die Spalten- und Zeilenanzahl beliebig festgelegt werden. Jeder Zelle der Tabelle kann ein sogenanntes *Widget* zugeordnet werden. *Widgets* sind konfigurierbare Menükarten, die mit verschiedenen Elementen des *openHAB* verbunden werden können. Auswählbar sind bspw. *Colorpicker*, *Slider*, *Switch*, *Label* etc., die es entweder erlauben *openHAB*-Elemente über die Benutzeroberfläche zu steuern (z. B. über das Anlegen eines *Switch Widgets* verknüpft mit dem *Item* einer Glühbirne zum Licht-Schalten) oder Informationen der Elemente anzeigen.

Da die Benutzeroberfläche die Funktion erfüllen soll die Daten der *SensFloor-Items* anzuzeigen, wird in der neu angelegten *SensFloor-Page* eine drei-spaltige Tabelle erstellt, die drei *LabelCards* beinhaltet. Jede davon, ist mit einer der *SensFloor-Items* verknüpft und kann in Echtzeit die Daten des *Items* (*Raum leer* oder *Person im Raum*) anzeigen. Außerdem wird die Ansicht der *SensFloor Web App* als *WebFrameCard* eingebunden. Diese kann die URL der *SensFloor Web App*[3] aufrufen und direkt in der Benutzeroberfläche des *openHAB* anzeigen.

Empfangen die *Items* des *SensFloor* über lange Zeit keine Statusänderungen, so wird der Stand auf *NULL* gesetzt. Damit der Nutzer über die Benutzeroberfläche eine verständliche Meldung erhält, gibt es auch hierfür eine *Rule*, die ausgelöst wird, wenn der Status eines beliebigen *SensFloor-Items* auf *NULL* geändert wird. Tritt dieser Fall ist, wird der Status der *Items* aus *Uninitialized* gesetzt, damit dem Nutzer klar wird, dass aktuell keine Verbindung zum *SensFloor* besteht.

Benutzeroberfläche der RotoFlex-Daten

Für die Anzeige der *RotoFlex*-Daten kann die bereits existierende Ansicht des Schlafzimmers genutzt werden, da das Bett semantisch in die Ansicht passt. Hierfür wird ebenfalls eine *LabelCard* erstellt, die die Daten des *RotoFlex-Items* anzeigt. Auch der A.S.T. *CANopen Logger* wird über eine *WebFrameCard* mit der zugehörigen URL[4] in die Ansicht integriert.

[3] http://192.168.172.22:8000
[4] http://192.168.172.26

3.8.11 Langzeittests

Um das System zu testen wird ein Langzeittest gestartet. Dieser wird über 24 Stunden Daten loggen. Das Ziel des Langzeittests ist es Schwachstellen des Programms festzustellen. Nicht nur wird die Zuverlässigkeit der Anbindung über viele Stunden hinweg überprüft, es können auch potentielle Fehlerquellen ermittelt werden.

Der Test ergab folgende Erkenntnisse: Die Anbindung ist stabil und es gab im gesamten Testdurchlauf keine unerwarteten Verbindungsabbrüche. Alle Daten wurden erfolgreich übermittelt. Die Benutzeroberfäche des *openHAB* hat alle Daten korrekt abgebildet. Insgesamt gab es jedoch in den 24 Stunden 949 Präsenzalarme zu Zeiten, wo sich niemand in der Wohnung aufgehalten hat. Davon kamen 926 Fehlalarme aus dem Wohnzimmer, 22 aus dem Flur und ein Fehlalarm aus dem Schlafzimmer.

Daraus lässt sich schließen, dass aufgrund unterschiedlicher Ursachen die in 2.5.6 beschriebenen Fehlerquellen der kapazitiven Sensoren einen deutlich höheren Einfluss auf die Datenübermittlung haben, als erwartet. Das Problem konnte auch durch Beseitigung möglicher Auslöser, wie z. B. auf dem Boden liegende Stromverteiler, nicht gelöst werden.

Analyse des Problems

Eine genaue Betrachtung der fehlerhaften Daten hat das Ziel mögliche Muster zu erkennen, um diese sogenannten *Ghosts* zuverlässig erkennen zu können. Aus der Datensammlung wird folgendes ersichtlich:

1. *Ghosts* entstehen prinzipiell überall in der Wohnung. Das bedeutet, dass es nicht möglich sein wird allein über die Lage der *Ghosts* ausschließen zu können, dass es sich um einen Fehlalarm handelt.

2. *Ghosts* sind nicht statisch. Sie bewegen sich gelegentlich und können teilweise ihren Standort auch komplett verändern. Jedoch ist der Bewegungsradius in der Regel kleiner und *Ghosts* sind insgesamt statischer, als das normale Bewegungsverhalten eines Menschen, welcher sich mit hoher Wahrscheinlichkeit nicht mehrere Stunden in einem Radius von unter zwei Metern bewegen wird.

3. *Ghosts* sind meistens langlebig. Wenn ein *Ghost* einmal entsteht wird dieser üblicherweise eine lange Lebensdauer haben. Der *SensFloor* vergibt jedem erkannten Objekt eine eindeutige ID. So lange, wie das Objekt auf dem Boden erkannt wird, wird sich auch die ID nicht ändern. Über diese Information lässt sich ein erkannter *Ghost* bis zu seinem Verschwinden aus der Analyse der Daten filtern.

4. *Ghosts* verursachen meistens geringere kapazitive Ladungen der Sensoren im Gegensatz zu einer physischen Präsenz. Auch diese Information kann genutzt werden, um *Ghosts* zu erkennen.

5. Objekte können kurzzeitig verschwinden, und wieder auftauchen. Die Kontinuität der Daten ist nicht immer gegeben. Es treten Fälle auf , wo sowohl Personen, als auch *Ghosts* für kurze Zeit aus der Datenliste verschwinden, danach aber wieder auftauchen.

6. Jedes erkannte Objekt des *SensFloor* bestitzt eine eindeutige ID. Jedoch besitzt der *SensFloor* kein eigenes Objekt-Tracking. Das bedeutet, dass wenn ein Objekt verschwindet, der *SensFloor* keine Informationen mehr dazu anbietet. Ein Objekt wird zwar seine ID nicht verändern, solange es existiert, in einem eher unwahrscheinlichem Randfall kann es aber passieren, dass zwei Objekte zur gleichen Zeit verschwinden und in unterschiedlicher Reihenfolge wieder auftauchen. Weil bereits festgestellt wurde, dass *Ghosts* zwischenzeitlich für kurze Zeit verschwinden können, kann dieser theoretisch eintreten. Abbildung 3.8 demonstriert, wie ein Objekt mit der ID *1* und ein Objekt mit der ID *2* gleichzeitig verschwinden und wieder auftauchen. Ihre IDs wurden jedoch neu vergeben und sind in dem Beispiel anschließend getauscht worden.

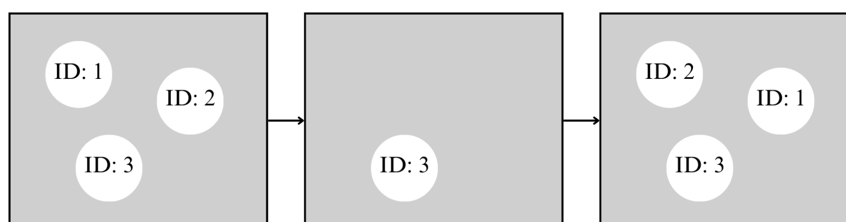

Abbildung 3.8: Änderung der ID von Objekten bei kurzzeitigem Verschwinden

7. Das Badezimmer ist besonders anfällig für *Ghosts*. Aufgrund der hohen Luftfeuchtigkeit sind die Messergebnisse der kapazitiven Sensoren im Badezimmer konstant erhöht und bilden sehr häufig *Ghosts*.

Aus den Erkenntnissen kann man schlussfolgern, dass es notwendig ist das Programm umfangreich anzupassen. Obwohl Komponenten, wie die main-Datei, der *OpenHABConnectionHandler*, das *RotoFlex*-Modul und der Logger keine besondere Anpassung erfordern werden, muss insbesondere das *SensFloor*-Modul neue Funktionalitäten erhalten, um die ermittelten Fehler zu behandeln.

Für eine zuverlässige *Ghost*-Erkennung muss ein Algorithmus entwickelt werden, der dynamisch auf neue Objekte reagiert und diese anhand verschiedener Kriterien, wie Langlebigkeit, Aktivität, Positionsänderung, usw. klassifiziert. Weil die Fehleranfälligkeit des *SensFloor* im Badezimmer (vermutlich aufgrund der hohen Luftfeuchtigkeit, siehe 2.5.6) so hoch ist, wird das Badezimmer vorerst aus der weiteren Entwicklung ausgenommen werden müssen. Die Messwerte des *SensFloor* im Badezimmer sind aufgrund der vielen *Ghosts* zu ungenau um eine Aussage über die Anwesenheit von Personen treffen zu können.

3.8.12 Korrektur Ghost-Erkennung

Die Klasse *SensFloorMonitor*, also das Modul, welches die *SensFloor*-Verbindung aufbaut und die Daten verarbeitet (siehe 3.8.5) besitzt eine vollständig überarbeitete Datenverarbeitungslogik. Ziel des Moduls wird es sein Objekte anhand mehrerer Kriterien erfolgreich in *Ghost* oder *Human* klassifizieren zu können.

Weil für die Entscheidung, ob ein Objekt ein *Ghost* ist, die Alarms des *SensFloor* nicht genügend Informationen bieten, wird für die neue Logik eine Kombination aus den Informationen der Ereginisse *clusters-update*, *alarms-active* und *objects-update* genutzt. Der *SensFloorMonitor* wird nach wie vor für jeden Raum initialisiert und jedes *SensFloorMonitor*-Objekt kommuniziert eigenständig (auf einem eigenen Thread) mit der *SensFloor*-API. Die Klassifizierung der von den Sensoren erkannten Objekte basiert auf einer detaillierten Analyse ihrer Bewegungs- und Verhaltensmuster. Durch die Bewertung von Parametern, wie Bewegungsintensität, Alter des Objekts, die vergangene Zeit seit seiner letzten Aktivität etc. kann das System zwischen menschlichen und nicht-menschlichen Verhaltensmustern differenzieren. Die detaillierte Analyse der gemessen Objekte ermöglicht es Fehlalarme auf ein Minimum zu reduzieren.

Genutzte Informationen der Events

Die *Events clusters-update*, *alarms-active* und *objects-update* liefern essentielle Informationen für die Klassifizierung eines Objektes. Aus dem *clusters-update-Event* wird die Information unter dem Schlüssel *weight* benötigt. Das Attribut liefert den durchschnittlichen kapazitiven Wert der Punkte, aus denen das Cluster besteht (Future-Shape GmbH (2022b)). Dieser Wert hilft die Intensität der Aktivität eines Standortes einzuschätzen. Ein durchschnittlich höherer Kapazitätswert deutet auf eine stärkere physische Präsenz, bzw. einen intensiven Kontakt mit dem Boden hin. Da durch die Datenanalyse (siehe 3.8.11) festgestellt wurde, dass *Ghosts* in der Regel niedrigere kapazitive Werte haben, wird diese Information als ein Erkennungskriterium genutzt.

Das *Event alarms-active* wird weiterhin genutzt. Aufgrund der mangelnden Aufbereitung der erkannten Präsenzen, kann keine zuverlässige Aussage darüber getroffen werden, ob ein Präsenzalarm durch einen Menschen oder einen *Ghost* verursacht wurde. Die Anpassungen der Datenverarbeitung hat jedoch keinen Einfluss auf die Art der Sturzerkennung. Nach wie vor kann dafür der *fall*-Alarm genutzt werden.

Die meistens Kriterien für die *Ghost*-Erkennung stammen aus dem *objects-update-Event*. Aus diesem *Event* werden Objektattribute wie die ID, getätigte Schritte, Zeitpunkt der Entstehung, Zeitpunkt der letzten Änderung, Bewegungsgeschwindigkeit usw. evaluiert.

Beschreibung der angepassten SensFloorMonitor-Klasse

Wie zuvor, ist das Herzstück des *SensFloorMonitors* die *connect_and_receive_sensfloor_data*-Methode, die bei der Erstellung des *Equipment* als Verbindungsmethode übergeben wird. Ebenfalls gleich geblieben ist die Verbindung zur *SensFloor*-API und die *Eventhandler*-Funktionen für die Bestätigung des Verbindungsaufbaus, bzw. des Verbindungsabbruchs durch den Logger. Außerdem besitzt die *connect_and_receive_sensfloor_data*-Methode drei weitere interne *Eventhandler*-Funktionen für die drei *Events*. Da die *Events objects-update* und *clusters-update* über Koordinaten zugeordnet werden müssen, erfolgt bei jedem gesendeten *Event* eine Raumzuordnung (Listing 33). Die Koordinaten der einzelnen Räume lassen sich in der Konfigurationsdatei festlegen.

```
1 objects_in_room = []
2 for obj in data:
3     x_coord = obj.get('x')
4     y_coord = obj.get('y')
5     if self.x_start <= x_coord <= self.x_end and
6             self.y_start <= y_coord <= self.y_end:
7         objects_in_room.append(obj)
```

Listing 33: Raumzuordnung der Alarme

Die *Eventhandler*-Funktion für das *alarms-active-Event* beinhaltet nur noch die Reaktion auf Sturzalarme und behandelt nicht wie vorher Präsenzalarme. Die Raumzuordnung funktioniert anders als in Listing 33 dargestellt, da Alarms für jeden Raum einen eigenen Index besitzen. Wird also ein Alarm mit dem Index *2* empfangen, steht dieser für einen Sturz im Wohnzimmer. Ein Alarm mit dem Index *6* hingegen wird bei einem Sturz im Flur gesendet.

```
1 @sio.on('alarms-active')
2 def on_alarms_active(data: list):
3     fall_alarm_index = self.room_dict[name]['fall-alarm']
4     for obj in data:
5         if obj['index'] == fall_alarm_index:
6             queue.put((name + 'fall', True))
```

Listing 34: Raumzuordnung der Alarms zu Beginn des *objects-update-Eventhandler*

Entspricht also ein Alarm dem korrekten Index für den Sturzalarm des überwachten Raums, so wird der Sturz zur Weiterverarbeitung in die Queue gelegt (Listing 34).

Die *Eventhandler*-Funktion des *clusters-update-Events* stellt zu jedem erkannten Cluster eine Hypothese auf, ob basierend auf dem *weight*-Attribut (siehe 3.8.12) das Cluster auf einen Kontakt mit einer physischen Person hindeutet, oder nicht (Listing 35). Dafür wird ein *weight_threshold*, also ein Schwellenwert für den kapazitiven Wert festgelegt. Besitzt das Cluster einen größeren Wert, als der Schwellenwert, dann handelt es sich vermutlich um eine Person und nicht einen *Ghost*. Der genaue Festlegung der Schwellenwerte erfolgt in 3.8.13.

```
for cluster in clusters_in_room:
    if cluster['weight'] >= self.weight_threshold:
        self.cluster_human_hypothese = True
    else:
        self.cluster_human_hypothese = False
```

Listing 35: Schwellenwertprüfung für das *weight*-Attribut eines Clusters

Die meisten für die Klassifizierung relevanten Informationen werden durch das *objects-update-Event* übermittelt. Die *Eventhandler*-Funktion ermittelt zu Beginn gleichermaßen, ob das *Event* zu dem Raum gehört, für das auch der *SensFoorMonitor* initialisiert wurde (Listing 33). Weil dieser *Event* die meiste Daten die für die Klassifizierung relevant sind liefert, übernimmt die *Eventhandler*-Funktion die meiste Logik.

Dass die Klassifizierungslogik in der *on_objects _update*-Funktion erfolgt hat den Grund, dass hier die meisten relevanten Daten geliefert werden. Würde man die Logik in eine der anderen *Eventhandler*-Funktionen auslagern, besteht das Risiko, dass ein Großteil der Daten leer oder veraltet bleiben, weil das *Event* für *objects-update* später im Programm ankommt, als die Ausführung der Logik.

Dadurch, dass die Klassifizierung in der *on_objects _update*-Funktion erfolgt, kann sichergestellt werden, dass immer die neusten Informationen vorliegen. Zwar können nach dem gleichen Prinzip die Daten aus dem *clusters-update-Event* leer oder veraltet sein, da dies aber nur ein einziger Bestandteil der Klassifizierung ist, hat das einen weniger großen Einfluss.

Die Verarbeitungslogik der *on_objects _update*-Funktion besteht aus drei Bestandteilen.

```
for obj in objects_in_room:
    obj_id = obj['id']
    if self.object_states.get(obj_id):
        self.object_states[obj_id]['empty_updates_count'] = 0
    self.track_object_location(obj_id, obj['x'], obj['y'])
    classification = self.evaluate_object(obj)
    if classification == self.last_state:
        continue
    if classification == 'Human':
        queue.put((name, True))
        self.last_state = classification
    elif classification == 'Ghost':
        queue.put((name, False))
    self.last_state = classification
```

Listing 36: Klassifizierungslogik in *on_objects _update*

Listing 36 zeigt einen der drei Bestandteile. Für jedes Objekt des zugehörigen Raumes wird zuerst geprüft, ob dieses Objekt bereits in dem *object_states*-Dictionary vorhanden ist. Das Dictionary wird bei Objektkonstruktion initialisiert und ist ein umfangreicher Speicher für alle Objekte. Das Dictionary beinhaltet zu jedem Objekt Informationen, wie die ID, den aktuellen Zustand, den vorherigen Zustand und eine Standort-Historie. Unter dem Schlüssel *empty_updates_count* wird ein Zahlenwert gespeichert, der angibt, wie oft hintereinander das Objekt nicht in *Events* gesendet wurde (siehe 3.8.11 Aufzählung 5). Der Wert wird auf '0' zurückgesetzt, weil das Objekt Teil des aktuellen *Events* ist. Die *track_objects_location*-Methode (siehe Listing 37) erstellt eine Referenz des *object_states*-Dictionary und fügt der Standort-Historie ein Tuple aus den aktuellen x- und y-Koordinaten an. Als letztes wird die Liste gekürzt, sodass nur die letzten n Standorte verfolgt werden, um den Speicherplatz wieder freizugeben. Die Anzahl n der verfolgten Standorte ist über die Variable *location_tracking_amount* festlegbar.

```
obj = self.object_states.setdefault(obj_id, {'current_state': self.last_state,
                                             'counter': 0, 'locations': []})
obj['locations'].append((x, y))
obj['locations'] = obj['locations'][-self.location_tracking_amount:]
```

Listing 37: *track_objects_location*-Methode

Der Hauptbestandteil der Klassifizierung erfolgt anschließend in der *evaluate_object*-Methode (Listing 38). Die Methode beinhaltet zwei Hypothesen-Listen, mit Kriterien, ob ein Objekt wahrscheinlich ein *Human* oder ein *Ghost* ist.

Die Hypothese *likely_human* ist dann erfüllt, wenn die Summe der wahren Aussagen des Objektes aus der Kriterien-Liste des Menschen größer ist, als die der Kriterien-Liste des *Ghosts*. Die Funktion gibt einen String zurück, mit der Information, welche Kassifizierung wahrscheinlicher ist. Sollten beide Listen gleich-viele erfüllte Kriterien besitzen, kann keine eindeutige Aussage getroffen werden. In diesem Fall wird der letzte Zustand (*last_state*) zurückgegeben, mit der Annahme, dass der Zustand sich nicht verändert haben wird.

```
1  current_time_millis = lambda: int(round(time.time() * 1000))
2  age_since_birth = current_time_millis() - obj['birthTime']
3  time_since_last_change = current_time_millis() - obj['lastChangeTime']
4  significant_movement = self.has_moved_significantly(obj_id)
5
6  likely_human_criteria = [
7      obj['steps'] > self.min_steps_for_human,
8      age_since_birth < self.age_since_birth,
9      time_since_last_change < self.time_since_last_change,
10     significant_movement,
11     not obj['isStatic'],
12     obj['mV'] > 0,
13     self.clusters_human_hypothese
14 ]
15
16 likely_ghost_criteria = [
17     obj['steps'] < self.min_steps_for_human,
18     age_since_birth > self.age_since_birth,
19     time_since_last_change > self.time_since_last_change,
20     not significant_movement,
21     obj['isStatic'],
22     obj['mV'] == 0,
23     not self.clusters_human_hypothese
24 ]
```

Listing 38: *evaluate_object*-Methode

Die Kriterien, dass ein Objekt als *Human* klassifiziert wird, sind folgendermaßen:

1. Das Objekt hat viele Schritte (*steps*) gemacht.
2. Das Objekt ist vor nicht allzu langer Zeit entstanden (*age_since_birth*).
3. Das Objekt hat sich in einem bestimmten Zeitintervall verändert (*time_since_last_change*).
4. Der vergangene Bewegungsradius ist bedeutend groß gewesen (*significant_movement*).
5. Das Objekt wird vom *SensFloor* nicht als statisch (*isStatic*) eingeschätzt.
6. Das Objekt besitzt eine Geschwindigkeit größer 0 (*mV*).
7. Die aus dem *clusters-update-Event* definierten *weight*-Informationen (*clusters_human_hypothese*) sprechen für einen Menschen (Listing 35).

Nach der Entscheidung der Zuordnung (*Human* oder *Ghost*, Listing 39), ist die *update_object_state*-Methode (siehe Anhang A, Listing 47) dafür verantwortlich die Zustands-Historie zu aktualisieren und vor allem zu prüfen, ob der Objektzustand konsistent ist. Wenn ein Objekt die ganze Zeit als ein *Ghost* eingestuft wurde, plötzlich aber als *Human* klassifiziert wird, soll der Zustand nicht direkt geändert werden. Damit kann verhindert werden, dass falsche Klassifizierungen einen deutlichen Einfluss auf die *openHAB*-Anbindung haben, indem fehlerhafte Klassifizierungen vorerst ignoriert werden. Zustände werden nur geändert, wenn konsistente Änderungen festgestellt werden.

```
1 if sum(likely_human_criteria) >= sum(likely_ghost_criteria):
2     if self.update_object_state(obj_id, 'Human'):
3         return 'Human'
4 elif sum(likely_ghost_criteria) > sum(likely_human_criteria):
5     if self.update_object_state(obj_id, 'Ghost'):
6         return 'Ghost'
7 return self.last_state
```

Listing 39: Klassifizierungsentscheidung der *evaluate_object*-Methode

Die *current_time_millis*-Variable besitzt den Rückgabewert einer Lambda-Funktion.
Das sorgt dafür, dass jedes mal, wenn die Variable aufgerufen wird, sie den aktuellen Zeitwert besitzt, anstatt, dass dieser zu einem Zeitpunkt definiert werden muss. Dies wird genutzt, um das Alter und die Inaktivitätszeit des Objektes zu bestimmen.
Die *has_moved_significantly*-Methode gibt einen Boolean zurück, der anhand der Standort-Historie des *object_states*-Dictionary bestimmt wird und angibt, ob die Bewegung des Objektes stark genug ist, um sie als 'menschlich' einzustufen. Die Bewegungsintensität wird

errechnet durch die Distanzermittlung (Listing 40) des Initalstandortes und des aktuellen Standortes in einem 2D-Raum.

```
dx = l2[0] - l1[0]
dy = l2[1] - l1[1]
return (dx**2 + dy**2)**0.5
```

Listing 40: Distanzberechnung zwischen zwei Punkten

Die Distanz zwischen den beiden Punkten wird unter Nutzung der Pythagoräischen Formel ($\sqrt{(dx^2 + dy^2)}$) berechnet (Strick (2019)). Die Differenzen dx und dy stellen die Unterschiede der x- und y-Koordinaten der beiden Punkte dar. Das Quadrat daraus stellt die Summe der Quadrate der Katheten eines rechtwinkligen Dreiecks dar. Die Hypotenuse (die Seite des Dreiecks, die dem rechten Winkel gegenüber liegt), ist die direkte Distanz der beiden Punkte. Die Quadratwurzel daraus gibt schließlich die Länge der Hypotenuse an, welche gleich der Entfernung der beiden Punkte ist (Abb. 3.9).

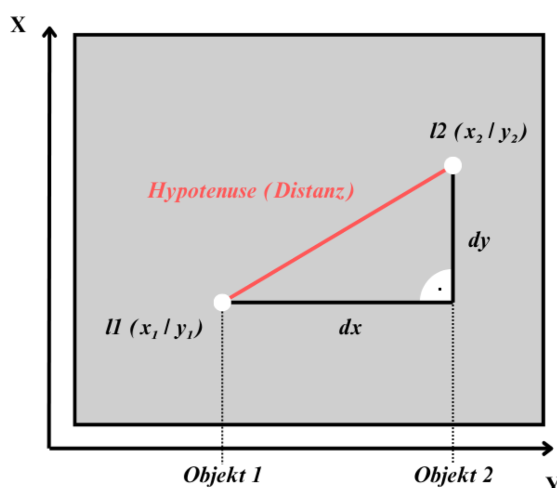

Abbildung 3.9: Distanzberechnung zwischen zwei Punkten (Objekten) mittels Hypotenusenbestimmmung

Nach der erfolgreichen Klassifizierung (Listing 36) wird geprüft, ob eine Zustandsänderung vorliegt. Ist der neue Zustand dem alten gleich (*classification == self.last_state*), dann geschieht keine weitere Datenverarbeitung. Ist der neue Zustand *Human* oder *Ghost*, dann wird dies zur Weiterverarbeitung in die Queue gelegt.

Vor der Klassifizierungslogik der *on_objects_update*-Methode gibt es zwei weitere Vorprüfungen. Zum einen kann es vorkommen, dass *objects_in_room* leer ist, weil das *SensFloor-Event* keine Objekte im Raum anzeigt. Dafür ist der Code in Listing 41 zuständig.

```
1  if not objects_in_room and self.last_state != 'No presence':
2      if self.no_object_counter > self.no_objects_threshold:
3          self.no_object_counter = 0
4          self.object_states = {}
5          self.last_state = 'No presence'
6          queue.put((name, False))
7      else:
8          self.no_object_counter += 1
```

Listing 41: Vorprüfung 1 leeres *Event on_objects_update*-Methode

Wenn das *Event* für den Raum leer ist und der letzte Raumzustand ebenfalls leer war, soll nicht sofort der Raum als leer betrachtet werden. Auch hier gibt es einen Schwellenwert, der angibt, wie viele leere *Events* hintereinander kommen müssen, bevor der Raum endgültig als leer eingestuft wird. Dies sorgt ebenfalls für Konsistenz und verhindert, dass kurzzeitig verschwundene Objekte den Raumzustand ändern. Ist aber der Raum tatsächlich länger als der Schwellenwert leer, dann werden alle Variablen zurückgesetzt und die fehlende Präsenz in die Queue gelegt. Weil *objects-update* ein synchrones *Event* ist, kann das Hoch-zählen von *no_objects_counter* iterativ erfolgen.

Einzelne Objekte können aber schon eher als 'verschwunden' eingestuft werden. Dafür wird alternativ geprüft, ob das aktuelle *Event* keine Objekte hatte, jedoch ohne zu prüfen, was der vorhergehende Zustand des Raumes war.

```
1  elif not objects_in_room:
2      objects_to_delete = []
3      for obj_id, obj_state in self.object_states.items():
4          obj_state['empty_updates_count'] = \
5          obj_state.get('empty_updates_count', 0) + 1
6          if obj_state['empty_updates_count'] >= self.empty_updates_threshold:
7              objects_to_delete.append(obj_id)
8      for obj_id in objects_to_delete:
9          del self.object_states[obj_id]
```

Listing 42: Vorprüfung 2 leeres *Event on_objects_update*-Methode

In Listing 42 erkennt man, wie durch alle Elemente des *object_states*-Dictionary iteriert wird, um den Zahlenwert für die Anzahl der *Events*, in denen das Objekt nicht mehr präsent war, zu erhöhen. Ähnlich wie in der ersten Vorprüfung ist das Ziel Objekte weiterhin zu überwachen, auch wenn zwischenzeitlich der *SensFloor* diese nicht mehr erkennt. Ist ein Objekt in mehr *Events* hintereinander nicht präsent gewesen als der Schwellenwert maximal festlegt dann wird das Objekt endgültig aus der Überwachung entfernt.

3.8.13 Konfiguration der Ghost-Erkennung

Damit die Konfiguration der einzelnen Schwellenwerte von außen einfach geschehen kann wird im *config*-Verzeichnis eine weitere Datei erstellt. Die *ghost_detection_settings.py*-Konfigurationsdatei beinhaltet alle in dem Programm genutzten Schwellenwerte für die *Ghost*-Erkennung. Da die Festlegung der Schwellenwerte je nach Anwendungsfall unterschiedlich sein kann und es keine eindeutig korrekten Schwellenwerte gibt, ist es notwendig die Schwellenwertfestlegung extern über eine Konfigurationsdatei zu ermöglichen. Mithilfe der Durchführung von praktischen Tests sind letztendlich geeignete Einstellungswerte ermittelt und konfiguriert worden (siehe Anhang, Listing 48).

3.8.14 Inbetriebnahme

Die Struktur des Projekt-Verzeichnisses ist in Abbildung 3.10 abgebildet. Orangefarbene Felder stehen für Verzeichnisse, graue Felder für Dateien.

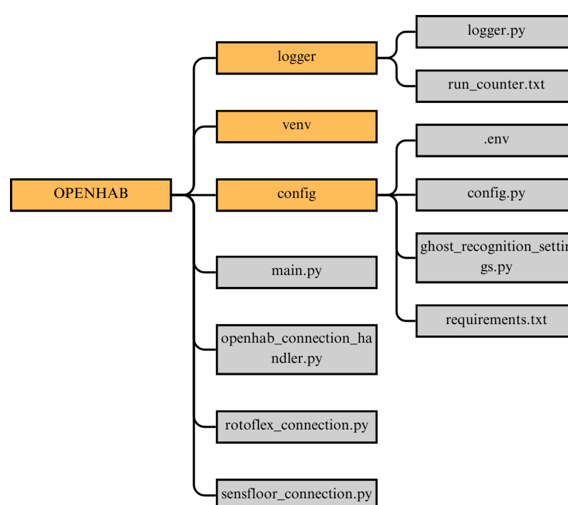

Abbildung 3.10: Verzeichnisbaum des Projekts

Die Inbetriebnahme kann direkt erfolgen, das Programm kann aber auch im Hintergrund (*headless*) gestartet werden. Für beide Varianten muss zunächst sichergestellt werden, dass die virtuelle Umgebung (*venv*-Verzeichnis) des Projektes aktiviert ist. Dies erfolgt über den Befehl *source venv/bin/activate*. Über den Befehl *pip install -r config/requirements.txt* können bei Bedarf alle für das Projekt notwendigen Abhängigkeiten installiert werden.

Beim ersten Start wird der Nutzer aufgefordert ein Passwort einzugeben. Dieses entschlüsselt die *.env.gpg*-Datei, welche den Zugriffstoken und die URL des *openHAB* beinhaltet. Solange die darauf entstandene *.env*-Datei nicht gelöscht wird, wird das Programm ohne erneute Passworteingabe auf diese zurückgreifen. Konfigurationen können über die *config.py*- und die *ghost_detection_settings.py*-Dateien vorgenommen werden.

Soll das Programm direkt im aktiven Terminal gestartet werden reicht der Befehl *python main.py* aus. Soll das Programm jedoch headless, also unsichtbar gestartet werden, dann kann der Befehl *nohup python main.py >/dev/null 2>&1 &* genutzt werden. Das Kommando *nohup* wird verwendet um Programme über die Terminalverbindung hinaus zu starten. Selbst wenn das Terminal geschlossen wird läuft das Programm weiter. Die Log-Meldungen erscheinen zwar nicht in der Konsole, können aber in der *Log.txt*-Datei eingesehen werden. Die Parameter *>/dev/null 2>&1 &* verwerfen die Standardausgabe und die Standardfehlerausgabe. Sie werden ignoriert. Da das Programm sowieso eine eigene Form der Ausgabe nutzt (über den Logger) ist die weitere Ausgabe irrelevant. Das abschließende *&* am Ende des Befehls bedeutet, dass der Prozess im Hintergrund gestartet wird und das Terminal direkt wieder freigegeben wird.

Der Befehl wird eine Ausgabe haben mit der ID des erstellten Prozesses. Diese ID kann genutzt werden, um über den Befehl *kill [process_id]* das Programm zu beenden, wobei *[process_id]* ein Platzhalter für die eigentliche Prozess-ID ist. Um die Prozess-ID nachträglich zu ermitteln, kann der Befehl *ps aux | grep python* ausgeführt werden, der alle aktuell laufenden Python Programme mit entsprechender ID anzeigt.

3.8.15 Einrichtung des Systems in anderen Umgebungen

Räume lassen sich über die Anpassung der Koordinaten flexibel über die Konfigurationsdatei hinzufügen und ändern. Somit kann das System mit wenig Anpassungsbedarf in neue Umgebungen integriert werden. Um die volle Funktion nutzen zu können müssen folgende Integrationsschritte befolgt werden[5]:

1. Es muss ein *openHAB* System in der Version 3 installiert sein, oder eine Version, die mit den Funktionalitäten des *openHAB* 3 kompatibel ist.

2. Es muss Python, in der Version 3.9[6] und eine virtuelle Umgebung basierend auf der requirements.txt-Daten aufgesetzt sein.

3. Die Raum-Informationen müssen entsprechend der Umgebung in der Konfigurationsdatei angepasst sein.

4. Im *openHAB* müssen die *Items* für *SensFloor* und *RotoFlex* (siehe 3.8.1), sowie *Rules* für das Dimmen (siehe 3.8.6) und die Re-Initialisierung der *Items* bei längerem Nicht-Betrieb erstellt werden.

5. Die Zugangsdaten und Adressen von *openHAB* sind anzupassen. Wird das Programm von einem separaten Server gestartet, und nicht von dem Server auf dem der *openHAB* installiert ist, müssen entsprechend vor den API-Anfragen Secure Shell (SSH)-Verbindungen zu dem openHAB-Server aufgebaut werden, von dem aus die Anfragen gesendet werden. Terminals mit SSH-Verbindungen können mithilfe der *subprocess*-Bibliothek aus einem Python-Skript heraus gestartet werden

[5] die Reihenfolge spielt keine Rolle
[6] bei höheren Versionen ist die Abwärtskompatibilität der genutzten Bibliotheken zu prüfen

(Python Software Foundation (2023)). Um die Leistung des Programms zu verbessern bietet es sich an die Terminals persistent auf einem eigenen Thread geöffen zu lassen, um ständiges auf- und abbauen von SSH-Verbindungen zu vermeiden.

6. Geräte, mitsamt Verbindungsfunktion und Datenverarbeitungslogik können über das Erstellen weiterer *Equipments* in das Programm integriert werden.

3.9 Ergebnisse

3.9.1 Skalierbarkeit

Das Programm ist so konzipiert, dass eigene Module nahtlos in die Verarbeitung einbezogen werden können. In der *main.py*-Datei können weitere *Equipment*-Objekte nach belieben erstellt werden. Diese erhalten eine eigene Queue und Verbindungsfunktion. Die Verarbeitungslogik kann in der *evaluate_data*-Methode des *OpenHABHandlers* als separate Methode eingeführt werden. Auf die Weiterleitung der Daten muss dabei nicht geachtet werden. Diese funktioniert universell für alle *Equipment*-Objekte, solange die Daten über die Queue weitergeleitet werden. Der *OpenHABHandler* dient in dem System als Hauptschnittstelle für die Kommunikation zwischen Geräten und *openHAB*. Die eigentlichen Anbindungen geschehen in gesonderten Modulen. Die Datenübertragung ist über die *Queues* möglich. Damit gibt es eine klare Trennung zwischen gerätespezifischen-Funktionen (die nach belieben erweitert und angepasst werden können) und der *openHAB*-Datenverarbeitung.

Über das Konfigurationsverzeichnis lassen sich grundlegende Einstellungen des Programms, sowie der *Ghost*-Erkennung ändern. Die Informationen *.env*-Umgebungsdatei können ebenfalls individuell angepasst werden, um mit anderen *openHAB*-System kommunizieren zu können. Wichtig ist dabei zu beachten, dass die *.env.gpg*-Datei verschlüsselt ist und erst durch das Programm entschlüsselt wird. Ist die Verschlüsselung für die Anwendung irrelevant, reicht es nur eine *.env*-Datei zu erstellen. Das Programm wird beim Start automatisch auf diese zurückgreifen. Beim Verändern der Lichtfarbwerte in der Konfigurationsdatei muss unbedingt darauf geachtet werden, diese auch in den *openHAB Rules* anzupassen, da sonst die Dimm-Funktion nicht funktionieren wird.

3.9.2 Bewertung der RotoFlex-Anbindung

Die *RotoFlex*-Anbindung besteht aus einem Modul, welches Daten von der *RotoFlex*-API erhält und diese auswertet. Je nachdem, ob die eingebaute Wiegezelle des Pflegebetts ein bestimmtes Gewicht registriert, wird an den *openHAB* vermittelt, ob eine Person im Bett ist, oder nicht.

3.9.3 Bewertung der SensFloor-Anbindung

Die parallele Erkennung mehrerer Personen in unterschiedlichen Räumen funktioniert - dank der Aufteilung des *SensFloor* in einzelne Räume - fehlerfrei. Die Erstellung eigener Anbindungen für jeden Raum sorgt erfolgreich dafür, dass die Räume auf eigenen Threads verarbeitet werden und somit gleichzeitig in der Lage sind Daten zu verarbeiten. Die festgestellten Fehler des *SensFloor* erforderten eine Erweiterung des *SensFloor*-Moduls. Beide Varianten der Datenverarbeitung des *SensFloor* haben spezifische Vor- und Nachteile in Bezug auf Genauigkeit und Effektivität der Personenerkennung.

Ergebnisse der Variante ohne Ghost-Erkennung

Die Basisvariante ohne *Ghost*-Erkennung zeichnet sich durch die einfache Konfiguration und die direkte Reaktion auf einkommende Sensordaten aus. Die direkte Reaktion ohne aufwendige Verarbeitungslogik bietet jedoch nur in Umgebungen mit geringem elektromagnetischen Interferenzniveau eine hohe Zuverlässigkeit. Durch praktische Analysen wurde festgestellt, dass die Sensoren fälschlicherweise nicht-menschliche Objekte als menschliche Präsenzen (*Ghosts*) registrieren. Diese Fehler führen zur Auslösung falscher Alarme, die insbesondere in einer AAL-Umgebung negative Folgen haben können. Beispielsweise führt ein Sensorfehler in der Nacht dazu, dass das Licht eingeschaltet wird und Personen aufgeweckt werden. Durch die Detailarmut der vordefinierten und -kalibrierten *SensFloor*-Alarme durch den Hersteller *Future-Shape*, ist es nicht möglich unter Nutzung dieser eine effektive *Ghost*-Erkennung zu entwickeln.

Ergebnisse der Variante mit Ghost-Erkennung

Die erweiterte Variante des *SensFloor*-Moduls mit *Ghost*-Erkennung integriert Algorithmen zur Unterscheidung zwischen echten Präsenzen und fälschlicherweise detektierten *Ghosts*. Eine praktische Analyse des Programms zeigt eine deutliche Reduktion der falsch positiven Erkennungen. Innerhalb eines 24-stündigen Testzeitraums wurden lediglich 12 *Ghosts* fälschlicherweise als menschliche Präsenz erkannt, statt den 949 falsch-positiven Alarms der Basisvariante (siehe 3.8.11). Jedoch wurde festgestellt, dass die Hypothesenbildung basierend auf limitierten Sensordaten in einigen Fällen zu einer verzögerten oder fehlenden Erkennung echter menschlicher Präsenzen geführt hat. Durch die hohe Filterung von erkannten Objekten durch den Klassifizierungsalgorithmus kommt es vor, dass ein still-stehender Mensch alle Kriterien erfüllt um als *Ghost* registriert zu werden.

Eine Laufzeitanalyse beider Varianten (durch Messung der benötigten Zeit eines Schleifendurchlaufs der *evaluate_data*-Methode) zeigt außerdem eine vierfache Erhöhung der Laufzeit. Diese hat zwar in praktischen Tests keinen merklich Nachteil dargestellt, eine lange Verarbeitungsdauer kann aber bei einer hohen Anzahl an Aktivitäten zu Verzögerungen führen.

Vergleich der Varianten

Der Vergleich der beiden Varianten zeigt, dass beide Varianten nicht fehlerfrei funktionieren und die Wahl der Variante davon abhängt, was in dem aktuellen Kontext wichtiger ist: Reduktion von falsch-positiven *Ghosts* oder zuverlässige Erkennung von echten Personen. In einer Umgebung, wo mit wenig Interferenzen zu rechnen ist, oder Sensorfehler keinen signifikanten Einfluss auf das System haben, ist die Variante ohne *Ghost*-Erkennung zuverlässiger. Andererseits ist in Umgebungen mit vielen Störfaktoren und einer hohen Fehlersensibilität die *Ghost*-Erkennung essentiell.

4 Schlussbemerkung

4.1 Diskussion

4.1.1 Auswertung des SensFloor-Systems

Das Hauptproblem der *SensFloor*-Anbindung ist die *Ghost*-Erkennung. Um diese Problematik weiter zu adressieren, gibt es einige Ansätze, die die Zuverlässigkeit der *SensFloor*-Datenauswertung verbessern können.

Ein Verbesserungsmöglichkeit besteht darin, den Klassifizierungsalgorithmus durch maschinelles Lernen und Mustererkennungsalgorithmen zu erweitern. Diese müssen auf großen Datensätzen trainiert werden, um erfolgreich zwischen menschlichen Präsenzen und Störquellen unterscheiden zu können. Diese Algorithmen können des weiteren durch Deep Learning Techniken verbessert werden, die in der Lage sind aus komplexen Zusammenhängen Datenfehler zu erkennen und diese zu ignorieren. Die Studie Shi et al. (2020) stellt eine Bodenüberwachungstechnologie vor, die auf Deep Learning basierende Datenanalysen nutzt, um Standort, Art der Aktivität und Identität der Person basierend auf Gangmustern zu erkennen. Die hohe Genauigkeit bei der Erkennung verschiedener Aktivitäten und Identitäten zeigt das Potenzial von Deep Learning Algorithmen im Kontext von Bodensensorsystemen und bieten damit eine mögliche Verbesserung für die *Ghost*-Erkennung des *SensFloor*.

Des weiteres könnte die Verwendung von fusionierter Sensorstrategie eine Lösung für das Problem bieten. Dabei werden mehrere Sensortypen kombiniert, um ein genaueres Bild der Raumaktivität zu erhalten. Beispielsweise kann die Verwendung von Infrarot-Sensoren dazu beitragen, um bei einem positiven *SensFloor*-Signal den Raum auf Bewegung zu überprüfen. Passiv-Infrarot-Sensor (PIR) wurden bereits erfolgreich in anderen Bewegungserkennungssystemen eingesetzt. Usikalu (2018) beschreibt, wie energiesparende PIR-Sensoren eingesetzt wurden, um Bewegungen in Räumen zu erkennen und basierend auf den Sensordaten das Licht ein- und auszuschalten, je nachdem, ob Personen im Raum sind, oder nicht. Diese kostengünstige Erweiterung hat das Potenzial die Zuverlässigkeit der Präsenzerkennungen deutlich zu verbessern. Diese Sensoren können ebenfalls unter Nutzung der *OpenHABHandler*-Klasse in das *openHAB* integriert werden. Die hohe Fehleranfälligkeit der kapazitiven Sensoren im Badezimmer verhinderte die Entwicklung des Beleuchtungssystems für den Raum. Neben der fusionierten Sensorstrategie und Deep Learning Verfahren kann untersucht werden, ob Kompensationstechniken die Umwelteinflüsse im Badezimmer auf die Sensoren reduzieren. Beispielsweise kann die Luftfeuchtigkeit im Badezimmer gemessen werden und die Reaktion des *SensFloor* bei hohen Messwerten entsprechend reduziert werden. Außerdem kann der Einsatz eines

Luftentfeuchters die Messergebnisse des *SensFloor* potentiell verbessern. Dafür muss vorher aber die genaue Ursache der hohen Messunstimmigkeiten im Badezimmer ermittelt werden.

Die Lösung das Beleuchtungssystem nur dann Lichter anschalten zu lassen, wenn die Daten des *RotoFlex* zeigen, dass sich keine Person im Bett befindet, sorgt dafür, dass das System jetzt schon angewendet werden kann, da es bei falsch-erkannten Objekten keine schlafende Person aufweckt. Beherbergt die Wohnung mehrere Personen, muss jedes einzelne Bett integriert und geprüft werden. Die Möglichkeit zu prüfen, ob eine Person im Bett liegt oder nicht bietet eine stabile Aussage dazu, ob das Licht in der Wohnung tatsächlich geschaltet werden soll, oder nicht und minimiert das Problem der fehlerhaften *Ghost*-Erkennung.

4.1.2 Auswertung der openHAB-Anbindung

Die gewählte Anbindungsstrategie der Geräte in den *openHAB* über die REST-API funktioniert zuverlässig und zeigt hohe Skalierbarkeit. Limitierende Faktoren sind die beschränkte Auswahl der Datentypen und somit eine Notwendigkeit Informationen in String zu versenden. Obwohl das für die Informationen zu *SensFloor*- und *RotoFlex*-Zuständen ausreicht, kann es bei komplexeren Systemen notwendig sein die Integration zu erweitern. Entweder es wird die Integration über ein *Custom Binding* gewählt, welches über mehrere *Channels* eines *Thing* eine Vielzahl an Daten pro Gerät speichern kann, oder es werden - wie am Beispiel des *SensFloor* demonstriert - mehrere *Items* zu einem Gerät erstellt, welche einzelne Teile des Geräts abbilden (Räume im Falle des *SensFloor*).

Ein weiterer Nachteil der Anbindung ist die unidirektionale Kommunikation. Es werden aktuell nur Daten von den Sensoren an das *openHAB* geleitet, es ist jedoch nicht möglich Befehle von dem *openHAB* aus an die Geräte zu leiten. Dies ist zwar in der Integration von *RotoFlex* und *SensFloor* nicht notwendig, kann bei der Anbindung anderer Geräte aber erforderlich sein. Dafür kann entweder die *Thing*-Integration in Betracht gezogen werden, oder aber die *OpenHABHandler*-Klasse wird erweitert, sodass sie über die REST-API auch bspw. auf *Rule*-Ausführungen oder Zustandsänderungen von *Items* hören kann. Damit wäre die bidirektionale Kommunikation zwischen *openHAB* und externem Gerät geschaffen.

Obwohl das Programm größtenteils über die Konfigurationsdateien steuerbar ist, gibt es einen Aspekt der bei Änderungen manuell anzupassen ist, nämlich der Farbwert, der für die Auslösung der Dimmer-*Rule* zuständig ist. Zwar ist der Farbwert, der an das Farbe-*Item* des LED-Streifens im *openHAB* gesendet wird über die Konfigurationsdatei beliebig anpassbar, bei dieser Änderung ist es jedoch wichtig die *Rule-Trigger* manuell anzupassen, da das Licht sonst nicht korrekt abgeschalten wird. Außerdem gibt es für jeden Raum eine eigene Dimmer-*Rule*, die als Auslöser die Farbwertänderung eines einzelnen LED-Streifen hat. Bei drei Räumen ist der Anpassungssaufwand der *Rules* überschaubar. Leider ist es nicht möglich eine einzelne Dimmer-*Rule* für alle Räume zu entwickeln, da diese auf einem Thread laufen würde und es somit zu Problemen bei paralleler Auslösung kommen kann. Bei Umgebungen mit sehr vielen Räumen ist es sinnvoll eine Lösung zu entwickeln, in der auch die Dimmer-*Rule* in ein externes Python-Skript ausgelagert wird.

4.2 Fazit

Die Implementierung des *SensFloor*-Systems und des smarten Pflegebetts *RotoFlex* in die bestehende Smart Home Umgebung von *openHAB* 3 hat gezeigt, dass die Integration solcher Technologien erhebliche Verbesserungen für den Betrieb und das Management von AAL-Umgebungen ermöglichen kann. Die Ergebnisse der prototypischen Entwicklung eines nächtlichen Beleuchtungssystems basieren auf den realen Testdaten und den Simulationen, die im Rahmen dieser Diplomarbeit durchgeführt wurden. Die Ergebisse der Arbeit bestätigen, dass die Verwendung von *SensFloor* und *RotoFlex* als Grundlage für die Entwicklung von AAL-Automatisierungen, wie ein automatisiertes Beleuchtungssystem, genutzt werden können und das Potenzial haben, die Lebensqualität von Bewohnern zu verbessern. Die erfolgreiche Integration der Geräte in das zentrale Smart Home System *openHAB* zeigt, dass dieses auch für die Verwaltung weiterer Technologien genutzt werden kann und unterschiedliche Möglichkeiten zur Anbindung externer Geräte anbietet. Die Ergebnisse der Arbeit bieten eine Grundlage für weitere Integrationen von AAL-Technologien in den *openHAB* und die Ausarbeitung neuer Automatisierungen basierend auf den bereits angebundenen Geräten. Das spezifische Problem, dass die *SensFloor*-Daten Fehler beinhalten können, ist mit den zur Verfügung stehenden Mitteln nicht eindeutig lösbar, kann jedoch unter Verwendung von Deep Learning oder PIR-Sensoren verbessert werden.

Literatur

A.S.T. – Angewandte System Technik GmbH, Mess- und Regeltechnik (2024). *Leben im Alter mit Zukunft: Pflegebett*. Dresden, Germany. URL: https://www.ast.de.

A.S.T. GmbH (2024). *Homepage of A.S.T. Angewandte System Technik GmbH, Mess- und Regeltechnik*. https://www.ast.de/de/. Zuletzt aufgerufen am 01.04.2024.

Aarts, Emile, Rick Harwig und Martin Schuurmans (2001). „Ambient intelligence“. In: *The Invisible Future: The Seamless Integration of Technology into Everyday Life*. McGraw-Hill, Inc., S. 235–250. ISBN: 0071382240. DOI: 10.5555/504949.504964.

Arshad, Atika et al. (Apr. 2017). „A Review: Electric Field Sensing for Human-Computer Interaction Applications“. In: *International Journal of Smart Home* 11, S. 1–10. DOI: 10.14257/ijsh.2017.11.4.01.

Astral Documentation (2022). Online. Zuletzt aufgerufen am 03.03.2024. URL: https://astral.readthedocs.io/en/latest/.

Becks, Dr. Thomas et al. (2007). *Ambient Assisted Living. Neue 'intelligente' Assistenzsysteme für Prävention, Homecare und Pflege*. Frankfurt: Verband der Elektrotechnik Elektronik Informationstechnik.

Bertelsmann Stiftung Tobias Bürger, Regina Siedel (2020). *Jetzt Alle?! Digitale Souveränität von Älteren Eine Befragung zu digitalen Kompetenzen*. DOI: 10.11586/2020070.

Biehl, M. (2016). *RESTful API Design*. API-University Series. CreateSpace Independent Publishing Platform. ISBN: 9781514735169. URL: https://books.google.de/books?id=DYC3DwAAQBAJ.

BMBF (Apr. 2008). *Bekanntmachung des BMBF*. Zuletzt aufgerufen am 04.01.2024. URL: https://www.bmbf.de/bmbf/shareddocs/bekanntmachungen/de/2008/04/338_bekanntmachung.html.

Braun, A. et al. (Sep. 2016). „Ambient Assisted Living“. In: S. 203–222. ISBN: 978-3-662-49503-2. DOI: 10.1007/978-3-662-49504-9_10.

Braun, Andreas, Henning Heggen und Reiner Wichert (Jan. 2012). „CapFloor – A Flexible Capacitive Indoor Localization System“. In: S. 26–35. ISBN: 978-3-642-33532-7. DOI: 10.1007/978-3-642-33533-4_3.

Cheng, Yusi et al. (2020). „Design and Application of a Smart Lighting System Based on Distributed Wireless Sensor Networks“. In: *Applied Sciences* 10.23. ISSN: 2076-3417. DOI: 10.3390/app10238545.

Console (o. D.). Zuletzt aufgerufen am 16.03.2024. URL: https://www.openhab.org/docs/administration/console.html.

deCONZ REST API (o. D.). dresden-elektronik.de. Zuletzt aufgerufen am 05.05.2024. URL: https://dresden-elektronik.github.io/deconz-rest-doc/endpoints/lights/.

Destatis (2009). *Bevölkerung Deutschlands bis 2060 – 12. koordinierte Bevölkerungsvorausberechnung*. Wiesbaden: Statistisches Bundesamt.

Destatis (2010). *Demografischer Wandel in Deutschland. Auswirkungen auf Krankenhausbehandlungen und Pflegebedürftige im Bund und in den Ländern*. Wiesbaden: Statistisches Bundesamt.

– (2013). *Pflegestatistik 2011 - Pflege im Rahmen der Pflegeversicherung - Deutschlandergebnisse*. Wiesbaden. Zuletzt aufgerufen am 31.12.2023. URL: https://www.destatis.de/DE/Themen/Gesellschaft-Umwelt/Gesundheit/Pflege/Publikationen/Downloads-Pflege/pflege-kreisvergleich-5224103119004.pdf?__blob=publicationFile (besucht am 31. 12. 2023).

– (2023a). *Geburten in Deutschland*. Zuletzt aufgerufen am 30.12.2023. Statistisches Bundesamt. URL: https://www.destatis.de/DE/Themen/Gesellschaft-Umwelt/Bevoelkerung/Geburten/_inhalt.html (besucht am 30. 12. 2023).

– (2023b). *Sterbefälle und Lebenserwartung*. Zuletzt aufgerufen am 29.12.2023. Statistisches Bundesamt. URL: https://www.destatis.de/DE/Themen/Gesellschaft-Umwelt/Bevoelkerung/Sterbefaelle-Lebenserwartung/_inhalt.html (besucht am 29. 12. 2023).

Developing a Binding (o. D.). Zuletzt aufgerufen am 12.03.2024. URL: https://www.openhab.org/docs/developer/bindings/.

Einrichtung (o. D.). Zuletzt aufgerufen am 16.03.2024. URL: https://www.openhab.org/docs/tutorial/first_steps.html.

Emile Aarts, José Encarnação (2006). *True Visions. The Emergence of Ambient Intelligence*. Springer Berlin, Heidelberg. DOI: 10.1007/978-3-540-28974-6.

Events (o. D.). Zuletzt aufgerufen am 13.03.2024. URL: https://www.openhab.org/docs/developer/utils/events.html.

Exec Binding (o. D.). Zuletzt aufgerufen am 17.03.2024. URL: https://www.openhab.org/addons/bindings/exec/.

Fette, Ian und Alexey Melnikov (2011). *The WebSocket Protocol*. RFC 6455. RFC 6455. IETF. URL: https://datatracker.ietf.org/doc/html/rfc6455.

Fielding, Roy Thomas (2000). „Architectural Styles and the Design of Network-based Software Architectures“. Diss. University of California, Irvine.

Future-Shape GmbH (2022a). *SensFloor Demo Kit Manual*. Accessed: 2024-05-02. Future-Shape GmbH. Hoehenkirchen-Siegertsbrunn, Germany. URL: https://www.future-shape.com.

– (Feb. 2022b). *SensFloor Live API Documentation*. Technical Documentation. Hoehenkirchen-Siegertsbrunn, Germany: Future-Shape GmbH.

Georgieff, P. (2008). *Ambient Assisted Living - Marktpotenziale IT-unterstützter Pflege für ein selbstbestimmtes Altern*. MFG Stiftung Baden-Württemberg.

GmbH, Future Shape (2024). *Über Uns - Future Shape*. https://future-shape.com/uber-uns/. Zuletzt aufgerufen am 03.01.2024.

Hedtke-Becker, A. et al. (2012). „Zu Hause wohnen wollen bis zuletzt“. In: *Altern mit Zukunft*. Hrsg. von S. Pohlmann. Wiesbaden: VS Verlag für Sozialwissenschaften. DOI: 10.1007/978-3-531-19418-9_6.

Heinle, Stefan (2015). *Heimautomation mit KNX, DALI,1-Wire und Co*. Rheinwerk Computing. ISBN: 978-3-8362-3461-0.

Hoffmann, R. et al. (2015). „Increasing the Reliability of Applications in AAL by Distinguishing Moving Persons from Pets by Means of a Sensor Floor“. In: DOI: 10.5162/sensor2015/C5.4.

Hu, X. und W. Yang (2010). „Planar capacitive sensors – designs and applications". In: *Sensor Review* 30.1, S. 24–39. DOI: 10.1108/02602281011010772.

Items (o. D.). Zuletzt aufgerufen am 12.03.2024. URL: https://www.openhab.org/docs/concepts/items.html.

Klinger, Mathias und Prof. Hans-Joachim Böhme (Dez. 2020a). *Sachbericht AAL-Living Lab Cultus Bühlau*. Projektbericht. Zusammenfassung des Auf-, Aus- und Umbaugeschehens des AAL-Labors in der Cultus Wohnanlage in Bühlau im Zeitraum vom 01.02.2020 bis zum 31.12.2020. Dresden: HTW Dresden.

– (Juni 2020b). *Sachbericht AAL-Living Lab Cultus Bühlau*. Projektbericht. Bericht über das Auf-, Aus- und Umbaugeschehen des AAL-Labors in der Cultus Wohnanlage Bühlau für den Zeitraum vom 01.07.2019 bis zum 31.01.2020. Dresden: HTW Dresden.

– (Mai 2021). *Sachbericht AAL-Living Lab Cultus Bühlau*. Projektbericht. Bericht über das Auf-, Aus- und Umbaugeschehen des AAL-Labors in der Cultus Wohnanlage Bühlau für den Zeitraum vom 01.02.2020 bis zum 31.12.2020. Dresden: HTW Dresden.

Klingner, Mathias et al. (2022). „Experiences of Co-Creative Development within Co-Care Settings". In: *Proceedings of the 20th European Conference on Computer-Supported Cooperative Work: The International Venue on Practice-centred Computing on the Design of Cooperation Technologies - Exploratory Papers*. European Society for Socially Embedded Technologies.

Kreuzer, Kai (2010). *openHAB is out!* Zuletzt aufgerufen am 22.02.2024. URL: http://kaikreuzer.blogspot.com/2010/02/.

Kumar, Saurabh et al. (2023). *python-dotenv: Add .env support to your django/flask apps*. Zuletzt aufgerufen am 01.03.2024. URL: https://pypi.org/project/python-dotenv/.

Lauterbach, C. et al. (2013). „SensFloor - Sensitiver Bodenbelag zur Unterstützung selbständigen Lebens im Alter". In: DOI: 10.2314/GBV:785229361.

Logs (o. D.). Zuletzt aufgerufen am 16.03.2024. URL: https://www.openhab.org/docs/installation/linux.html.

Main UI (o. D.[a]). Zuletzt aufgerufen am 13.03.2024. URL: https://www.openhab.org/docs/ui/.

MainUI (o. D.[b]). Zuletzt aufgerufen am 16.03.2024. URL: https://next.openhab.org/docs/mainui/.

Memon, Mukhtiar et al. (März 2014). „Ambient Assisted Living Healthcare Frameworks, Platforms, Standards, and Quality Attributes". In: *Sensors (Basel, Switzerland)* 14, S. 4312–41. DOI: 10.3390/s140304312.

Models (o. D.). Zuletzt aufgerufen am 13.03.2024. URL: https://www.openhab.org/docs/tutorial/model.html.

Ni, Qin, Ana García Hernando und Iván Pau (Mai 2015). „The Elderly's Independent Living in Smart Homes: A Characterization of Activities and Sensing Infrastructure Survey to Facilitate Services Development". In: *Sensors* 15, S. 11312–11362. DOI: 10.3390/s150511312.

OAuth Community (2024). *OAuth 2.0*. https://oauth.net/2/. Zuletzt aufgerufen am 04.05.2024.

openHAB Foundation (2024a). *openHAB 3 MainUI*. Lokale Installation, erreichbar über localhost:8080. Screenshot im Rules-Menü.

openHAB Foundation (2024b). *openHAB 3 MainUI*. Lokale Installation, erreichbar über localhost:8080. Item-Liste des Things für LED-Streifen.

Osoinach, Bryce (2008). *Proximity Capacitive Sensor Technology for Touch Sensing Applications*. White Paper. Document Number: PROXIMITYWP, REV 1. Freescale Semiconductor, Inc. URL: http://www.freescale.com.

Philips Binding (o. D.). Zuletzt aufgerufen am 16.03.2024. URL: https://www.openhab.org/addons/bindings/hue/.

PhysioNova GmbH (2024). *Rotoflex - PhysioNova*. https://www.physionova.de/de/produkte/rotoflex/. Zuletzt aufgerufen am 03.03.2024.

Piccardi, M. (2004). „Background subtraction techniques: a review“. In: *2004 IEEE International Conference on Systems, Man and Cybernetics (IEEE Cat. No.04CH37583)*. Bd. 4, 3099–3104 vol.4. DOI: 10.1109/ICSMC.2004.1400815.

Plitnick, Barbara et al. (2010). „The effects of red and blue light on alertness and mood at night“. In: *Lighting Research & Technology* 42.4, S. 449–458. DOI: 10.1177/1477153509360887.

Preißler, J. et al. (2016). *Akzeptanz von Ambient-Assisted-Living-Lösungen: Befragung von Seniorinnen und Senioren im Landkreis Görlitz*. Techn. Ber. Zuletzt aufgerufen am 01.12.2023. Görlitz: Forschungsbericht. URL: https://nbn-resolving.org/urn:nbn:de:0168-ssoar-47109-5.

Python, Why (2021). „Python“. In: *Python releases for windows* 24.

Python Software Foundation (2023). *subprocess — Subprocess management*. https://docs.python.org/3/library/subprocess.html. Zuletzt aufgerufen am 01.05.2024.

– (2024a). *18. Asynchronous I/O*. https://docs.python.org/3/library/asyncio.html. Zuletzt aufgerufen am 05.03.2024.

– (2024b). *8. Data Classes*. https://docs.python.org/3/library/dataclasses.html. Zuletzt aufgerufen am 05.03.2024.

– (2024c). *Event Loop — Python 3 documentation*. https://docs.python.org/3/library/asyncio-eventloop.html. Zuletzt aufgerufen am 29.04.2024.

Radtke, R. (Aug. 2023). *Anzahl der Pflegebedürftigen in Deutschland in den Jahren 1999 bis 2021*. URL: https://de.statista.com/statistik/daten/studie/2722/umfrage/pflegebeduerftige-in-deutschland-seit-1999/ (besucht am 01. 12. 2023).

Ramkumar, M. et al. (2019). „Survey of Cognitive Assisted Living Ambient System Using Ambient Intelligence as a Companion“. In: *Proceedings of the International Conference on System, Computation, Automation and Networking (ICSCAN)*. DOI: 10.1109/ICSCAN.2019.8878707.

Rashidi, Parisa und Alex Mihailidis (2013). „A Survey on Ambient-Assisted Living Tools for Older Adults“. In: *IEEE Journal of Biomedical and Health Informatics* 17.3, S. 579–590. DOI: 10.1109/JBHI.2012.2234129.

REST API (o. D.). Zuletzt aufgerufen am 12.03.2024. URL: https://www.openhab.org/docs/configuration/restdocs.html.

Ruettgers, Oscar (2022). *Philips Hue Lampen per Homematic Skript steuern*. Zuletzt aufgerufen am 27.04.2024. URL: https://www.oscarruettgers.de/hausautomation/philips-hue-lampen-per-homematic-skript-steuern/.

Rules (o. D.). Zuletzt aufgerufen am 12.03.2024. URL: https://www.openhab.org/docs/concepts/rules.html.

Scripts (o. D.). Zuletzt aufgerufen am 13.03.2024. URL: https://www.openhab.org/addons/automation/jsscripting/.

Shi, Qiongfeng et al. (Sep. 2020). „Deep learning enabled smart mats as a scalable floor monitoring system“. In: *Nature Communications* 11.1. ISSN: 2041-1723. DOI: 10.1038/s41467-020-18471-z. URL: https://doi.org/10.1038/s41467-020-18471-z.

Socket.IO Team (2024). *Socket.IO Documentation (v4)*. https://socket.io/docs/v4/. Zuletzt aufgerufen am 26.04.2024.

Statista Research Department (2023). *Umfrage zur Nutzung des Internets bei Personen ab 60 Jahren in Deutschland im Jahr 2023*. Zuletzt aufgerufen am 31.10.2023. URL: https://de.statista.com/statistik/daten/studie/1100772/umfrage/internetnutzung-von-senioren/ (besucht am 31. 10. 2023).

Strick, Heinz Klaus (2019). „Der Satz des Pythagoras“. In: *Mathematik ist schön: Anregungen zum Anschauen und Erforschen für Menschen zwischen 9 und 99 Jahren*. Berlin, Heidelberg: Springer Berlin Heidelberg, S. 333–370. ISBN: 978-3-662-59060-7. DOI: 10.1007/978-3-662-59060-7_17. URL: https://doi.org/10.1007/978-3-662-59060-7_17.

Strube, A. (2011). „Ich würde gerne hier bleiben...“ In: *Sozial Extra* 35, S. 25–29. DOI: 10.1007/s12054-011-0186-9.

Sun, Hong et al. (Jan. 2009). „Promises and Challenges of Ambient Assisted Living Systems“. In: S. 1201–1207. DOI: 10.1109/ITNG.2009.169.

Things (o. D.). Zuletzt aufgerufen am 12.03.2024. URL: https://www.openhab.org/docs/concepts/things.html.

Tsakalidis, Sotirios et al. (Juni 2023). *Design and Implementation of a Versatile openHab IoT Testbed with a Variety of Wireless Interfaces and Sensors*. DOI: 10.20944/preprints202306.0343.v1.

Usikalu, Mojisola (Jan. 2018). „International Conference on Science and Sustainable Development“. In.

Vogler, Christine und Annemarie Fajardo (Nov. 2023). *DPR Newsletter November 2023*. Published by Deutscher Pflegerat e.V. Available online: [URL]. Alt-Moabit 91, 10559 Berlin, Germany.

Walter, U. et al. (2006). „Krankheitslast und Gesundheit im Alter“. In: *Bundesgesundheitsblatt - Gesundheitsforschung - Gesundheitsschutz*, S. 537–546.

Wang, X. et al. (2018). „A Highly Stretchable Transparent Self-Powered Triboelectric Tactile Sensor with Metallized Nanofibers for Wearable Electronics“. In: *Advanced Materials*. DOI: 10.1002/adma.201706738.

Welcome to openHAB (o. D.). Zuletzt aufgerufen am 22.02.2024. URL: https://www.openhab.org/docs/.

Wikipedia (2024). *HSV-Farbraum — Wikipedia, The Free Encyclopedia*. http://de.wikipedia.org/w/index.php?title=HSV-Farbraum&oldid=242801276. Zuletzt aufgerufen am 29.04.2024.

Yang, C. und Y. Hsu (2010). „A Review of Accelerometry-Based Wearable Motion Detectors for Physical Activity Monitoring“. In: *Sensors*. DOI: 10.3390/s100807772.

Yang, J. et al. (2020). „Ultrasoft liquid metal elastomer foams with positive and negative piezopermittivity for tactile sensing“. In: *Advanced Functional Materials* 30.36. DOI: 10.1002/adfm.202002611.

Zuk, S., A. Pietrikova und I. Vehec (2018). „Possibilities of planar capacitive rain sensor manufacturing by thick film technology“. In: *Acta Electrotechnica Et Informatica* 18.4, S. 11–16. DOI: 10.15546/aeei-2018-0027.

A Erweiterte Codeausschnitte

```
1 now = datetime.now()
2 logger.debug(now)
3 sum = 1
4 for count in range(1000):
5     sum += 1
6 logger.debug(datetime.now())
```

Listing 43: Performance Test Python

```
1 logger.debug((zdt.now()));
2 var sum = 1;
3 for (var count = 0; count < 1000; count++) {
4   sum += 1;
5 }
6 logger.debug((zdt.now()));
```

Listing 44: Performance Test *openHAB Script*

```
async def run_tasks():
    openhab_connection_handler = OpenHABConnectionHandler()

    sensfloor_monitor_hall = SensFloorMonitor()
    sensfloor_monitor_livingroom = SensFloorMonitor()
    sensfloor_monitor_bedroom = SensFloorMonitor()

    sensors: dict[str, Equipment] = {
        'Hall': Equipment(
                queue=Queue(),
                connect_method=
                sensfloor_monitor_hall.connect_and_receive_sensfloor_data,
                data=('Bootup SensFloor', False),
                prev_data=('', False),
                name='Hall'),
        'LivingRoom': Equipment(
                queue=Queue(),
                connect_method=
                sensfloor_monitor_livingroom.connect_and_receive_sensfloor_data,
                data=('Bootup SensFloor', False),
                prev_data=('', False),
                name='LivingRoom'),
        'BedRoom': Equipment(
                queue=Queue(),
                connect_method=
                sensfloor_monitor_bedroom.connect_and_receive_sensfloor_data,
                data=('Bootup SensFloor', False),
                prev_data=('', False),
                name='BedRoom'),
        'RotoFlex': Equipment(
                queue=Queue(),
                connect_method=
                connect_and_receive_rotoflex_data,
                data='Bootup RotoFlex',
                prev_data='',
                name='RotoFlex')
    }

    tasks = [sensor.start_connection() for sensor in sensors.values()]
    await asyncio.gather(*tasks)
    await openhab_connection_handler.evaluate_data(sensors)
```

Listing 45: *run_tasks*-Methode

```
class Logger:
    _instance = None
    def __new__(cls, log_file_path: Optional[str] = 'log.txt') -> 'Logger':
        if not cls._instance:
            cls._instance = super(Logger, cls).__new__(cls)
            cls._instance._initialize_logger(log_file_path)
        return cls._instance

    def _initialize_logger(self, log_file_path: str):
        self.logger = logging.getLogger('project_logger')
        self.logger.handlers.clear()
        self.logger.setLevel(logging.DEBUG)
        file_handler = logging.FileHandler(log_file_path)
        file_handler.setLevel(logging.DEBUG)
        console_handler = logging.StreamHandler()
        console_handler.setLevel(logging.INFO)
        formatter = logging.Formatter(
                    '%(asctime)s - %(levelname)s - %(message)s')
        file_handler.setFormatter(formatter)
        console_handler.setFormatter(formatter)
        self.logger.addHandler(file_handler)
        self.logger.addHandler(console_handler)
        self.logger.propagate = False
        run_counter = 0
        if os.path.exists('run_counter.txt'):
            with open('run_counter.txt', 'r') as file:
                run_counter = int(file.read().strip())
            if run_counter < MAX_RUNS:
                run_counter += 1
            else:
                self.info('Clearing log.txt file')
                run_counter = 0
                open(log_file_path, 'w').close()
            with open('run_counter.txt', 'w') as file:
                file.write(str(run_counter))
```

Listing 46: Initialisierung der Logger-Klasse als Singleton

```
def update_object_state(self, obj_id: str, new_state: str) -> bool:
    state_history = self.object_states.setdefault(
                    obj_id,
                    {
                        'current_state': self.last_state,
                        'counter': 0,
                        'locations': []
                    }
                )
    if state_history['current_state'] == new_state:
        state_history['counter'] = 0
    else:
        consistency = 0

        if state_history['current_state']:
            state_history['counter'] += 1
            if state_history['current_state'] == 'Human':
                consistency = self.consistency_threshold_human
            elif state_history['current_state'] == 'Ghost' and
                new_state == 'Human':
                consistency = self.consistency_threshold_ghost
            elif state_history['current_state'] == 'No presence' and
                new_state == 'Human':
                consistency = self.consistency_threshold_no_presence
            else:
                consistency = 10
            if state_history['counter'] >= consistency:
                state_history['current_state'] = new_state
                state_history['counter'] = 0
                return True
        else:
            state_history['current_state'] = new_state
            state_history['counter'] = 0
            return True
    return False
```

Listing 47: *update_objects_state*-Methode

```
consistency_threshold_human: int = 50
consistency_threshold_ghost: int = 20
consistency_threshold_no_presence: int = 2
min_steps_for_human: int = 5
significant_movement_threshold: float = 0.02
empty_updates_threshold: int = 20
activation_time_threshold: int = 60_000
weight_threshold: int = 50
no_objects_threshold: int = 50
age_since_birth: int = 300_000
time_since_last_change: int = 45_000
location_tracking_amount: int = 20
```

Listing 48: Einstellungen für die *Ghost*-Erkennung

Erklärung über die eigenständige Erstellung der Arbeit

Hiermit erkläre ich, dass ich die vorgelegte Arbeit mit dem Titel 'Integration eines Sensorfußbodens und eines smarten Pflegebetts in openHAB 3' selbstständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie alle wörtlich oder sinngemäß übernommenen Stellen in der Arbeit als solche und durch Angabe der Quelle gekennzeichnet habe. Dies gilt auch für Zeichnungen, Skizzen, bildliche Darstellungen sowie für Quellen aus dem Internet. Mir ist bewusst, dass die Hochschule für Technik und Wirtschaft Dresden Prüfungsarbeiten stichprobenartig mittels der Verwendung von Software zur Erkennung von Plagiaten überprüft.

Ort, Datum   Unterschrift