Universität Bremen

Fachbereich 3: Mathematik / Informatik

Integration eines Debuggers in ein Unreal Engine Projekt

Bachelorarbeit
im Studiengang Informatik

Jannik Bohling
Matrikelnummer: 2925458

Erstgutachter: Prof. Dr. rer.nat. Rainer Koschke
Zweitgutachter: Prof. Dr.-Ing. Gabriel Zachmann

13. März 2018

Inhaltsverzeichnis

0 Vorwort

Die vorliegende schriftliche Ausarbeitung setzt voraus, dass der Leser über grundlegende Unreal Engine und Java Kenntnisse verfügt. So wird unter anderen nicht genauer erörtert was zum Beispiel eine Blueprint[1] ist.

Des Weiteren gilt es anzumerken, dass für die Bearbeitung der Bachelorarbeit kein Budget zur Verfügung stand.

Die für die Bearbeitung der Bachelorarbeit benutzten Programme finden Sie im Anhang unter A.23 Programmübersicht. Außerdem beinhaltet der Anhang unter A.24 eine Auflistung der Inhalte der beiliegenden Disk.

1 Problemstellung

Für ein potentielles zukünftiges Forschungsprojekt der Software AG gilt es zu prüfen, ob beziehungsweise wie ein Debugger in Einklang mit einem Unreal Engine Projekt gebracht werden kann.

Ziel des Forschungsprojektes ist es zu untersuchen, inwiefern Virtual Reality in der Softwareentwicklung als Hilfsmittel eingesetzt werden kann. Dabei sind die genauen Anwendungsfälle noch nicht definiert, ein solches Tool könnte bei der Qualitätssicherung oder bei der Verbesserung der User Experience unterstützen. So soll es zum Beispiel möglich sein, dass sich mehrere Personen, unter anderem Entwickler, Designer, Tester und Endverbraucher, per Virtual Reality treffen und eine abstrahierte Form eines Programmes betrachten, debuggen und benutzen können. Dazu muss geprüft werden, wie ein Debugger in ein Virtual Reality Projekt integriert werden kann.

Es wird explizit die Verknüpfung eines Debuggers mit einem Unreal Engine Projekt untersucht, da es bereits ein Bachelorprojekt gibt, welches auf der Unreal Engine basiert und sich mit der Darstellung von Codecitys in Virtual Reality befasst.

2 Zielsetzung

Damit ein Unreal Engine Projekt in die Lage versetzt wird ein Programm zu debuggen, muss es um Debug-Funktionalitäten ergänzt werden. Hierbei besteht die erste Herausforderung im Wesentlichen in der Prüfung der verschiedenen Arten einer möglichen Einbindung.

Die vielversprechende Möglichkeit, einer solchen Integration, muss im Anschluss genauer untersucht werden. Dafür ist es notwendig, dass diese implementiert wird.

Im Mittelpunkt der Implementierung steht die Kommunikation einer Debug-Schnittstelle mit einer aktiven Unreal Instanz. Dabei ist es ausreichend, wenn es in Unreal möglich ist einfache Debug-Tätigkeiten, wie zum Beispiel das Beobachten einer Variablen, auszuüben. Die Darstellung von Programmen und Programmcode (Klassen, Methoden etc.) ist kein Bestandteil der Implementierung. Diese Darstellung wird höchstens in Form von Pseudo-Projektdaten integriert, damit sich das Verhalten der Debug-Funktionalitäten einfacher testen und nachvollziehen lässt. Es wird sich zunächst auf das Debuggen von Java-Programmen beschränkt.

Die Debug-Funktionalitäten sind zunächst auf Attribute Watcher (Beobachtungsanker) und Breakpoints beschränkt. Dabei soll es einem Nutzer ermöglicht werden diese innerhalb des Unreal Engine Projektes zu setzen. Zusätzlich sollen die Ergebnisse der Watcher und Breakpoints textuell dargestellt werden. Zudem soll es möglich sein das zu debuggende Programm bis zum nächsten Haltepunkt/Beobachtungshalt weiterlaufen zulassen. Weiterführende Debug Funktionalitäten, wie z.B. die Navigation innerhalb von Codeblöcken oder die Ausführung von Code Snippets, sind kein Bestandteil der Arbeit, da diese erweiterten Funktionalitäten den zeitlichen Rahmen sprengen würden. Selbiges gilt auch für inline Breakpoints.

Da es später ggf. möglich sein soll einen Abhängigkeitsgraph in Virtual Reality darzustellen, soll außerdem untersucht werden, inwiefern es möglich ist Live-Trace-Daten zu übermitteln. Die Live-Trace-Daten beschränken sich hierbei im ersten Schritt auf die Methoden- und Klassennamen.

3 Integrationsmöglichkeiten

Dieses Kapitel beinhaltet die Betrachtung, sowie die Bewertung verschiedener Integrationsmöglichkeiten eines Debuggers in ein Unreal Engine 4 Projekt. Außerdem wird abschließend die vielversprechende Integrationsart bestimmt.

3.1 Identifizierung verschiedener Integrationsmöglichkeiten

Zunächst einmal gibt es mindestens zwei verschiedene Herangehensweisen an solch eine Integration eines Java Debuggers. Zum einen könnte ein in C++ geschriebener Debugger direkt in ein Unreal Projekt inkludiert werden, zum anderen könnte einen Debugger Microservice[A] erstellen werden, welcher die entkoppelte Debugger Logik enthält.

Die erste Variante kann nicht umgesetzt werden, da die dafür benötigten C++ Kenntnisse nicht vorhanden sind und eine Aneignung dieser, unter Berücksichtigung der zeitlichen Beschränkung der Arbeit, als unrealistisch eingestuft wird.

Die zweite Art der Integration scheint hingegen realisierbar und bietet zusätzlich den Vorteil, dass der Microservice bei entsprechendem Schnittstellendesign, später durch eine andere Debugger Komponente ersetzt werden kann, welche das Debuggen einer anderen Programmsprache ermöglicht.

Es gilt also zu klären, wie das Unreal Projekt mit dem zu erstellenden Microservice kommuniziert. Die Kommunikation zwischen den beiden Komponenten muss bidirektional sein, da es ihnen möglich sein muss Nachrichten an die jeweils andere Komponente zu schicken. So muss der Microservice das Unreal Projekt zum Beispiel über das Erreichen eines Haltepunktes informieren und das Unreal Projekt muss logischerweise auch in der Lage sein Haltepunkte anzulegen. Im Folgenden werden mögliche Arten einer solchen Kommunikation aufgelistet und genauer beleuchtet.

Mögliche Kommunikationsformen sind:

- Kommunikation per Dateisystem
- Datenbankanbindung
- Nutzung einer Warteschlange (Queue)
- HTTP Kommunikation
- Socket Kommunikation

[A] Ein Microservice ist eine erweiterbare unabhängige Softwarekomponente. Das Entwurfsmuster der Microservices kann dazu benutzt werden komplexe Architekturen zu entkoppeln.

3.1.1 Kommunikation per Dateisystem

Eine rudimentäre Kommunikation per Dateisystem könnte wie folgt aussehen: Beiden Komponenten wird ein **Event Creation**-Pfad und einen **Event Listener**-Pfad zugewiesen. Dabei ist der **Event Creation**-Pfad der einen Komponente der **Event Listener**-Pfad der anderen Komponente und umgekehrt. Eine Komponente könnte in diesem Fall Event-Dateien unter ihrem **Event Creation**-Pfad ablegen. Diese könnten dann wiederum von der anderen Komponente zeitnah verarbeitet werden. Diese Kommunikationsform würde allerdings erfordern, dass beide Systeme permanent ihren **Event Listener**-Pfad, wie in der „Abbildung 1: Event Listener Skizze“ skizziert, auf neue Events überprüfen.

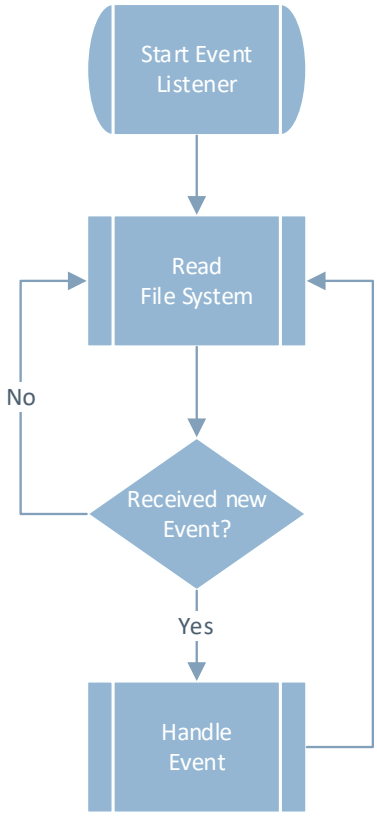

Abbildung 1: Event Listener Skizze

Es ist außerdem noch anzumerken, dass die Unreal Engine eine einfach zu nutzende Form des Dateimanagements beinhaltet. Diese ist ausführlich im Unreal Engine Wiki dokumentiert: *File Management, ...* [2].

3.1.2 Datenbankanbindung

Auch die Nutzung einer Datenbank als Kommunikationsmittel ist vorstellbar. Diese Kommunikation würde dann ähnlich wie die Kommunikation per Dateisystem ablaufen. Statt der Pfade würden den jeweiligen Komponenten jedoch Äquivalente in Form von Tabellen zugeordnet. Diese Tabellen würden dann die abzuarbeitenden Events enthalten. Auch diese Kommunikationsform würde allerdings erfordern, dass beide Systeme permanent ihre Event-Tabellen überprüfen, ähnlich wie in der „Abbildung 1: Event Listener Skizze“ skizziert.

Leider bietet Unreal von Haus aus keine direkte Datenbankintegration an. Jedoch ist es möglich vorhandene C / C++ Bibliotheken zu nutzen. Alternativ dazu kann auch auf Plugins zurückgegriffen werden, welche auf dem *Unreal Engine Marketplace* [B] [3] angeboten werden. Eines dieser Datenbank Plugins ist under anderem *MySQL Integration* [4].

3.1.3 Nutzung einer Warteschlange (Queue)

Es wäre auch möglich eine Warteschlange (Queue) als Bindeglied zwischen den zwei Systemen zu installieren. Als mögliche Queue kommt zum Beispiel die *RabbitMQ* [5] in Frage, welche unter der *Mozilla Public License* [6] läuft, da diese sowohl eine *C++ Anbindung* [7] als auch eine *Java Anbindung* [8] bereitstellt.

3.1.4 HTTP Kommunikation

Der Datenaustausch könnte auch per HTTP [C] erfolgen. Unreal bietet ein umfangreiches HTTP Modul. Die Verwendung dieses Moduls ist ebenfalls im Unreal Engine Wiki dokumentiert (*How To Make HTTP GET Requests in C++* [9]). Zu erwähnen ist, dass die Unreal Engine nicht direkt als HTTP Server fungieren kann. Sie bietet zwar wie zuvor beschrieben ein HTTP Modul, dieses beschränkt sich aber auf HTTP Calls. Wenn das Unreal Projekt dennoch einen HTTP Server bereitstellen soll, dann muss auf eine C / C++ Bibliothek zurückgegriffen werden oder auf das kostenpflichtige *Unreal Web Server Plugin* [10].

3.1.5 Socket Kommunikation

Auch TCP [D] Sockets könnten als Kommunikationsendpunkte dienen. Die Unreal Engine bietet auch hier ein Modul, welches ebenfalls im Wiki dokumentiert ist. Eine beispielhafte Nutzung wird im *TCP Socket Listener, Receive Binary Data From an IP/Port Into UE4, (Full Code Sample)* [11] Tutorial beschrieben.

[B] Der Unreal Engine Marketplace ermöglicht es Entwicklern ihre Unreal Plugins anzubieten. Nicht alle Plugins sind kostenpflichtig.
[C] HTTP = Hypertext Transfer Protocol
[D] TCP = Transmission Control Protocol

3.2 Bewertung der identifizierten Kommunikationsarten

Zur Bewertung der verschiedenen Kommunikationsformen werden gewichtete Entscheidungsmatrizen, welche auf der von [BEP18] vorgestellten Vorlage basieren, genutzt. Dafür werden zunächst die wichtigsten Kriterien, sowie deren Gewichtung festgelegt. Anschließend werden die jeweiligen Kommunikationsarten nacheinander bewertet. Dabei werden Ausreißer innerhalb der Bewertungen genauer erörtert.

[Gewichtungen] Kriterien:

- $[5]$ Aufwand (C++): Da die C++ Kenntnisse des Entwicklers höchstens als ausreichend beschrieben werden können, gilt es einen hohen C++ Entwicklungsaufwand zu vermeiden, da sonst gegebenenfalls die beispielhafte Implementierung nicht innerhalb der Bearbeitungszeit abgeschlossen werden kann.
- $[2]$ Aufwand (Java): Die Java Kenntnisse des Entwicklers können als umfangreich angesehen werden. Deshalb ist ein hoher Java Entwicklungsaufwand nicht so risikobehaftet wie ein hoher C++ Aufwand.
- $[2]$ Trennbarkeit: Es sollte möglich sein die beiden zu entwickelnden Komponenten später separat auf verschiedenen Maschinen laufen zu lassen.
- $[2]$ Wartbarkeit: Auf die Wartbarkeit der zu erstellenden Komponenten sollte kein all zu großer Fokus gelegt werden, da das Endprodukt lediglich als Machbarkeitsstudie fungieren soll.
- $[4]$ Nachvollziehbarkeit: Das zu entwickelnde Produkt soll als Machbarkeitsstudie dienen. Deshalb muss es verständlich und sauber sein, so dass es späteren Betrachtern möglich ist die Entscheidungen des Entwicklers anhand des Quellcodes nachzuvollziehen.

Die Aufwandsschätzungen, *Aufwand (C++)* und *Aufwand (Java)*, basieren auf den für das Kapitel 3.1 Identifizierung verschiedener Integrationsmöglichkeiten recherchierten Informationen, sowie auf den Berufserfahrungen[E] des Entwicklers.

Bewertungsskala:
Die jeweiligen Kriterien werden unter Berücksichtigung folgender Skala bewertet:

5 = sehr gut, 4 = gut, 3 = befriedigend, 2 = ausreichend, 1 = mangelhaft

[E] Software Entwickler im Bereich E-Commerce seit 2012 bei der neuland – Büro für Informatik GmbH [12]

3.2.1 Bewertung: Kommunikation per Dateisystem

Kriterien	Gewichtung	Bewertung	gewichteter Wert
Aufwand (C++)	**5**	**3**	**15**
Aufwand (Java)	**2**	**3**	**6**
Trennbarkeit	**2**	**1**	**2**
Wartbarkeit	**2**	**2**	**4**
Nachvollziehbarkeit	**4**	**2**	**8**
Summe	**-**	**-**	**35**

Tabelle 1: Entscheidungsmatrix - Kommunikation per Dateisystem

Die Kommunikation per Dateisystem kommt nach der Bewertung auf einen gewichteten Wert von 35 (siehe „Tabelle 1: Entscheidungsmatrix - Kommunikation per Dateisystem“). Die Trennbarkeit wurde mit 1 (mangelhaft) bewertet, da hierfür die Nutzung eines Netzwerklaufwerkes oder ähnlicher Technologien notwendig wäre.

3.2.2 Bewertung: Datenbankanbindung

Kriterien	Gewichtung	Bewertung	gewichteter Wert
Aufwand (C++)	**5**	**1**	**5**
Aufwand (Java)	**2**	**3**	**6**
Trennbarkeit	**2**	**4**	**8**
Wartbarkeit	**2**	**3**	**6**
Nachvollziehbarkeit	**4**	**2**	**8**
Summe	**-**	**-**	**33**

Tabelle 2: Entscheidungsmatrix - Kommunikation per Datenbankanbindung

Das Ergebnis der „Tabelle 2: Entscheidungsmatrix - Kommunikation per Datenbankanbindung“ lautet 33. Da kein Budget für den Kauf von Unreal Engine Plugins eingeplant ist, schreckt insbesondere die Anbindung des Unreal Engine Projektes an eine Datenbank stark ab, da der dafür geschätzte Aufwand enorm hoch ist.

3.2.3 Bewertung: Nutzung einer Warteschlange (Queue)

Kriterien	Gewichtung	Bewertung	gewichteter Wert
Aufwand (C++)	**5**	**2**	**10**
Aufwand (Java)	**2**	**3**	**6**
Trennbarkeit	**2**	**4**	**8**
Wartbarkeit	**2**	**3**	**6**
Nachvollziehbarkeit	**4**	**3**	**12**
Summe	**-**	**-**	**42**

Tabelle 3: Entscheidungsmatrix - Nutzung einer Warteschlange (Queue)

Die Nutzung einer Warteschlange (Queue) als Kommunikationsmittel kommt auf einen gewichteten Wert von 42 (siehe „Tabelle 3: Entscheidungsmatrix - Nutzung einer Warteschlange (Queue)“).

3.2.4 Bewertung: HTTP Kommunikation

Die Bewertung der HTTP Kommunikation wurde in zwei Subbewertungen unterteilt, da sich der C++ Aufwand je nach Kommunikationsrichtung (Java Server $\leftrightarrow$ Unreal Engine Client bzw. Unreal Engine Server $\leftrightarrow$ Java Client) stark unterscheidet. Beide Varianten verfügen aber über gemeinsame Stärken, so gibt es zahlreiche Java HTTP Frameworks, unter anderem das *Spring Framework* [13], welche als HTTP Server und Client fungieren können. Außerdem bietet die Verwendung von HTTP eine hohe Trennbarkeit. Jedoch sind beide Kommunikationsrichtungen jeweils unidirektional und nur gemeinsam bidirektional, da ein HTTP Server zwar mit einer Reponse auf einen Request eines Clients antworten kann, es ist ihm aber weiterhin nicht möglich dem Client eine Nachricht zu senden, ohne dass diese Teiler einer Reponse ist.

Kriterien	Gewichtung	Bewertung	gewichteter Wert
Aufwand (C++)	**5**	**5**	**25**
Aufwand (Java)	**2**	**5**	**10**
Trennbarkeit	**2**	**5**	**10**
Wartbarkeit	**2**	**3**	**6**
Nachvollziehbarkeit	**4**	**4**	**16**
Summe	**-**	**-**	**67**

Tabelle 4: Entscheidungsmatrix - HTTP Kommunikation: C++ Client $\leftrightarrow$ Java Server

Die HTTP Kommuikation (C++ Client ↔ Java Server) kommt auf einen gewichteten Wert von 67 (siehe „Tabelle 4: Entscheidungsmatrix - HTTP Kommunikation: C++ Client ↔ Java Server“). Gerade der C++ Aufwand wurde als 5 (sehr gut) bewertet, da die Unreal Engine das Erstellen von HTTP Requests in Form eines Moduls direkt anbietet und somit eine potenzielle Integration einer Fremdbibliotheken obsolet macht.

Kriterien	Gewichtung	Bewertung	gewichteter Wert
Aufwand (C++)	**5**	**1**	**5**
Aufwand (Java)	**2**	**5**	**10**
Trennbarkeit	**2**	**5**	**10**
Wartbarkeit	**2**	**3**	**6**
Nachvollziehbarkeit	**4**	**4**	**16**
Summe	**-**	**-**	**47**

Tabelle 5: Entscheidungsmatrix - HTTP Kommunikation: C++ Server ↔ Java Client

Der resultierende gewichtete Wert der „Tabelle 5: Entscheidungsmatrix - HTTP Kommunikation: C++ Server ↔ Java Client“ beträgt hingegen nur 47, da der C++ Aufwand mit 1 (mangelhaft) bewertet wurde. Diese Bewertung resultiert aus der Notwendigkeit einer Einbindung eines C++ HTTP Server Frameworks. Es gilt auch hier zu betonen, dass kein Budget für den Erwerb eines HTTP Server Plugins zur Verfügung steht.

3.2.5 Bewertung: Socket Kommunikation

Kriterien	Gewichtung	Bewertung	gewichteter Wert
Aufwand (C++)	**5**	**4**	**20**
Aufwand (Java)	**2**	**4**	**8**
Trennbarkeit	**2**	**5**	**10**
Wartbarkeit	**2**	**3**	**6**
Nachvollziehbarkeit	**4**	**4**	**16**
Summe	**-**	**-**	**60**

Tabelle 6: Entscheidungsmatrix - Socket Kommunikation

Die TCP Socket Kommunikation erreichte einen gewichteten Wert von 60 (siehe „Tabelle 6: Entscheidungsmatrix - Socket Kommunikation“). Ähnlich wie die HTTP Kommunikation erzielt auch diese Kommunikationsart eine hohe Trennbarkeit. Auch die Entwicklungsaufwände wurden mit 2 (gut) bewertet, da ein Unreal Engine Network Modul vorhanden ist, welches TCP Socket Kommunikationsmittel bereitstellt. Auch Java beinhaltet Hilfsmittel zur Socket Kommunikation (*java.net*[14]).

3.3 Bestimmung des zu implementierenden Integrationsverfahrens

Zur besseren Übersicht wurden die Ergebnisse der Entscheidungsmatrizen aus dem Kapitel 3.2 Bewertung der identifizierten Kommunikationsarten noch einmal in einer separaten Tabelle aufgelistet (siehe „Tabelle 7: Ergebnisse der Entscheidungsmatrizen“).

Kommunikationsart	Kommunikation per Dateisystem	Kommunikation per Datenbank	Nutzung einer Warteschlange (Queue)
gewichteter Wert	35	33	42
Kommunikationsart	**HTTP Kommunikation: C++ Client ↔ Java Server**	**HTTP Kommunikation: C++ Server ↔ Java Client**	**Socket Kommunikation**
gewichteter Wert	67	47	60

Tabelle 7: Ergebnisse der Entscheidungsmatrizen

Die *Kommunikation per Dateisystem* (gewichteter Wert 35) und die *Kommunikation per Datenbank* (gewichteter Wert 33), sowie die *Nutzung einer Warteschlange (Queue)* (gewichteter Wert 42) werden nicht weiter betrachtet, da diese Kommunikationsarten am schlechtesten bewertet wurden.

Das am besten bewertete Kommunikationsverfahren, mit einem gewichten Wert von 67, ist die *HTTP Kommunikation: C++ Client ↔ Java Server*. Dieses Verfahren ist jedoch unidirektional, und somit nicht als alleiniges Kommunikationsverfahren ausreichend, da die Kommunikation zwischen den Systemen bidirektional sein muss. Das Gegenstück, die *HTTP Kommunikation: C++ Server ↔ Java Client*, kommt nur auf einen gewichteten Wert von 47 und scheint ein hohes Risikopotential zu beinhalten, da der C++ Aufwand mit 1 (mangelhaft) bewertet wurde.

Es wäre auch möglich auf eine *Socket Kommunikation* zu setzen, da diese einen gewichteten Wert von 60 erzielt und damit als bestbewertetes Kommunikationsverfahren abschneidet, wenn die Bewertungen der beiden HTTP Kommunikationsrichtungen zusammenfasst werden:

$$
\begin{aligned}
GW_{HTTP_1} &= \text{gewichteter Wert HTTP Kommunikation: C++ Client} \leftrightarrow \text{Java Server} = 67 \\
GW_{HTTP_2} &= \text{gewichteter Wert HTTP Kommunikation: C++ Server} \leftrightarrow \text{Java Client} = 47 \\
GW_{Socket} &= \text{gewichteter Wert Socket Kommunikation} = 60
\end{aligned}
$$

$$
\frac{GW_{HTTP_1}+GW_{HTTP_2}}{2} < GW_{Socket} = \frac{67+47}{2} < 60 = \frac{114}{2} < 60 = 57 < 60
$$

Bei genauerer Überlegung lässt sich feststellen, dass gegebenenfalls eine Mischform der beiden am besten bewerteten Kommunikationsarten (*HTTP Kommunikation: C++ Client ↔ Java Server* & *Socket Kommunikation*) das bestmögliche Endprodukt gewährleisten könnte. Bei dieser könnte das Unreal Engine Projekt dann per HTTP Requests mit dem HTTP Server der Java Debugger Anwendung kommunizieren, welche wiederum Nachrichten per TCP Socket an das Unreal Engine Projekt übermitteln könnte (siehe „Abbildung 2: Kommunikationsmix Skizze“).

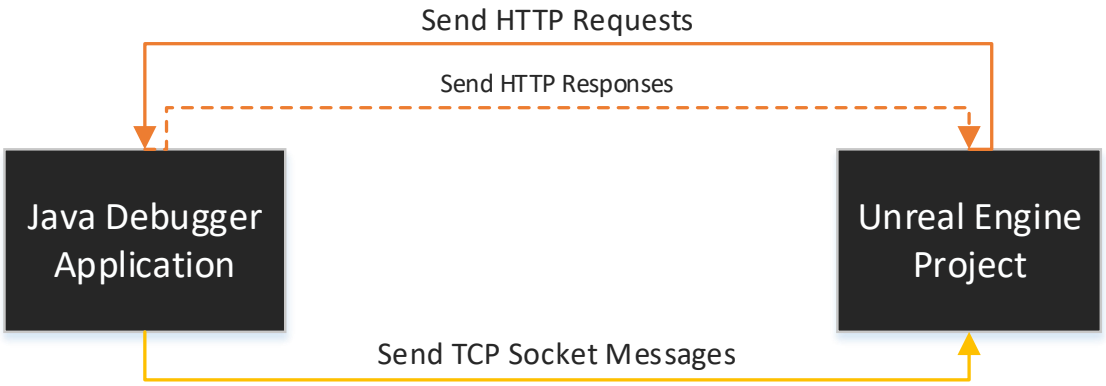

Abbildung 2: Kommunikationsmix Skizze

Wenn bei dieser Integrationsart jetzt auch noch dieselbe Serialisierung und Deserialisierung der Payload [F] für die HTTP Requests, HTTP Responses und Socket Messages genutzt wird, dann könnte später notfalls auf die in der Praxis besser funktionierende Kommunikationsart mit einem relativ geringen Entwicklungsaufwand umgeschwenkt werden.

Außerdem bietet diese Mischkommunikation Einblicke in gleich zwei, von der Unreal Engine bereitgestellten, Kommunikationsformen. Diese sind beide umfangreich dokumentiert und werden in Tutorials, welche hilfreiche Einführungen in die jeweilige Kommunikationstechnik bieten, ausführlich erklärt.

Aufgrund der aufgeführten Argumente wurde der beschriebene Kommunikationsmix als Kommunikationsverfahren der zu erstellenden Beispielimplementierung festgelegt. Die Integration des Debuggers erfolgt nun also, wie in der „Abbildung 2: Kommunikationsmix Skizze“ skizziert, per Java Debugger Microservice welcher einen HTTP Server bereitstellt und per TCP Socket Nachrichten an das Unreal Engine Projekt sendet.

Die genaue Festlegung der Schnittstellen und der gewählten Frameworks erfolgt im Kapitel 4 Implementierung.

[F] Übermittelte Nutzdaten

4 Implementierung

Das folgende Kapitel beinhaltet die Planung, sowie die Umsetzung der Implementierung. Dabei wird zunächst das Implementierungsvorgehen festgelegt, anschließend der Implementierungsverlauf kurz vorgestellt und näher auf die einzelnen Implementierungsabschnitte eingegangen.

4.1 Implementierungsvorgehen

Die Anforderungen an die Beispielimplementierung werden zunächst in fachliche User Stories überführt und in einer User Story Map geordnet, woraus sich wiederum die Implementierungs-Meilensteine ergeben. Anschließend wird eine abstrakte Entwurfsskizze visualisiert, die nach der Festlegung der zu nutzenden Technologien um selbige ergänzt und genauer erörtert wird. Abschließend werden die User Stories um technische Subtasks erweitert.

4.1.1 User Stories

Die verwendeten User Stories, welche über einen Namen und eine Kurzbeschreibung verfügen, basieren auf der „Abbildung 3: User Story Template“. Die User Stories orientieren sich an dem Prinzip:

Stories get their name from how they should be used, not what should be written - [Pat14a]]

As a [type of user] I want to [do something] So that I can [get some benefit]

Abbildung 3: User Story Template [Pat14b]

Die nachfolgenden User Stories, welche auch mit US abgekürzt werden, beschränken sich zunächst nur auf die Sicht eines Entwicklers, welcher mit dem zu erstellenden Produkt eine Java Anwendung debuggen möchte. Im folgenden Abschnitt ist deshalb mit *Entwickler* nicht der Entwickler der Beispielimplementierung, sondern der User-Typ der jeweiligen Stories gemeint. Die technischen Subtasks der User Stories werden erst nach der Erstellung der 4.1.5 Technologische Entwurfsskizze festgelegt.

US1 - Debugging Session: Verbindung herstellen

Als *Entwickler* möchte ich eine Debugging-Verbindung zu einer Java Anwendung, welche auf derselben Maschine wie der Debugger läuft, unter einem frei wählbaren Port aufbauen können, damit ich die Java Anwendung später debuggen kann.

US2 - Debugging Session: Verbindung trennen

Als *Entwickler* möchte ich eine bestehende Debugging-Verbindung trennen können, damit ich später eine neue Verbindung aufbauen kann.

US3 - Debugging Session: Starten

Als *Entwickler* möchte ich den Debugging-Prozess einer bestehenden Debugging-Verbindung starten können, damit mich der Debugger über zukünftig entstehende Debugging-Events des Debugging-Prozesses informiert.

US4 - Debugging Session: Pausierung

Als *Entwickler* möchte ich informiert werden, wenn der Debugging-Prozess vom Debugger pausiert wurde, damit ich diesen später fortsetzen kann.

US5 - Debugging Session: Fortsetzen

Als *Entwickler* möchte ich einen pausierten Debugging-Prozess fortsetzen können, damit ich vom Debugger über neue Debugging-Ereignisse informiert werden kann.

US6 - Watcher: Anzeige aller potenzieller Watcher

Als *Entwickler* möchte ich abfragen können, welche Watcher potenziell erstellt werden könnten, damit ich später einen davon auswählen kann.

US7 - Watcher: Auswahl eines potenziellen Watchers

Als *Entwickler* möchte ich einen potenziellen Watcher auswählen können, damit ich diesen später erstellen kann.

US8 - Watcher: Erstellung eines Watchers

Als *Entwickler* möchte ich einen ausgewählten Watcher erstellen können, damit ich später informiert werde, wenn sich das beobachtete Feld ändert.

US9 - Watcher: Anzeige aller aktiven Watcher

Als *Entwickler* möchte ich abfragen können, welche Watcher zur Zeit aktiv sind, damit ich eine Übersicht erhalte und aktive Watcher später löschen kann.

US10 - Watcher: Löschung eines aktiven Watchers

Als *Entwickler* möchte ich gezielt einen aktiven Watcher löschen können, damit dieser nicht länger aktiv ist.

US11 - Watcher: Mitteilung der Änderungsinformationen

Als *Entwickler* möchte ich über die Änderung eines von einem Watcher beobachteten Feldes informiert werden, dabei interessieren mich sowohl der alte, als auch der neue Wert des Feldes.

US12 - Breakpoint: Anzeige der vorhandenen Klassen

Als *Entwickler* möchte ich die vorhanden Klassen des Beispielprogrammes aufgelistet bekommen, damit ich später einen Breakpoint innerhalb einer dieser Klassen erstellen kann.

US13 - Breakpoint: Erstellung eines Breakpoints

Als *Entwickler* möchte ich einen Breakpoint erstellen können, indem ich eine der aufgelisteten Klassen des Beispielprogrammes auswähle und eine Zeilennummer angebe, damit ich informiert werde sobald der erstellte Breakpoint erreicht wird.

US14 - Breakpoint: Anzeige aller aktiven Breakpoints

Als *Entwickler* möchte ich abfragen können, welche Breakpoints zur Zeit aktiv sind, damit ich eine Übersicht erhalte und aktive Breakpoints später löschen kann.

US15 - Breakpoint: Löschung eines Breakpoints

Als *Entwickler* möchte ich gezielt einen aktiven Breakpoint löschen können, damit dieser nicht länger aktiv ist.

US16 - Breakpoint: Mitteilung Breakpoint erreicht

Als *Entwickler* möchte ich informiert werden sobald ein aktiver Breakpoint erreicht wird, damit ich den Programmablauf besser nachvollziehen kann.

Abbildung 4: User Stories: US1 bis US16

US17 - Live Stack Daten: Aktivierung
Als *Entwickler* möchte ich die Mitteilung von Live Stack Daten aktiveren können, sodass diese mir später mitgeteilt werden.

US18 - Live Stack Daten: Mitteilung
Als *Entwickler* möchte ich die Live Stack Daten mitgeteilt bekommen, wenn das Live Stack Daten Feature aktiviert ist, sodass ich den Callstack des Programmes nachvollziehen kann.

US19 - Live Stack Daten: Visualisierung
Als *Entwickler* möchte ich, dass die mitgeteilten Live Stack Daten simpel visualisiert werden, damit ich ein besseres Gefühl für die Veränderung des Stacks bekomme.

Abbildung 5: User Stories: US17 bis US19

Die User Story „US19 - Live Stack Daten: Visualisierung“ wurde am 13.12.2017 auf Wunsch des Erstprüfers nachträglich eingeplant.

4.1.2 User Story Map

Die Stories aus „Abbildung 4: User Stories: US1 bis US16“ und „Abbildung 5: User Stories: US17 bis US19“ werden nun in einer User Story Map (siehe „Abbildung 6: User Story Map“) geordnet. Dabei werden sie in verschiedene Lanes unterteilt, diese fassen die jeweils beinhaltenden User Stories unter einem Namen als Story-Verlauf zusammen. Sie basieren auf der jeweils vorherigen Lane und können damit als Meilensteine der Implementierung betrachtet werden. Aus den Lanes der User Story Map lassen sich also folgende sieben Implementierungs-Meilensteine ableiten:

- Meilenstein 1: *Initiale Verbindung*
- Meilenstein 2: *Watcher Erstellung*
- Meilenstein 3: *Watcher Mitteilung*
- Meilenstein 4: *Watcher Löschung*
- Meilenstein 5: *Breakpoint Erstellung*
- Meilenstein 6: *Breakpoint Löschung & Mitteilung*
- Meilenstein 7: *Live Stack Daten*

Innerhalb des Implementierungszeitraumes sollen die soeben aufgeführten Implementierungs-Meilensteine abgearbeitet werden. Dabei gilt es zu beachten, dass der Meilenstein 7 *Live Stack Daten* als experimentell angesehen wird, es muss also zunächst geprüft werden, ob sich eine rudimentäre Umsetzung überhaupt realisieren lässt.

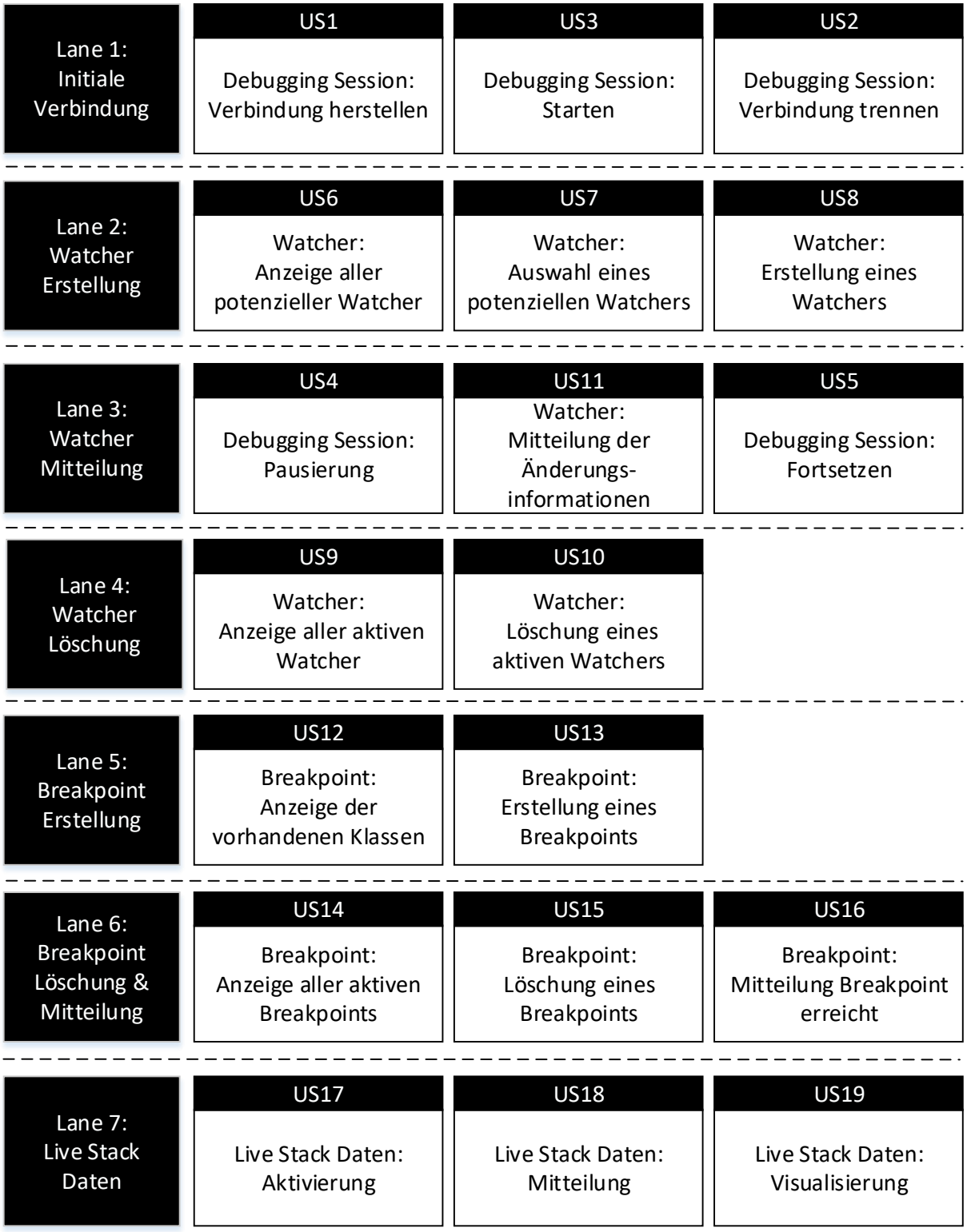

Abbildung 6: User Story Map

4.1.3 Entwurfsskizze

Die „Abbildung 7: Abstrakte Entwurfsskizze“ dient als abstrakte Visualisierung eines ersten Systementwurfes. Nach der 4.1.4 Festlegung der zu verwendenden Technologien wird diese Entwurfskizze um die verwendeten Technologien im Abschnitt 4.1.5 Technologische Entwurfsskizze erweitert und genauer erörtert.

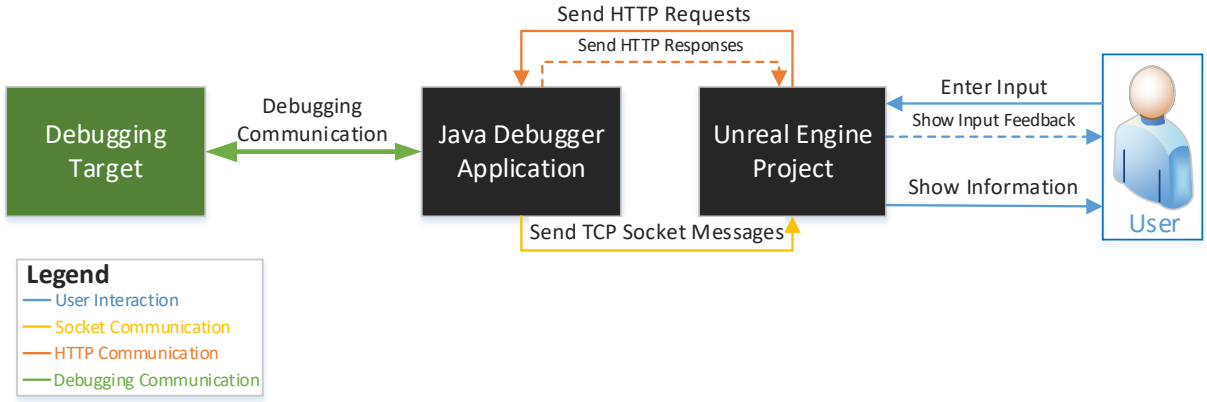

Abbildung 7: Abstrakte Entwurfsskizze

4.1.4 Festlegung der zu verwendenden Technologien

Im folgenden Abschnitt werden die ausgewählten Technologien, sowie dessen Einsatzgebiete, kurz aufgelistet. Dabei steht *JDA* für *Java Debugger Application*, sowie *UEP* für *Unreal Engine Project*, dies dient lediglich der Übersicht.

JDA - Grundgerüst: Spring Boot Die *Java Debugger Application* soll auf einer Spring Boot [15] Application basieren, da diese einen schnellen Entwicklungsstart ermöglicht, denn Spring Boot bringt unter anderem einen Webserver in Form eines eingebetteten Tomcat[16], Jetty[17] oder Undertow[18] Servers, welcher für die HTTP Kommunikation benötigt wird, mit sich.

JDA - Debugger Kommunikation: JDI Um eine Debugging Kommunikation zwischen der *Java Debugger Application* und dem *Debugging Target* zu ermöglichen wird auf das Java™ Debug Interface (JDI)[19] zurückgegriffen.

JDA - HTTP Kommunikation: Spring Da die *Java Debugger Application* auf Spring Boot basiert, wird für die HTTP Kommunikation das Spring Framework verwendet.

UEP - HTTP Kommunikation: Unreal Engine HTTP Modul Zum Senden von HTTP Requests wird das *Unreal Engine Project* auf die von der Unreal Engine in Form des HTTP[20] Moduls bereitgestellten Standardroutinen zurückgreifen.

JDA - Socket Kommunikation: *java.net* Die *Java Debugger Application* soll zur Socket Kommunikation die Java interne *java.net*[14] API verwenden.

UEP - Socket Kommunikation: Unreal Engine Sockets/Networking Modul Das *Unreal Engine Project* wiederum soll auf die von der Unreal Engine bereitgestellten Module, Sockets[21] und Networking[22] , zurückgreifen um Socket Nachrichten zu empfangen.

Payload Die Payloads der HTTP Requests, HTTP Responses, sowie der Socket Messages sollen dem JSON[G] Format entsprechen.

JDA - Payload De-/Serialisierung: GSON Zur Serialisierung beziehungsweise zur Deserialisierung von HTTP Bodies und Socket Messages soll die *Java Debugger Application* das GSON [23] Framework nutzen.

UEP - Payload De-/Serialisierung: Unreal Engine Json/JsonUtilities Modul Zur Serialisierung beziehungsweise zur Deserialisierung von HTTP Bodies und Socket Messages soll das *Unreal Engine Project* auf die von der Unreal Engine bereitgestellten Module, Json[24] und JsonUtilities[25] , zurückgreifen.

Der Einsatz anderer Technologien wäre natürlich auch vorstellbar, so könnte zum Beispiel Play[26] statt Spring Boot oder Jackson[27] statt GSON genutzt werden. Jedoch basiert die obige Auflistung auf den Ergebnissen der Recherche für das Kapitel 3 Integrationsmöglichkeiten, sowie auf den Erfahrungen des Entwicklers, dies dient zur Vermeidung von potenziellen Entwicklungsrisiken. Auf eine dedizierte Risikoanalyse wird explizit verzichtet, da es sich um eine Machbarkeitsprüfung handelt und auch ein technisches Scheitern als valides Endresultat angesehen wird.

4.1.5 Technologische Entwurfsskizze

Die im Abschnitt 4.1.3 erstellte abstrakte Entwurfsskizze wird nun um die gerade bestimmten Technologien ergänzt (siehe „Abbildung 8: Technologische Entwurfsskizze“). Zusätzlich werden den jeweiligen Komponenten ihre späteren Projektnamen zugewiesen. Somit wird aus der *Java Debugger Application* das *Debugger Backend*, sowie aus dem *Unreal Engine Project* der *Unreal VR Debugger (UVRD)*. Das *VR (Virtual Reality)* im Projektnamen resultiert lediglich aus der vorhandenen Problemstellung.

[G] JavaScript Object Notation

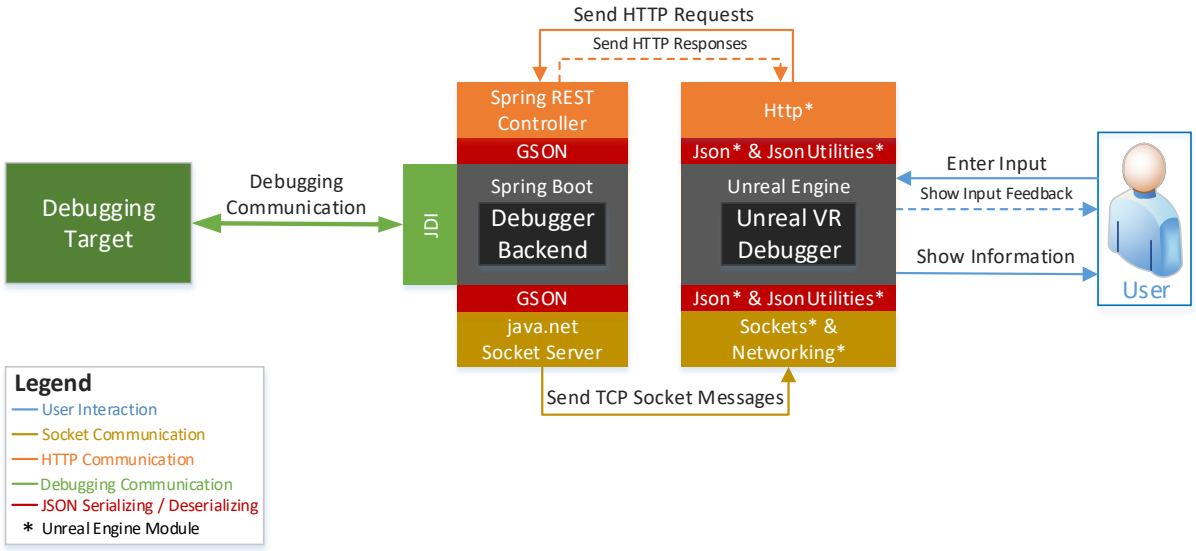

Abbildung 8: Technologische Entwurfsskizze

Der „Abbildung 8: Technologische Entwurfsskizze“ ist zu entnehmen, dass das *Debugger Backend* per *JDI* mit dem *Debugging Target* kommunizieren soll. Des Weiteren soll das *Debugger Backend* Daten per GSON serialisieren können, um diese anschließend per Socket Message an den *UVRD* zu schicken. Dieser wiederum soll zum Lesen der Socket Messages die Unreal Engine Module *Sockets* und *Networking* benutzen. Der Inhalt einer Socket Message soll daraufhin per *Json* und *JsonUtilities* Modul deserialisiert werden. Demgegenüber soll der *UVRD* mit dem *Debugger Backend* per HTTP kommunizieren. Die Payload der HTTP Requests und Responses soll ähnlich wie bei den Socket Messages mittels GSON, *Json* und *JsonUtilities* serialisiert beziehungsweise deserialisiert werden.

4.1.6 Ergänzung der User Stories um technische Subtasks

Die technischen Subtasks der User Stories sind im Anhang unter A.1 Technische Subtasks der User Stories zu finden, bei der Erstellung dieser galt es zu beachten, dass Grundmechanismen natürlich nur einmal Implementiert werden müssen und danach inkrementell erweitert werden können. So muss zum Beispiel das Versenden bzw. die Verarbeitung von Socket Messages nur einmal initial implementiert werden. Die Reihenfolge der User Stories ergibt sich aus der User Story Map (siehe „Abbildung 6: User Story Map“).

4.2 Implementierungsverlauf

Zur Visualisierung des Implementierungsverlaufes wurde ein Meilenstein Diagramm erstellt, welches im Anhang unter A.2 Meilenstein Diagramm zu finden ist. Während der Bearbeitungzeit konnten sämtliche User Stories abgeschlossen werden, jedoch verlängerte sich die Implementierungsphase, da eine Code-Dokumentation in Form von javadoc[28] und Doxygen[29] vom Erstprüfer als notwendig erachtet wurde. Diese war zum Zeitpunkt des Exposés allerdings nicht Teil der Implementierung. Glücklicherweise führte dies aber zu keiner Verzögerung im Zeitplan, da der Abschnitt Integrationsmöglichkeiten frühzeitig fertiggestellt werden konnte.

4.3 Architektur Überblick

Der folgende Abschnitt der Ausarbeitung befasst sich mit der resultierenden Systemarchitektur, dabei wird zunächst auf die Architektur des *Debugger Backend*s eingegangen. Anschließend wird der *Unreal Virtual Reality Debugger* genauer betrachtet. Die allgemeine Architektur lässt sich auf die in Abschnitt 4.1.5 Technologische Entwurfsskizze erwähnte Skizze herunterbrechen. Hierbei fungiert der *UVRD* als Front-End System, welches Nutzerinteraktionen verarbeitet und mittels HTTP Requests mit dem *Debugger Backend* kommuniziert. Dieses wiederum sendet Socket Messages an das Front-End um es über Debugging-Ereignisse zu informieren.

4.3.1 Debugger Backend Architektur

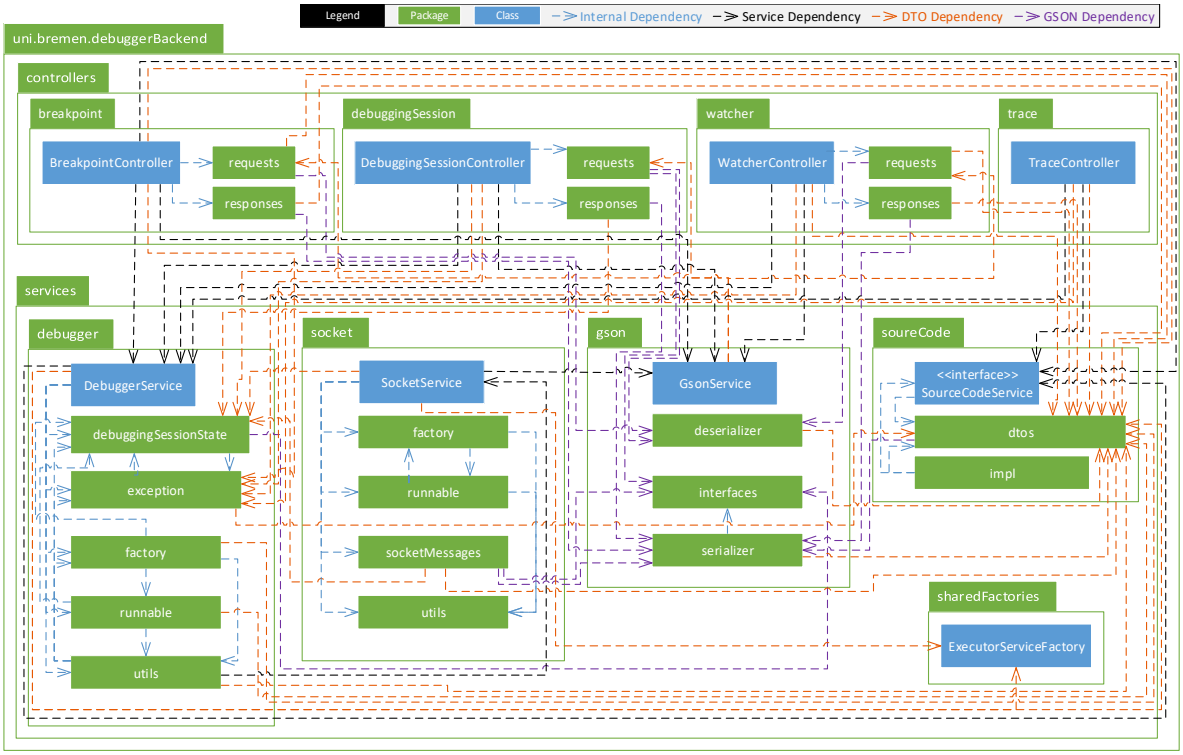

Abbildung 9: Abhängigkeitsskizze: Debugger Backend
Eine vergrößerte Version dieser Abbildung finden Sie im Anhang unter A.3 Abhängigkeitsskizze: Debugger Backend.

Für einen groben Systemüberblick wurde eine Abhängigkeitsskizze erstellt, siehe „Abbildung 9: Abhängigkeitsskizze: Debugger Backend“. Die verschiedenen Abhängigkeiten wurden zur besseren Übersicht in vier verschiedene Kategorien unterteilt: Interne Abhängigkeit, Service Abhängigkeit, DTO Abhängigkeit, GSON Abhängigkeit. Auf den ersten Blick wirkt die Skizze dennoch überladen, deshalb wird nun genauer auf die Architektur eingegangen. Im wesentlichen wurde die auf Spring Boot basierende Anwendung in zwei grundlegende Packages aufgeteilt: **controllers** und **services**.

Das **controllers**-Package beinhaltet sämtliche Logik, welche für die HTTP Kommunikation benötigt wird. Die einzelnen Schnittstellen wiederum wurden fachlich voneinander getrennt. Eine genauere Erörterung der HTTP Kommunikation finden Sie im Abschnitt 4.5.2 Java HTTP Server.

Das **services**-Package hingegen beinhaltet die Businesslogik der Anwendung in Form des `DebuggerServices`, sowie den für die Socket Kommunikation benötigten `SocketService` und den `GsonService`, welcher zur Serialisierung bzw. Deserialisierung von Datensätzen benötigt wird. Außerdem enthält es den `SourceCodeService`. Hierbei handelt es sich um ein Interface, wessen eigentliche Implementierung zur Zeit lediglich statische Daten zurückliefert.

Auf den `DebuggerServices` wird im Abschnitt 4.4 Debugger genauer eingegangen, selbiges gilt für den *SocketService* und Abschnitt 4.6.2 Socket Kommunikation: Java.

Allgemein lässt sich noch ergänzen, dass jeder Service über dazugehörige fachlich Komponenten verfügt, welche sich innerhalb seines Packages befinden, zu diesen zählen unter anderem *Data Transfer Object*s (DTOs), sowie aus dem jeweiligen Service ausgelagerte Logik. Die Services entsprechen aufgrund ihres *default Scopes*[30] dem *Singleton Pattern*[31].

Die internen Softwareabhängigkeiten der jeweiligen Spring Komponenten werden per *Spring dependency injection*[32] aufgelöst, hierfür wird die `@Autowired`[33] Annotation benutzt. Der dafür benötigte *Component Scan* wird ebenfalls per Annotation in der Main-Klasse der Softwarekomponente (`DebuggerBackendApplication.java`) konfiguriert. Die erwähnte Main-Klasse, welche sich im Root-Package (`uni.bremen.debuggerBackend`) befindet, ist nicht Bestandteil der Abhängigkeitsskizze, da sie über keine expliziten Abhängigkeiten verfügt und lediglich zur Konfiguration und zum Starten der Spring Boot Anwendung dient. Die Konfigurationsdatei des *Debugger Backend*s, die den HTTP-Serverport, den Serversocketport, sowie die maximale Anzahl der gleichzeitig verbundenen Client-Sockets beinhaltet, finden Sie in dessen Verzeichnis unter `src/main/resources/application.properties`.

Es gilt darüber hinaus anzumerken, dass es sich bei dem *Debugger Backend* um ein *Apache Maven*[34] Projekt handelt. Die Abhängigkeiten zu Software Dritter wurden in der `pom.xml` des Projektes vermerkt (siehe A.4 Debugger Backend: pom.xml), diese sind ebenfalls nicht Bestandteil der Abhängigkeitsskizze. Als Entwicklungsumgebung für diese Softwarekomponente diente die *IntelliJ IDEA*[35] in der Community Edition.

Bei der Entwicklung des *Debugger Backend*s wurde versucht nach dem Software-Mantra *Test Driven Development* (TDD) vorzugehen, dabei wurden für die vorhandenen Java Klassen insgesamt 283 Unit Tests geschrieben, welche das gewünschte Verhalten der implementierten Logik definieren. Diese Vorgehensweise führt unweigerlich zu einer relativ hohen Testabdeckung (siehe „Abbildung 10: Testabdeckung: Debugger Backend“) obwohl DTOs und Factory-Klassen explizit nicht unter Test gestellt wurden. Als Testframeworks wurden *Mockito 2*[36] und *JUnit 4*[37] eingesetzt.

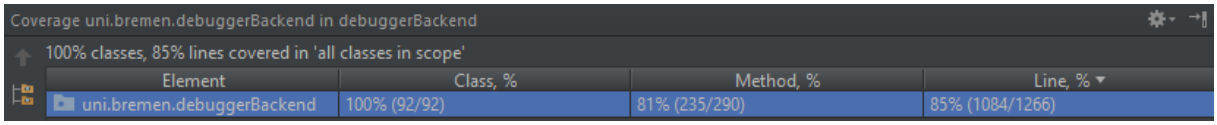

Abbildung 10: Testabdeckung: Debugger Backend
Die Testabdeckung wurde vom IntelliJ IDEA interne Testabdeckungstool ermittelt.

Eine ausführliche Dokumentation des Codes in Form von JavaDoc finden Sie übrigens auf der beiliegenden Disk im Order `DebuggerBackend_JavaDoc`. Ebenfalls auf der Disk enthalten ist der komplette Sourcecode des Anwendung. Diesen finden Sie im Ordner `DebuggerBackend`.

4.3.2 Unreal Virtual Reality Debugger Architektur

Der UVRD basiert auf der Unreal Engine 4.16.3. Sämtliche Kommunikationsschnittstellen wurden in C++ implementiert. Die Blueprints des UVRD enthalten zum Großteil nur Front-End Funktionalitäten, deshalb bezieht sich die folgende Abhängigkeitsskizze (siehe „Abbildung 11: Abhängigkeitsskizze: UVRD“) ausschließlich auf den C++ Code des UVRD. Die Blueprints werden anschließend im Kapitel 4.3.2.1 Blueprints betrachtet.

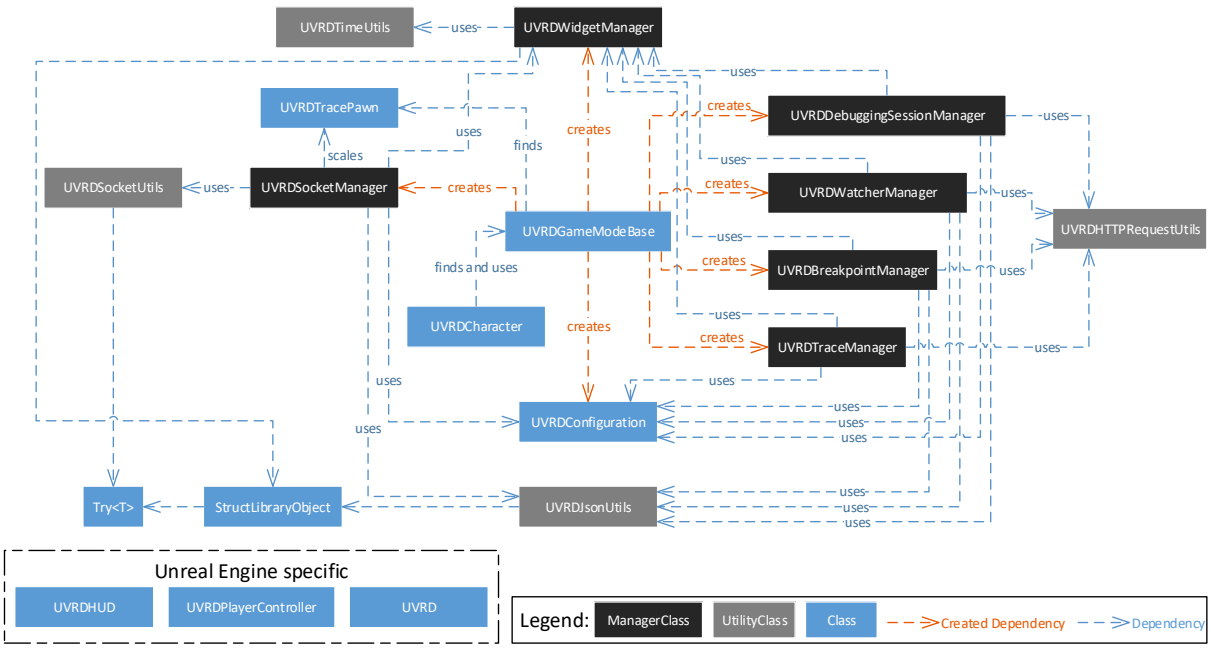

Abbildung 11: Abhängigkeitsskizze: UVRD
Eine vergrößerte Version dieser Abbildung finden Sie im Anhang unter A.5 Abhängigkeitsskizze: UVRD.

Es gilt anzumerken, dass die Abhängigkeitsskizze nur die direkte Abhängigkeiten beinhaltet. Die Komponenten des UVRD werden jedoch per `#include` importiert, dies hat zur Folge, dass zum Beispiel die `UVRDGameModeBase` Zugriff auf das `StructLibraryObject` hat, da sie unter anderem den `UVRDWatcherManager` inkludiert, welcher wiederum die `UVRDJsonUtils` und damit auch das `StructLibraryObject` inkludiert.

Den eigentlichen Kern der vorliegenden Softwarekomponente bildet die `UVRDGameModeBase`. Sie kristallisierte sich während des Entwicklungsstarts als initiale Klasse heraus. In ihr wurden zunächst neue Logiken testweise implementieren, diese wurden dann jeweils im Nachhinein in dazu passende Manager ausgelagert. Die Manager werden allesamt von der `UVRDGameModeBase` erstellt und greifen selber auf Utility-Klassen zurück, welche statische Hilfsmethoden beinhalten. Bei der Erstellung eines Managers übergibt die `UVRDGameModeBase` alle vom Manager benötigten Komponenten an dessen statische `CreateNew`-Methode. Durch dieses Entwurfs-Pattern werden alle internen Abhängigkeiten aufgelöst. Dabei kann es auch vorkommen, dass ein Manager einen anderen Manager benötigt: Z.B. den `UVRDWidgetManager`. Um dieses Manager-Pattern hervorzuheben, wurden die *Manager-Klassen*, die *Utility-Klassen*, sowie die *„erstellenden Abhängigkeiten“* in der Abhängigkeitsskizze farblich markiert. Anzumerken ist, dass sämtliche Manager vom `UObject`[38] abgeleitet werden.

Eine Ausnahme bei der Erstellung der Manager bildet der `UVRDWidgetManager`, dieser wird nämlich von der `BP_UVRDGameModeBase` erstellt. Dies ist notwendig, da der `UVRDWidgetManager` auf Werten basiert, welche in der auf ihm basierenden Blueprint konfiguriert sind.

Der `UVRDWidgetManager` stellt die Schnittstelle zwischen dem C++ Code und den UMG[39] Widgets dar. Sämtliche vom C++ Code hervorgerufenen Änderungen an den Widgets werden über den `UVRDWidgetManager` getätigt. Auf die UMG Widgets wird genauer im Kapitel 4.7 UVRD: Front-End eingegangen.

Neben den Managern erstellt die `UVRDGameModeBase` auch noch eine Instanz der `UVRDConfiguration`, dieser wiederum beinhaltet unter anderem die Schnittstellenkonfiguration der Socket und HTTP Verbindungen. Sie basiert auf der `DefaultDebuggerConfig.ini`, welche sich im Unterverzeichnis `Config` des *UVRD*-Projektes befindet.

Die `UVRDGameModeBase` stellt außerdem sämtliche von den Blueprints benötigte Manager, sowie die `UVRDConfiguration` in Form von *blueprintcallable UFunction-Getter* bereit.

Auf den `UVRDDebuggingSessionManager`, den `UVRDWatcherManager`, den `UVRDBreakpointManager`, sowie auf den `UVRDTraceManager` wird genauer im Abschnitt 4.5.3 C++ HTTP Client eingegangen, da diese Bestandteil der HTTP Kommunikation sind.

Ähnliches gilt für den `UVRDSocketManager`, da dieser für die Socket Kommunikation zuständig ist, wird er erst im Kapitel 4.6.3 Socket Kommunikation: C++ genauer betrachtet.

Eine weitere Besonderheit des UVRDs ist, dass das `StructLibraryObject` sämtliche benötigten DTOs als C++ Structs beinhaltet. Eine Ausnahme hiervon bildet jedoch die getypte `Try` Klasse. Im Wesentlich handelt es sich bei ihr auch um ein DTO, jedoch ist es stark an das *Scala Try*[40] angelehnt. Es enthält im Erfolgsfall einen Content vom Typ des Trys und im Fehlerfall eine Fehlermeldung. Bei dieser handelt es sich entweder um die Standartfehlermeldung oder um eine dediziert bei der Erstellung des Trys übergebenen Fehlermeldung. Durch dieses

Try-Pattern soll die Nutzung von Exceptions vermieden werden, da diese in C++ Kreisen häufig als *bad Practice* angesehen wird.

Bei dem vorliegenden `UVRDCharacter` handelt es sich um einen sehr rudimentären Spielercharakter, welcher lediglich die Kamera mittels der Maus bewegen kann und per Tastenbefehl (F12) das Log-Widget ein- bzw. ausblenden kann. Auch das `UVRDHUD` entspricht einer sehr einfachen Version eines Head-up-Display. Es basiert auf dem klassischen *Unreal Engine First Person Shooter Tutorial*[41] und dient lediglich dazu ein rotes Fadenkreuz im Mittelpunkt des Bildschirmes darzustellen. Der `UVRDPlayerController` dient nur als C++ Basis für die auf ihm basierende Blueprint, da vom Entwickler festgelegt wurde, dass sämtliche verwendeten Blueprints, bis auf die Widgets, auf einer selbst erstellten C++ Klasse basieren müssen, damit diese bei Bedarf um entsprechende Logik erweitert werden kann. Das `UVRDTracePawn` dient übrigens zur Visualisierung der Live-Call-Stacks-Informationen.

Neben den Standard Unreal Engine Modulen wurde für die Nutzeroberfläche, sowie für die Socket und HTTP Kommunikation noch weitere Module bemüht. Eine vollständige Modulauflistung befindet sich im Anhang unter A.6 UVRD: UVRD.Build.cs.

Ähnlich wie für das *Debugger Backend* liegt auch für den C++ Code des UVRD eine Codedokumentation in Form von Doxygen vor, diese befindet sich auf der beiliegenden Disk im Ordner `UVRD_Doxygen`.

4.3.2.1 Blueprints

Im folgenden Kapitel werden die einzelnen Blueprints (siehe „Abbildung 12: Blueprints“) kurz betrachtet. Sämtliche UMG Widgets fallen aus dieser Betrachtung heraus, da sie explizit im Abschnitt 4.7 UVRD: Front-End behandelt werden. Wie bereits erwähnt wurde basieren sämtliche Blueprints, bis auf die UMG Widgets, auf einer C++ Klasse. Der Name einer Blueprint resultiert aus dem Präfix `BP_` in Kombination mit dem Namen der jeweiligen Basis Klasse.

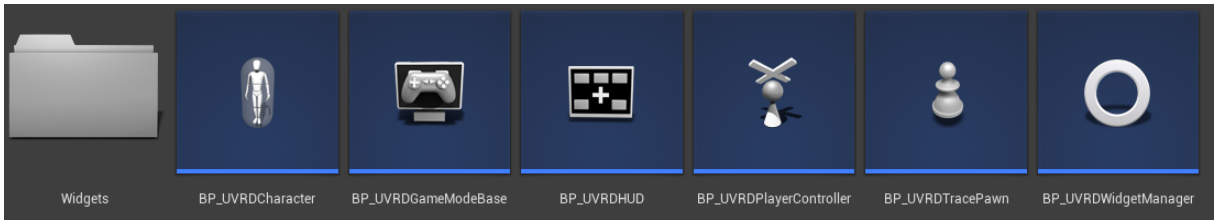

Abbildung 12: Blueprints

Bei dem `BP_UVRDCharacter` handelt es sich lediglich um eine Default-Blueprint, welche aus Gründen der Kontinuität erstellt wurde, da für das vorliegende Projekte festgelegt wurde, dass unter *Maps & Modes* möglichst nur Blueprints konfiguriert werden sollten.

Die Funktionalitäten der `BP_UVRDGameModeBase` wurden bereits im vorherigen Abschnitt beschrieben.

In der `BP_UVRDHUD` wird die Textur des Fadenkreuzes, welches Bestandteil des Head-up-Display

ist, konfiguriert (siehe Anhang A.7.1 Blueprint Auszug: BP_UVRDHUD).

Die Anzeige des Mauszeiger wird in der `BP_UVRDPlayerController` (siehe Anhang A.7.2 Blueprint Auszug: BP_UVRDPlayerController) so konfiguriert, dass der Mauszeiger permanent sichtbar ist. Dies führt dazu, dass der Nutzer einfacher mit der Benutzeroberfläche interagieren kann. Jedoch muss der Nutzer nun die rechte oder linke Maustaste gedrückt halten, wenn er sich umsehen möchte.

Die `BP_UVRDTracePawn` setzt per *Construction Script* das `UStaticMesh` ihrer `VisibleStackComponent`, welche wiederum über die vorhandene Scale-Methode skaliert werden kann. Das zusetzende *Mesh* wird in der Blueprint selbst konfiguriert (siehe Anhang A.7.2 Blueprint Auszug: BP_UVRDPlayerController).

Sämtliche vom `UVRDWidgetManager` benötigten *UMG Widget Classes*, werden in der `BP_UVRDWidgetManager` gesetzt (siehe A.7.4 Blueprint Auszug: BP_UVRDWidgetManager). Zusätzlich zu dieser Konfiguration wurden außerdem alle *BlueprintImplementableEvent UFUNCTION*s des `UVRDWidgetManager` in der auf ihm basierenden Blueprint implementiert, sodass der C++ Code Zugriff auf die Funktionalitäten der vom Manager verwalteten UMG Widgets erhält.

Um einen genauen Einblick in die Implementierung der Blueprints zu erhalten, empfiehlt es sich die Blueprints direkt im Unreal Editor zu betrachten. Das Unreal Engine Projekt des UVRD befindet sich auf der beiliegenden Disk im Ordner `UVRD`.

4.4 Debugger

Das nun folgende Kapitel behandelt die Implementierung des Java Debuggers. Zunächst wird auf die Realisierung der Ermittlung von Live-Call-Stack-Daten eingegangen, anschließend auf die Funktionsweise der Debugger Logik und abschließend auf den Lebenszyklus der Debugging Session.

4.4.1 Ermittlung von Live Call Stack

Während der Bearbeitung der User Story Map *Lane 7: Live Stack Daten* wurde sich genauer mit den vom JDI bereitgestellten Events auseinandergesetzt. Dabei wurde der Entschluss gefasst, dass die Events `MethodEntryEvent`[42] und `MethodExitEvent`[43] als sehr viel versprechend einzustufen sind. Diese werden nämlich unter anderem von der JVM[H] erstellt, wenn eine Methode betreten bzw. verlassen wird, welche sich innerhalb des Beobachtungsspekturms eines aktiven `MethodEntryRequest`[44] bzw. `MethodExitRequest`[45] befindet. Die beiden soeben erwähnten Requests bieten hierbei mehrere Arten der Einschränkung des jeweiligen Beobachtungsspekturms in Form von verschiedenen Filtermethoden an. Es wurde sich schlussendlich für die `addClassFilter`[46] Methode entschieden, welche als Input ein sogenanntes *Class Pattern* erwartet. Hierbei handelt es sich um einen regulärer Ausdruck, der entweder genau auf eine Klasse zutrifft oder welcher mit einem Wildcard-Operator anfängt bzw. endet.

Die Aktualität des Live Call Stack hängt bei dieser Art der Ermittlung somit also von der Wahl der *Class Pattern* ab. Dabei müssen sich selbige nicht auf den Programmcode des verbundenen *Debugging Target*s an sich beschränken, denn sie können theoretisch auch auf Fremdbibliotheken ausgedehnt werden.

Die aktuelle Implementierung des `DebuggerServices` ist zur Zeit sehr flexibel, sie erlaubt es mehrere *Class Pattern* in Form von *Package Names* zu definieren. Diese werden dann bei der Erstellung der entsprechenden Requests um den Wildcard-Operator ergänzt, somit wird zum Beispiel aus dem *Package Name* `"uni.bremen.debuggerBackend"` das *Class Pattern* `"uni.bremen.debuggerBackend.*"`.

Es gilt jedoch anzumerken, dass sich der `TraceController` aktuell noch auf das Root-Package des Beispielprojektes beschränkt. Genauere Informationen über das erwähnte Beispielprojekt erhalten Sie im Abschnitt 5.2.1 Beispiel Projekt.

4.4.2 Funktionsweise

Die erstellte Abhängigkeitsskizze, „Abbildung 13: Abhängigkeitsskizze: Debuger Backend - debugger“, dient als grobe Architektur-Visualisierung, dabei beinhaltet sie ausschließlich die Abhängigkeiten des `DebuggerServices` und dessen Hilfsklassen. Im wesentlich lässt dich die gesamte Debugger Logik auf den `DebuggerServices` zurückführen. Er verwaltet die aktiven Watcher und Breakpoints, sowie die Methodenablaufverfolgung. Zusätzlich beinhaltet er den

[H] Java Virtual Machine

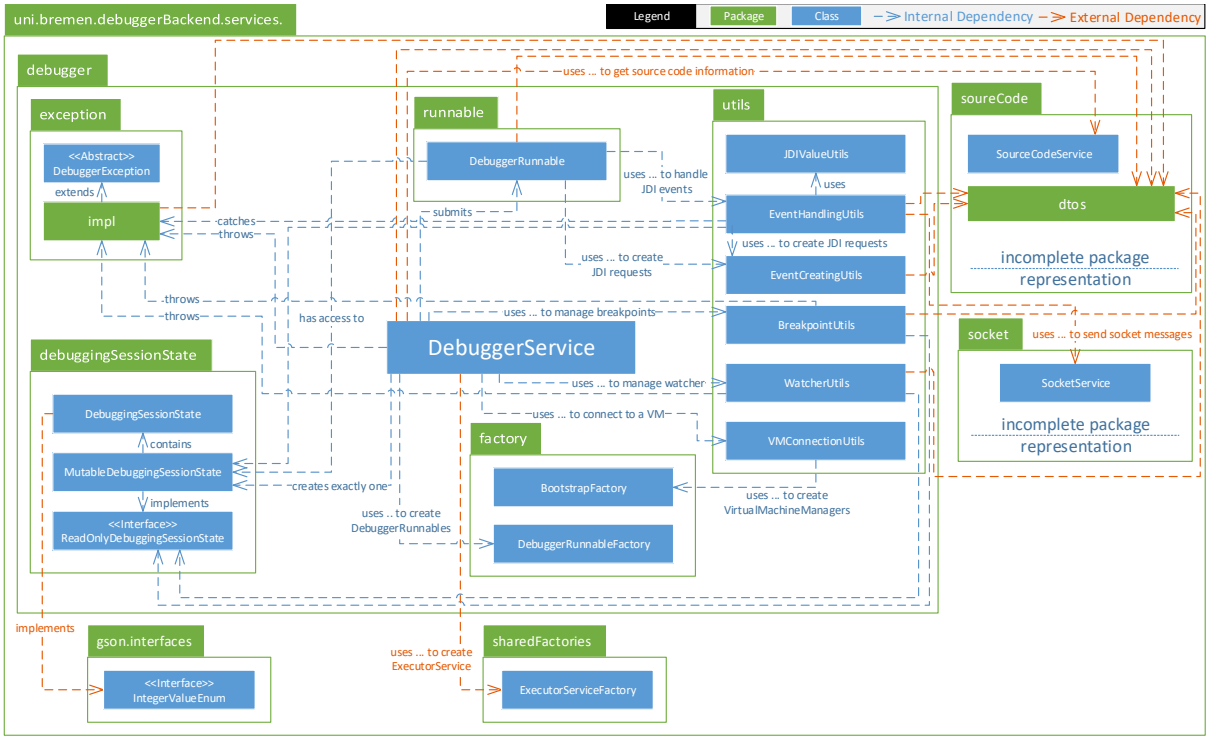

Abbildung 13: Abhängigkeitsskizze: Debuger Backend - debugger
Eine vergrößerte Version dieser Abbildung finden Sie im Anhang unter A.10 Abhängigkeitsskizze: Debuger Backend - debugger.

Zustand des aktuellen Debugging-Prozesses, in Form des `DebuggingSessionState`s. Auf diesen wird im nächsten Abschnitt genauer eingegangen.

Wie in Abschnitt 4.3.1 Debugger Backend Architektur beschrieben kommuniziert das *Debugger Backend* per JDI mit dem *Debugging Target*. Die eigentliche JDI Logik wurde dabei aus dem `DebuggerServices` in die Hilfsklassen `JDIValueUtils`, `EventHandlingUtils`, `EventCreatingUtils` und `VMConnectionUtils`, sowie in den `DebuggerRunnable` ausgelagert.

Nachdem eine Debugging-Verbindung zu einer JVM mittels der `VMConnectionUtils` hergestellt wurde, kann der Debugging-Prozess gestartet werden, hierbei erzeugt der `DebuggerServices` einen `DebuggerRunnable` mithilfe der `DebuggerRunnableFactory`. Bei der Erzeugung an sich werden dem `DebuggerRunnable` per Konstruktor sämtliche vom `DebuggerServices` verwaltete Informationen übergeben und zusätzlich werden die `EventHandlingUtils`, sowie die `EventCreatingUtils` injiziert.

Sobald der `DebuggerRunnable` gestartet wird erzeugt er zunächst sämtliche für die Ermittlung des Live Call Stacks benötigten `MethodEntryRequest`s und `MethodExitRequest`s. Anschließend vereint er die Klassennamen der Klassen, für die er im späteren Verlauf des Debugging Prozesses Breakpoints oder Watcher erstellen soll, in eine Liste und erstellt für die in dieser Liste enthaltenden Elemente `ClassPrepareEvent`s[47]. Danach wird permanent der `DebuggingSessionState` des `MutableDebuggingSessionState`s überprüft. Wenn der Zustand

nicht CONNECTED_AND_RUNNING oder CONNECTED_AND_WAIT_FOR_RESUME entspricht, dann wird der DebuggerRunnable beendet. Solange der Zustand CONNECTED_AND_RUNNIN entspricht, verarbeitet die implementierte Logik die Debugging-Events der EventQueue[48] der verbundenen VirtualMachine[49] wie folgt:

- VMDeathEvent[50]: Informiere den Client per Socket Nachricht über die Beendigung der JVM.
- VMDisconnectEvent[51]: Informiere den Client per Socket Nachricht darüber, dass die Verbindung zur JVM getrennt wurde.
- MethodEntryEvent: Informiere den Client per Socket Nachricht über das Betreten der Methode.
- MethodExitEvent: Informiere den Client per Socket Nachricht über das Verlassen der Methode.
- ClassPrepareEvent: Prüfe ob für die vorliegende Klasse BreakpointRequests[52] oder ModificationWatchpointRequests[53] zu erstellen sind.
- ModificationWatchpointEvent[54]: Pausiere den Debugging-Prozess, sodass der Client die Information zunächst in Ruhe betrachten kann und informiere den Client per Socket Nachricht über das Erreichen des Watchers, sowie über die vorliegende Pausierung.
- BreakpointEvent[55]: Informiere den Client per Socket Nachricht über das Erreichen des Breakpoints, sowie über die Pausierung des Debugging-Prozesses.

Sollte sich der DebuggingSessionState in CONNECTED_AND_WAIT_FOR_RESUME ändern, wartet der DebuggerRunnable $500ms$ bis er den Zustand des MutableDebuggingSessionState erneut überprüft.

Eine explizite Dokumentation der einzelnen Klassen liegt als JavaDoc vor. Diese finden Sie wie bereits erwähnt auf der beiliegenden Disk oder unter der URL des entsprechenden online Verweises.

4.4.3 Lebenszyklus: Debugger Session

Der Lebenszyklus der Debugger Session kann Anhand des DebuggingSessionState des MutableDebuggingSessionStates nachvollzogen werden. Zu erwähnen ist, dass dieser während der gesamten Laufzeit der Anwendung nur genau einmal vom *DebuggerServices* initial erstellt wird. Der angesprochene Lebenszyklus wurde in „Abbildung 14: Zustandsdiagramm: Debugger Session“ visualisiert.

Des Weiteren muss betont werden, dass die Methoden des MutableDebuggingSessionState auf illegale Modifikationen mit einer IllegalDebuggingSessionStateModificationException reagierten. So ist es zum Beispiel nicht möglich den Zustand von NOT_CONNECTED in CONNECTED_AND_RUNNING zu ändern.

Der DebuggingSessionState wirkt sich außerdem auf die Verwaltung der Methodenab-

laufverfolgung, sowie auf die Verwaltung der Watcher und Breakpoints aus. Änderungen an diesen können nur durchgeführt werden, wenn der Zustand `NOT_CONNECTED` oder `CONNECTED_AND_WAIT_FOR_START` beträgt. Diese Beschränkung erfolgt aus der aktuellen Implementierung und könnte perspektivisch entfernt werden (siehe 5.1 Implementierte Funktionalität).

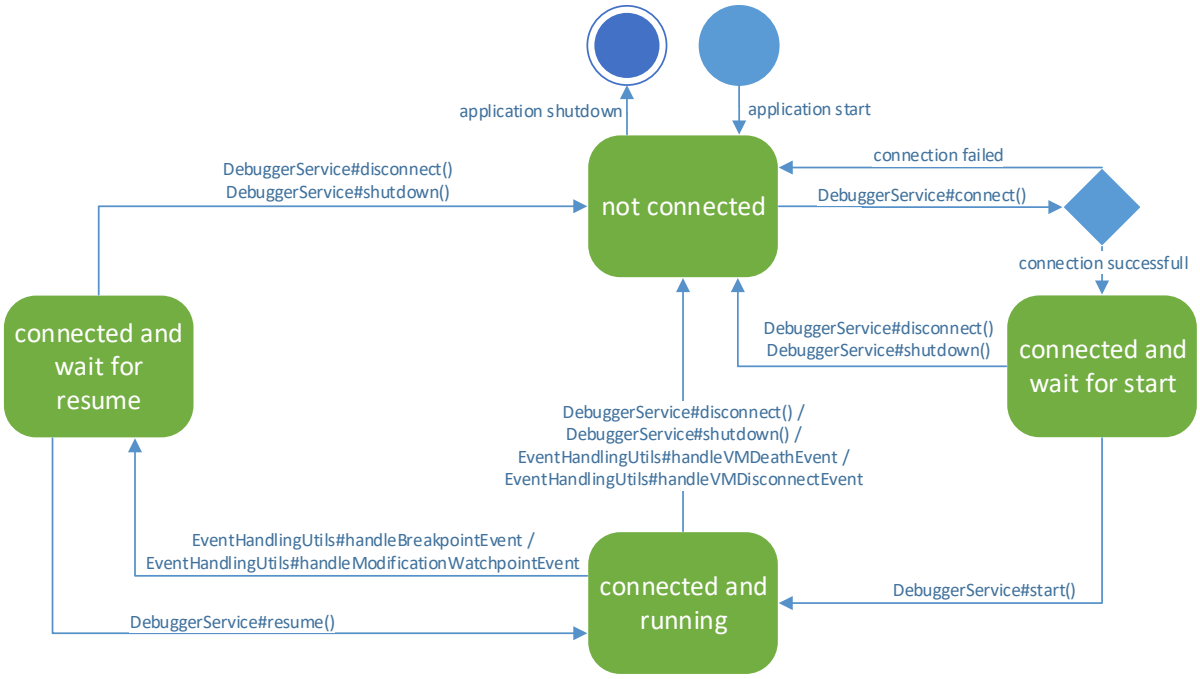

Abbildung 14: Zustandsdiagramm: Debugger Session

Eine vergrößerte Version dieser Abbildung finden Sie im Anhang unter A.11 Zustandsdiagramm: Debugger Session.

4.5 HTTP Kommunikation

Dieses Teilkapitel der Ausarbeitung wird zunächst das Schnittstellendesign der HTTP Kommunikation betrachten. Anschließend wird genauer auf die technische Realisierung eingegangen.

4.5.1 Schnittstellendesign

Das Schnittstellendesign der HTTP Kommunikation ist stark an das Programmierparadigma *REST*[I] angelehnt. Das Konzept der Ressourcen, welches unter anderem von [Gri14a] genauer erörtert wird, wurde jedoch nicht vollständig eingehalten. Die Wahl der HTTP Methoden basiert auf der Beschreibung von [Gri14b]. Die resultierenden Ressourcen wurden in der „Abbildung 8: Ressourcen“ zusammengetragen. Eine ausführliche Schnittstellendokumentation finden Sie im Anhang unter A.8 HTTP Kommunikation: Schnittstellenbeschreibung. Bei den in der Schnittstellendokumentation erwähnten Payloads handelt es sich um Payloads im JSON Format.

Ressource	HTTP Methoden	Kurzbeschreibung
`/debuggingSession`	`GET, POST, PUT, DELETE`	Verwaltung der Debugging Session
`/breakpoints/classNames`	`GET`	Bereitstellung der Klassennamen des Beispielprojektes
`/breakpoints/active`	`GET, POST, DELETE`	Verwaltung der aktiven Breakpoints
`/watchers/available`	`GET`	Bereitstellung der potentiellen Watcher des Beispielprojektes
`/watchers/active`	`GET, POST, DELETE`	Verwaltung der aktiven Watcher
`/trace/methodEntry`	`POST`	Aktivierung der Verfolgung von Methoden-Betretungen
`/trace/methodExit`	`POST`	Aktivierung der Verfolgung von Methoden-Verlassungen

Tabelle 8: Ressourcen

4.5.2 Java HTTP Server

Die Java seitige HTTP Kommunikation des *Debugger Backend*s basiert auf dem *Spring Web MVC*[56]. Das im vorherigen Abschnitt beschriebene Schnittstellendesign wurde mittels

[I] Representational State Transfer

`RestController`[57] abgebildet, wobei die Ressourcen ihrem Pfad entsprechend in die folgenden vier Controller aufgeteilt wurden:

- `DebuggingSessionController`: Endpunkt zur Verwaltung der Debugging Session
- `BreakpointController`: Endpunkt zur Verwaltung der Breakpoints
- `WatcherController`: Endpunkt zur Verwaltung der Watcher
- `TraceController`: Endpunkt zur Verwaltung der Ablaufverfolgung

Für die in der Schnittstellenbeschreibung enthaltenden Requests und Responses wurden DTO Klassen erzeugt. Diese können von den jeweiligen Controllern mittels des `GsonService` serialisiert bzw. deserialisiert werden. Die Controller an sich wurden in einzelne Packages verteilt, die neben den Controller auch noch die dazugehörigen DTOs beinhalten, siehe „Abbildung 9: Abhängigkeitsskizze: Debugger Backend“.

Außerdem ist anzumerken, dass der HTTP Serverport[J] in der Konfigurationsdatei des *Debugger Backend*s angepasst werden kann.

4.5.3 C++ HTTP Client

Die HTTP Kommunikation des *UVRD* beschränkt sich als Client auf das Erstellen von HTTP Requests, sowie auf die Verarbeitung der resultierenden HTTP Responses.

Die verschiedenen Ressourcen des Schnittstellendesigns, welche in der `DefaultDebuggerConfig.ini` konfiguriert werden können, wurden entsprechend ihres Pfades auf die folgenden vier Manager aufgeteilt:

- `UVRDDebuggingSessionManager`: Verwaltung der Debugging Session
- `UVRDBreakpointManager`: Verwaltung der Breakpoints
- `UVRDWatcherManager`: Verwaltung der Watcher
- `UVRDTraceManager`: Verwaltung der Ablaufverfolgung

Zur Erstellung der HTTP Requests nutzen die soeben aufgeführten Manager die `UVRDHTTPRequestUtils`. Diese basieren auf der in Form des HTTP Moduls bereitgestellten HTTP Standartroutinen der Unreal Engine.

Die Payloads der Requests und Responses werden von den Managern unter Nutzung der `UVRDJsonUtils`, welche wiederum ebenfalls auf den entsprechenden Standartroutinen der Unreal Engine basieren, serialisiert bzw. deserialisiert. Die erwähnten Payloads wurden allesamt als C++ Structs innerhalb des `StructLibraryObject`s abgebildet.

Abschließend sei erwähnt, dass die IP und der Port des Zielserver der HTTP-Schnittstelle in der `DefaultDebuggerConfig.ini` konfiguriert werden können.

[J] application.properties - Konfigurationsparameter: `server.port` (Default = 8080)

4.6 Socket Kommunikation

Bei der Erörterung der Socket Kommunikation wird ähnlich wie im vorherigen Kapitel zunächst auf das Schnittstellendesign und anschließend auf die technische Realisierung eingegangen. Es gilt zu betonen, dass für Socket Kommunikation TCP Sockets benutzt werden.

4.6.1 Schnittstellendesign

Für die Socket Kommunikation wurde ein relativ simples Schnittstellendesign gewählt, da der Entwickler über so gut wie keinerlei Vorwissen verfügte. Damit die Schnittstelle perspektivisch leicht auf eine andere Kommunikationsart umgestellt werden kann, werden die Nutzdaten der Nachrichten ebenfalls wie bei der HTTP Kommunikation im JSON Format übermittelt.

Das Schnittstellendesign selbst sieht vor, dass genau eine Nachrichtenart existiert. Diese entspricht der im Anhang unter A.9.15 SocketMessage aufgeführten JSON Definition. Sie beinhaltet einen `messageType`, sowie ein `messageJson`. Ersteres dient zur Identifizierung des `messageJson`, welches wiederum ein JSON Objekt ist, das den eigentlichen Inhalt der Socket Nachricht enthält. Dieser Aufbau ermöglicht es verschiedene Arten von Inhalten mittels einer Art von Socket Nachricht zu transportieren. Im Folgenden werden die verschiedenen `messageJson`s kurz aufgelistet[K]:

- `PAUSE_DEBUGGING_SESSION[0]` / A.9.16 PauseDebuggingSessionSocketMessageJson / Dient zur Übermittlung einer Pausierung des Debugging Prozesses.
- `END_DEBUGGING_SESSION[1]` / A.9.17 EndDebuggingSessionSocketMessageJson / Dient zur Übermittlung der Beendigung des Debugging Prozesses.
- `METHOD_ENTRY[2]` / A.9.18 MethodEntrySocketMessageJson / Dient zur Übermittlung der beim Betreten einer unter Beobachtung stehenden Methode ermittelten Daten.
- `METHOD_EXIT[3]` / A.9.19 MethodExitSocketMessageJson / Dient zur Übermittlung der beim Verlassen einer unter Beobachtung stehenden Methode ermittelten Daten.
- `WATCHER_CHANGED[4]` / A.9.20 WatcherChangedSocketMessageJson / Dient zur Übermittlung der bei einer Änderung eines von einem Watcher beobachteten Feldes ermittelten Daten.
- `BREAKPOINT[5]` / A.9.21 BreakpointSocketMessageJson / Dient zur Übermittlung der bei einer Erreichung eines Haltepunktes ermittelten Daten.

Damit der Anfang, sowie das Ende einer Socket Nachricht eindeutig zu identifizieren ist, wird sie um Steuerzeichen der ASCII-Zeichentabelle ergänzt. Daraus resultiert, dass sämtliche Socket Nachrichten dem folgenden Aufbau entsprechen:

STX[L] + Playload im JSON Format + ETX[M]

[K] Auflistungsformat: `messageType[Value]` / JSON Definition des jeweiligen `messageJson`s / Kurzbeschreibung
[L] STX = Start of Text - ASCII-Steuerzeichen
[M] ETX = End of Text - ASCII-Steuerzeichen

4.6.2 Socket Kommunikation: Java

Die gesamte Implementierung der Socket Kommunikation befindet sich im `uni.bremen.debuggerBackend.services.socket` Paket, selbiges wurde in der „Abbildung 15: Abhängigkeitsskizze: Debugger Socket“ visualisiert. Es beinhaltet den `SocketService`, dessen Hilfsklassen, sowie die Socket-Nachrichten-DTOs, welche auf dem Schnittstellendesign der Socket Kommunikation basieren. Letztere befinden sich im Teilpaket `uni.bremen.debuggerBackend.services.socket.socketMessages`.

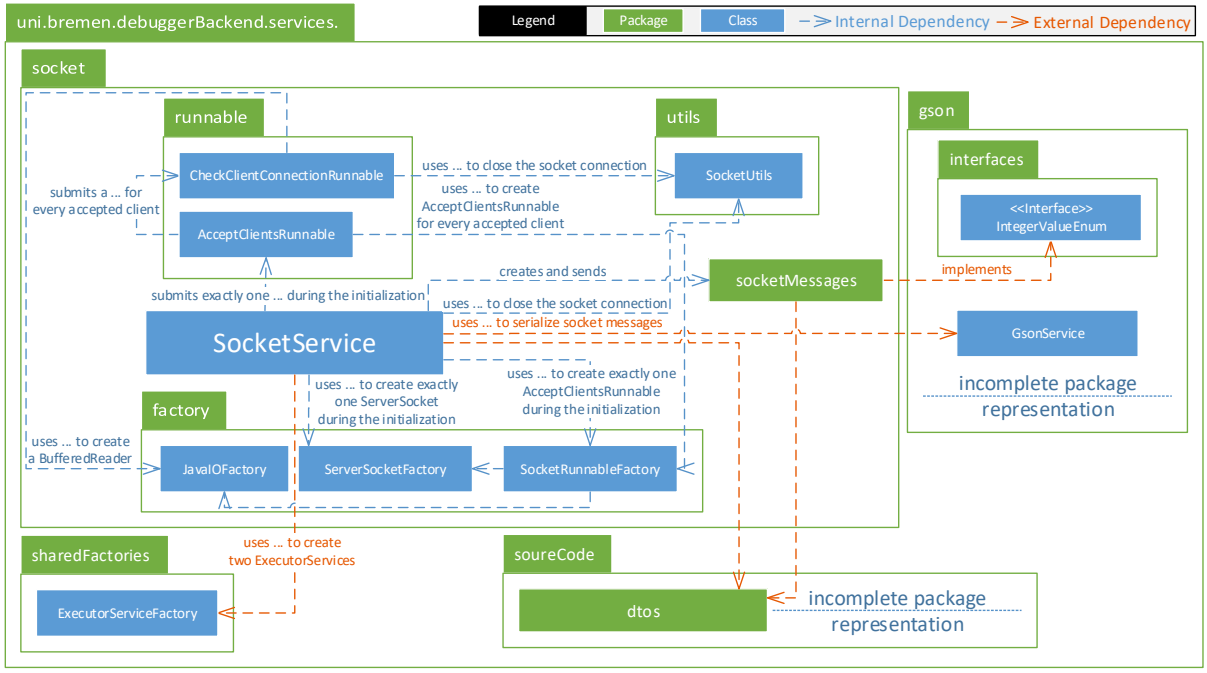

Abbildung 15: Abhängigkeitsskizze: Debuger Backend - Socket
Eine vergrößerte Version dieser Abbildung finden Sie im Anhang unter A.12 Abhängigkeitsskizze: Debuger Backend - Socket.

Die eigentliche Kernkomponente der Socket Kommunikation bildet der `SocketService`. Er verwaltet die verbundenen Sockets und ermöglicht es anderen Komponenten Socket-Nachrichten zu verschicken. Beim Systemstart wird die `init()`-Methode des `SocketService`s durch die *javax* Annotation `@PostConstruct`[58] getriggert. Diese öffnet einen `ServerSocket`[59] unter dem in der Konfiguration angegebenen Port[N]. Zusätzlich erstellt sie anschließend mittels der `SocketRunnableFactory` einen `AcceptClientsRunnable`, dieser akzeptiert bis zu `SocketService#maxClients`[O] Clients. Der `AcceptClientsRunnable` erzeugt für jeden akzeptierten Client eine `CheckClientConnectionRunnable`. Hierfür wird erneut auf die `SocketRunnableFactory` zurückgegriffen. Die erstellten `CheckClientConnectionRunnable`s

[N] application.properties - Konfigurationsparameter: `socketServerPort` (Default = 8040)
[O] application.properties - Konfigurationsparameter: `numberOfMaxClients` (Default = 1)

entfernt den von ihnen beobachteten Client Socket aus der vom `SocketService` verwalteten Socket Client Menge, sobald die Socket-Verbindung unterbrochen wird.

Die Socket-Nachrichten werden als Byte-Arrays übertragen. Solch ein Byte Array beinhaltet das aus der Socket-Nachricht resultierende JSON. Des Weiteren ist dessen Anfang und Ende mittels STX und ETX Steuerzeichen maskiert. Diese Art der Nachrichtenversendung resultiert aus dem Schnittstellendesign der Socket Kommunikation.

Abschließend gilt es anzumerken, dass der Serversocket, sowie sämtliche mit ihm verbundenen Clients, von der `shutdown()`-Methode des `SocketServices`, welche mit der `PreDestroy`[60] Annotation ausgestattet wurde, bei Beendigung der Anwendung geschlossen werden.

Auch für diesen Abschnitt des *Debugger Backend*s finden Sie eine ausführliche Code-Dokumentation in Form von JavaDoc auf der beiliegenden Disk im Ordner *Debugger-Backend_JavaDoc*.

4.6.3 Socket Kommunikation: C++

Die Socket Kommunikation des *UVRD* verläuft über den `UVRDSocketManager`, dieser stellt eine Verbindung zu dem Serversocket unter dem in der `UVRDConfiguration` angegebenen Port her. Der `UVRDSocketManager` selbst, sowie die von ihm benutzte Hilfsklasse, die `UVRDSocketUtils`, basieren auf den von der Unreal Engine bereitgestellten Standardroutinen, die in Form des *Sockets*, sowie des *Networking* Moduls in das Projekt eingebunden wurden.

Der `UVRDSocketManager` beinhaltet einen `FSocket*`, namentlich `DebuggerBackendSocket`, welcher immer den aktuellen `FSocket`[61] der Socket Kommunikation referenziert, und zwei `FTimerHandler`[62]:

`ReConnectIfNecessaryTimerHandler`
`HandleTCPSocketMessageTimerHandler`

Der `ReConnectIfNecessaryTimerHandler` führt alle $3000ms$ die `ReConnectIfNecessary`-Methode des `UVRDSocketManagers` aus. Diese wiederum überprüft die Verbindung zum Serversocket, indem sie einen *Heartbeat* an den Server übermittelt. Als *Heartbeat*-Payload wurde das ASCII Steuerzeichen `ENQ`[P] gewählt. Sollte keine Verbindung bestehen, versucht die Methode außerdem selbige selbst aufzubauen. Wenn der Verbindungsaufbau fehlschlägt, dann öffnet sie mittels des `UVRDWidgetManager` das `BP_ConnectToSocketInfoLayer`-Widget. Der vom dem Widget dargestellte `Fail Counter` entspricht der Anzahl an Verbindungsfehlschläge. Des Weiteren wird der `HandleTCPSocketMessageTimerHandler` pausiert, solange keine Socket-Verbindung hergestellt werden kann. Nachdem eine Verbindung hergestellt werden konnte, resettet die Methode den `Fail Counter` der `BP_ConnectToSocketInfoLayer` und setzt abschließend den `HandleTCPSocketMessageTimerHandler` fort.

Der `HandleTCPSocketMessageTimerHandler` führt alle $1ms$ die `HandleTCPSocketMessage`-Methode des `UVRDSocketManagers` aus. Diese Methode holt sich sämtliche ausstehenden Da-

[P] ENQ = enquiry (Anfrage)

ten in Form eines Bytes-Arrays vom `DebuggerBackendSocket` ab. Das resultierende Byte Array wird dann in einzelne Socket-Nachrichten unterteilt. Hierfür wird das Array an den STX und ETX Steuerzeichen, welche die einzelnen Socket-Nachrichten voneinander trennen, in einzelne Abschnitte unterteilt, welche dann zunächst in einen *FString*[63] und abschließend mittels der `UVRDJsonUtils` in eine `FSocketMessage`-Struct überführt werden. Unvollständige Socket-Nachrichten werden bei der darauffolgenden Iteration der Methode vervollständigt. Die vollständigen Socket-Nachrichten werden wiederum vom `UVRDSocketManager` in Abhängigkeit zum `ESocketMessageType` der jeweiligen `FSocketMessage` wie folgt verarbeitet:

- `ESocketMessageType::MT_METHOD_ENTRY`: Skaliere den `UVRDTracePawn` basierend auf dem Call Stack der erhaltenden `FMethodEntrySocketMessage` und füge eine auf dem Inhalt der Nachricht basierende Zeile im Log-Widget hinzu, wenn das Logging von *Trace-Socket-Messages* aktiviert ist.
- `ESocketMessageType::MT_METHOD_EXIT`: Skaliere den `UVRDTracePawn` basierend auf dem Call Stack der erhaltenden `FMethodExitSocketMessage` und füge eine auf dem Inhalt der Nachricht basierende Zeile im Log-Widget hinzu, wenn das Logging von *Trace-Socket-Messages* aktiviert ist.
- `ESocketMessageType::MT_WATCHER_CHANGED`: Füge eine auf dem Inhalt der erhaltenden `FWatcherChangedSocketMessage` basierende Zeile im Log-Widget hinzu.
- `ESocketMessageType::MT_BREAKPOINT`: Füge eine auf dem Inhalt der erhaltenden `FBreakpointSocketMessage` basierende Zeile im Log-Widget hinzu.
- `ESocketMessageType::MT_PAUSE_DEBUGGING_SESSION`: Füge eine auf dem Inhalt der erhaltenden `FPauseDebuggingSessionSocketMessage` basierende Zeile im Log-Widget hinzu und update das `BP_DebuggerControlPanel`-Widget mit dem erhaltenen `EDebuggingSessionState`.
- `ESocketMessageType::MT_END_DEBUGGING_SESSION`: Füge eine auf dem Inhalt der erhaltenden `FEndDebuggingSessionSocketMessage` basierende Zeile im Log-Widget hinzu und update das `BP_DebuggerControlPanel`-Widget mit dem erhaltenen `EDebuggingSessionState`.

Sämtliche für die Socket Kommunikation benötigten DTO-Structs befinden sich übrigens im `StructLibraryObject`. Die `FinishDestroy`-Methode des `UVRDSocketManager`s sorgt dafür, dass die Socket-Verbindung bei der Beendigung des *UVRD*s geschlossen wird.

4.7 UVRD: Front-End

Der folgende Abschnitt der Ausarbeitung behandelt das Front-End des *Unreal VR Debugger*s. Bevor auf das eigentlich Front-End eingegangen wird, sei erwähnt, dass das resultierende User Interface lediglich während der Erstellung der User Stories händisch skizziert wurde. Bei der Erstellung der Skizzen wurde nur teilweise auf Aspekte der *User Experience* eingegangen.

Das Front-End an sich besteht zum Großteil aus UMG-Widgets, sowie aus der `BP_UVRDTracePawn`, welche zur Visualisierung des Call Stacks genutzt wird. Zusätzlich beinhaltet es eine einfache Implementierung des `AHUD`. Auf die `BP_UVRDTracePawn` und die `BP_UVRDHUD` wurde bereits im Abschnitt 4.3.2.1 Blueprints genauer eingegangen. Zunächst wird die initiale Benutzeroberfläche des *UVRD*s betrachtet. Sie besteht aus den folgenden vier Komponenten, die in der „Abbildung 4.7: Initiale Benutzeroberfläche“ hervorgehoben wurden.

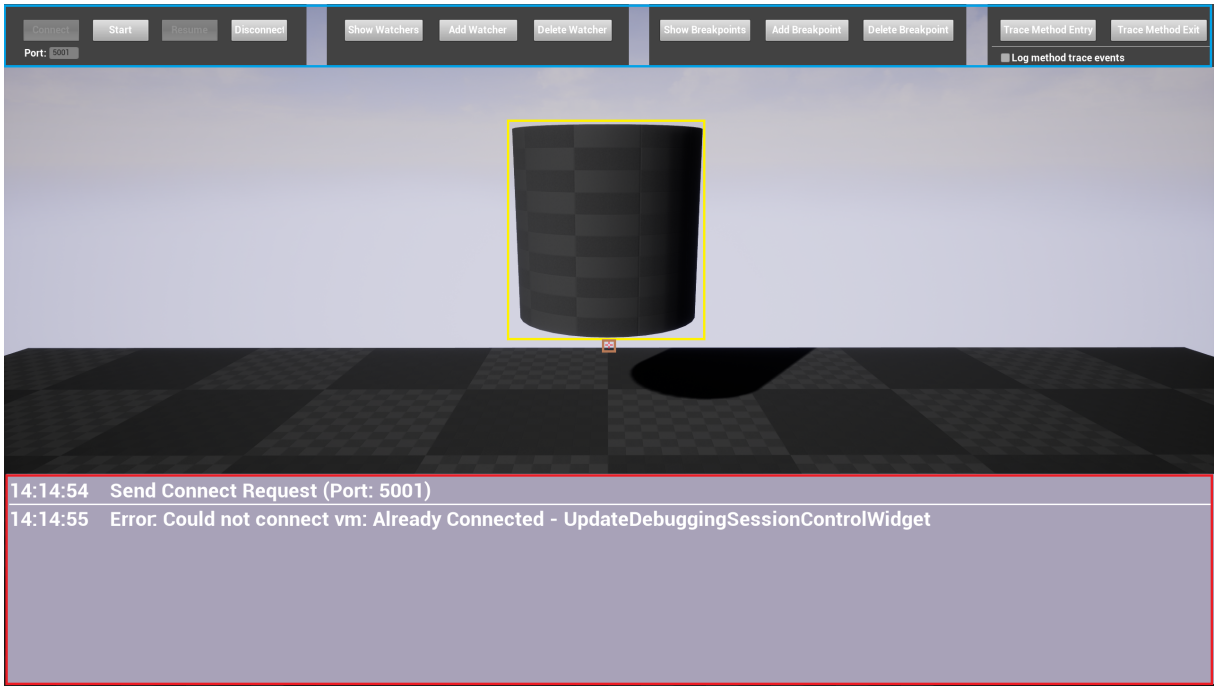

Abbildung 16: Initiale Benutzeroberfläche

Blaue Umrandung: `BP_DebuggerControlPanel`
Gelbe Umrandung: `BP_UVRDTracePawn`
Braune Umrandung: `BP_UVRDHUD`
Rote Umrandung: `BP_Log`

Eine vergrößerte Version dieser Abbildung finden Sie im Anhang unter A.13 Initiale Benutzeroberfläche.

Die folgenden Abschnitte der Ausarbeitung behandeln die einzelnen Front-End-Komponenten.

4.7.1 BP_DebuggerControlPanel

Das BP_DebuggerControlPanel-Widget dient zur allgemeinen Steuerung des Debuggers. Weshalb es den Großteil der Front-End-Logik enthält. Es besteht aus vier fachlichen Rubriken, welche durch Zwischenräume voneinander abgetrennt werden. Dabei verbergen sich hinter den verschiedenen Kontrollknöpfen der jeweiligen Rubriken folgende Funktionalitäten:[Q]

BP_DebuggerControlPanel: Debugging Session

- *Connect*: Wenn der Port aus dem zum *Connect*-Button dazugehörigen Textinput einer positiven Zahl entspricht, dann deaktiviert die Betätigung dieses Buttons sämtliche Kontrollknöpfe. Anschließend wird die `Connect(int32 Port)`-Methode des `UVRDDebuggingSessionManager` aufgerufen.
- *Start*: Die Betätigung dieses Buttons deaktiviert sämtliche Kontrollknöpfe. Anschließend wird die `Start()`-Methode des `UVRDDebuggingSessionManager` aufgerufen.
- *Resume*: Die Betätigung dieses Buttons deaktiviert sämtliche Kontrollknöpfe. Anschließend wird die `Resume()`-Methode des `UVRDDebuggingSessionManager` aufgerufen.
- *Disconnect*: Die Betätigung dieses Buttons deaktiviert sämtliche Kontrollknöpfe. Anschließend wird die `Disconnect()`-Methode des `UVRDDebuggingSessionManager` aufgerufen.

Die Funktionalitäten der *Debugging Session*-Rubrik werden anderen Komponenten als öffentliche Methoden der Blueprint bereitgestellt, damit diese zukünftig ggf. auch per Hotkey genutzt werden können.

BP_DebuggerControlPanel: Watcher

- *Show Watchers*: Die Betätigung dieses Buttons führt zum Aufruf der `RequestActiveWatchersAndShowActiveWatchersDialog()`-Methode des `UVRDWatcherManager`. Wenn dieser erfolgreich aktive Watcher vom *Debugger Backend* abfragen konnte, dann wird im Anschluss vom Manager das `BP_Active_Watchers_Overlay`-Widget mit den erworbenen Daten geöffnet.
- *Add Watcher*: Die Betätigung dieses Buttons führt zum Aufruf der `RequestAvailableWatchersAndShowAddWatcherDialog()`-Methode des `UVRDWatcherManager`. Wenn dieser erfolgreich Informationen über die verfügbaren Watcher vom *Debugger Backend* abfragen konnte, dann wird im Anschluss vom Manager das `BP_Add_Watcher_Overlay`-Widget geöffnet.
- *Delete Watcher*: Die Betätigung dieses Buttons führt intern die Funktionalität des *Show Watchers*-Buttons aus, da Watcher im dadurch geöffneten `BP_Active_Watchers_Overlay`-Widget gelöscht werden können.

[Q] Auflistungsformat: *Knopfbeschriftung*: Funktionalität

`BP_DebuggerControlPanel`**: Breakpoint**

- *Show Breakpoints*: Die Betätigung dieses Buttons führt zum Aufruf der `RequestActiveBreakpointsAndShowActiveBreakpointsDialog()`-Methode des `UVRDBreakpointManager`. Wenn dieser erfolgreich Informationen über die aktive Breakpoints vom *Debugger Backend* abfragen konnte, dann wird im Anschluss vom Manager das `BP_Active_Breakpoints_Overlay`-Widget mit den erworbenen Daten geöffnet.
- *Add Breakpoint*: Die Betätigung dieses Buttons führt zum Aufruf der `RequestClassNamesAndShowAddBreakpointDialog()`-Methode des `UVRDBreakpointManager`. Wenn dieser erfolgreich Informationen über die verfügbaren *ClassNames*, für welche Breakpoints erstellt werden könnten, vom *Debugger Backend* abfragen konnte, dann wird im Anschluss vom Manager das `BP_Add_Breakpoint_Overlay`-Widget geöffnet.
- *Delete Breakpoint*: Die Betätigung dieses Buttons führt intern die Funktionalität des *Show Breakpoints*-Buttons aus, da Breakpoints im dadurch geöffneten `BP_Active_Breakpoints_Overlay`-Widget gelöscht werden können.

`BP_DebuggerControlPanel`**: Method Tracing**

- *Trace Method Entry*: Die Betätigung dieses Buttons führt zum Aufruf der `ActivateMethodEntryTraceForRootPackage()`-Methode des `UVRDTraceManager`.
- *Trace Method Exit*: Die Betätigung dieses Buttons führt zum Aufruf der `ActivateMethodExitTraceForRootPackage()`-Methode des `UVRDTraceManager`.
- *Log method trace events*: Diese Checkbox dient zur Konfiguration des Loggings von Socket Nachrichten, welche den *UVRD* über das Betreten bzw. das Verlassen von Methoden informieren. Wenn die Checkbox angehakt ist, dann werden die eben beschriebenen Socket Nachrichten mittels des `BP_Log` mitgeloggt.

Zusätzlich zu den bereits erwähnten Funktionalitäten verfügt die `BP_DebuggerControlPanel` auch noch über eine öffentliche `Update(EDebuggingSessionState DebuggingSessionState)`-Methode, welche es anderen Komponenten erlaubt das Widget in Abhängigkeit zum übergebenen `EDebuggingSessionState` zu aktualisieren. Dabei wird innerhalb der Blueprint festgelegt, welche Kontrollknöpfe aktiviert bzw. deaktiviert werden müssen. Die Logik der Methode wurde in Form der folgenden Tabelle (siehe „Tabelle 9: Zustand der Kontrollköpfe in Abhängigkeit zum *Debugging Session State*“) für den Leser zusammengefasst. Sie basiert auf dem in Kapitel 4.4.3 Lebenszyklus: Debugger Session bereits erwähnten Einschränkungen des *Debugger Backend*s.

Button	not connected	connected and wait for start	connected and running	connected and wait for resume
Connect	aktiviert	deaktiviert	deaktiviert	deaktiviert
Start	deaktiviert	aktiviert	deaktiviert	deaktiviert
Resume	deaktiviert	deaktiviert	deaktiviert	aktiviert
Disconnect	deaktiviert	aktiviert	aktiviert	aktiviert
Show Watchers	aktiviert	aktiviert	aktiviert	aktiviert
Add Watcher	aktiviert	aktiviert	deaktiviert	deaktiviert
Delete Watcher	aktiviert	aktiviert	deaktiviert	deaktiviert
Show Breakpoints	aktiviert	aktiviert	aktiviert	aktiviert
Add Breakpoint	aktiviert	aktiviert	deaktiviert	deaktiviert
Delete Breakpoint	aktiviert	aktiviert	deaktiviert	deaktiviert
Trace Method Entry	aktiviert	aktiviert	deaktiviert	deaktiviert
Trace Method Exit	aktiviert	aktiviert	deaktiviert	deaktiviert

Tabelle 9: Zustand der Kontrollköpfe in Abhängigkeit zum *Debugging Session State*

4.7.2 BP_Log

Das BP_Log-Widget verfügt über eine öffentlich addLogLine(FString time, FString data)-Methode. Mittels dieser Methode ist es anderen Komponenten möglich Logzeilen zu erstellen. Die einzelnen Zeilen des Logs bestehen aus BP_LogLine-Widgets, welche vom BP_Log erzeugt werden. Das Log dient zur Darstellung von benutzerrelevanten Informationen. Das Loggen von Ablaufverfolgungsevents kann dazu führen, dass das Front-End vorübergehend einfriert, da hierbei eine große Anzahl an Logzeilen innerhalb halb einer kurzen Zeitspanne erzeugt werden.

4.7.3 BP_Active_Watchers_Overlay

Die BP_Active_Watchers_Overlay dient zur Darstellung der aktiven Watcher. Dabei verfügt sie über eine öffentliche Init(...)-Methode, welche die zur Initialisierung benötigte FWatcherFileInformation-Sammlung übergeben bekommt. Diese Methode erstellt für jede FWatcherFileInformation ein BP_Active_Watchers_File_Listing-

Widget, das wiederum für jeden Feldnamen ein `BP_Active_Field_Watcher`-Widget erzeugt. Schlussendlich resultiert das in „Abbildung 4.7.3: Active Watcher Overlay“ abgebildete Overlay. Jedes `BP_Active_Field_Watcher`-Widget verfügt außerdem über einen *Delete*-Button, welcher bei Betätigung das Overlay schließt und im Anschluss die `DeleteWatcher(const FDeleteWatcherRequestData &DeleteWatcherRequestData)`-Methode des `UVRDWatcherManager` aufruft. Das Overlay verfügt außerdem über einen *Close*-Button, welcher die Schließung des Overlays jederzeit ermöglicht.

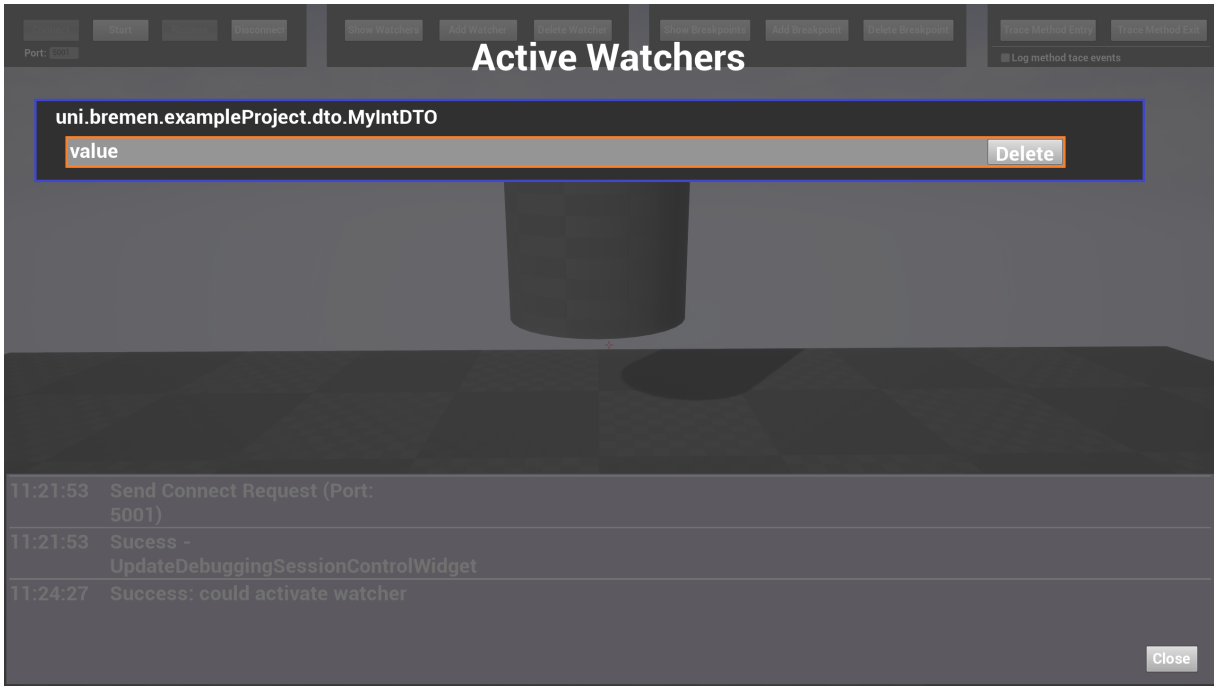

Abbildung 17: Active Watcher Overlay
Blaue Umrandung: `BP_Active_Watchers_File_Listing`
Orange Umrandung: `BP_Active_Field_Watcher`
Eine vergrößerte Version dieser Abbildung finden Sie im Anhang unter A.14 Active Watcher Overlay.

4.7.4 `BP_Add_Watcher_Overlay`

Das `BP_Add_Watcher_Overlay`-Widget, siehe „Abbildung 18: Add Watcher Overlay“, verfügt über zwei Selectboxen, sowie einen *Submit*- und einen *Cancel*-Button. Es dient zur Erstellung von Watchern. Zunächst wird die Klasse und anschließend das zu beobachtete Feld ausgewählt. Wenn beide Selectboxen befüllt sind, dann kann die Erstellung mittels des *Submit*-Buttons angestoßen werden. Dabei wird das Overlay geschlossen und anschließend wird die `ActivateWatcher(const FActivateWatcherRequestData &ActivateWatcherRequestData)`-Methode des `UVRDWatcherManagers` aufgerufen. Ein Klick auf den *Cancel*-Button schließt hingegen das Overlay.

Die Initialisierung des Overlay erfolgt mittels einer öffentlichen `InitSelectors(...)`-Methode. Diese bekommt eine `FWatcherFileInformation`-Sammlung übergeben, welche potentielle Watcher enthält.

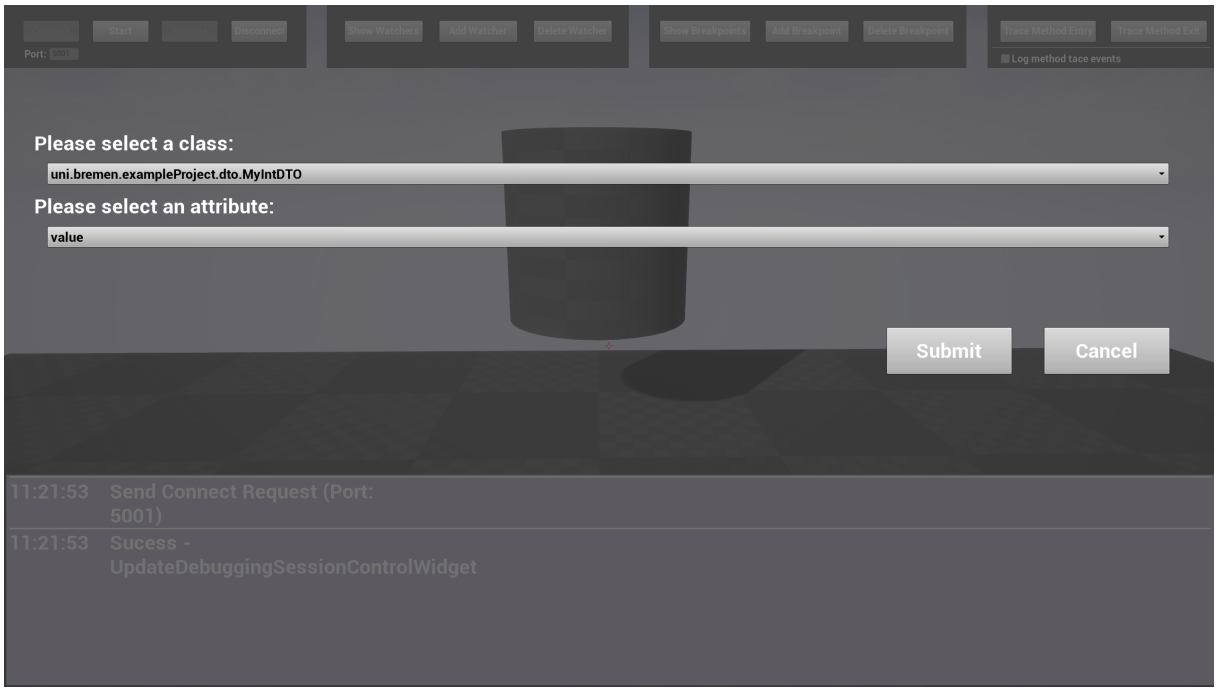

Abbildung 18: Add Watcher Overlay
Eine vergrößerte Version dieser Abbildung finden Sie im Anhang unter A.15 Add Watcher Overlay.

4.7.5 `BP_Active_Breakpoints_Overlay`

Das `BP_Active_Breakpoints_Overlay`-Widget, siehe „Abbildung 4.7.5: Active Breakpoints Overlay“, dient zur Darstellung der aktiven Breakpoints. Dabei erzeugt es ähnlich wie das `BP_Active_Watchers_Overlay`-Widget für sämtliche bei der Initialisierung übergebenen `FBreakpointFileInformations` `BP_Active_Breakpoints-_File_Listing`-Widgets, welche `BP_Active_Breakpoint`-Widgets beinhalten. Letztere enthalten die Zeilenangabe des jeweiligen Breakpoints, sowie einen *Delete*-Button, dessen Betätigung schließt das Overlay und führt anschließend zum Aufruf der `DeleteBreakpoint(const FDeleteBreakpointRequestData &DeleteBreakpointRequestData)`-Methode des `UVRDBreakpointManager`. Das Overlay kann außerdem durch einen Klick auf den `Close`-Button jederzeit vom Nutzer geschlossen werden.

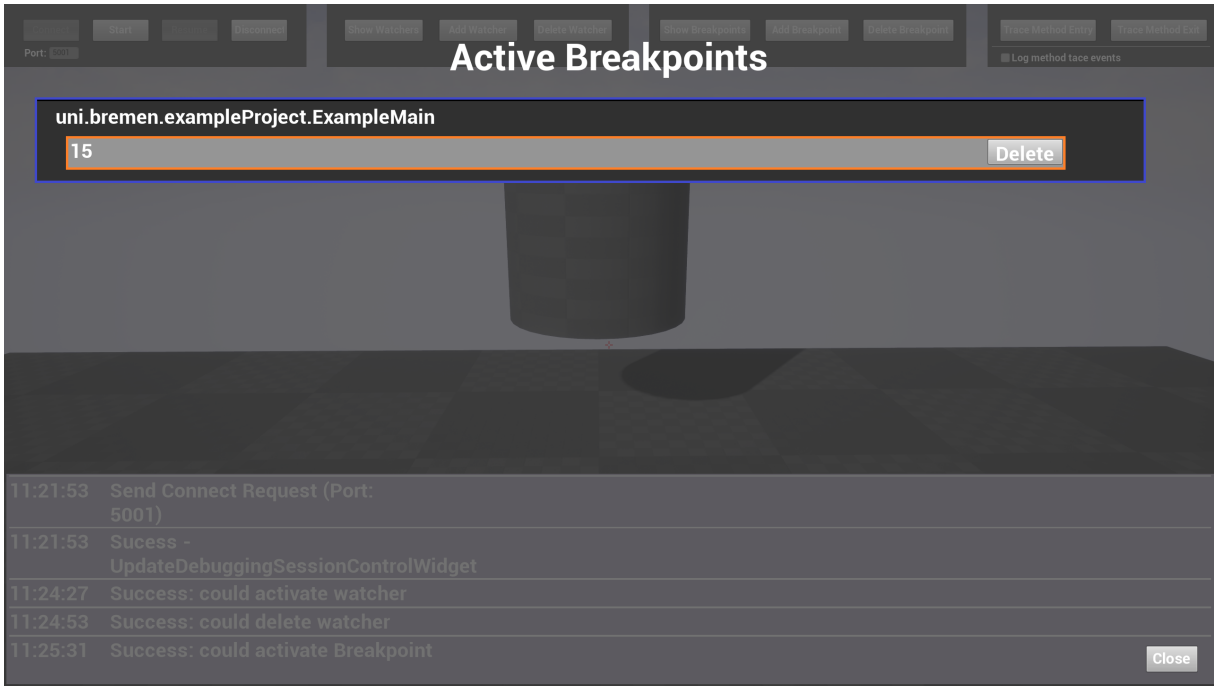

Abbildung 19: Active Breakpoints Overlay
Blaue Umrandung: `BP_Active_Breakpoints_File_Listing`-Widgets
Orange Umrandung: `BP_Active_Breakpoints`-Widgets
Eine vergrößerte Version dieser Abbildung finden Sie im Anhang unter A.16 Active Breakpoints Overlay.

4.7.6 `BP_Add_Breakpoint_Overlay`

Das `BP_Add_Breakpoint_Overlay`-Widget, siehe „Abbildung 20: Add Breakpoint Overlay“, verfügt über eine Selectbox, ein Textinput-Feld, einen *Submit*- und einen *Cancel*-Button. Es dient zur Erstellung von Breakpoints. Hierbei wird zunächst die Klasse per Selectbox ausgewählt und anschließend wird die gewünschte Zeile des Breakpoints in das Textinput-Feld eingegeben. Anschließend kann die Erstellung mittels des *Submit*-Buttons angestoßen werden. Dabei wird das Overlay geschlossen und die `ActivateBreakpoint(const FActivateBreakpointRequestData &ActivateBreakpointRequestData)`-Methode des `UVRDBreakpointManager`s wird aufgerufen. Ein Klick auf den *Cancel*-Button schließt hingegen das Overlay.

Auch dieses Overlay verfügt über einer öffentlichen `InitSelectors(...)`-Methode. Diese bekommt eine `FString`-Sammlung übergeben, welche die vollständigen Klassennamen des *Debugging Target*s beinhaltet.

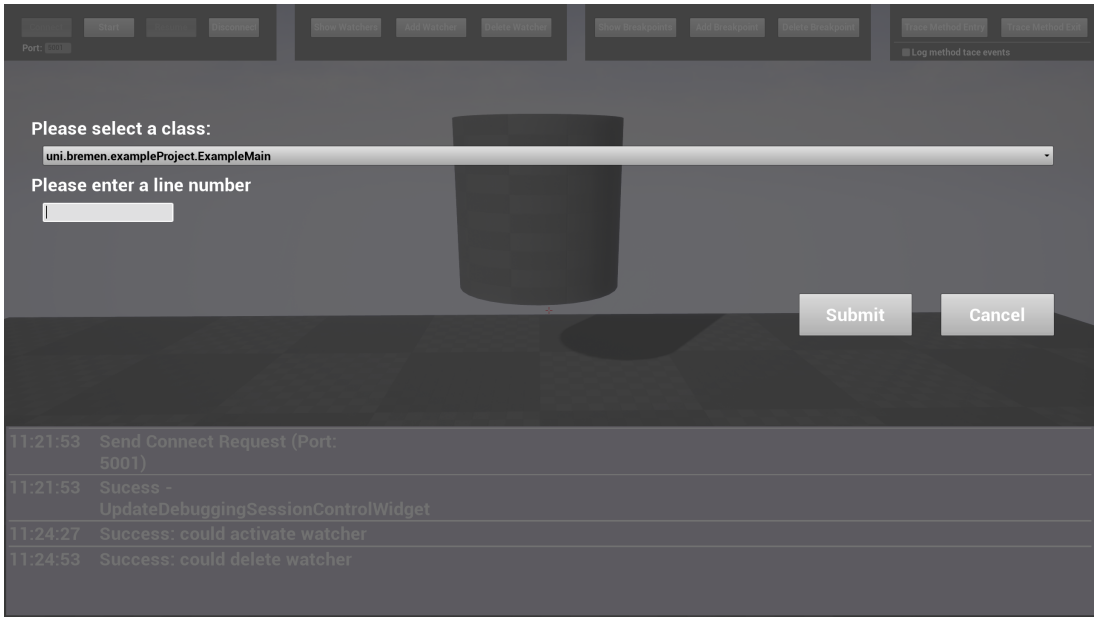

Abbildung 20: Add Breakpoint Overlay
Eine vergrößerte Version dieser Abbildung finden Sie im Anhang unter A.17 Add Breakpoint Overlay.

4.7.7 `BP_ConnectToSocketInfoLayer`

Das `BP_ConnectToSocketInfoLayer`-Widget, siehe „Abbildung 21: Connect To Socket Info Layer“, dient dazu den Nutzer über eine Unterbrechung der Socket Kommunikation zu informieren. Es besitzt zwei öffentliche Methoden. Zum einen die `IncreaseFailCounter()`-Methode, die den *Fail Counter*, welcher in der Abbildung 21 orange umrandet wurde, erhöht und zum anderen die `ResetFailCounter()`-Methode, welche den *Fail Counter* wiederum zurücksetzt.

Abbildung 21: Connect To Socket Info Layer
Eine vergrößerte Version dieser Abbildung finden Sie im Anhang unter A.18 Connect To Socket Info Layer.

5 Evaluation

Während der Evaluation werden zunächst die implementieren Funktionalitäten betrachtet. Im Anschluss wird sich genauer mit den Auswirkungen des *UVRD*s auf das *Debugging Target* auseinandergesetzt.

5.1 Implementierte Funktionalität

Die implementieren Funktionalitäten entsprechen den im Abschnitt 4.1.1 aufgeführten User Stories, welche auf der im Kapitel 2 beschriebenen Zielsetzung basieren. Es gilt jedoch zu berücksichtigen, dass während der Entwicklung bestimmte Einschränkungen in Kauf genommen wurden, auf die zwei wesentlichsten wird im Folgenden eingegangen.

Debugging-Ablauf: Die aktuelle Implementierung des *Debugger Backend*s führt dazu, dass der Nutzer keinerlei Watcher, Breakpoints oder Ablaufverfolgungen während des Debugging-Prozesses erstellen oder modifizieren kann. Aufgrund der Tatsache, dass es sich bei dem Debugging-Ablauf um einen Kernbestandteil des *Debugger Backend*s handelt, wird der Änderungsaufwand basierend auf den Entwicklungswerten auf 10 Personentage geschätzt. Innerhalb dieses Zeitraumes sollte es möglich sein die Logik so anzupassen, dass der Nutzer sämtliche Debugging-Inhalte auch während einer aktuellen Debugging Session verwalten kann.

Verwaltung der Ablaufverfolgung: Zur Zeit ist es nur möglich die Ablaufverfolgung von Methoden für das Root-Package zu aktivieren. Es wäre wünschenswert, wenn der Nutzer die Ablaufverfolgung auch deaktivieren könnte. Hierfür wäre eine Erweiterung der HTTP-Schnittstelle, sowie der Debugging-Logik notwendig. Der geschätzte Entwicklungsaufwand für diese Änderung beträgt drei Personentage.

5.2 Auswirkungen auf das Debug Target

Es gilt zu ermitteln, inwiefern sich die Nutzung des *UVRD*s auf das *Debugging Target* auswirkt. Insbesondere die Auswirkung der Live Call Stack Ermittlung in Form der Ablaufverfolgung der Methoden ist zu untersuchen. Dafür wurden die folgenden vier verschiedene Messreihen angelegt:

- **MR1:** Ausführung des Beispielprojektes ohne Debugger-Verbindung
- **MR2:** Ausführung des Beispielprojektes mit IntelliJ IDEA Remote-Debugger-Verbindung
- **MR3:** Ausführung des Beispielprojektes mit UVRD-Verbindung (deaktivierte Methoden-Ablaufverfolgung)
- **MR4:** Ausführung des Beispielprojektes mit UVRD-Verbindung (aktivierte Methoden-Ablaufverfolgung)

Zunächst wird kurz auf das Beispielprojekt eingegangen, welches bei den Messungen als *Debugging Target* diente. Anschließend werden die Ergebnisse der verschiedenen Messreihen betrachtet.

5.2.1 Beispiel Projekt

Während der Entwicklung wurde parallel ein sogenanntes Beispielprojekt unter dem Projektnamen *exampleProject* implementiert. Bei diesem Projekt handelt es sich ebenfalls wie bei dem *Debugger Backend* um ein Maven Projekt, dessen *pom.xml* finden Sie im Anhang unter A.20 `exampleProject:` *pom.xml*. In ihr wurden explizit `-g:source,lines,vars`[64] als `compilerArgs`[65] angegeben. Dies führt dazu, dass das resultierenden Kompilat die benötigten *Debugging-Informationen* beinhaltet.

Bei dem *exampleProject* handelt es sich um eine sehr einfache Java-Anwendung, dessen Klassendiagramm finden Sie im Anhang unter A.21 `exampleProject:` Klassendiagramm. Die Hauptaufgabe der Anwendung besteht darin auf einer unnötig komplizierten Art und Weise von 0 bis 1000 zu zählen, dabei loggt sie die Startzeit und die Endzeit des Zählvorganges, sowie dessen Differenz.

Die resultierende `exampleProject-1.0.jar` diente während der Implementierung und der Evaluation der entwickelten Lösung als *Test Debugging Target*. Zum Starten des *Debugging Target*s wurden zwei Hilfsskripte geschrieben:

- `runExampleProject.bat`: Führt die aktuelle `exampleProject-1.0.jar`, welche sich im `target`-Verzeichnis befindet, aus.
- `runExampleProject_debug.bat`: Führt die aktuelle `exampleProject-1.0.jar`, welche sich im `target`-Verzeichnis befindet, im Debug-Modus (Port 5001) aus.

Sie befinden sich beide im Hauptverzeichnis des *exampleProject*s und erleichtern die Ausführung der resultierenden Anwendung unter dem Betriebssystem *Microsoft Windows 10*.

Das komplette *exampleProject*, inklusive der Hilfsskripte, ist auf der beiliegenden DVD im Verzeichnis `exampleProject` zu finden.

5.2.2 Betrachtung der Messergebnisse

Zunächst wird der Ablauf der Messungen genauer erörtert, im Anschluss werden dann die eigentlichen Messergebnisse betrachtet.

Das *Debugging Target* wurde für die erste Messreihe mittel des bereits erwähnten Hilfsskriptes `runExampleProject.bat` ausgeführt. Für die anderen Messreihen wurde es mittels des `runExampleProject_debug.bat` gestartet, dies ermöglicht es den Debuggern eine Remote-Debugging-Verbindung aufzubauen.

Um Messungenauigkeiten zu vermeiden, wurde unter anderem drauf geachtet, dass bei sämtlichen Messungen dieselben Haupt- und Hintergrundprozesse abliefen.

Für die Messungen wird kein zusätzlicher Aufwand durch die Einführung von zusätzlichen

Zeitabfragen produziert, da die Messergebnisse auf den Logausgaben des *exampleProject*s basieren, welche wiederum ein allgemeiner Bestandteil der Anwendung sind. Aufgrund des einfaches Aufbaues des *exampleProject*s könnte es jedoch sein, dass die aus der Messung resultierenden Ergebnisse nicht repräsentativ sind, da die interne Implementierung des JDI nicht genauer bekannt ist. Außerdem sind die ermittelten Messwerte zusätzlich mit Vorsicht zu genießen, da die Messungen nur auf einem spezifischen System[R] durchgeführt wurde und auch für die durchgeführten Messungen gilt:

> *Jede Messung ist fehlerbehaftet. Der Grund liegt in bestimmten Eigenschaften des Messgegenstandes, in Unvollkommenheiten der Messeinrichtung und des Messverfahrens, in wechselnden Umwelteinflüssen und in Fehlern des Beobachters.“*
>
> – [Leo15]

Betrachten wir nun die Zusammenfassung der Messwerte:

Messreihe	Dauer des Zählvorganges (Mittelwert) [ms]
MR1	360,41
MR2	406,86
MR3	399,08
MR4	4938,89

Tabelle 10: Messreihenzusammenfassung

Die Mittelwerte der jeweiligen Messreihen wurden auf zwei Nachkommastellen gerundet. Eine vollständige Auflistung der Messreihen finden Sie im Anhang unter A.19 Messungen.

Wenn die Messreihen *MR2* und *MR3* miteinander verglichen werden, dann fällt auf, dass sich der *UVRD* wahrscheinlich nicht negativer als der IntelliJ IDEA Remote-Debugger auf das *Debugging Target* auswirkt. Bei der Betrachtung der *MR4* fällt jedoch auf, dass die Methoden-Ablaufverfolgung die durchschnittliche Laufzeit des *Debugging Target* erheblich beeinträchtigt. Dabei muss bedacht werden, dass für die Ablaufverfolgung von der JVM jedes Mal ein Event erstellt werden muss, wenn eine Methode betreten bzw. verlassen wird. Dies führt dazu, dass das *Debugger Backend* bei einem Zähldurchlauf des *Debugging Target*s unter anderem insgesamt 4004 *MethodEntryEvent*s und 4004 *MethodExitEvent*s verarbeiten und dessen enthaltene Informationen per Socket-Nachricht an den *UVRD* übertragen muss. Unter diesem Gesichtspunkten, erscheint die verlängerte Laufzeit als vertretbar. Es gilt zu erwähnen, dass während der Zielsetzung und der Entwicklungsphase kein besonderes Augenmerk auf die Performance der Ablaufverfolgung gelegt wurde.

[R] Systemspezifikation finden Sie im Anhang unter A.22 Systemspezifikation

6 Fazit

Ziel dieser Bachelorarbeit war es einen Java Debugger in ein Unreal Engine Projekt zu integrieren. Dabei wurden zunächst verschiedene Integrationsarten auf ihre Stärken und Schwächen analysiert. Anschließend wurde die vielversprechendste Integrationsart beispielhaft implementiert.

Der für das Projekt nötige Arbeitsaufwand wurde vom Entwickler während der Einarbeitungszeit stark unterschätzt. Im Exposé wurde ein Entwicklungszeitraum von circa zehn Wochen eingeplant. Dieser konnte zwar auch mit leichter Verzögerung eingehalten werden, dafür wurden während der Entwicklungsphase nicht wie zunächst geplant 35 Stunden pro Woche sondern circa 60 bis 70 Stunden pro Woche in das Projekt investiert. Glücklicherweise war dies durch das kulante Entgegenkommen des Arbeitgebers des Entwicklers möglich. Es ist nicht abzustreiten, dass sich der Autor bezüglich des Umfangs der Arbeit stark übernommen hat. Das im Exposé unter Punkt *7 Mögliche Risiken* bereits angesprochene Risiko der *Falsche Einschätzung der Komplexität* trat also wirklich ein, es führte jedoch nicht zum Projektabbruch. Unter diesen Umständen erscheint das resultierende Ergebnis dem Autor als noch zufriedenstellender. Dieses besteht aus zwei Anwendungen, welche sich einfach starten lassen und zusammen als simpler Debugger für das Beispielprojekt eingesetzt werden können. Damit wurde nach Ansicht des Autors die Zielsetzung im vollen Umfang erfüllt.

6.1 Ausblick

Die aus der Bachelorarbeit entstandenen Softwarekomponenten könnten auf die unterschiedlichsten Arten weiter entwickelt werden, es folgt eine Auflistung, welche potenzielle Erweiterungen beinhaltet:

- **HTTP C++ Server:** Die aktuelle TCP Socket Schnittstelle könnte durch einen HTTP Schnittstelle abgelöst werden, da die aktuelle Socket Implementierung des *UVRD*s nur genau mit einem Serversocket verbunden sein kann. Hierfür könnte ggf. auf das bereits erwähnte kostenpflichtige *Unreal Web Server*-Plugin zurückgegriffen werden.
- **SourceCodeService:** Damit der *UVRD* auch zum Debuggen von anderen Java-Anwendungen genutzt werden kann, muss die aktuelle `SourceCodeService` Mock-Implementierung durch eine wirkliche Implementierung der enthaltenen Funktionalitäten ersetzt werden.
- **Konzeption & Entwicklung einer Virtual Reality User Interface:** Die wohl größte Herausforderung bei der Realisierung eines vollständigen Virtual Reality Debuggers liegt wohl in der Konzeption, sowie der Entwicklung einer für die Virtual Reality geeigneten Benutzeroberfläche.
- **Erweiterte Debugging-Funktionalitäten:** Die aktuelle Logik könnte um die in der Zielsetzung ausgeklammerten erweiterten Debugging-Funktionalitäten ergänzt werden, damit der *UVRD* als vollwertiger Debugger genutzt werden kann.

Verweise

Online-Verweise

[1] Epic Games Inc. *Blueprints Visual Scripting*.
Online abrufbar unter : https://docs.unrealengine.com/latest/INT/Engine/Blueprints/ [Stand: 11.03.2018].

[2] Rama und Zyr. *File Management, Create Folders, Delete Files, and More*.
Online abrufbar unter : https://wiki.unrealengine.com/File_Management,_Create_Folders,_Delete_Files,_and_More [Stand: 11.03.2018].

[3] Epic Games Inc. *Unreal Engine Marketplace*.
Online abrufbar unter : https://www.unrealengine.com/marketplace [Stand: 11.03.2018].

[4] Sameek Kundu. *MySQL Integration*. 2017
Online abrufbar unter : https://unrealengine.com/marketplace/mysql-integration [Stand: 11.03.2018].

[5] Pivotal Software Inc. *RabbitMQ*.
Online abrufbar unter : https://www.rabbitmq.com/changelog.html [Stand: 11.03.2018].

[6] Pivotal Software Inc. *Mozilla Public License*.
Online abrufbar unter : https://www.rabbitmq.com/mpl.html [Stand: 11.03.2018].

[7] Pivotal Software Inc. *RabbitMQ devtools C / C++*.
Online abrufbar unter : https://www.rabbitmq.com/devtools.html#c-dev [Stand: 11.03.2018].

[8] Pivotal Software Inc. *RabbitMQ devtools Java and Spring*.
Online abrufbar unter : https://www.rabbitmq.com/devtools.html#java-dev [Stand: 11.03.2018].

[9] GotSomePills OzoneBG Marki217q. *UE4.10 How To Make HTTP GET Request in C++*.
Online abrufbar unter : https://wiki.unrealengine.com/UE4.10_How_To_Make_HTTP_GET_Request_in_C%2B%2B [Stand: 11.03.2018].

[10] Isara Tech. *Unreal Web Server*. 2017
Online abrufbar unter : https://www.unrealengine.com/marketplace/unreal-web-server [Stand: 11.03.2018].

[11] Rama. *TCP Socket Listener, Receive Binary Data From an IP/Port Into UE4, (Full Code Sample)*. Online abrufbar unter : https://wiki.unrealengine.com/TCP_Socket_Listener,_Receive_Binary_Data_From_an_IP/Port_Into_UE4,_(Full_Code_Sample) [Stand: 11.03.2018].

[12] neuland – Büro für Informatik GmbH. *neuland – Büro für Informatik GmbH*.
Online abrufbar unter : https://www.neuland-bfi.de/ [Stand: 29.01.2018].

[13] Pivotal Software Inc. *Spring Framework 5.0*.
Online abrufbar unter : https://spring.io/ [Stand: 11.03.2018].

[14] Oracle. *Java 8 API: java.net*.
Online abrufbar unter : https://docs.oracle.com/javase/8/docs/api/java/net/package-summary.html [Stand: 11.03.2018].

[15] Pivotal Software Inc. *Spring Boot*.
Online abrufbar unter : https://projects.spring.io/spring-boot/ [Stand: 11.03.2018].

[16] Apache Software Foundation. *Apache Tomcat®*.
Online abrufbar unter : http://tomcat.apache.org/ [Stand: 11.03.2018].

[17] Eclipse Foundation. *Eclipse Jetty*.
Online abrufbar unter : https://www.eclipse.org/jetty/ [Stand: 11.03.2018].

[18] Red Hat Inc. *Undertow*.
Online abrufbar unter : http://undertow.io/ [Stand: 11.03.2018].

[19] Oracle. *Java 8 API: Java™ Debug Interface*.
Online abrufbar unter : https://docs.oracle.com/javase/8/docs/jdk/api/jpda/jdi/ [Stand: 11.03.2018].

[20] Epic Games Inc. *Unreal Engine API: HTTP*.
Online abrufbar unter : https://docs.unrealengine.com/latest/INT/API/Runtime/HTTP/ [Stand: 11.03.2018].

[21] Epic Games Inc. *Unreal Engine API: Sockets*.
Online abrufbar unter : https://docs.unrealengine.com/latest/INT/API/Runtime/Sockets/ [Stand: 11.03.2018].

[22] Epic Games Inc. *Unreal Engine API: Networking*.
Online abrufbar unter : https://docs.unrealengine.com/latest/INT/API/Runtime/Networking [Stand: 11.03.2018].

[23] Google LLC. *Gson User Guide*.
Online abrufbar unter : https://github.com/google/gson/blob/master/UserGuide.md [Stand: 11.03.2018].

[24] Epic Games Inc. *Unreal Engine API: Json*.
Online abrufbar unter : https://docs.unrealengine.com/latest/INT/API/Runtime/Json/ [Stand: 11.03.2018].

[25] Epic Games Inc. *Unreal Engine API: JsonUtilities*.
Online abrufbar unter : https://docs.unrealengine.com/latest/INT/API/Runtime/JsonUtilities/ [Stand: 11.03.2018].

[26] Lightbend Inc. *Play Framework*.
Online abrufbar unter : https://www.playframework.com/ [Stand: 11.03.2018].

[27] FasterXML. *jackson*.
Online abrufbar unter : https://github.com/FasterXML/jackson [Stand: 11.03.2018].

[28] Oracle. *javadoc*.
Online abrufbar unter : https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html [Stand: 11.03.2018].

[29] Dimitri van Heesch. *Doxygen*.
Online abrufbar unter : http://www.stack.nl/~dimitri/doxygen/index.html [Stand: 11.03.2018].

[30] Pivotal Software Inc. *Spring Framework Documentation. 1.5. Bean scopes*.
Online abrufbar unter : https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-scopes [Stand: 11.03.2018].

[31] Pivotal Software Inc. *Spring Framework Documentation. 1.5.1. The singleton scope*.
Online abrufbar unter : https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-scopes-singleton [Stand: 11.03.2018].

[32] Pivotal Software Inc. *Part III. Using Spring Boot. 17. Spring Beans and dependency injection*.
Online abrufbar unter : https://docs.spring.io/spring-boot/docs/current/reference/html/using-boot-spring-beans-and-dependency-injection.html [Stand: 11.03.2018].

[33] Pivotal Software Inc. *Annotation Type Autowired*.
Online abrufbar unter : https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/beans/factory/annotation/Autowired.html [Stand: 11.03.2018].

[34] Apache Software Foundation. *Welcome to Apache Maven*.
Online abrufbar unter : https://maven.apache.org/ [Stand: 11.03.2018].

[35] JetBrains. *IntelliJ IDEA*.
Online abrufbar unter : https://www.jetbrains.com/idea/ [Stand: 11.03.2018].

[36] Szczepan Faber. *mockito: Tasty mocking framework for unit tests in Java*.
Online abrufbar unter : http://site.mockito.org/ [Stand: 11.03.2018].

[37] JUnit. *JUnit 4*.
Online abrufbar unter : https://junit.org/junit4/ [Stand: 11.03.2018].

[38] Epic Games Inc. *Unreal Engine API: UObject*.
Online abrufbar unter : https://docs.unrealengine.com/latest/INT/API/Runtime/CoreUObject/UObject/UObject/ [Stand: 11.03.2018].

[39] Epic Games Inc. *UMG UI Designer*.
Online abrufbar unter : https://docs.unrealengine.com/latest/INT/Engine/UMG/ [Stand: 11.03.2018].

[40] École polytechnique fédérale de Lausanne. *Scala Standard Library: scala.util.Try*.
Online abrufbar unter : https://www.scala-lang.org/api/2.12.3/scala/util/Try.html [Stand: 11.03.2018].

[41] Epic Games Inc. *First Person Shooter. 3.5 - Adding Crosshairs to your Viewport*. Online abrufbar unter : https://docs.unrealengine.com/latest/INT/Engine/UMG/ [Stand: 11.03.2018].

[42] Oracle. *Java 8 API: MethodEntryEvent*. Online abrufbar unter : https://docs.oracle.com/javase/8/docs/jdk/api/jpda/jdi/com/sun/jdi/event/MethodEntryEvent.html [Stand: 11.03.2018].

[43] Oracle. *Java 8 API: MethodExitEvent*. Online abrufbar unter : https://docs.oracle.com/javase/8/docs/jdk/api/jpda/jdi/com/sun/jdi/event/MethodExitEvent.html [Stand: 11.03.2018].

[44] Oracle. *Java 8 API: MethodEntryRequest*. Online abrufbar unter : https://docs.oracle.com/javase/8/docs/jdk/api/jpda/jdi/com/sun/jdi/request/MethodEntryRequest.html [Stand: 11.03.2018].

[45] Oracle. *Java 8 API: MethodExitRequest*. Online abrufbar unter : https://docs.oracle.com/javase/8/docs/jdk/api/jpda/jdi/com/sun/jdi/request/MethodExitRequest.html [Stand: 11.03.2018].

[46] Oracle. *Java 8 API: MethodExitRequest#addClassFilter(java.lang.String)*. Online abrufbar unter : https://docs.oracle.com/javase/8/docs/jdk/api/jpda/jdi/com/sun/jdi/request/MethodExitRequest.html#addClassFilter-java.lang.String- [Stand: 11.03.2018].

[47] Oracle. *Java 8 API: ClassPrepareEvent*. Online abrufbar unter : https://docs.oracle.com/javase/8/docs/jdk/api/jpda/jdi/com/sun/jdi/event/ClassPrepareEvent.html [Stand: 11.03.2018].

[48] Oracle. *Java 8 API: EventQueue*. Online abrufbar unter : https://docs.oracle.com/javase/8/docs/jdk/api/jpda/jdi/com/sun/jdi/event/EventQueue.html [Stand: 11.03.2018].

[49] Oracle. *Java 8 API: VirtualMachine*. Online abrufbar unter : https://docs.oracle.com/javase/8/docs/jdk/api/jpda/jdi/com/sun/jdi/VirtualMachine.html [Stand: 11.03.2018].

[50] Oracle. *Java 8 API: VMDeathEvent*. Online abrufbar unter : https://docs.oracle.com/javase/8/docs/jdk/api/jpda/jdi/com/sun/jdi/event/VMDeathEvent.html [Stand: 11.03.2018].

[51] Oracle. *Java 8 API: VMDisconnectEvent*. Online abrufbar unter : https://docs.oracle.com/javase/8/docs/jdk/api/jpda/jdi/com/sun/jdi/event/VMDisconnectEvent.html [Stand: 11.03.2018].

[52] Oracle. *Java 8 API: BreakpointRequest*. Online abrufbar unter : https://docs.oracle.com/javase/8/docs/jdk/api/jpda/jdi/com/sun/jdi/request/BreakpointRequest.html [Stand: 11.03.2018].

[53] Oracle. *Java 8 API: ModificationWatchpointRequest*. Online abrufbar unter : https://docs.oracle.com/javase/8/docs/jdk/api/jpda/jdi/com/sun/jdi/request/ModificationWatchpointRequest.html [Stand: 11.03.2018].

[54] Oracle. *Java 8 API: ModificationWatchpointEvent*.
Online abrufbar unter : https://docs.oracle.com/javase/8/docs/jdk/api/jpda/jdi/com/sun/jdi/event/ModificationWatchpointEvent.html [Stand: 11.03.2018].

[55] Oracle. *Java 8 API: BreakpointEvent*.
Online abrufbar unter : https://docs.oracle.com/javase/8/docs/jdk/api/jpda/jdi/com/sun/jdi/event/BreakpointEvent.html [Stand: 11.03.2018].

[56] Pivotal Software Inc. *Spring Framework Documentation. 1. Spring Web MVC*.
Online abrufbar unter : https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html#mvc [Stand: 11.03.2018].

[57] Pivotal Software Inc. *Spring Framework API: Annotation Type RestController*.
Online abrufbar unter : https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/bind/annotation/RestController.html [Stand: 11.03.2018].

[58] Oracle. *Java 8 API: Annotation Type PostConstruct*.
Online abrufbar unter : https://docs.oracle.com/javase/8/docs/api/javax/annotation/PostConstruct.html [Stand: 11.03.2018].

[59] Oracle. *Java 8 API: ServerSocket*.
Online abrufbar unter : https://docs.oracle.com/javase/8/docs/api/java/net/ServerSocket.html [Stand: 11.03.2018].

[60] Oracle. *Java 8 API: Annotation Type PreDestroy*.
Online abrufbar unter : https://docs.oracle.com/javase/8/docs/api/javax/annotation/PreDestroy.html [Stand: 11.03.2018].

[61] Epic Games Inc. *Unreal Engine API: FSocket*.
Online abrufbar unter : https://docs.unrealengine.com/latest/INT/API/Runtime/Sockets/FSocket/ [Stand: 11.03.2018].

[62] Epic Games Inc. *Unreal Engine API: FTimerHandler*.
Online abrufbar unter : https://docs.unrealengine.com/latest/INT/API/Runtime/Engine/Engine/FTimerHandle/ [Stand: 11.03.2018].

[63] Epic Games Inc. *Unreal Engine API: FString*.
Online abrufbar unter : https://docs.unrealengine.com/latest/INT/Programming/UnrealArchitecture/StringHandling/FString/ [Stand: 11.03.2018].

[64] Oracle. *javac - Java programming language compiler*.
Online abrufbar unter : https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javac.html [Stand: 11.03.2018].

[65] Apache Software Foundation. *Welcome to Apache Maven*.
Online abrufbar unter : https://maven.apache.org/plugins/maven-compiler-plugin/examples/pass-compiler-arguments.html [Stand: 11.03.2018].

[66] Microsoft. *Visual Studio-IDE*.
Online abrufbar unter : https://www.visualstudio.com/de/vs/ [Stand: 11.03.2018].

[67] Sublime HQ. *Sublime Text 2*.
Online abrufbar unter : https://www.sublimetext.com/2 [Stand: 11.03.2018].

[68] Microsoft. *Visio 2013*.
Online abrufbar unter : https://products.office.com/de-de/microsoft-visio-2013 [Stand: 11.03.2018].

[69] TeX Users Group. *TeX Live*.
Online abrufbar unter : https://www.tug.org/texlive/ [Stand: 11.03.2018].

[70] Pascal Brachet. *Texmaker*.
Online abrufbar unter : http://www.xm1math.net/texmaker/ [Stand: 11.03.2018].

[71] Akshathkumar Shetty. *SocketTest v3.0.0*.
Online abrufbar unter : sockettest.sourceforge.net [Stand: 11.03.2018].

[72] Postdot Technologies. *Postman Makes API Development Simple*.
Online abrufbar unter : https://www.getpostman.com/ [Stand: 11.03.2018].

[73] Epic Games Inc. *What is unreal engine 4?*
Online abrufbar unter : https://www.unrealengine.com/en-US/what-is-unreal-engine-4 [Stand: 11.03.2018].

Literaturverzeichnis

[BEP18] Joachim H. Becker, Helmut Ebert, and Sven Pastoors. *Praxishandbuch berufliche Schlüsselkompetenzen : 50 Handlungskompetenzen für Ausbildung, Studium und Beruf*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2018, pp. 100–101. ISBN: 9783662549254. Online abrufbar unter : https://link.springer.com/book/10.1007%2F978-3-662-54925-4 [Stand: 27.01.2018].

[Pat14a] Jeff Patton. *User Story Mapping: Discover the Whole Story, Build the Right Product*. O'Reilly Media, 2014, p. 3. ISBN: 1491904909.

[Pat14b] Jeff Patton. *User Story Mapping: Discover the Whole Story, Build the Right Product*. O'Reilly Media, 2014, p. 99. ISBN: 1491904909.

[Gri14a] Miguel Grinberg. *Flask Web Development: Developing Web Applications with Python*. O'Reilly Media, 2014, pp. 175–176. ISBN: 978-1-449-37262-0.

[Gri14b] Miguel Grinberg. *Flask Web Development: Developing Web Applications with Python*. O'Reilly Media, 2014, p. 177. ISBN: 978-1-449-37262-0.

[Leo15] Fernando Puente Leon. *Messtechnik: Systemtheorie für Ingenieure und Informatiker*. 10. Auflage. Berlin: Springer Vieweg, 2015, pp. 17–18. ISBN: 9783662448212.

Abbildungsverzeichnis

Tabellenverzeichnis

A - Anhang

A.1 Technische Subtasks der User Stories

US1 - Debugging Session: Verbindung herstellen

- *Debugger Backend:* Erstellung der initialen Spring Boot Apllication
- *UVRD:* Erstellung des initialen Unreal Engine Projektes
- *Schnittstellendesign:* HTTP - „Create Debugging Session“
- *UVRD:* Front-End - Eingabe einer Port Nummer
- *UVRD:* Front-End - „Connect“ Button
- *UVRD:* Back-End - Absenden von „Create Debugging Session“-Requests
- *UVRD:* Back-End - Verarbeitung von „Create Debugging Session“-Responses
- *Debugger Backend:* Debugging Session Controller - „Create Debugging Session“
- *Debugger Backend:* JDI - Verbindungsaufbau

US3 - Debugging Session: Starten

- *Schnittstellendesign:* HTTP - „Start Debugging Session“
- *UVRD:* Front-End - „Start“ Button
- *UVRD:* Back-End - Absenden von „Start Debugging Session“-Requests
- *UVRD:* Back-End - Verarbeitung von „Start Debugging Session“-Responses
- *Debugger Backend:* Debugging Session Controller - „Start Debugging Session“
- *Debugger Backend:* JDI - Start Debugging Process

US2 - Debugging Session: Verbindung trennen

- *Schnittstellendesign:* HTTP - „Delete Debugging Session“
- *UVRD:* Front-End - „Delete“ Button
- *UVRD:* Back-End - Absenden von „Delete Debugging Session“-Requests
- *UVRD:* Back-End - Verarbeitung von „Delete Debugging Session“-Responses
- *Debugger Backend:* Debugging Session Controller - „Delete Debugging Session“
- *Debugger Backend:* JDI - Exit Debugging Process

US6 - Watcher: Anzeige aller potenzieller Watcher

- *Schnittstellendesign:* HTTP - „Potential Watcher“
- *UVRD:* Front-End - „Add Watcher“ Button
- *UVRD:* Back-End - Absenden von „Potential Watcher“-Requests
- *UVRD:* Back-End - Verarbeitung von „Potential Watcher“-Responses, bei einer erfolgreichen Verarbeitung soll ein „Add Watcher“ Overlay geöffnet werden, die Ausimplementierung dieses Overlays ist jedoch nicht Bestandteil dieser Story (siehe US7)
- *Debugger Backend:* Watcher Controller - „Potential Watcher“
- *Debugger Backend:* Source Code Mock - Potential Watcher

US7 - Watcher: Auswahl eines potenziellen Watchers

- *UVRD:* Front-End - „Add Watcher“ Overlay

US8 - Watcher: Erstellung eines Watchers

- *Schnittstellendesign:* HTTP - „Add Watcher“
- *UVRD:* Back-End - Absenden von „Add Watcher“-Requests
- *UVRD:* Back-End - Verarbeitung von „Add Watcher“-Responses
- *Debugger Backend:* Watcher Controller - „Add Watcher“
- *Debugger Backend:* JDI - Create Watcher

US4 - Debugging Session: Pausierung

- *Debugger Backend:* JDI - Pausierung erkennen
- *Debugger Backend:* JDI - Manuelles pausieren ermöglichen
- *Debugger Backend:* Rudimentäre Server Socket Implementierung
- *UVRD:* Rudimentäre Client Socket Implementierung
- *Schnittstellendesign:* Socket Message - „Pause Debugging Session“
- *Debugger Backend:* Server Socket - Senden von „Pause Debugging Session“ Socket Messages
- *UVRD:* Client Socket - Verarbeiten von „Pause Debugging Session“ Socket Messages

US11 - Watcher: Mitteilung der Änderungsinformationen

- *Debugger Backend:* JDI - Watcher Änderungen erkennen
- *Schnittstellendesign:* Socket Message - „Watcher Changed“
- *Debugger Backend:* Server Socket - Senden von „Watcher Changed“ Socket Messages
- *UVRD:* Client Socket - Verarbeiten von „Watcher Changed“ Socket Messages

US5 - Debugging Session: Fortsetzen

- *Schnittstellendesign:* HTTP - „Resume Debugging Session“
- *UVRD:* Front-End - „Resume“ Button
- *UVRD:* Back-End - Absenden von „Resume Debugging Session“-Requests
- *UVRD:* Back-End - Verarbeitung von „Resume Debugging Session“-Responses
- *Debugger Backend:* Debugging Session Controller - „Resume Debugging Session“
- *Debugger Backend:* JDI - Resume Debugging Process

US9 - Watcher: Anzeige aller aktiven Watcher

- *Schnittstellendesign:* HTTP - „Active Watcher“
- *UVRD:* Front-End - „Active Watcher“ Button
- *UVRD:* Back-End - Absenden von „Active Watcher“-Requests
- *UVRD:* Back-End - Verarbeitung von „Active Watcher“-Responses, bei einer erfolgreichen Verarbeitung soll ein „Active Watcher“ Overlay geöffnet werden
- *UVRD:* Front-End - „Active Watcher“ Overlay
- *Debugger Backend:* Watcher Controller - „Active Watcher“
- *Debugger Backend:* JDI - Find Active Watcher

US10 - Watcher: Löschung eines aktiven Watchers

- *Schnittstellendesign:* HTTP - „Delete Watcher“
- *UVRD:* Front-End - „Delete Watcher“ Button im „Active Watcher“ Overlay
- *UVRD:* Back-End - Absenden von „Delete Watcher“-Requests
- *UVRD:* Back-End - Verarbeitung von „Delete Watcher“-Responses
- *Debugger Backend:* Watcher Controller - „Delete Watcher“
- *Debugger Backend:* JDI - Delete Watcher

US12 - Breakpoint: Anzeige der vorhandenen Klassen

- *Schnittstellendesign:* HTTP - „Potential Breakpoint Classes“
- *UVRD:* Front-End - „Add Breakpoint“ Button
- *UVRD:* Back-End - Absenden von „Potential Breakpoint Classes“-Requests
- *UVRD:* Back-End - Verarbeitung von „Potential Breakpoint Classes“-Responses, bei einer erfolgreichen Verarbeitung soll ein „Add Breakpoint“ Overlay geöffnet werden, dieses ist nicht Bestandteil dieser Story (siehe US13)
- *Debugger Backend:* Breakpoint Controller - „Potential Breakpoint Classes“
- *Debugger Backend:* Source Code Mock - Potential Breakpoint Classes

US13 - Breakpoint: Erstellung eines Breakpoints

- *UVRD:* Front-End - „Add Breakpoint“ Overlay
- *Schnittstellendesign:* HTTP - „Add Breakpoint“
- *UVRD:* Back-End - Absenden von „Add Breakpoint“-Requests
- *UVRD:* Back-End - Verarbeitung von „Add Breakpoint“-Responses
- *Debugger Backend:* Breakpoint Controller - „Breakpoint Watcher“
- *Debugger Backend:* JDI - Create Breakpoint

US14 - Breakpoint: Anzeige aller aktiven Breakpoints

- *Schnittstellendesign:* HTTP - „Active Breakpoints“
- *UVRD:* Front-End - „Active Breakpoints“ Button
- *UVRD:* Back-End - Absenden von „Active Breakpoints“-Requests
- *UVRD:* Back-End - Verarbeitung von „Active Breakpoints“-Responses, bei einer erfolgreichen Verarbeitung soll ein „Active Breakpoints“ Overlay geöffnet werden
- *UVRD:* Front-End - „Active Breakpoints“ Overlay
- *Debugger Backend:* Breakpoint Controller - „Active Breakpoints“
- *Debugger Backend:* JDI - Find Active Breakpoints

US15 - Breakpoint: Löschung eines Breakpoints

- *Schnittstellendesign:* HTTP - „Delete Breakpoint“
- *UVRD:* Front-End - „Delete Breakpoint“ Button im „Active Breakpoints“ Overlay
- *UVRD:* Back-End - Absenden von „Delete Breakpoint“-Requests
- *UVRD:* Back-End - Verarbeitung von „Delete Breakpoint“-Responses
- *Debugger Backend:* Breakpoint Controller - „Delete Breakpoint“
- *Debugger Backend:* JDI - Delete Breakpoint

US16 - Breakpoint: Mitteilung Breakpoint erreicht

- *Debugger Backend:* JDI - Erreichung eines Breakpoints erkennen
- *Schnittstellendesign:* Socket Message - „Breakpoint Reached“
- *Debugger Backend:* Server Socket - Senden von „Breakpoint Reached“ Socket Messages
- *UVRD:* Client Socket - Verarbeiten von „Breakpoint Reached“ Socket Messages

US17 - Live Stack Daten: Aktivierung

- *Schnittstellendesign:* HTTP - „Trace Live Stack“
- *UVRD:* Front-End - „Trace Live Stack“ Button
- *UVRD:* Back-End - Absenden von „Trace Live Stack“-Requests
- *UVRD:* Back-End - Verarbeitung von „Trace Live Stack“-Responses
- *Debugger Backend:* Trace Controller - „Trace Live Stack“
- *Debugger Backend:* JDI - Trace Live Stack

US18 - Live Stack Daten: Mitteilung

- *Debugger Backend:* JDI - Live Stack Daten ermitteln
- *Schnittstellendesign:* Socket Message - „Live Stack Data“
- *Debugger Backend:* Server Socket - Senden von „Live Stack Data“ Socket Messages
- *UVRD:* Client Socket - Verarbeiten von „Live Stack Data“ Socket Messages

US19 - Live Stack Daten: Visualisierung

- *UVRD:* Front-End - Einfache Visualisierung der Live Stack Daten

A.2 Meilenstein Diagramm

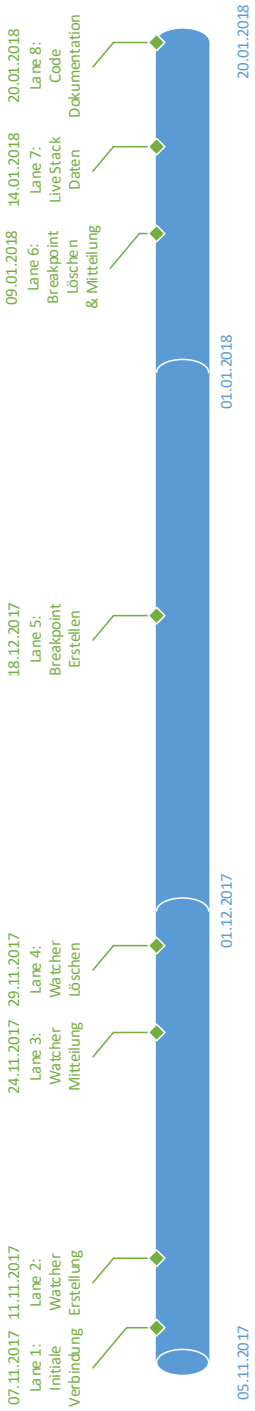

Abbildung 22: Meilenstein Diagramm

A.3 Abhängigkeitsskizze: Debugger Backend

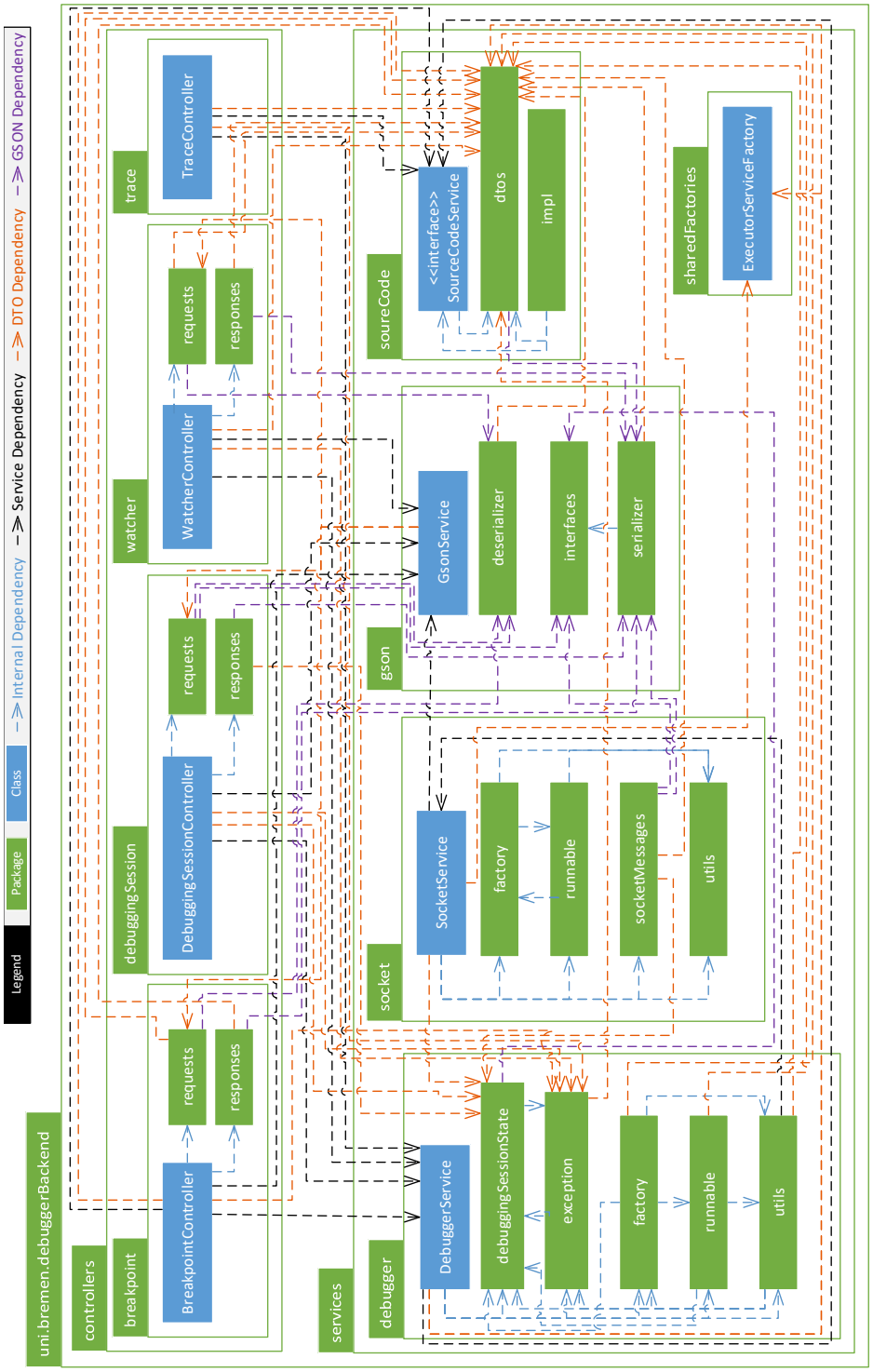

Abbildung 23: Abhängigkeitsskizze: Debugger Backend

A.4 Debugger Backend: pom.xml

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/
     XMLSchema-instance"
3    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven
       -4.0.0.xsd">
4    <modelVersion>4.0.0</modelVersion>
5    <groupId>uni.bremen</groupId>
6    <artifactId>debuggerBackend</artifactId>
7    <version>0.0.1-SNAPSHOT</version>
8    <packaging>jar</packaging>
9    <name>debuggerBackend</name>
10   <description>debugger backend powered by Spring Boot</description>
11   <parent>
12     <groupId>org.springframework.boot</groupId>
13     <artifactId>spring-boot-starter-parent</artifactId>
14     <version>1.5.8.RELEASE</version>
15     <relativePath/>
16   </parent>
17   <properties>
18     <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
19     <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
20     <java.version>1.8</java.version>
21   </properties>
22   <dependencies>
23     <dependency>
24       <groupId>org.springframework.boot</groupId>
25       <artifactId>spring-boot-starter-web</artifactId>
26     </dependency>
27     <dependency>
28       <groupId>org.springframework.boot</groupId>
29       <artifactId>spring-boot-starter-test</artifactId>
30       <scope>test</scope>
31     </dependency>
32     <dependency>
33       <groupId>com.google.code.gson</groupId>
34       <artifactId>gson</artifactId>
35       <version>2.8.2</version>
36     </dependency>
37     <dependency>
38       <groupId>org.mockito</groupId>
39       <artifactId>mockito-core</artifactId>
40       <version>2.12.0</version>
41       <scope>test</scope>
42     </dependency>
43     <dependency>
44       <groupId>org.apache.commons</groupId>
45       <artifactId>commons-lang3</artifactId>
46       <version>3.7</version>
47     </dependency>
48     <dependency>
49       <groupId>com.sun</groupId>
50       <artifactId>tools</artifactId>
51       <version>1.0</version>
52     </dependency>
53   </dependencies>
54   <build>
55     <plugins>
56       <plugin>
57         <groupId>org.apache.maven.plugins</groupId>
58         <artifactId>maven-install-plugin</artifactId>
```

```
59          <executions>
60            <execution>
61              <id>hack-binary</id>
62              <phase>validate</phase>
63              <configuration>
64                <file>${java.home}/../lib/tools.jar</file>
65                <repositoryLayout>default</repositoryLayout>
66                <groupId>com.sun</groupId>
67                <artifactId>tools</artifactId>
68                <version>1.0</version>
69                <packaging>jar</packaging>
70                <generatePom>true</generatePom>
71              </configuration>
72              <goals>
73                <goal>install-file</goal>
74              </goals>
75            </execution>
76          </executions>
77        </plugin>
78        <plugin>
79          <groupId>org.springframework.boot</groupId>
80          <artifactId>spring-boot-maven-plugin</artifactId>
81        </plugin>
82      </plugins>
83    </build>
84  </project>
```

Abbildung 24: Debugger Backend: pom.xml

A.5 Abhängigkeitsskizze: UVRD

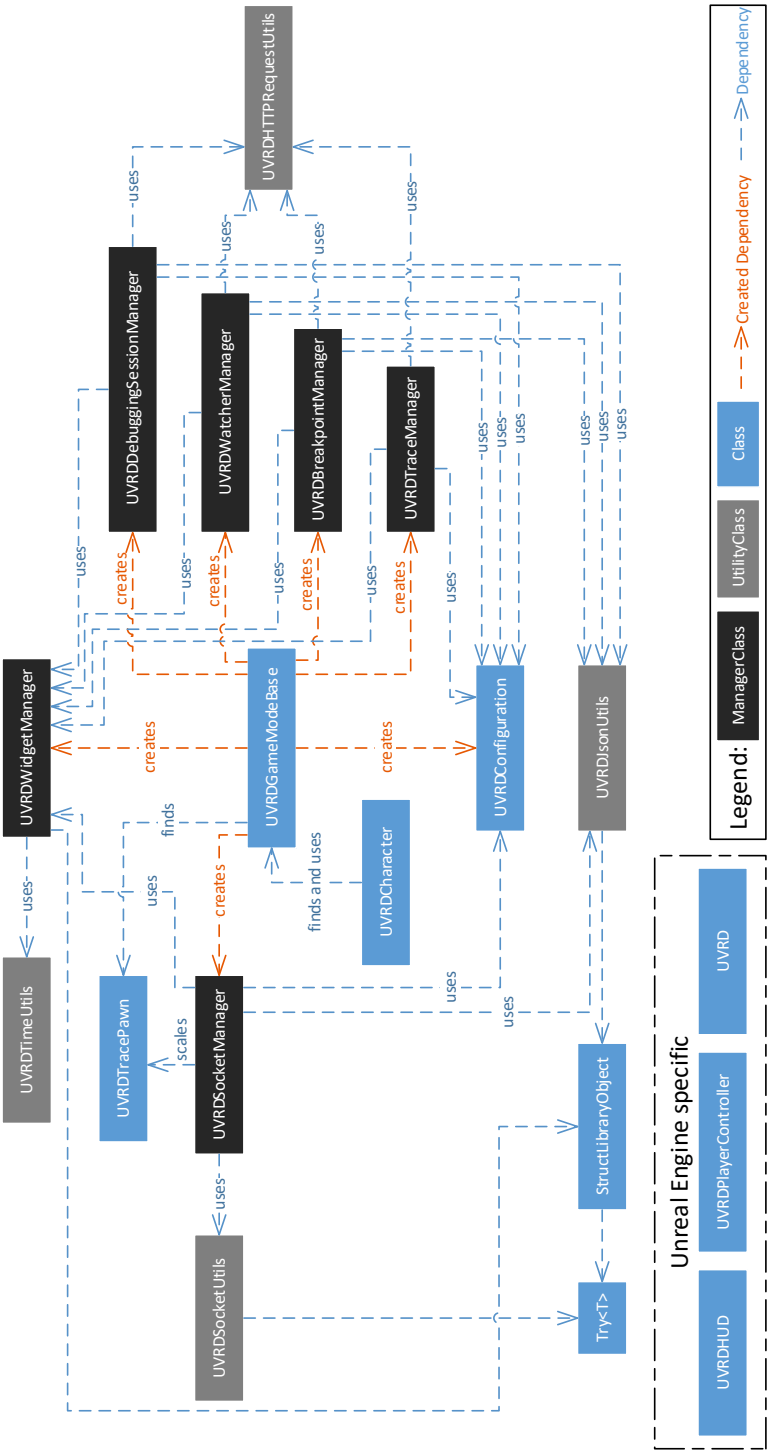

Abbildung 25: Abhängigkeitsskizze: UVRD

A.6 UVRD: UVRD.Build.cs

```
1  using UnrealBuildTool;
2
3  /** The UVRD module rules */
4  public class UVRD : ModuleRules
5  {
6
7      /**
8       * Dependencies:
9       * - UMG => User interface
10      * - Json / JsonUtilities => JSON handling
11      * - Http => http request
12      * - Sockets / Networking => socket communication
13      */
14     public UVRD(ReadOnlyTargetRules Target) : base(Target)
15   {
16     PCHUsage = PCHUsageMode.UseExplicitOrSharedPCHs;
17
18     PublicDependencyModuleNames.AddRange(new string[] {
19             "Core", "CoreUObject", "Engine", "InputCore",
20             "UMG",
21             "Json", "JsonUtilities",
22             "Http",
23             "Sockets","Networking"
24         });
25   }
26 }
```

Abbildung 26: UVRD: UVRD.Build.cs

A.7 Blueprint Auszüge

A.7.1 Blueprint Auszug: BP_UVRDHUD

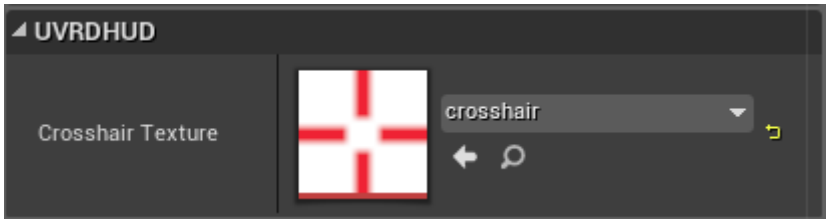

Abbildung 27: Blueprint Auszug: BP_UVRDHUD

A.7.2 Blueprint Auszug: BP_UVRDPlayerController

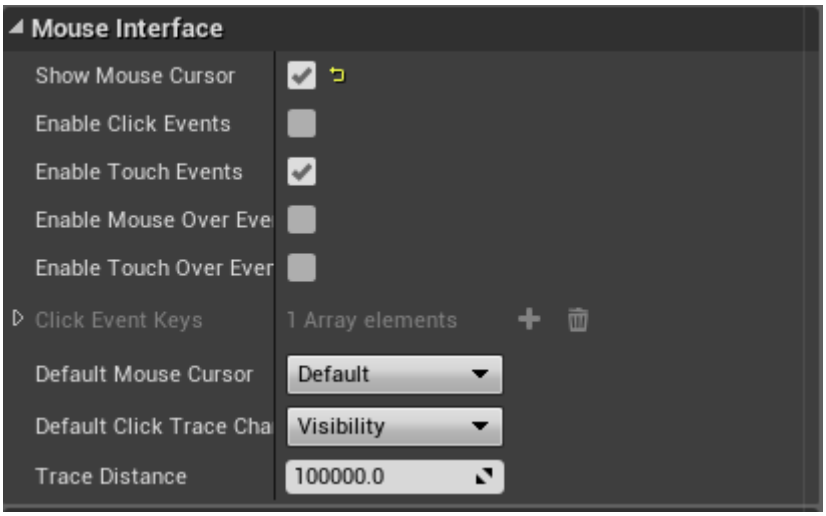

Abbildung 28: Blueprint Auszug: BP_UVRDPlayerController

A.7.3 Blueprint Auszug: BP_UVRDTracePawn

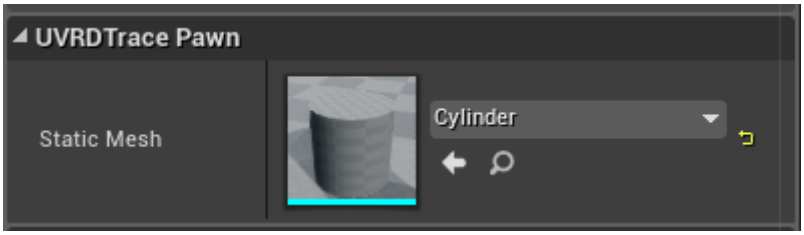

Abbildung 29: Blueprint Auszug: BP_UVRDTracePawn

A.7.4 Blueprint Auszug: BP_UVRDWidgetManager

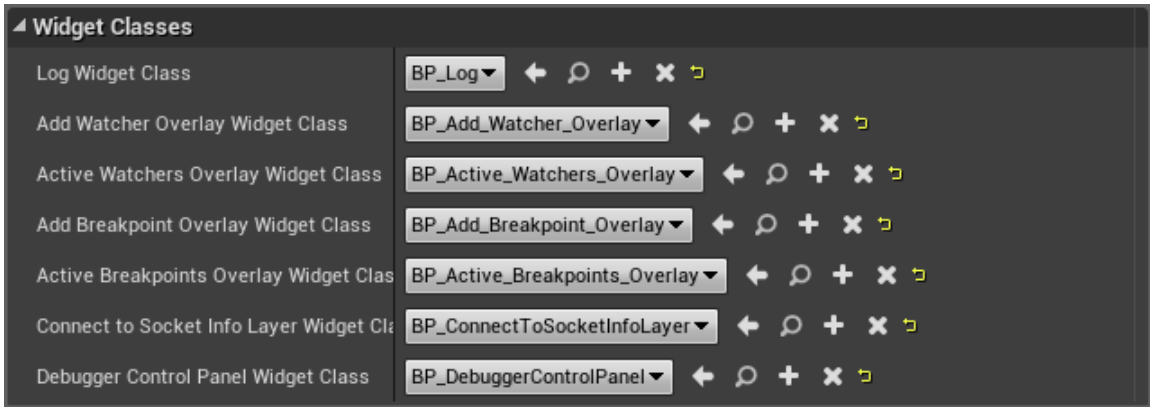

Abbildung 30: Blueprint Auszug: BP_UVRDWidgetManager

A.8 HTTP Kommunikation: Schnittstellenbeschreibung

A.8.1 `/debuggingSession`

A.8.1.1 `GET` - request current debugging session state

Request:

Resource: `/debuggingSession`
Method: `GET`
Payload: -

Response:

HTTP-Statuscode: 200
Payload: A.9.9 DebuggingSessionSuccessResponse

A.8.1.2 `POST` - create a new debugging session

Request:

Resource: `/debuggingSession`
Method: `POST`
Payload: A.9.6 CreateNewDebuggingSessionRequest

Success-Response:

HTTP-Statuscode: 200
Payload: A.9.9 DebuggingSessionSuccessResponse

Bad-Request-Response:

HTTP-Statuscode: 400
Payload: A.9.8 DebuggingSessionErrorResponse

Error-Response:

HTTP-Statuscode: 500
Payload: A.9.8 DebuggingSessionErrorResponse

A.8.1.3 `PUT` - modify the current debugging session (start / resume)

Request:

Resource: `/debuggingSession`
Method: `PUT`
Payload: A.9.7 ModifyDebuggingSessionRequest

Success-Response:

HTTP-Statuscode: 200
Payload: A.9.9 DebuggingSessionSuccessResponse

Bad-Request-Response:

HTTP-Statuscode: 400
Payload: A.9.8 DebuggingSessionErrorResponse

Error-Response:
HTTP-Statuscode: 500
Payload: A.9.8 DebuggingSessionErrorResponse

A.8.1.4 `DELETE` - end (delete) the current debugging session

Request:
Resource: `/debuggingSession`
Method: `DELETE`
Payload: -
Response:
HTTP-Statuscode: 200
Payload: A.9.9 DebuggingSessionSuccessResponse

A.8.2 `/breakpoints/classNames`

A.8.2.1 `GET` - get a list of all known class names (mock data)

Request:
Resource: `/breakpoints/classNames`
Method: `GET`
Payload: -
Response:
HTTP-Statuscode: 200
Payload: A.9.5 BreakpointsClassNamesResponse

A.8.3 `/breakpoints/active`

A.8.3.1 `GET` - get a list of all known active breakpoints

Request:
Resource: `/breakpoints/active`
Method: `GET`
Payload: -
Response:
HTTP-Statuscode: 200
Payload: A.9.3 ActiveBreakpointsResponse

A.8.3.2 `POST` - add a new breakpoint

Request:

Resource: `/breakpoints/active`
Method: `POST`
Payload: A.9.1 AddBreakpointRequest

Success-Response:

HTTP-Statuscode: 200
Payload: -

Bad-Request-Response:

HTTP-Statuscode: 400
Payload: A.9.4 BreakpointErrorResponse

Error-Response:

HTTP-Statuscode: 500
Payload: A.9.4 BreakpointErrorResponse

A.8.3.3 `DELETE` - delete a specific breakpoint

Request:

Resource: `/breakpoints/active`
Method: `DELETE`
Payload: A.9.2 DeleteBreakpointRequest

Success-Response:

HTTP-Statuscode: 200
Payload: -

Bad-Request-Response:

HTTP-Statuscode: 400
Payload: A.9.4 BreakpointErrorResponse

Not-Found-Response:

HTTP-Statuscode: 404
Payload: A.9.4 BreakpointErrorResponse

Error-Response:

HTTP-Statuscode: 500
Payload: A.9.4 BreakpointErrorResponse

A.8.3.4 `DELETE` - delete all breakpoints

Request:

Resource: `/breakpoints/active`
Method: `DELETE`
Payload: -

Success-Response:

HTTP-Statuscode: 200
Payload: -

Error-Response:

HTTP-Statuscode: 500
Payload: A.9.4 BreakpointErrorResponse

A.8.4 `/watchers/available`

A.8.4.1 `GET` - find all available watcher

Request:

Resource: `/watchers/available`
Method: `GET`
Payload: -

Response:

HTTP-Statuscode: 200
Payload: A.9.13 AvailableWatchersResponse

Error-Response:

HTTP-Statuscode: 500
Payload: A.9.14 WatcherErrorResponse

A.8.5 `/watchers/active`

A.8.5.1 `GET` - find all active watcher

Request:

Resource: `/watcher/active`
Method: `GET`
Payload: -

Response:

HTTP-Statuscode: 200
Payload: A.9.12 ActiveWatchersResponse

A.8.5.2 `POST` - add a new watcher

Request:

Resource: `/watcher/active`
Method: `POST`
Payload: A.9.10 AddWatcherRequest

Success-Response:

HTTP-Statuscode: 200
Payload: -

Bad-Request-Response:

HTTP-Statuscode: 400
Payload: A.9.14 WatcherErrorResponse

Not-Found-Response:

HTTP-Statuscode: 404
Payload: A.9.14 WatcherErrorResponse

Error-Response:

HTTP-Statuscode: 500
Payload: A.9.14 WatcherErrorResponse

A.8.5.3 `DELETE` - delete a specific watcher

Request:

Resource: `/watcher/active`
Method: `DELETE`
Payload: A.9.11 DeleteWatcherRequest

Success-Response:

HTTP-Statuscode: 200
Payload: -

Bad-Request-Response:

HTTP-Statuscode: 400
Payload: A.9.14 WatcherErrorResponse

Not-Found-Response:

HTTP-Statuscode: 404
Payload: A.9.14 WatcherErrorResponse

Error-Response:

HTTP-Statuscode: 500
Payload: A.9.14 WatcherErrorResponse

A.8.5.4 `DELETE` - delete all watcher

Request:

Resource: `/watcher/active`
Method: `DELETE`
Payload: -

Success-Response:

HTTP-Statuscode: 200
Payload: -

Error-Response:

HTTP-Statuscode: 500
Payload: A.9.14 WatcherErrorResponse

A.8.6 `/trace/methodEntry`

A.8.6.1 `POST` - enable the method entry tracing for the root package

Request:

Resource: `/trace/methodEntry`
Method: `POST`
Payload: -

Success-Response:

HTTP-Statuscode: 200
Payload: -

Error-Response:

HTTP-Statuscode: 500
Payload: -

A.8.7 `/trace/methodExit`

A.8.7.1 `POST` - enable the method exit tracing for the root package

Request:

Resource: `/trace/methodExit`
Method: `POST`
Payload: -

Success-Response:

HTTP-Statuscode: 200
Payload: -

Error-Response:

HTTP-Statuscode: 500
Payload: -

A.9 JSON Definitionen

A.9.1 AddBreakpointRequest

```
{
  "fullClassName":String,
  "lineNumber":Number
}
```

Abbildung 31: JSON Definition: AddBreakpointRequest

A.9.2 DeleteBreakpointRequest

```
{
  "fullClassName":String,
  "lineNumber":Number
}
```

Abbildung 32: JSON Definition: DeleteBreakpointRequest

A.9.3 ActiveBreakpointsResponse

```
{
  "breakpointFileInformations":[
    {
      "fullClassName":String,
      "lineNumbers":[
        Number, ...
      ]
    }, ...
  ]
}
```

Abbildung 33: JSON Definition: ActiveBreakpointsResponse

A.9.4 BreakpointErrorResponse

```
{
  "errorMessage":String
}
```

Abbildung 34: JSON Definition: BreakpointErrorResponse

A.9.5 BreakpointsClassNamesResponse

```
{
  "fullClassNames":[
    String, ...
  ]
}
```

Abbildung 35: JSON Definition: BreakpointsClassNamesResponse

A.9.6 CreateNewDebuggingSessionRequest

```
{
  "port":Number
}
```

Abbildung 36: JSON Definition: CreateNewDebuggingSessionRequest

A.9.7 ModifyDebuggingSessionRequest

```
{
  "modificationType":Number // 0 = START, 1 = RESUME
}
```

Abbildung 37: JSON Definition: ModifyDebuggingSessionRequest

A.9.8 DebuggingSessionErrorResponse

```
{
  "debuggingSessionState": Number, // 0 = NOT_CONNECTED,
                    // 1 = CONNECTED_AND_WAIT_FOR_START,
                    // 2 = CONNECTED_AND_RUNNING,
                    // 3 = CONNECTED_AND_WAIT_FOR_RESUME,
  "errorMessage":String
}
```

Abbildung 38: JSON Definition: DebuggingSessionErrorResponse

A.9.9 DebuggingSessionSuccessResponse

```
{
  "debuggingSessionState": Number // 0 = NOT_CONNECTED,
                   // 1 = CONNECTED_AND_WAIT_FOR_START,
                   // 2 = CONNECTED_AND_RUNNING,
                   // 3 = CONNECTED_AND_WAIT_FOR_RESUME,
}
```

Abbildung 39: JSON Definition: DebuggingSessionSuccessResponse

A.9.10 AddWatcherRequest

```
{
  "fullClassName":String,
  "fieldName":String
}
```

Abbildung 40: JSON Definition: AddWatcherRequest

A.9.11 DeleteWatcherRequest

```
{
  "fullClassName":String,
  "fieldName":String
}
```

Abbildung 41: JSON Definition: DeleteWatcherRequest

A.9.12 ActiveWatchersResponse

```
{
  "watcherFileInformations":[
    {
      "fullClassName":String,
      "fieldNames":[
        String, ...
      ]
    }, ...
  ]
}
```

Abbildung 42: JSON Definition: ActiveWatchersResponse

A.9.13 AvailableWatchersResponse

```
{
  "watcherFileInformations":[
    {
      "fullClassName":String,
      "fieldNames":[
        String, ...
      ]
    }, ...
  ]
}
```

Abbildung 43: JSON Definition: AvailableWatchersResponse

A.9.14 WatcherErrorResponse

```
{
  "errorMessage":String
}
```

Abbildung 44: JSON Definition: WatcherErrorResponse

A.9.15 SocketMessage

```
{
  "messageType":Number, // 0 = PAUSE_DEBUGGING_SESSION
              // 1 = END_DEBUGGING_SESSION
              // 2 = METHOD_ENTRY
              // 3 = METHOD_EXIT
              // 4 = WATCHER_CHANGED
              // 5 = BREAKPOINT
  "messageJson":Object
}
```

Abbildung 45: JSON Definition: SocketMessage

A.9.16 PauseDebuggingSessionSocketMessageJson

```
{
  "endMessage":String,
  "debuggingSessionState":Number // 0 = NOT_CONNECTED,
                    // 1 = CONNECTED_AND_WAIT_FOR_START,
                    // 2 = CONNECTED_AND_RUNNING,
                    // 3 = CONNECTED_AND_WAIT_FOR_RESUME,
}
```

Abbildung 46: JSON Definition: PauseDebuggingSessionSocketMessageJson

A.9.17 EndDebuggingSessionSocketMessageJson

```
{
  "endMessage":String,
  "debuggingSessionState":Number // 0 = NOT_CONNECTED,
                    // 1 = CONNECTED_AND_WAIT_FOR_START,
                    // 2 = CONNECTED_AND_RUNNING,
                    // 3 = CONNECTED_AND_WAIT_FOR_RESUME,
}
```

Abbildung 47: JSON Definition: EndDebuggingSessionSocketMessageJson

A.9.18 MethodEntrySocketMessageJson

```
{
  "orderedCallStack":[
    {
      "fullClassName":String,
      "methodName":String,
      "lineNumber":Number
    }, ...
  ]
}
```

Abbildung 48: JSON Definition: MethodEntrySocketMessageJson

A.9.19 MethodExitSocketMessageJson

```
{
  "orderedCallStack":[
    {
      "fullClassName":String,
      "methodName":String,
      "lineNumber":Number
    }, ...
  ]
}
```

Abbildung 49: JSON Definition: MethodExitSocketMessageJson

A.9.20 WatcherChangedSocketMessageJson

```
{
  "fullClassName":String,
  "fieldName":String,
  "oldValue":String,
  "newValue":String,
  "orderedCallStack":[
    {
      "fullClassName":String,
      "methodName":String,
      "lineNumber":Number
    }, ...
  ]
}
```

Abbildung 50: JSON Definition: WatcherChangedSocketMessageJson

A.9.21 BreakpointSocketMessageJson

```
{
  "fullClassName":String,
  "lineNumber":Number,
  "orderedCallStack":[
    {
      "fullClassName":String,
      "methodName":String,
      "lineNumber":Number
    }, ...
  ]
}
```

Abbildung 51: JSON Definition: BreakpointSocketMessageJson

A.10 Abhängigkeitsskizze: Debuger Backend - debugger

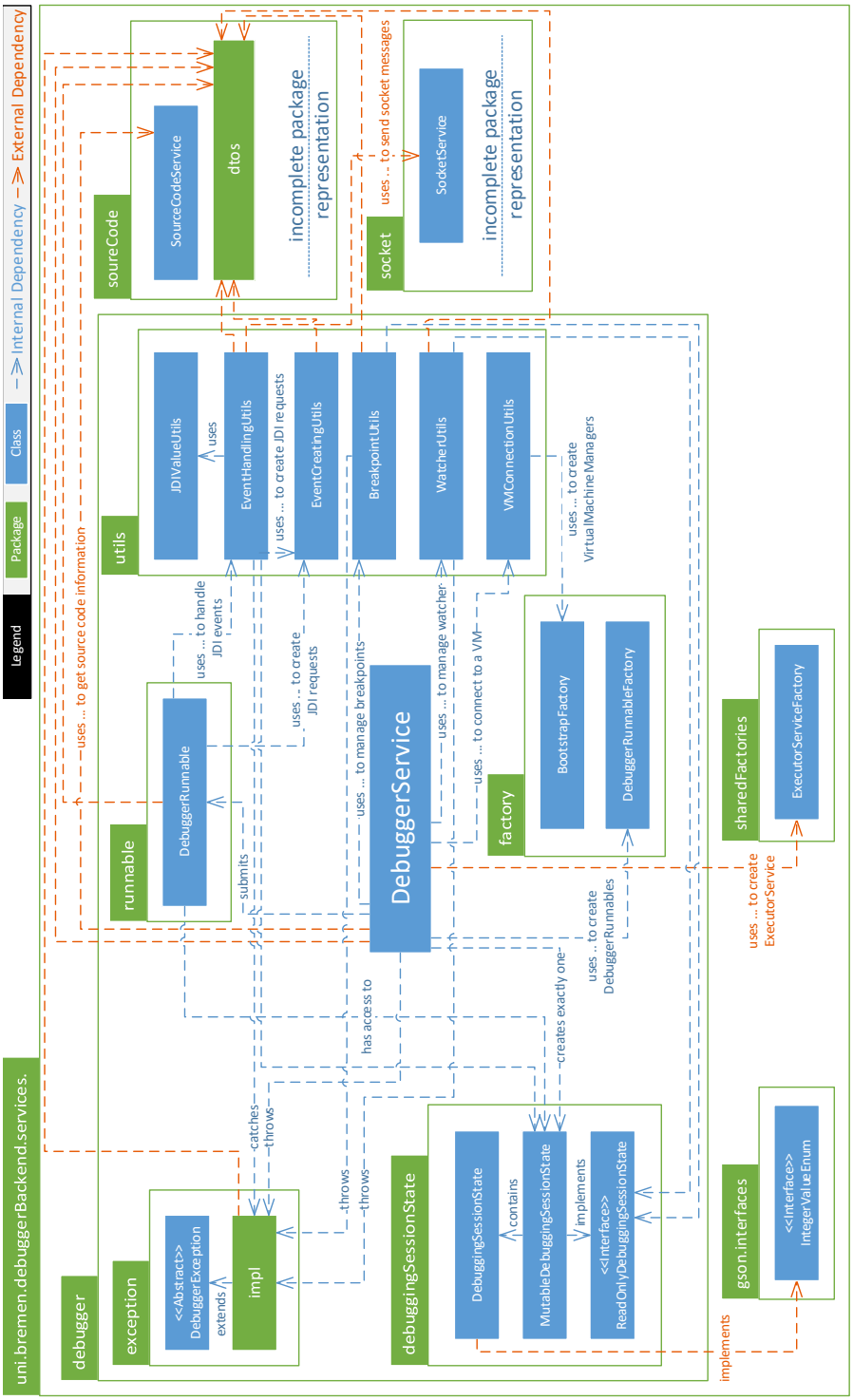

Abbildung 52: Abhängigkeitsskizze: Debuger Backend - debugger

A.11 Zustandsdiagramm: Debugger Session

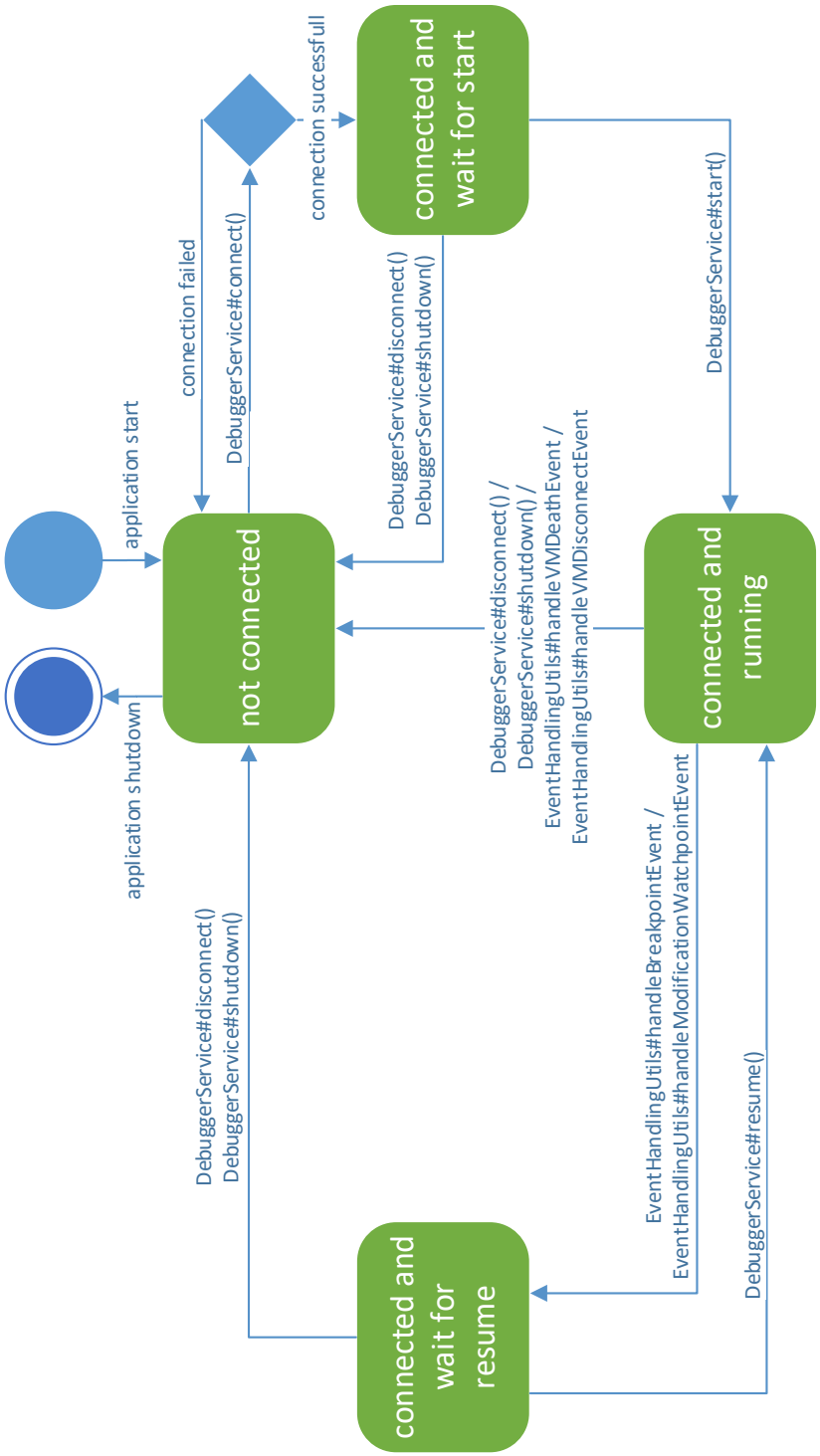

Abbildung 53: Zustandsdiagramm: Debugger Session

A.12 Abhängigkeitsskizze: Debuger Backend - Socket

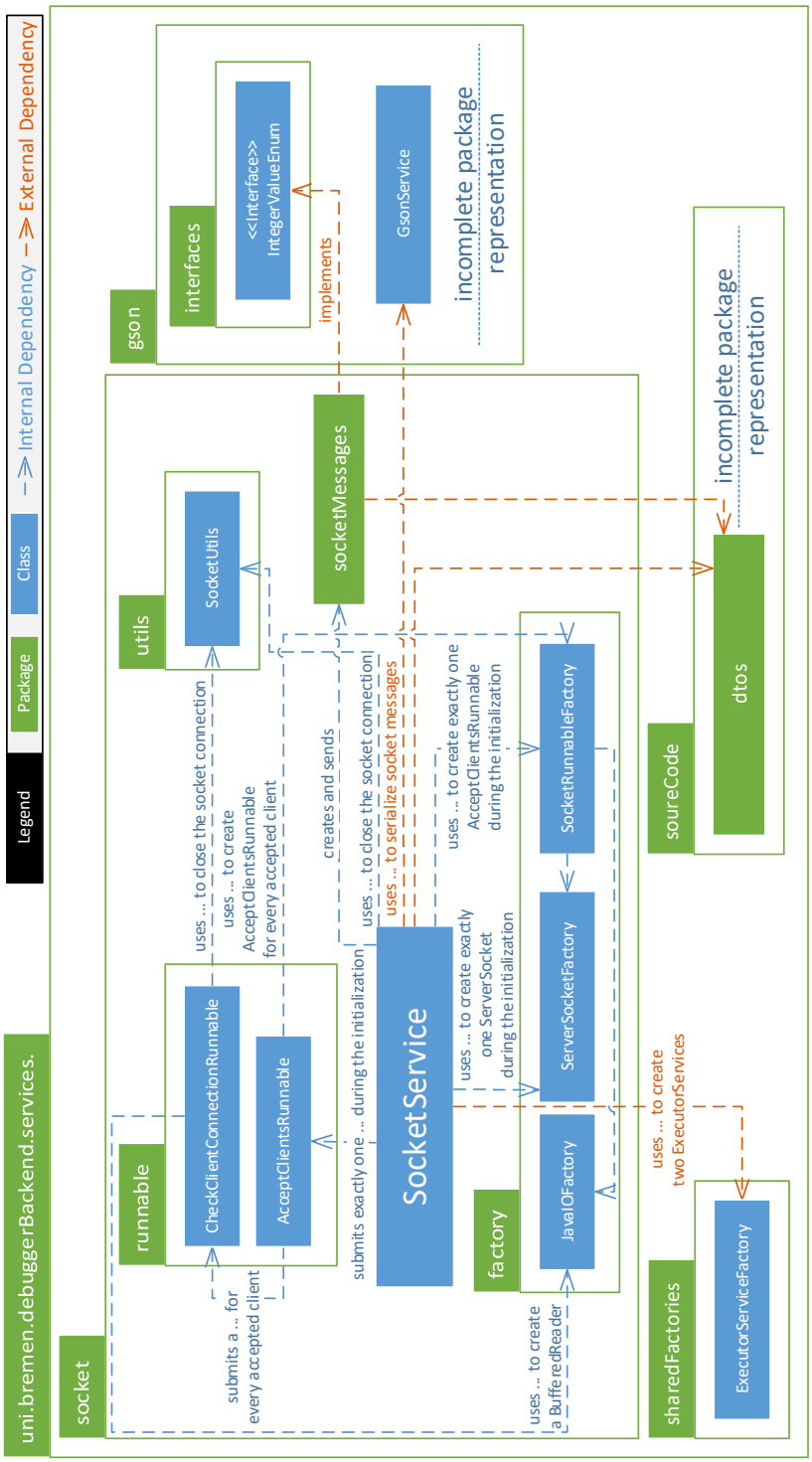

Abbildung 54: Abhängigkeitsskizze: Debuger Backend - Socket

A.13 Initiale Benutzeroberfläche

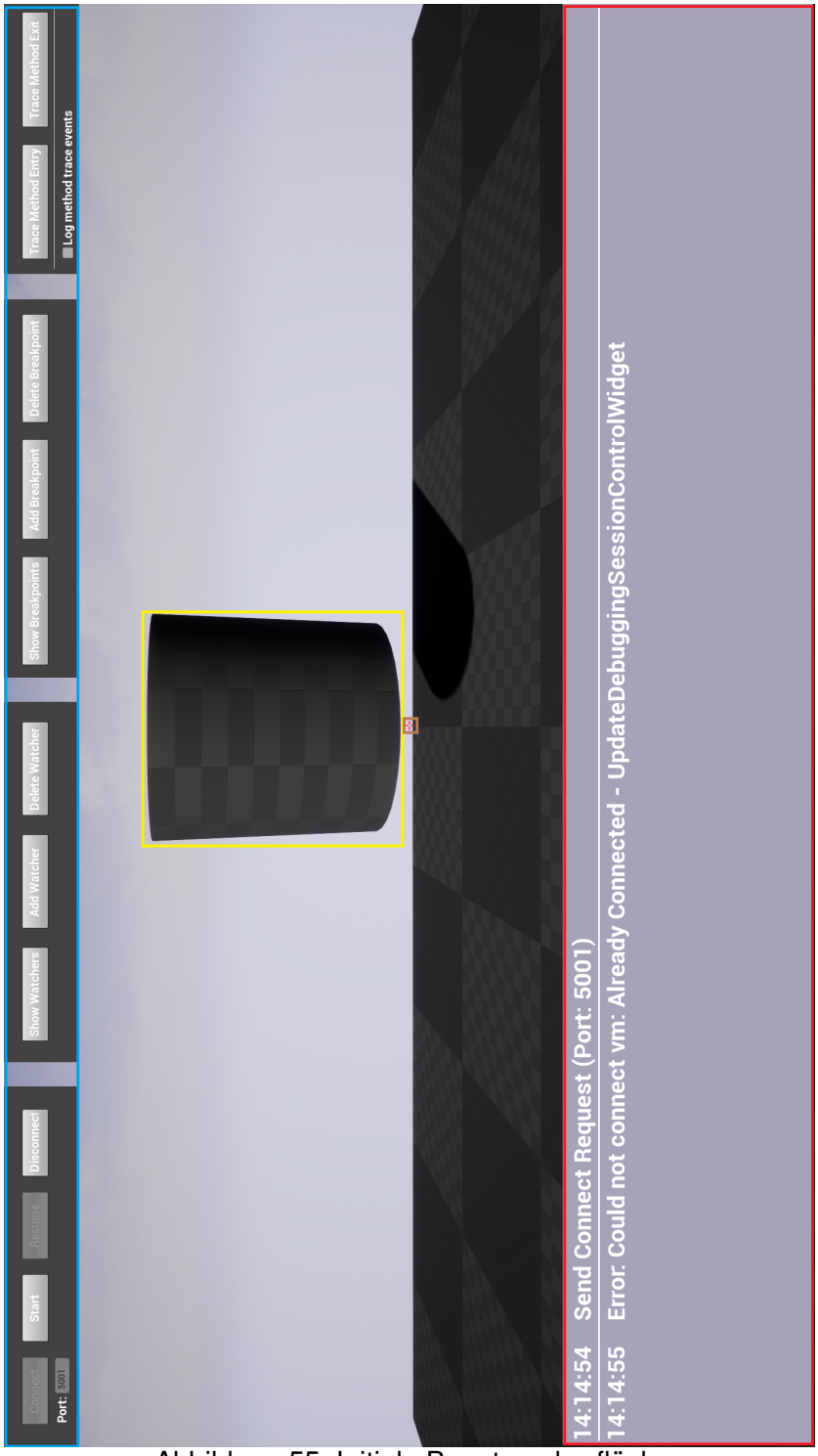

Abbildung 55: Initiale Benutzeroberfläche

A.14 Active Watcher Overlay

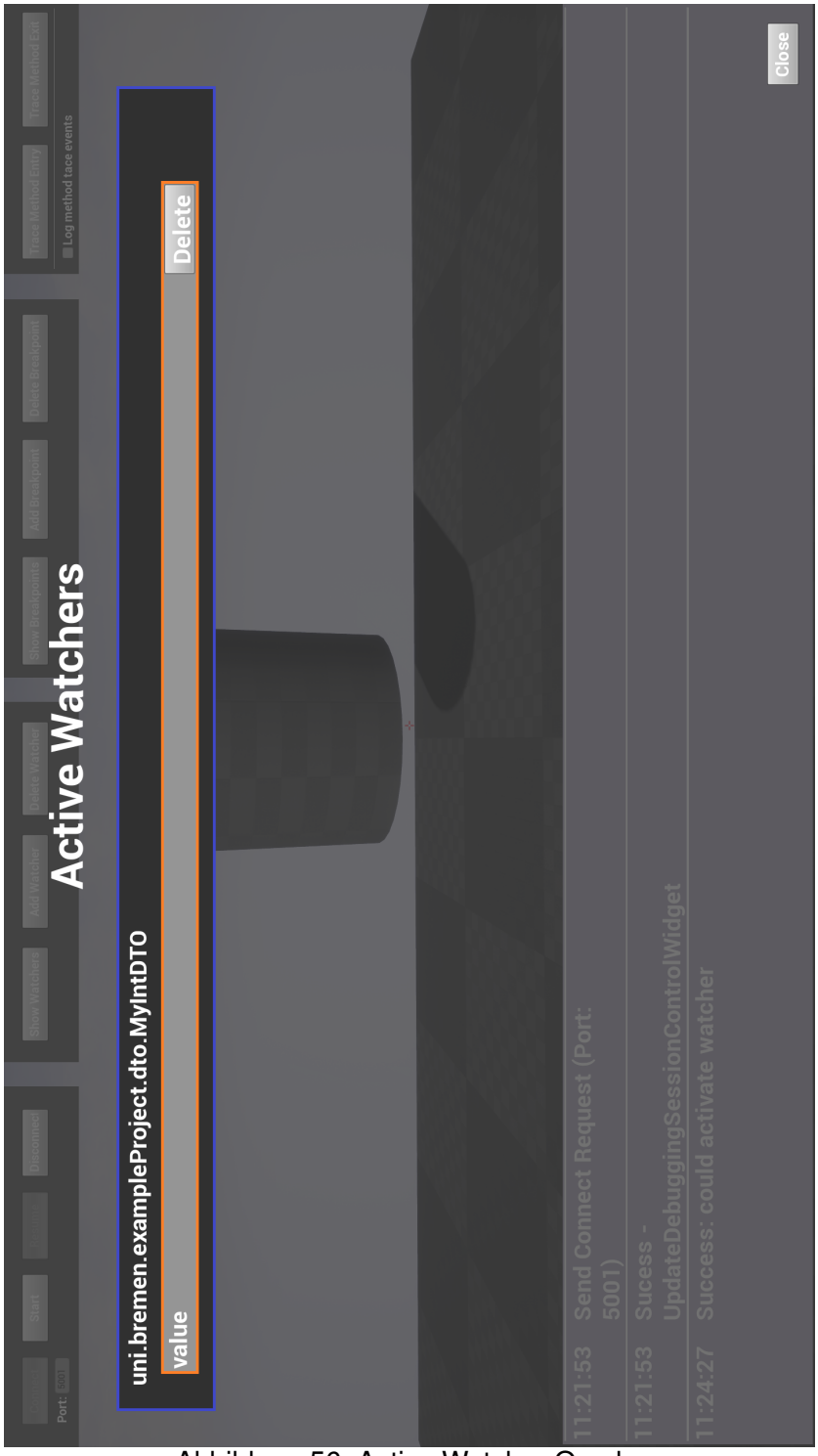

Abbildung 56: Active Watcher Overlay

A.15 Add Watcher Overlay

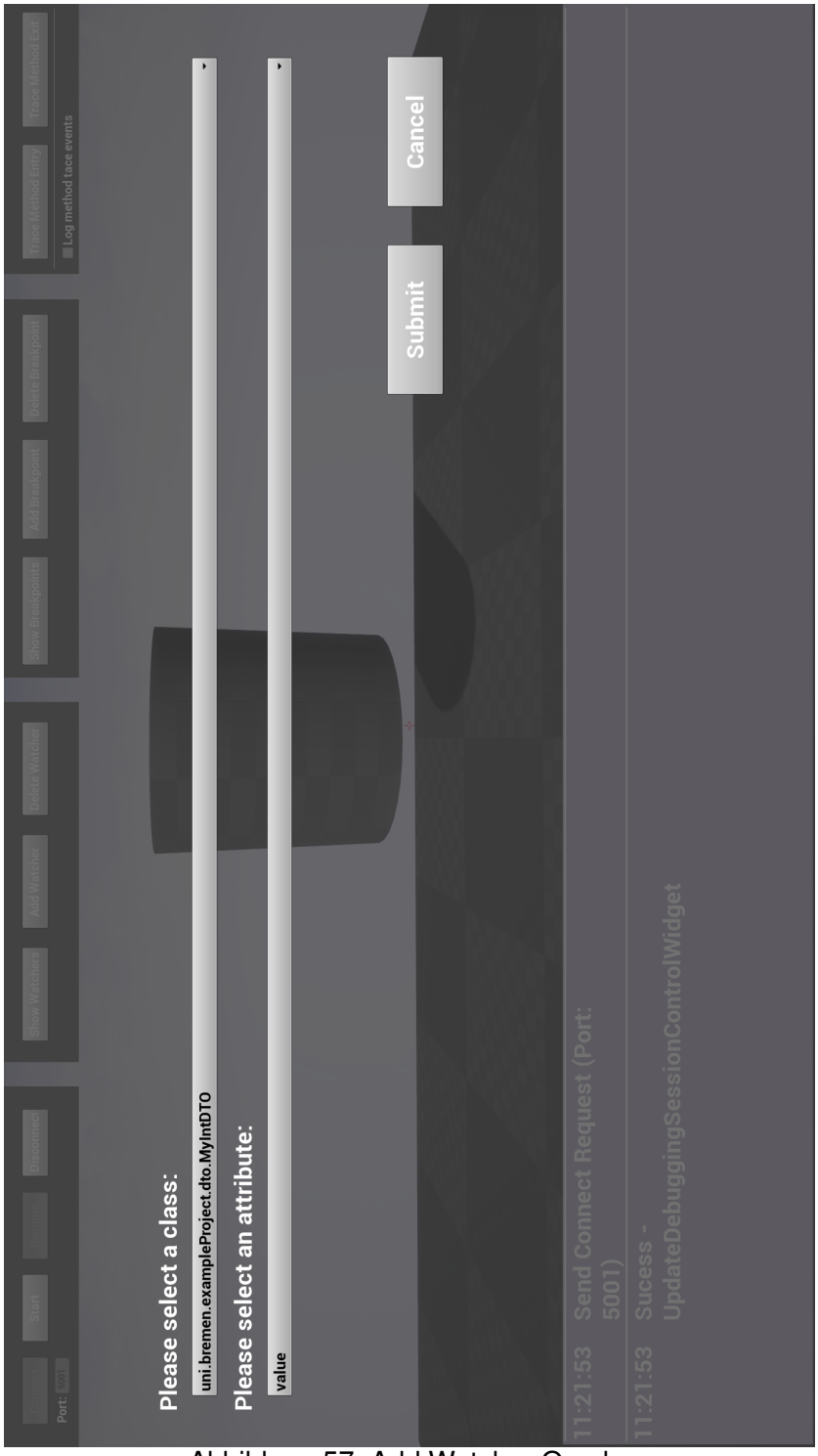

Abbildung 57: Add Watcher Overlay

A.16 Active Breakpoints Overlay

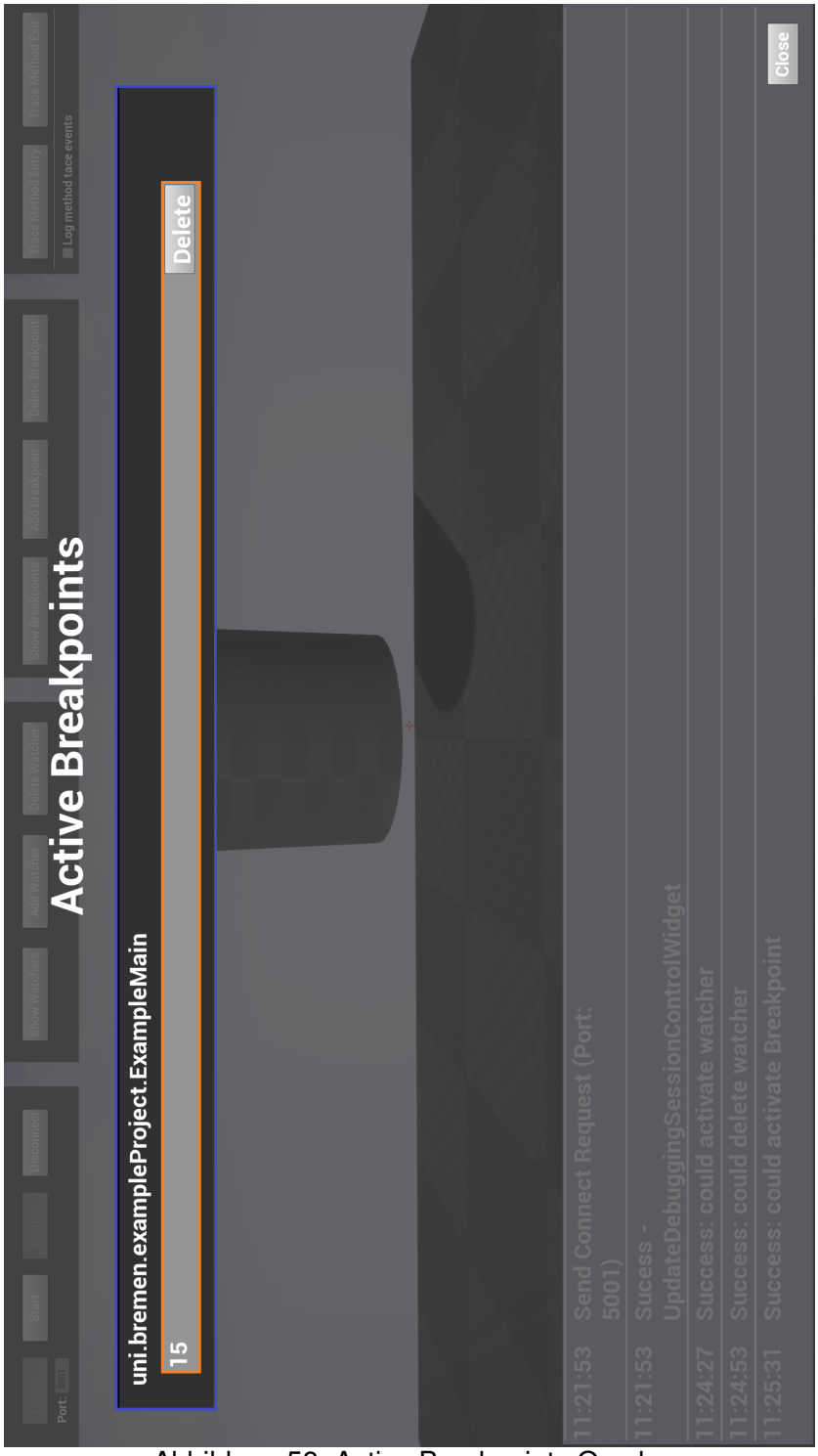

Abbildung 58: Active Breakpoints Overlay

A.17 Add Breakpoint Overlay

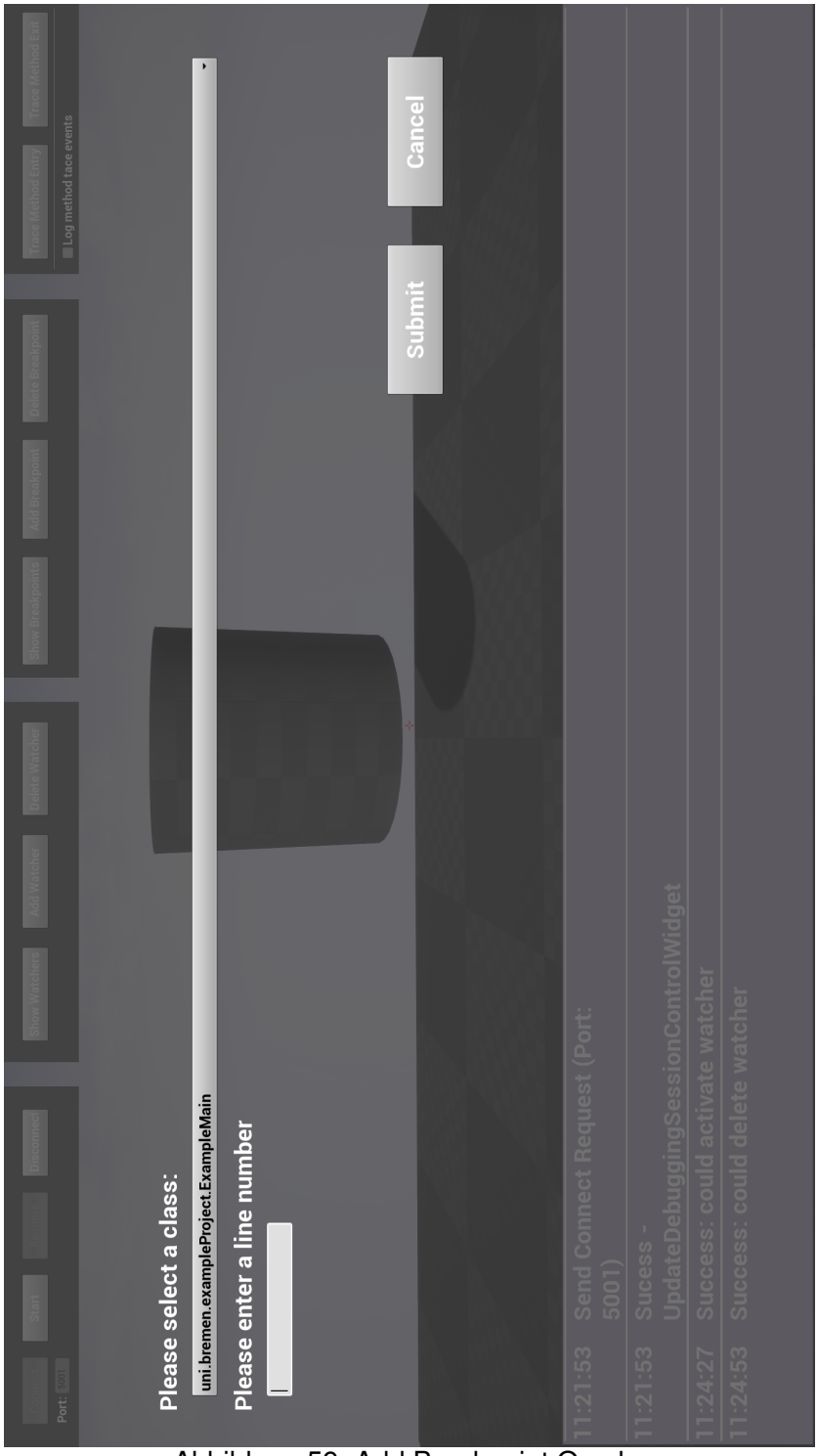

Abbildung 59: Add Breakpoint Overlay

A.18 Connect To Socket Info Layer

Abbildung 60: Connect To Socket Info Layer

A.19 Messungen

A-Nr. = *Ausführungsnummer*

A-Nr.	startTime [ns]	endTime [ns]	difference [ns]	difference [ms]
1	58211819051487	58212173547911	354496424	354,496424
2	58222734836436	58223103231234	368394798	368,394798
3	58227250115382	58227603755059	353639677	353,639677
4	58233081013585	58233446910796	365897211	365,897211
5	58243362689406	58243712725332	350035926	350,035926
6	58249553386163	58249912879468	359493305	359,493305
7	58256738920981	58257098350895	359429914	359,429914
8	58263502122241	58263864668532	362546291	362,546291
9	58269387789618	58269746821635	359032017	359,032017
10	58280020362744	58280391538680	371175936	371,175936
	Mittelwert [ns] / [ms]:		360414149,9	360,4141499
	Standardabweichung [ns] / [ms]:		6720319,354711	6,7203193547

Tabelle 11: Messungen 1: Ausführung mittels der runMainOfCurrentJar.bat

A-Nr.	startTime [ns]	endTime [ns]	difference [ns]	difference [ms]
1	63266871416210	63267289397763	417981553	417,981553
2	63276275072478	63276687180411	412107933	412,107933
3	63285895960292	63286299975692	404015400	404,0154
4	63294158065750	63294573913480	415847730	415,84773
5	63302402153912	63302807363736	405209824	405,209824
6	63309759523878	63310165627021	406103143	406,103143
7	63317453509019	63317870806685	417297666	417,297666
8	63328407840774	63328801941408	394100634	394,100634
9	63346984960109	63347388580049	403619940	403,61994
10	63355140917392	63355533255769	392338377	392,338377
	Mittelwert [ns] / [ms]:		406862220	406,86222
	Standardabweichung [ns] / [ms]:		9046291,206222	9,0462912062

Tabelle 12: Messungen 2: IntelliJ IDEA Remote-Debugger

A-Nr.	startTime [ns]	endTime [ns]	difference [ns]	difference [ms]
1	80904271628380	80904672650477	401022097	401,022097
2	80926009124073	80926404820157	395696084	395,696084
3	80953948535279	80954342026352	393491073	393,491073
4	80974144756869	80974554425906	409669037	409,669037
5	80989112644021	80989512511437	399867416	399,867416
6	81010591493202	81010995639715	404146513	404,146513
7	81025729536296	81026126074254	396537958	396,537958
8	81043265131594	81043664163477	399031883	399,031883
9	81056233574387	81056629539393	395965006	395,965006
10	81074198469267	81074593819386	395350119	395,350119
	Mittelwert [ns] / [ms]:		399077718,6	399,0777186
	Standardabweichung [ns] / [ms]:		4887119,765365	4,8871197654

Tabelle 13: Messungen 3: UVRD (deaktivierte Methoden-Ablaufverfolgung)

A-Nr.	startTime [ns]	endTime [ns]	difference [ns]	difference [ms]
1	89144765386253	89149923168389	5157782136	5157,782136
2	89224413301306	89229464998377	5051697071	5051,697071
3	89306857455211	89311630359079	4772903868	4772,903868
4	89388762601220	89393598601217	4835999997	4835,999997
5	89451438435115	89456485977432	5047542317	5047,542317
6	89551338605220	89556660658718	5322053498	5322,053498
7	89632791319025	89637501973514	4710654489	4710,654489
8	89694872563583	89699535930292	4663366709	4663,366709
9	89764813301355	89770031142000	5217840645	5217,840645
10	89805645766361	89810254818219	4609051858	4609,051858
	Mittelwert [ns] / [ms]:		4938889258,8	4938,8892588
	Standardabweichung [ns] / [ms]:		252103977,2299	252,1039772299

Tabelle 14: Messungen 4: UVRD (aktivierte Methoden-Ablaufverfolgung)

A.20 `exampleProject:` *pom.xml*

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <project xmlns="http://maven.apache.org/POM/4.0.0"
3           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4           xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/
              xsd/maven-4.0.0.xsd">
5      <modelVersion>4.0.0</modelVersion>
6
7      <groupId>uni.bremen</groupId>
8      <artifactId>exampleProject</artifactId>
9      <version>1.0</version>
10
11     <build>
12         <plugins>
13             <plugin>
14                 <groupId>org.apache.maven.plugins</groupId>
15                 <artifactId>maven-compiler-plugin</artifactId>
16                 <version>3.7.0</version>
17                 <configuration>
18                     <compilerArgs>
19                         <arg>-g:source,lines,vars</arg>
20                     </compilerArgs>
21                 </configuration>
22             </plugin>
23         </plugins>
24     </build>
25
26 </project>
```

Abbildung 61: `exampleProject:` *pom.xml*

A.21 `exampleProject:` Klassendiagramm

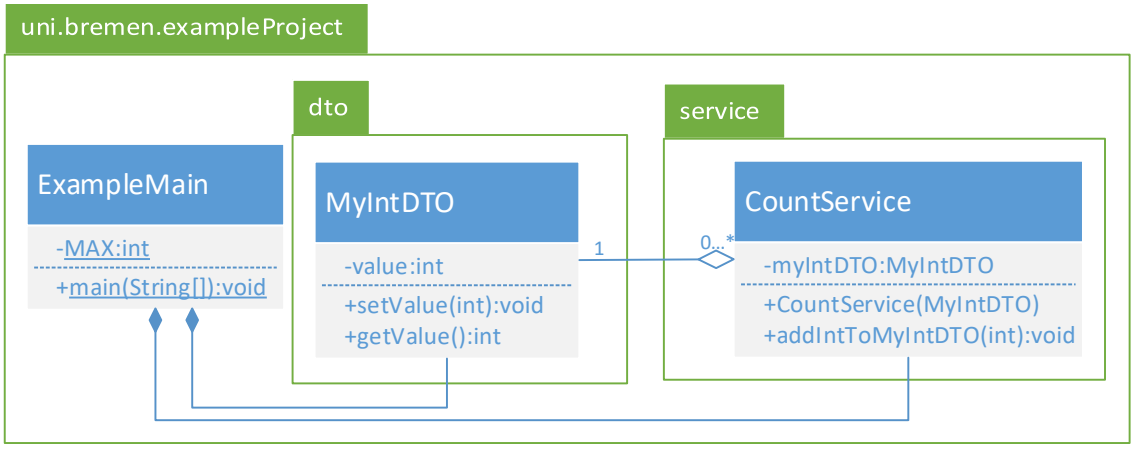

Abbildung 62: `exampleProject:` Klassendiagramm

A.22 Systemspezifikation

Geräteart:	Desktop-Computer
Betriebssystem:	Windows 10 Pro (Version: 1709 / Betriebssystembuild 162999.248)
Prozessor:	Intel(R) Core(TM) i7-7700k CPU 4.20 GHz
Arbeitsspeicher:	Corsair Vengeance® LPX Speicherkit (2 x 16 GB, 3000 MHz)
Grafikkarte:	NVIDIA GeForce GTX 1080 Ti (11 GB, GDDR5X, 352 Bit)
Motherboard:	ASRock Z170 Gaming K4
Festplatte:	Samsung SSD 850 PRO (512 GB)

Tabelle 15: Systemspezifikation

A.23 Programmübersicht

Programm	**Verwendungszweck**
IntelliJ IDEA Community 2017.2[35]	Java IDE
Visual Studio Community 2017[66]	C++ IDE
Sublime Text 2[67]	Texteditor für Notizen
Visio 2013[68]	Erstellung von Diagrammen
TeX Live[69]	TeX Distribution
Texmaker[70]	TeX Editor (Ausarbeitung)
SocketTest v3.0.0[71]	TCP Socket Schnittstellentests
Postman[72]	HTTP Schnittstellentests
Unreal Engine 4[73]	Basis des *UVRD*s
Mockito 2[36]	Java Testframework
JUnit 4[37]	Java Testframework

Tabelle 16: Programmübersicht

A.24 Inhalt der DVD

- `DebuggerBackend`: Sourcecode des *Debugger Backend*s
- `DebuggerBackend_JavaDoc`: javaDoc des *Debugger Backend*s
- `exampleProject`: Sourcecode des Beispielprojektes
- `UVRD`: Das komplette *UVRD* Unreal Engine 4 Projekt
- `UVRD_Doxygen`: Doxygen des *UVRD*s
- `Bachelorarbeit_Bohling.pdf`: Die Ausarbeitung im portable Document Format, welche skalierbare Grafiken beinhaltet.
- `Schnellstartanleitung.txt`: Eine simple Schnellstartanleitung