Lucerne University of
Applied Sciences and Arts

HOCHSCHULE
LUZERN

Team 5

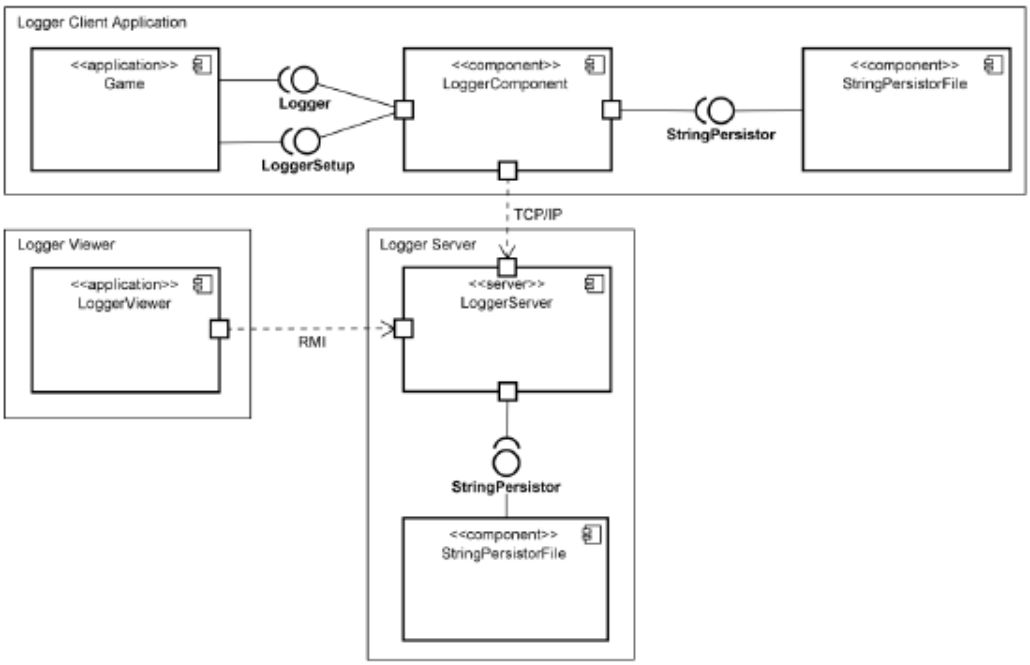
Fischer Frederico Küng Matthias Werlen Oliver Majkel Zivkovski

Message Logger

Version 3.0.0

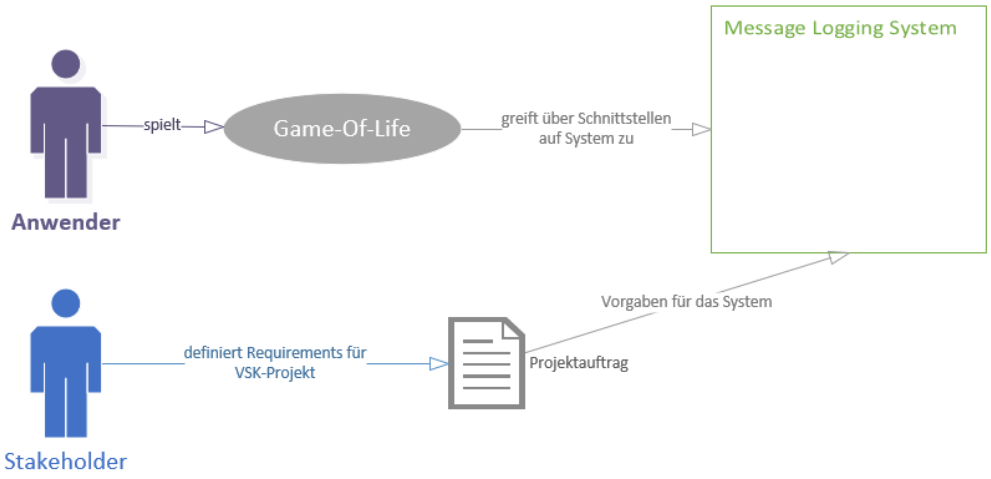
System-Spezifikation

1. Systemübersicht

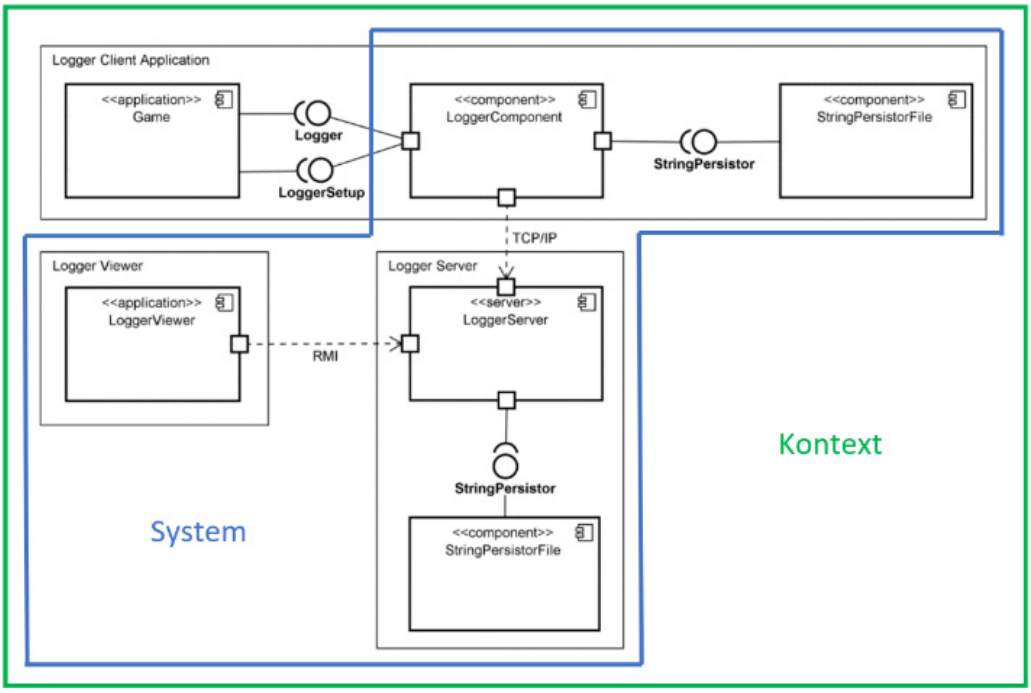


Die Systemarchitektur ist durch den Auftraggeber und seinen Anforderungen gegeben und im oberen Diagramm ersichtlich. Die Anwendung besteht dabei aus zwei Bereichen: dem Client sowie dem Server. Das Game nutzt dabei die Interfaces Logger sowie LoggerSetup, um mit der Logger-Komponente zu kommunizieren. Diese wiederum kommuniziert über TCP/IP-Sockets mit dem LoggerServer. Dieser verwendet wiederum die StringPersistor-Schnittstelle, um mit der StringPersistor-Komponenten zu kommunizieren.

1.1. Kontextdiagramm



Message Logging System (Blau gekennzeichnet)



Das Kontextdiagramm bietet einen abstrakten und grafischen Überblick über das Message Logging System sowie dessen Kontext.

Die Schnittstelle zwischen dem System und dem Anwender stellt die auf einem PC laufende Game-of-Life Applikation dar, welche gleichzeitiges spielen und loggen auf

dem System ermöglicht. Die Vorgaben zum System werden, wie oben in der Grafik abgebildet, im Projektauftrag von den Stakeholdern definiert, weshalb dies ebenfalls in den Kontext des Systems mit einfliesst.

2. Architektur und Designentscheid

Die Architektur wurde weitgehendst vom Auftraggeber vorgegeben. Diese kann man bei Interesse im Detail im Dokument Projektauftrag nachlesen.

2.1. Modell(e) und Sichten

In diesem Projekt wird zwischen drei verschiedenen Sichten unterschieden:

- Der *Anwender*, welcher die Game of Life-Applikation nutzt und seine Spielstände loggen möchte.
- Der *Administrator*, welcher die Konfigurationen für die LoggerKomponente vornimmt (LogLevel und Serverkonfigurationen).
- Der *Programmierer,* welcher die Software-Komponenten zur Verfügung haben möchte, um die gewünschten Anforderungen umsetzen zu können.

2.2. Daten (Mengengerüst & Strukturen)

Für den Austausch der Logmeldungen haben wir als Datenstruktur das Objekt LogMessage definiert. Dabei handelt es sich um ein serialisierbare Klasse, welche als Austauschcontainer für die Logmeldungen zwischen dem Client und dem Server dient. Dieses Objekt besteht aus folgenden Attributen:

- Level: Das LogLevel als String (DEBUG, INFO, WARN, ERROR, FATAL)
- Message: Die Logmeldung als String
- TimeLogged: Der Zeitpunkt an dem die LogMeldung kreiert wurde. Dies wird mit dem Instant-Objekt realisiert und über die toString()-Methode in das Logfile geschrieben.
- TimeServerReceivedLog: Der Zeitpunkt an dem die LogMeldung beim Server angekommen ist. Dies wird mit dem Instant-Objekt realisiert und über die toString()-Methode in das Logfile geschrieben.

Die Klasse PersistedString dient zum Abspeichern und zum späteren Auslesen von Logmeldungen durch den StringPersistor.

2.3. Konfiguration

2.3.1. Logger-Konfiguration

Clientseitig können folgende Konfigurationen vorgenommen werden:

- Angabe des Pfades der JAR-Datei der LoggerKomponente
- Angabe des vollqualifizierten Klassennamens der jeweiligen Implementierung des LoggerSetup
- Angabe des LogLevels ab dem geloggt werden soll (Meldungen mit diesem sowie schwereren Levels werden lokal abgespeichert und an den Server weitergeleitet)
- Angaben der Server-Koordinaten. Dazu gehören der Hostname (Anhand eines InetAdress-Objekts) sowie die Portnummer (anhand der Datenstruktur Int)

2.4. Entwurfsentscheide

2.4.1. Struktur der Logdatei

Die LogMessage-Instanzen werden auf dem Client und dem Server unterschiedlich abgespeichert. Der Grund hierfür ist, dass auf dem Client weniger Informationen gespeichert werden müssen da der Zeitstempel, welcher aussagt, wann die LogMessage-Instanz beim Server angekommen ist, entfällt und dazu das Auslesen der LogDatei dadurch auch schneller abläuft. Anhand der untenstehenden Bilder wird erläutert wie die LogMessage-Instanzen auf dem Client sowie auf dem Server abgespeichert werden:

Client:

```
2019-12-05T17:03:03.927757900Z | [INFO] Game was created
2019-12-05T17:03:06.059492400Z | [DEBUG] Test LogLevel DEBUG
2019-12-05T17:03:08.064366800Z | [WARN] Test LogLevel WARN
2019-12-05T17:03:10.068753400Z | [ERROR] Test LogLevel ERROR
2019-12-05T17:03:12.074449700Z | [FATAL] Test LogLevel FATAL
2019-12-05T17:03:14.493861700Z | [INFO] reset applet
2019-12-05T17:03:20.069710300Z | [INFO] next Generationorg.bitstorm.gameoflife.GameOfLifeGrid@49034156
2019-12-05T17:03:23.077660300Z | [INFO] next Generationorg.bitstorm.gameoflife.GameOfLifeGrid@49034156
2019-12-05T17:03:26.087659Z | [INFO] next Generationorg.bitstorm.gameoflife.GameOfLifeGrid@49034156
2019-12-05T17:03:29.092604500Z | [INFO] next Generationorg.bitstorm.gameoflife.GameOfLifeGrid@49034156
2019-12-05T17:03:32.097665800Z | [INFO] next Generationorg.bitstorm.gameoflife.GameOfLifeGrid@49034156
2019-12-05T17:03:35.105154900Z | [INFO] next Generationorg.bitstorm.gameoflife.GameOfLifeGrid@49034156
2019-12-05T17:03:38.112676200Z | [INFO] next Generationorg.bitstorm.gameoflife.GameOfLifeGrid@49034156
2019-12-05T17:03:41.119081300Z | [INFO] next Generationorg.bitstorm.gameoflife.GameOfLifeGrid@49034156
2019-12-05T17:03:44.125294100Z | [INFO] next Generationorg.bitstorm.gameoflife.GameOfLifeGrid@49034156
```

Als erstes kommt der Zeitstempel, an dem die LogMessage-Instanz erstellt wird. Anschliessend kommt das «|»-Zeichen gefolgt vom LogLevel, welches in «[]»-Klammern dargestellt wird. Zum Schluss kommt noch die entsprechende Message.

Server:

Beim Server wird die toString()-Methode des LogMessage-Instanz verwendet um die entsprechenden Parameter in der LogDatei abzuspeichern. Zuerst kommt wieder der Zeitstempel, an dem die LogMessage-Instanz im Game erstellt wurde. Danach folgt wieder das «|»-Zeichen gefolgt von der toString()-Methode der LogMessage-Instanz welche nach folgendem Format aufgebaut wurde:

LogMessage{loglevel= der Loglevel der Message, message= die Message, timeServerReceivedLog= Zeitstempel, an dem der Server die LogMessage-Instanz erhält}

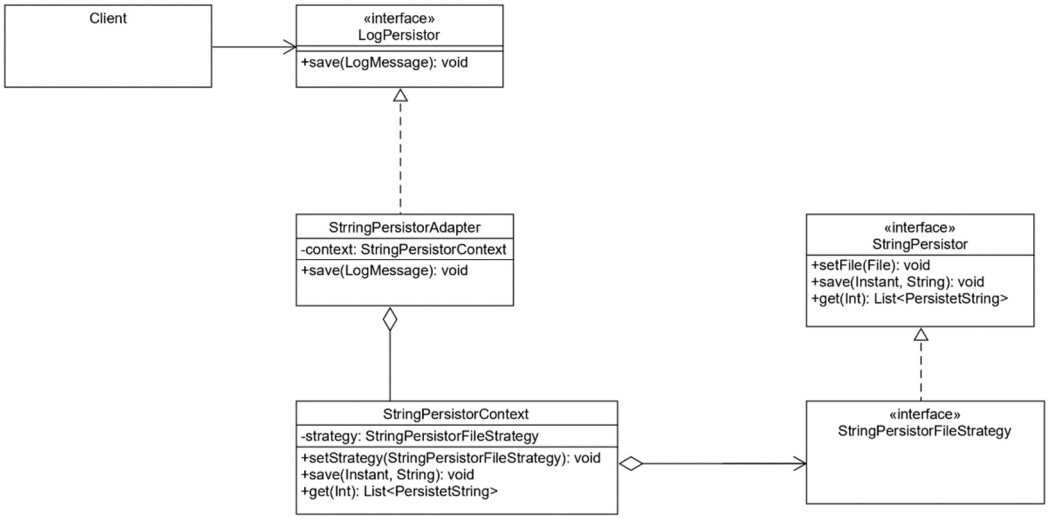
```
2019-12-08T19:50:55.937986300Z | LogMessage{loglevel= WARN, message= 'Test LogLevel WARN, timeServerReceivedLog= 2019-12-08T19:50:55.971181100Z}
2019-12-08T19:50:55.939981Z | LogMessage{loglevel= FATAL, message= 'Test LogLevel FATAL, timeServerReceivedLog= 2019-12-08T19:50:55.971181100Z}
2019-12-08T19:50:55.935992400Z | LogMessage{loglevel= DEBUG, message= 'Test LogLevel DEBUG, timeServerReceivedLog= 2019-12-08T19:50:55.971181100Z}
2019-12-08T19:50:55.899300300Z | LogMessage{loglevel= INFO, message= 'Game was created, timeServerReceivedLog= 2019-12-08T19:50:55.971181100Z}
2019-12-08T19:50:55.938984100Z | LogMessage{loglevel= ERROR, message= 'Test LogLevel ERROR, timeServerReceivedLog= 2019-12-08T19:50:55.971181100Z}
2019-12-08T19:50:57.319864700Z | LogMessage{loglevel= INFO, message= 'reset applet, timeServerReceivedLog= 2019-12-08T19:50:57.324803800Z}
```

Speicherort der LogDatei

Die LogDatei wird auf dem Client im Home-Verzeichnis des angemeldeten Users in einem erstellten Ordner namens «vsk_group5_HS19» im Format «YYYY.MM.DD» abgespeichert, welches den Zeitpunkt der Erstellung der Datei darstellt. Auf dem Server wird die Datei ebenfalls in einem erstellten Ordner im Home-Verzeichnis namens «vsk_group5_HS19» in einer LogDatei namens «yyyy.MM.dd.HH.mm» im TXT-Format abgespeichert. Auch hier handelt es sich um den Zeitpunkt der Erstellung LogDatei bzw. der Zeitpunkt an dem der Server gestartet wird.

2.4.2. Adapterpattern

Gemäss den Anforderungen des Auftraggebers musste das Adapter Pattern (GoF) verwendet werden, um den Payload dem StringPersistor zu übergeben. Im UML-Diagramm ist ersichtlich, wie wir dies umgesetzt haben.



In unserem Projekt nutzt der Logger-Server dieses Pattern. Der Nutzen davon ist, dass uns anhand des Adapters die Möglichkeit besteht eine andere Art für das Abspeichern der Messages zu implementieren.

Wichtig zu erwähnen ist, dass wir das Adapter Pattern mit dem Strategy Pattern kombiniert haben. Dabei wird über den StringPersistorContext festgelegt, welche

Strategie für den StringPersistor verwendet wird. Darauf wird aber im Kapitel «Strategy Pattern» genauer eingegangen.

2.4.3. Strategy Pattern

Beschreibung

Im Stringpersistor wurde das Strategy Pattern mit einem Interface sowie einem Context umgesetzt. Das Team hat sich entschieden, einen Context zu implementieren, da dieses Thema interessant klingt und wir das gesamte Konzept hinter dem Pattern umsetzen wollten und nicht nur einen Teil davon.

Verschiedene Implementation

StringPersistorFileDefault:
In den meisten Fällen wird zum Persistieren die Implementation von StringPersistorFileDefault genutzt. Diese Persistiert die Daten im bereits beschriebenen Format.

StringPersistorFileAsJson:
Als Alternative dazu lässt sich die Message auch im JSON-Format abspeichern. Hierzu wurde die Google GSON API verwendet. Für die Implementation wurde eine innere Klasse definiert. Diese regelt die Struktur des JSON Objekt. Da jedoch die Struktur des Log-Message Objekts nicht Vorgegeben ist, konnte dieses nur als Payload mitgegeben werden.

Das JSON Objekt sieht dabei immer folgendermassen aus:

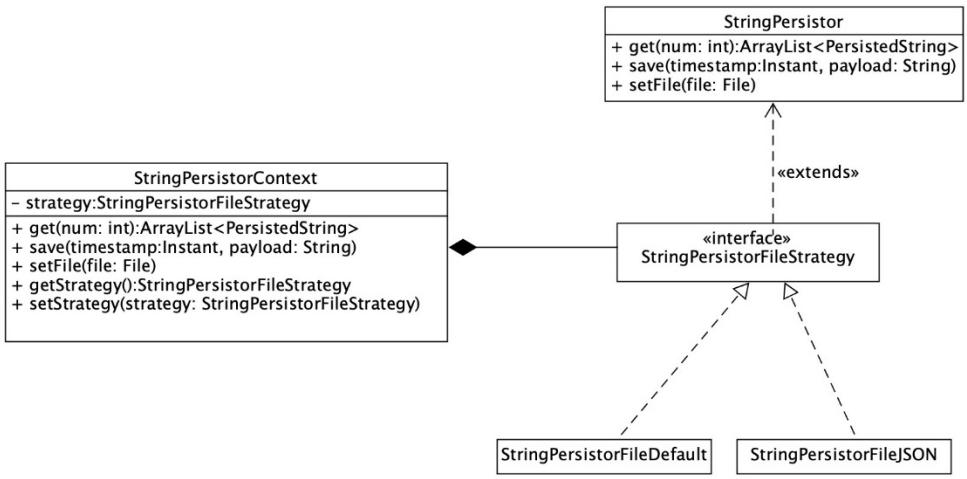
```
{"instantTimeReceived":{"seconds":1575906397,"nanos":606875000},"log
Message":"2019-11-25T16:47:49.298285Z | ./target/g05-
loggercomponent-2.0.0-SNAPSHOT.jar"}
```

```
private class JsonObject{

   Instant instantTimeReceived;
   String logMessage;

   public JsonObject(Instant instantTimeReceived, String logMessage) {
      this.instantTimeReceived = instantTimeReceived;
      this.logMessage = logMessage;
   }
}
```

Ein JSON-Objekt besteht somit immer aus Timestamp + payload.

Jedoch findet in unserer Applikation diese Strategy keinen produktiven Einsatz.

UML



Anwendung

In unserem Projekt haben wir bewusst darauf geachtet, dass ein Zugriff über den Context läuft. Unten ist das Aufrufen des Adapters auf die Strategy zu sehen.

```
this.context.save(logMessage.getTimeStamp(), logMessage.toString() +
"\n");
```

Fazit

Bei unserem Projekt macht der Einsatz einer Strategy nicht viel Sinn. Jedoch könnten noch weitere Implementationen, wie zum Beispiel eine StringPersistorFileSaveBinary dies ändern. Jedoch schränkt das Verwenden von PersistedStrings ein. Hier wäre ein Refactoring dieser Klasse nötig.

2.4.4. Konfiguration File

Einsatz des Konfig Files

Das Konfig File wird beim Game of Life genutzt, um den zu nutzenden Logger anzugeben.

Struktur des Konfig File

Das Konfigurationsfile ist folgendermassen aufgebaut:

1. jarPath
2. className
3. loggerID
4. portNumber
5. Adresse
6. logLevel

Vorgehen

Es wird zuerst überprüft, ob die config.txt Datei existiert. Falls dies zutrifft, die Datei aber leer ist, werden Default-Werte mittels BufferedWriter hineingeschrieben. Ansonsten werden die Werte mittels BufferedReader hinausgeladen und den betreffenden Variablen zugewiesen.

Fazit

Die Konfigurationsdatei wird bei Spielstart entweder generiert und mit Standardwerten gefüllt oder falls bereits vorhanden ausgelesen. Somit kann bei einem Neustart des Spiels die Logger Komponente ausgetauscht werden.

3. Schnittstellen

In diesem Projekt wurden diverse Schnittstellen zu Beginn definiert. Einerseits das Interface StringPersistor vom Auftraggeber (Spezifikation zu finden im Dokument VSK_InterfaceSpec_StringPersistor) und andererseits die Interfaces Logger und LoggerSetup vom Interface-Komitee(Spezifikation zu finden im Dokument Spezifikation_ Logger-Schnittstelle_InterfacekomiteeA).

3.1. Externe Schnittstellen

Damit der Austausch der Logger-Komponente unter den Gruppenmitgliedern ermöglicht werden kann, wurden folgende Schnittstellen definiert:

Konfigurationseinheit	Release 1 + 2
Logger	1.0.0
LoggerSetup	1.0.0
StringPersistor	5.0.1

3.2. wichtige interne Schnittstellen

Der Projektauftrag erforderte weitere Schnittstellen, welche durch uns wie folgt definiert wurden:

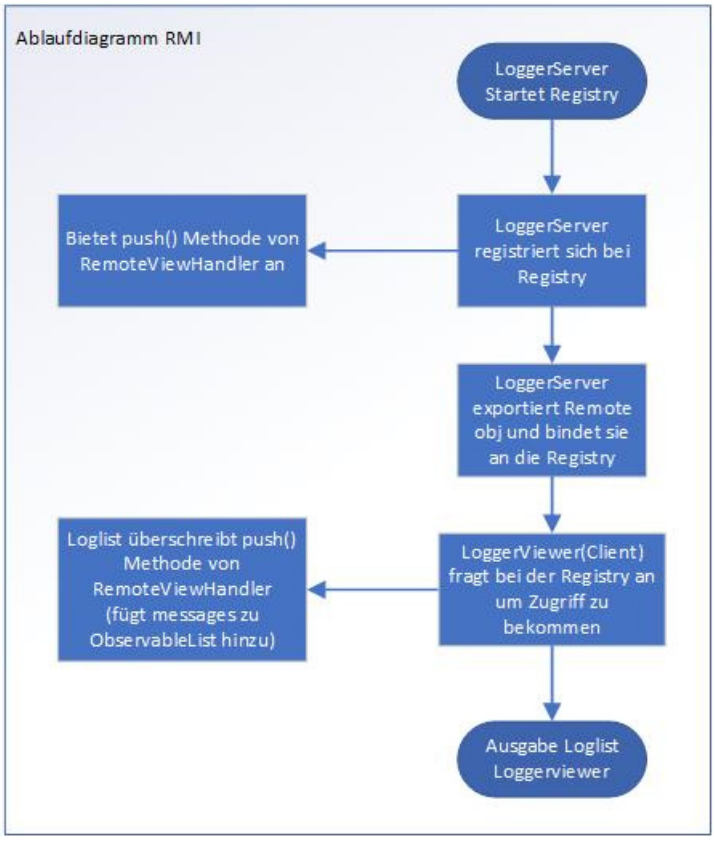
Konfigurationseinheit	Release 1 + 2
LogPersistor	1.0.0
StringPersistorFileStrategy	1.0.0
RemoteViewHandler	1.0.0
RMIRegistration	1.0.0

LogPersistor: hierbei handelt es sich um die Adapter-Schnittstelle, welche die StringPersistor-Schnittstelle für die Logger-Komponente und den Logger Server adaptiert(Adapter Pattern).

StringPersistorFileStrategy: Interface zum umsetzen des Strategy Pattern, welches oben bereits beschrieben wurde.

RemoteViewHandler: erlaubt dem Logger-Server, einem Logger-Viewer Meldung über ein Push-Verfahren zuzustellen.

RMIRegistration: ein Interface welches eine Registrierung des Logger Viewers auf den Logger-Server ermöglicht.



Auf die Implementierung wird an dieser Stelle verzichtet. Stattdessen wird auf den Source Code des Projektes verwiesen.

4. Environment-Anforderungen

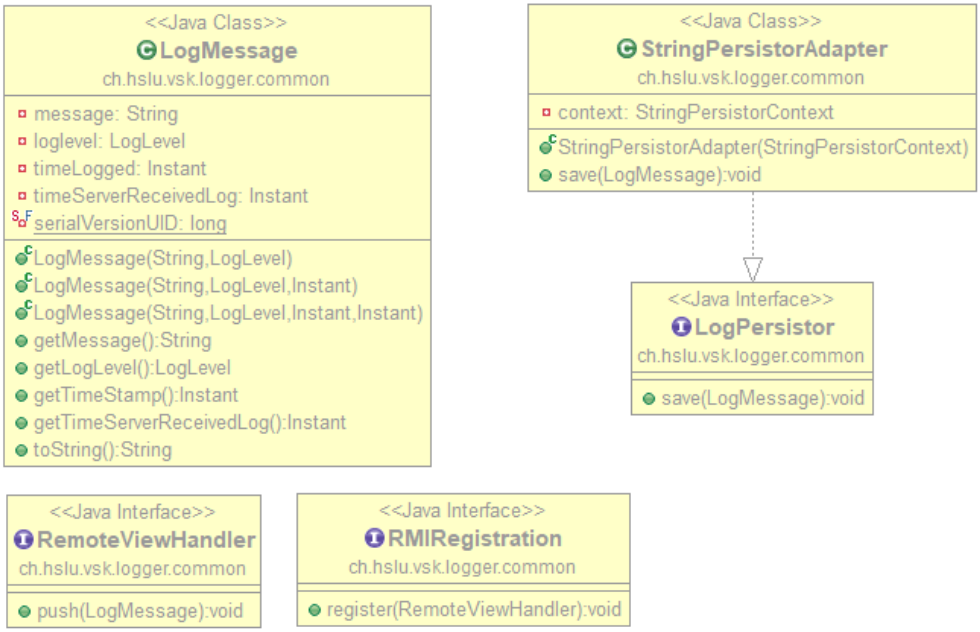
Damit die Anwendung Client- sowie Serverseitig ausgeführt werden kann, wird die Java SE Runtime Environment 8 mit JavaFX benötigt.

Für die Kompilierung der Anwendung wird Maven und das HSLU-Nexus-Repository benötigt.

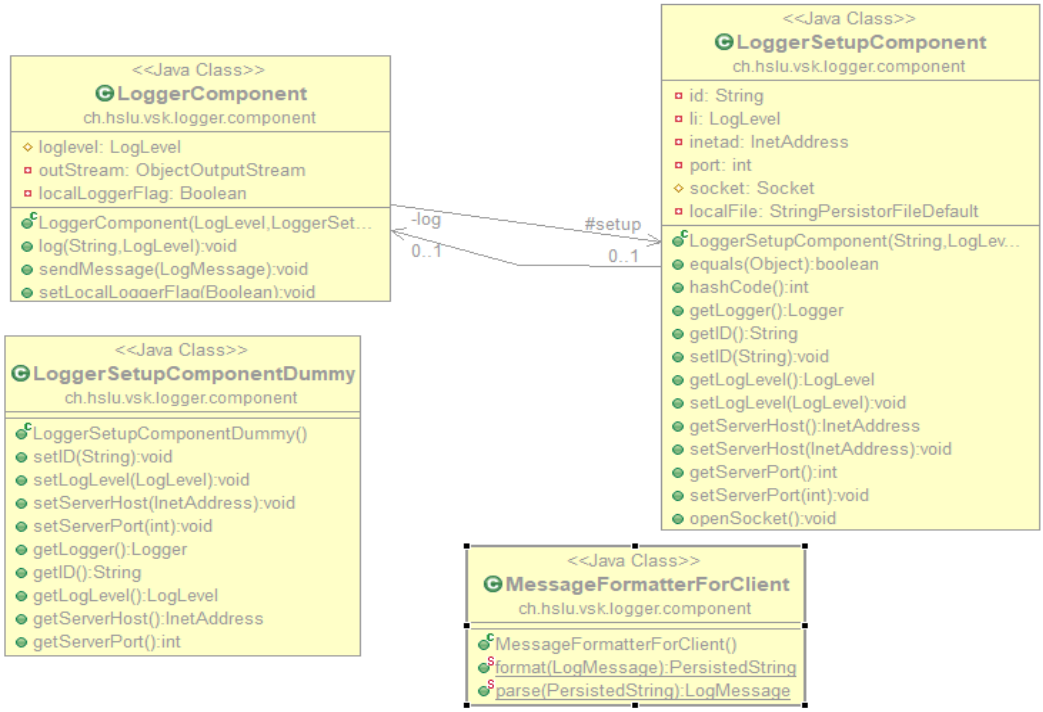
Damit die wechselseitige Kommunikation zwischen den Clients sowie Servers funktioniert, sollte darauf geachtet werden, dass sich die Systeme im gleichen Netzwerk befinden, da ansonsten Firewalls die entsprechenden Ports unterbinden könnten.

5. Klassendiagramme

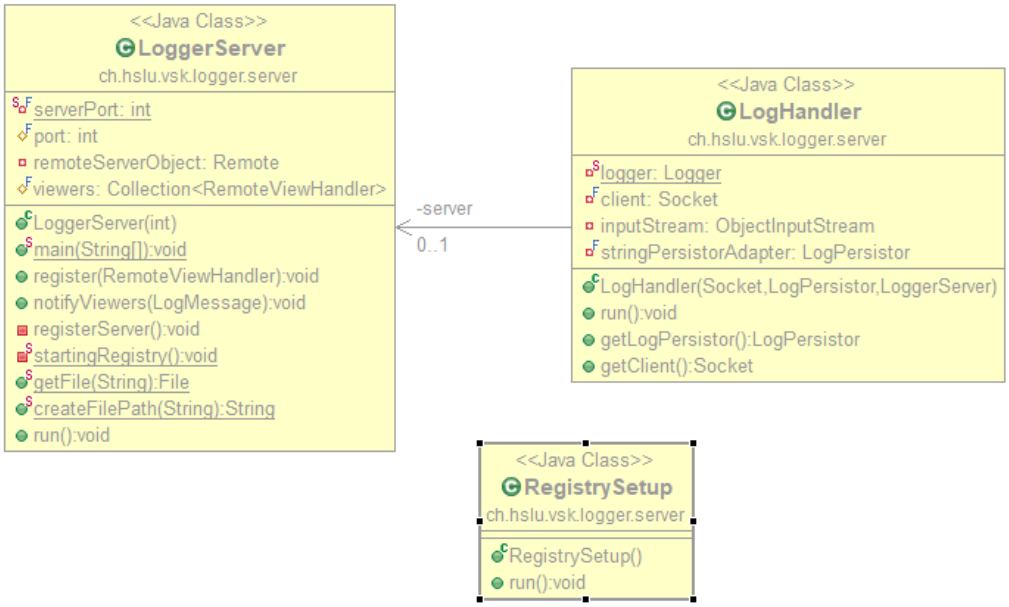
Logger-Commen:



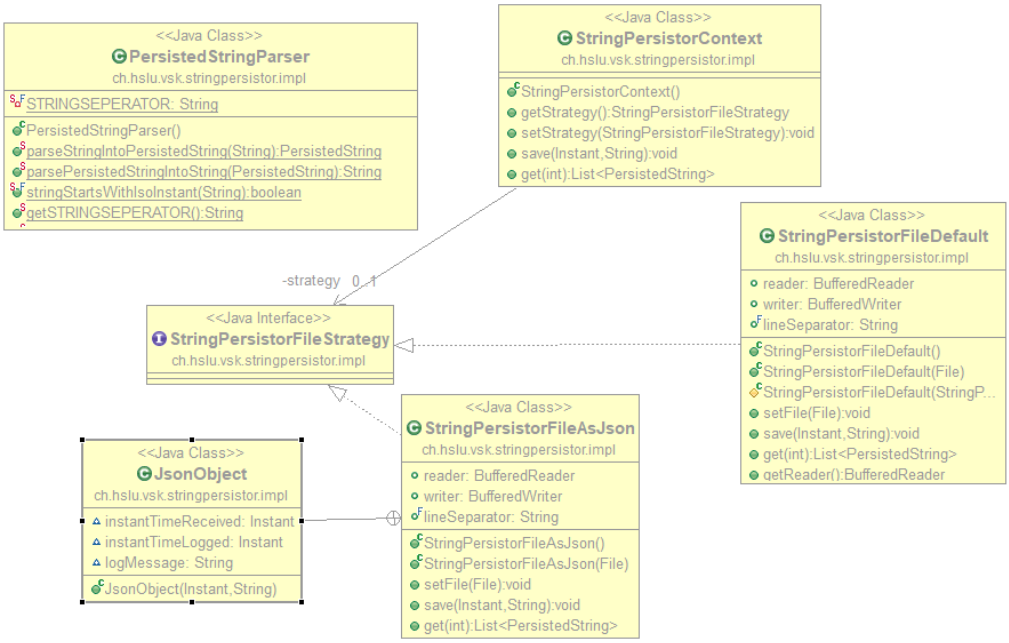
Logger-Component:



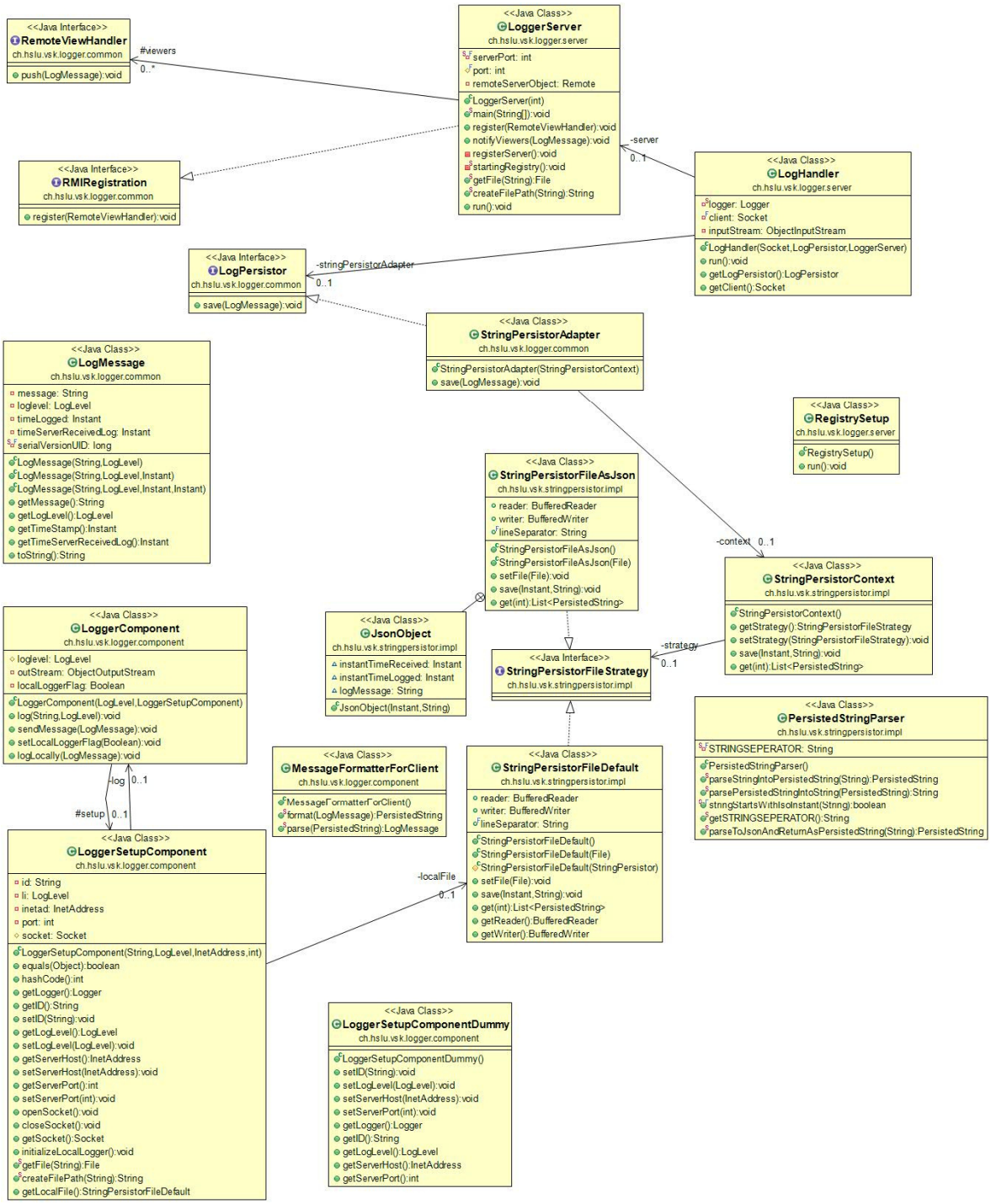
Logger-Server:



StringPersistor:



Message Logger:



6. Diskussion

6.1. Diskussion zur Serialisierung

1. Diskutieren Sie im Team, wie das Übertragungsprotokoll der Messages via TCP/IP aussehen könnte.

Die Messages werden als String abgespeichert und über einen Byte Stream versendet. Zudem muss eine TCP-Prüfsumme mitgeschickt werden, was zu einem grossen Aufwand führen würde.

2. Stellen Sie im Team Überlegungen zum Mengengerüst der Datenübertragung an.

Das Mengengerüst setzt sich zusammen aus LogMessage, LogLevel, Timestamps und Prüfsumme, welche in einem String abgespeichert werden.

In Bezug auf die Serialisierung könnte man anstelle der Prüfsumme eine serialVersionUID verwenden.

3. Diskutieren Sie im Team, ob und wie Sie die Konzepte und Konstrukte aus dem Input Serialisierung einsetzen könnten:

a) Messages via TCP/IP übertragen

Die Klasse LogMessage implementiert das Interface Serializable. LoggerComponent, welches den Logger implementiert schickt dann ein Serializable Objekt über den ObjectOutputStream weiter.

b) Messages speichern

Log Messages werden serialisiert und über den ObjectOutputStream an den Server weitergeleitet. Dieser bekommt ein Paket, welches er deserialisiert und mittels StringPersistor in ein File schreibt.

c) Messages anzeigen

Messages können mittels einer getter-Methode angezeigt werden.

d) Spezialfälle

Spezialfall 1 – Objekte können nicht deserialisiert werden:

In so einem Fall soll eine Fehlermeldung abgespeichert werden.

Spezialfall 2 Server funktioniert nicht:

Wenn dieser Fall eintritt, sollten die Log-Messages in einem ersten Schritt lokal abgespeichert und sobald Server wieder verfügbar ist nachgeschickt werden.

6.2. Diskussion zu Message Passing

6.2.1. TCP/IP-Schnittstelle

Skizzieren und Diskutieren Sie in Ihrem Projektteam die proprietäre TCP/IP Schnittstelle unter Beachtung der folgenden Aspekte:

- **Verbindungaufbau:**
  Der Verbindungsaufbau erfolgt über Sockets. Sollte dies nicht funktionieren tritt eine Exception auf

- **Konfiguration:**

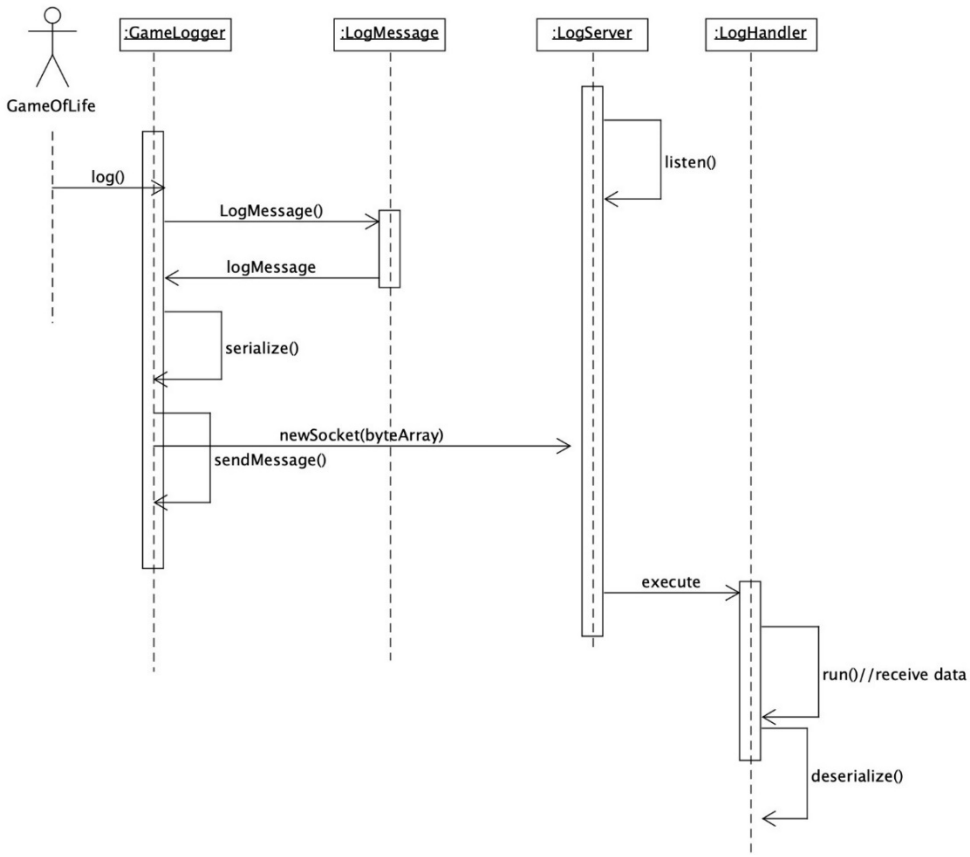
  Serverhost: localhost

  Port: beliebig (aktuell 3200)

  Konfiguration sind auf einem File gespeichert
- **Protokoll:**

  TCP/IP
- **Datenformat:**

  Byte Stream



6.2.2. Umsetzung im Message-Logger

a) Wie könnten die im VS_03_MessagePassing vorgestellten Implementationen der Message Passing Protokolle zum Einsatz kommen? Begründen Sie in jedem Fall Ihre

Antwort, warum Sie eine der vorgestellten Codeskizzen des Message Passing übernehmen und einsetzen oder warum Sie Message Passing in dieser Art nicht einsetzen.

Persistente asynchrone Kommunikation:

Die persistente asynchrone Kommunikation ist aufgrund von fehlender Middleware nicht machbar. Zudem müssen beim Logger Projekt beide Parteien A und B lauffähig sein.

Persistente synchrone Kommunikation:

Die persistente synchrone Kommunikation funktioniert beim Logger Projekt ebenfalls nicht, da wieder beide Parteien A und B lauffähig sein müssen.

Transiente asynchrone Kommunikation:

Die transiente asynchrone Kommunikation wird im Projekt nicht eingesetzt, da die Datenpersistenz nicht gewährleistet werden kann.

Empfangsbasierte transiente synchrone Kommunikation:

Die empfangsbasierte transiente synchrone Kommunikation ist das Ziel für unser Projekt, da die Persistenz für grosse Datenmengen gewährleistet werden kann.

Auslieferungsbasierte transiente synchrone Kommunikation:

Die auslieferungsbasierte transiente synchrone Kommunikation kommt nicht in Frage, da das Game-of-Life nicht warten kann, bis der Server eine Bestätigung sendet.

Antwortbasierte transiente synchrone Kommunikation:

Die antwortbasierte transiente synchrone Kommunikation kommt ebenfalls nicht in Frage, da es hier zu noch längeren Wartezeiten kommen würde.

b) Welchen Mehrwert ergibt ein Message Passing Protokoll im Projekt?

- Die Datenpersistenz kann gewährleistet werden
- Man kann auf bewährtes zugreifen
- Die Message Passing Protokolle bieten ein standardisiertes Verfahren

c) Sehen Sie andere Möglichkeiten wie Sie ein Message Passing Protokoll (in Ihrem Projekt) umsetzen? Welche?

Es wurde keine passende Alternative zu den Message Passing Protokollen gefunden.

6.3. Disskussion zu Uhren Synchronistation

a) Wo könnten logische Uhren zum Einsatz kommen? Begründen Sie in jedem Fall Ihre Antwort,

- **warum Sie logische Uhren einsetzen oder**
- **warum Sie logische Uhren nicht einsetzen**.

Logische Uhren könnten in unserem Projekt beim Game of life und beim Server vorkommen. Hierbei verweisen wir auf die beiden Zeitstempel in der LogMessage-Instanz welche erzeugt werden. Der erste Zeitstempel beim Auslösen der LogMessage auf dem Game und der zweite beim Erhalt auf dem Server.
Eine mögliche Implementation wäre somit das Abgleichen der Uhren von Game und Server bei jedem Log.

- LogMessages dürfen nicht in der Zukunft liegen. Das heisst, sie dürfen keinen Zeitstempel aus der Zukunft tragen. Dies würde der “Happened Bevore”-Relation von Lamport widersprechen. Somit müssten die Uhren zwischen GameOfLife und Server synchronisiert werden.
- Bei unserem banalen Logger ist die Konsistenz der Daten wichtiger als die Zeitstempel. Die Banalität der “Happened Before”-Relation kann vernachlässigt werden, da diese keinen direkt Einfluss auf die Funktion des Servers hat.

b) Welchen Mehrwert ergeben die logischen Uhren im Projekt?

Im Allgemeinen werden logische Uhren in Bereichen eingesetzt, in denen Kausalität und Verlässlichkeit eine grosse Rolle spielen. In unserem Projekt spielt die Verlässlichkeit der Logs eine sehr wichtige Rolle, weshalb diese logische Uhr einen hohen Mehrwert dazu beitragen könnte.

- Der genaue Zeitpunkt, zu welchem ein Log aufgerufen wurde, kann zu jedem Zeitpunkt auf dem Game und dem Server nachvollzogen werden.
- Zudem können verwirrende LogMessages vermieden werden.

c) Welche logische Uhr (mit Lamport-Zeitstempel oder Vektor-Zeitstempel) ist sinnvoll, bezüglich des Mehrwerts vs. Aufwand?

In unserem Projekt würde die Nutzung des Lamport-Zeitstempel am meisten Sinn machen. Dieser gibt wenig Aufwand und liefert eine ausreichende Genauigkeit, um die Ansprüche des Auftraggebers zu erfüllen.