Systemspezifikationen

Teammitglieder:

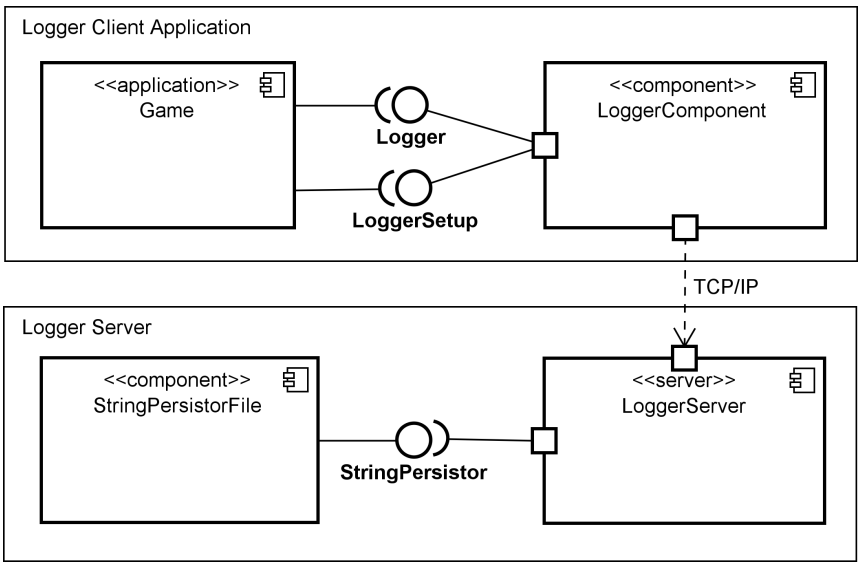
- Christopher Christensen
- Valentin Bürgler
- Lukas Arnold
- Melvin Werthmüller

// TODO Update this list

Rev.	Datum	Autor	Bemerkungen	Status
1.1	24.10.17	Valentin Bürgler	Erster Entwurf	done
1.2	31.10.17	Christopher Christensen	Erweiterung Kap.1/2	done
1.3	01.11.17	Valentin Bürgler	Bearbeitung Kap.2/4	done
1.4	03.11.17	Valentin Bürgler	Bearbeitung Kap.1/3, Diagramme + `config` -File	done
1.5	05.11.17	Christopher Christensen	Für Zwischenabgabe prüfen	done
1.6	05.11.17	Valentin Bürgler	Überarbeitung aller Kapitel	done
2.0	06.11.17	Christopher Christensen	Aufbereitung für Merge mit alter Dokumentation	done
2.1	10.11.17	Christopher Christensen	Merge SysSpec mit alter Dokumentation	done
2.2	10.11.17	Christopher Christensen	LogFile.txt Specs added	done
2.3	10.11.17	Melvin Werthmüller	Content organisation	done
2.4	10.11.17	Melvin Werthmüller	LoggerServer specs updated	done
2.5	10.11.17	Christopher Christensen	einige TODOs erledigt	done

1 Systemübersicht

1.1 Grobe Systemübersicht



Es soll eine Logger-Komponente implementiert werden, die eingebunden in eine bestehende Java-Applikation über Methodenaufrufe Meldungen aufzeichnet, welche dann per TCP/IP an einen Logger-Server gesendet werden, wo sie in einem wohldefinierten Format gespeichert werden.

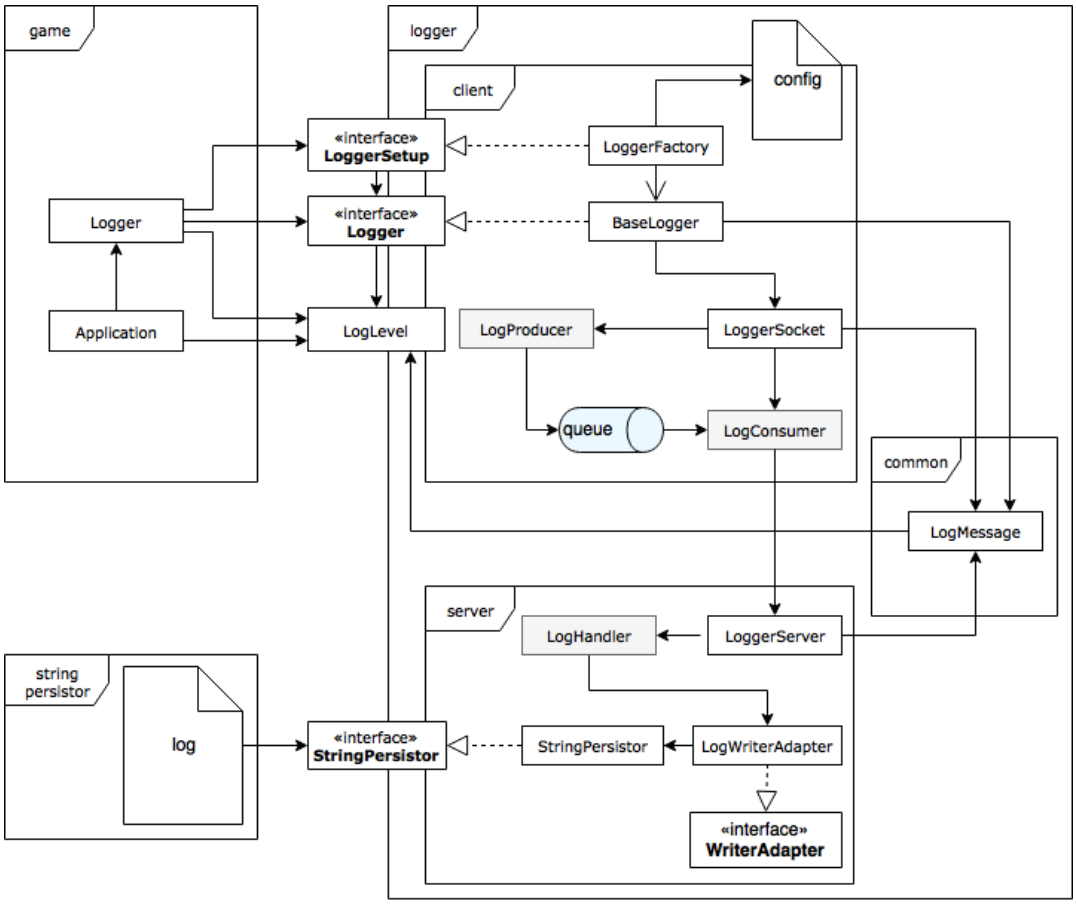
Sinnvolle Ereignisse und Situationen, die geloggt werden müssen, sind zu definieren und die entsprechenden Aufrufe in der Java-Applikation zu integrieren.

Die durch ein Interface-Team definierten LogLevels sind sinnvoll und konsistent zu nutzen. Weiter sind die vorgegebenen Schnittstellen Logger, LoggerSetup und StringPersistor einzuhalten. Es müssen sich mehrere Clients mit einem Server verbinden können.

Im späteren Verlauf des Projektes kommen weitere Anforderungen hinzu.

1.2 Vollständige Systemübersicht

Das folgende UML soll eine detaillierte Übersicht über das implementierte System schaffen.



Im beiliegenden Dokument DokumentationMessageLogger.pdf werden die einzelnen Komponenten detaillierter beschrieben. Auch die Relationen untereinander werden ausführlich aufgezeigt.

1.3 Ablauf auf dem Client

In der Applikation instanziiert ein `Logger`-Singleton über die `start`-Methode mit der `LoggerFactory` eine spezifische Logger-Implementierung. Dieses `Logger`-Objekt bietet dann Methoden um einen `String` oder ein `Throwable` mit dem entsprechenden `LogLevel` zu loggen. Damit die Verbindung asynchron ist, werden zuerst alle zu loggenden Meldungen mit einem eigenen Thread `LogProducer` in eine Queue geschrieben. Des Weiteren ist ein Thread `LogConsumer` dafür zuständig die Queue zu lesen und die Meldungen über eine TCP Verbindung zum Server zu schicken.

1.4 Ablauf auf dem Server

Der Server stellt einen Socket bereit und empfängt Meldungen vom Client. Für jede erhaltene Nachricht wird ein eigener `LogHandler` erstellt, welcher die Meldungen asynchron an den Adapter zum Stringpersistor weitergibt. Der Stringpersistor ermöglicht es dem `LogHandler` (via `LogWriterAdapter`) über die `save`-Methode eine Zeitinstanz mit einer Log-Message in ein Log-File zu schreiben. Das File wird durch einen Aufruf der Methode `setFile` im Logger-Server definiert.

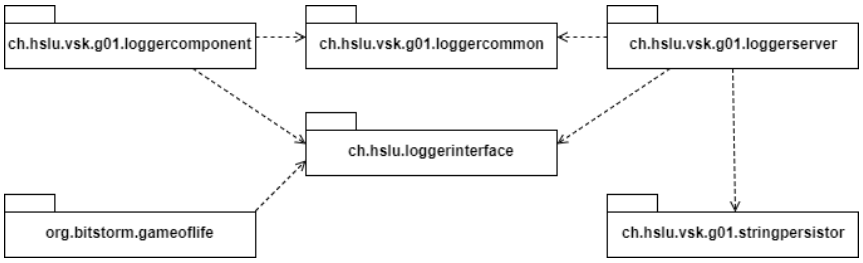
---

2 Architektur und Designentscheide

Wir versuchten, möglichst viele bewährte objektorientierte Entwurfsmuster zu verwenden, um eine saubere Architektur unseres MessageLoggers zu erreichen.

2.1 Modelle und Sichten

Packetdiagramm



Klassendiagramm

// TODO Melvin & Vali (erstellen und einfügem des kompletten klassendiagrams)

2.2 Entwurfsentscheide

Wir haben generell über das Projekt hinweg versucht uns an den Clean-Code-Prinzipien zu orientieren. Wir versuchten Vererbung zu vermeiden und das «Favour Composition over Inheritance»-Prinzip zu verfolgen. Dazu strebten wir an die Wiederverwendbarkeit zu erhöhen indem wir das DRY-Prinzip vor Augen hielten und die einzelnen Komponenten so zu gestalten, dass sie nur jeweils eine Aufgabe erfüllen (Seperation of Concerns).

Strategie-Pattern

// TODO vali (nur Strategie Pattern beschreiben, nicht Factory etc. zusammen. Und wo dieses verwendet wird in unserem Projket)

> Strategie-Pattern ist ein Verhaltensmuster

Singleton-Pattern

// TODO vali (singeltonprinzip erklären und wo dieses verwendet wird in unserem Projket)

> Strategie-Pattern ist ein Erzeugungsmuster

Fabrikmethode-Pattern

// TODO vali (fabrikprinzip erklären und wo dieses verwendet wird in unserem Projket)

> Strategie-Pattern ist ein Erzeugungsmuster

Adapter-Pattern

Das Adapter-Muster ist ein Strukturmuster und übersetzt eine Schnittstelle in eine andere. Dadurch kann die Kommunikation einer Klasse zu einer inkompatiblen Schnittstellen ermöglicht werden und gleichzeitig eine lose Kopplung zu gewährleisten.

Für die Übertragung der `LogMessage` vom `LogHandler` zum `StringPersistor`, verwenden wir das Adapter-Modell. So kann die Implementation der `StringPersistor`-Klasse ungeändert bleiben und wir können eine angepasste Implementation für den `LogHandler` erstellen. Dadurch erhalten wir die effektiv gewünschte Zielschnittstelle.

Konfigurationsdatei

// TODO luki (prinzip von konfigurationsdateien erklären und wo dieses verwendet wird in unserem Projken)

---

3 Schnittstellen

3.1 Externe Schnittstellen

Die folgenden Schnittstellen wurden uns vorgeschrieben.

- `Logger`
- `LoggerSetup`
- `LogLevel`
- `StringPersistor`

Logger

// TODO luki (erklärung der Logger schnittstelle ohni konkrete implementation)

> Verwendete Version: TODO luki

LoggerSetup

// TODO luki (erklärung der LoggerSetup schnittstelle ohne konkrete implementation)

> Verwendete Version: TODO luki

LogLevel

// TODO luki (erklärung weshabl es diese zentrale loglevels gibt und was logLevels sind)

LogLevel
`DEBUG`
`INFO`
`WARN`
`ERROR`
`CRITICAL`

> Verwendete Version: TODO luki

StringPersistor

> Verwendete Version: 1.0.0 (ch.hslu.vsk.g01.stringpersistor)

3.2 Interne Schnittstellen

Die folgenden Schnittstellen wurden von uns vorgeschrieben.

- `LogMessage`
- `WriteAdapter`
- `client.properties`
- `server.properties`
- TCP/IP Schnittstelle

LogMessage

Die LogMessage speichert Meldungen mit zusätzlichen Attributen. Folgende Tabelle gibt einen Überblick über die Klasse.

Attribut	Beschreibung	Datentyp
level	Log Stufe	LogLevel
message	Nachricht	String
createdAt	Zeitpunkt der Erstellung	Instant
receivedAt	Zeitpunkt des Erhaltens	Instant

WriteAdapter

Der WriteAdapter stellt die Schnittstelle vom Server zum Stringpersistor her und versteht sich somit als Adapter. Der Adapter definiert das File und das Format der zu speichernden `LogMessage`-Objekte. Der WriteAdapter verfügt über die Schreibmethode `void writeLogMessages(LogMessage logMessage)`. Es schreibt auch die Implementation der Methode `List<PersistedString> readLogMessages(int i)` vor. Der übergebene Parameter liefert die gewünschte Anzahl letzter `LogMessage`-Objekten aus dem LogFile zurück.

Der Server nutzt diesen Adapter über die Implementation `LogWriterAdapter`, um die LogMessages (unabhängig von der Implementation des StringPersistors) dem StringPersistor zu übergeben.

client.properties

TODO Beschreibung Luki

Verwendung: TODO luki wie & wo wird diese verwendet

server.properties

TODO Beschreibung Luki

Verwendung: TODO luki wie & wo wird diese verwendet

TCP/IP Schnittstelle

Der Logger beinhaltet die Funktion `log`, welche eine `LogMessage` an den Server schickt. Damit die Verbindung asynchron ist, werden zuerst alles zu loggenden Meldungen mit einem eigenen Thread `LogProducer` in eine Queue geschrieben. Desweiteren ist ein Thread `LogConsumer` dafür zuständig, die Queue zu lesen und die Meldungen über eine TCP Verbindung zum Server zu schicken.

Die Übertrag der Meldungen geschieht über den `ObjectInputStream` / `ObjectOutputStream`, welche die serialisierbare Klasse `LogMessage` als Objekte überträgt.

---

4 Implementation von Komponenten

LoggerComponent (Client)

Der Logger besteht hauptsächlich aus der Klasse `BaseLogger`, welcher das `Logger`-Interface implementiert. Er bietet die Methode `log` an, welche mit einem `LogLevel` als erstes Argument und einer Nachricht als String, aufgerufen werden kann um etwas zu loggen. Zusätzlich steht noch eine überladene Methode bereit, welche als zweites Argument ein `Throwable` akzeptiert, was es ermöglicht auch Exceptions zu loggen.

Durch die Instanzierung eines Loggers wird sofort ein `LoggerSocket` erstellt und gestartet. Er enthält

eine Queue mit den Meldungen, welche an den Server gesendet werden sollten. Er bietet ausserdem die Methode `queueLogMessage`, welche asynchron eine `LogMessage` in die Queue speichert. Beim Starten des Sockets wird ein `LogConsumer`-Thread gestartet, welcher ständig die Queue abarbeitet und die enthaltenen Nachrichten via einen `ObjectOutputStream` über einen TCP-Socket an den Server sendet.

LoggerServer

Der Server stellt einen Socket bereit und empfängt Meldungen vom Client. Für jede erhaltene Nachricht, wird ein eigener `LogHandler` erstellt, welcher die Meldungen asynchron an den Adapter zum Stringpersistor weitergiebt.

LoggerServer - Class

Der LoggerServer besitzt eine `main` Methode, welche für das Starten des Servers verantwortlich ist. Die Klasse bestitz ausserdem drei wichtige lokale Konstanten.

- ExecutorService

Dies ist ein ThreadPool, welcher für die einzelnen LogHandler abarbeitet. Genauer handelt es sich um einen `newFixedThreadPool` mit fünf Threads.

- LogWriterAdapter

Dies ist die Referenz zum Adapter, welche einmalig erzeugt wird und jedem `LogHandler` zur Verwendung mitgegeben wird. Diest ist die Schnittstelle zum `StringPersistor`.

- ServerSocket

Der Socket ist die Anlaufstelle des Servers. TCP Packete werden damit empfangen. Der LoggerServer erstellt für jede erhaltene Nachricht einen eigenen LogHandler. Der ServerSocket ist mit der Klasse `LoggerServerSocket` implementiert

LoggerServerSocket - Class

Der LoggerServerSocket erstellt einen ServerSocket. Dafür liest er die Konfiguarionen mit der Methode `loadConfigFile()` aus dem Konfigurationsfile. Falls das File `config.properties` nicht existiert, werden standard Werte verwendet. Mit diesen Werten wird ein ServerSocket erstellt. Der Socket wird mit der statischen Methode `create()` erstellt.

Die standard Werte sind wie folgt definiert:

Name	Value
host	127.0.0.1
port	54321
amount	10

LogHandler - Class

Der LogHandler wird vom LoggerServer erstellt. Dieser ist für die asynchron Weitergabe an den LogWriterAdapter verantwortlich. Dementsprechend ist die impementierung auch einfach gehalten. Die Run-Methode sieht wie folgt aus:

```
public void run() {
    logWriterAdapter.writeLogMessage(message);
}
```

StringPersistor

In der StringPersistor-Komponente wird dafür gesorgt, dass die `LogMessage`-Objekte in ein `File` geschrieben werden.

StringPersistor - Class

Der Stringpersistor schreibt eine Zeitinstanz mit einer Log-Message in ein Log-File. Dazu muss der LogHandler im StringPersistor auch das Log-File an den StringPersistor übergeben mit der Methode `void setFile(final File file)`. Mit der Methode `void save(final Instance instance, final String s)` wird die Zeitinstanz und Log-Message in das zuvor festgelegte Log-File gespeichert. Die Methode `List<PersistedString> get(int i)` liefert die mit dem Parameter `i` gewünschte Anzahl letzten Log-Einträge als `List` des Typs `PersistedString` aus dem Log-File zurück.

LogWriterAdapter - Class

Der `LogWriterAdapter` implementiert das Interface `WriteAdapter` und überschreibt die Methoden `writeLogMessage(LogMessage logMessage)` und die Methode `readLogMessages(int i)`. Die Methoden haben dieselbe Funktion, wie die Methoden der `StringPersistor`-Klasse (`save` und `get`), sind jedoch auf den `LogHandler` angepasst.

LogFile.txt

Das `LogFile.txt` ist das Text-Dokument, in welches alle `LogMessage`-Objekte gespeichert werden.Es wird durch den `LogWriterAdapter` erstellt und dem `StringPersistor` übergeben.

Danach werden die `LogMessage`-Objekte über den `StringPersistor` mit Hilfe des `LogWriterAdapter` in das `LogFile.txt`.

Format

Das Format mit dem die `LogMessage`-Objekte in das `LogFile.txt` geschrieben werden sieht folgendermassen aus.

1. Datum & Zeit vom Erhalten der `LogMessage`
2. Datum & Zeit vom Erstellen der `LogMessage`
3. LogLevel der `LogMessage`
4. Nachricht in der `LogMessage`

```
String message = logMessage.getReceivedAt() + ";"
+ logMessage.getCreatedAt() + ";"
+ logMessage.getLogLevel() + ";"
+ logMessage.getMessage();
```

5 Verwendung des Loggers

5.1 Einbinden auf einem Client

Um den Logger in einer Client-Applikation in Betrieb zu nehmen, muss dafür mit der `LoggerFactory` ein `LoggerSetup`-Objekt geholt werden. Hierfür muss der Factory-Methode `getLoggerSetup` der "Fully Qualified Class Name" einer Klasse übergeben werden, die das `LoggerSetup` Interface implementiert. Über das `LoggerSetup`-Objekt können dann verschiedene `Logger` erstellt werden.

Zum besseren Verständnis folgt eine Beispiel-Implementierung:

```
String fqn = "ch.hslu.vsk.g01.loggercomponent.LoggerFactory";
String server = "127.0.0.1";
Integer port = 54321;

try {
    LoggerSetup loggerFactory = LoggerFactory.getLoggerSetup(fqn);
    Logger logger = loggerFactory.createLogger(server, port);
} catch (ClassNotFoundException | IllegalAccessException | InstantiationException
e) {
    // Implement error handling here
}
```

5.2 GameOfLife Einbindung

Der GameOfLife Applikation wurde eine neue Klasse hinzugefügt, das `Logger`-Singleton. Um den Logger in Betrieb zu nehmen, wird über die statische `start`-Methode mit der `LoggerFactory` eine spezifische Logger-Implementierung instanziiert. Dafür wird die Konfigurationsdatei `config.properties` eingelesen, worin sich der "Fully Qualified Class Name" der `LoggerFactory`, die IP Adresse des Servers und die Portnummer in dieser Reihenfolge befinden muss. Mit dieser Konfigurationsdatei lässt sich die Logger-Komponente austauschen. Zur Veranschaulichung folgt der mögliche Inhalt von `config.properties`:
`fqn=ch.hslu.vsk.g01.loggercomponent.LoggerFactory server=127.0.0.1 port=54321`
Danach kann der Logger dazu verwendet werden, mit der statischen `log` Methode ein `LogLevel` und entweder ein `String` oder `Throwable` loggen.

Die Applikation wurden ausserdem um Aufrufe dieser `log` Methode mit entsprechenden LogLevels erweitert. Die `LogLevels` finden folgende Verwendung:

LogLevel	Verwendung
`DEBUG`	Jegliche Information, die in irgendeiner Form nützlich sein könnte, wie Methodenaufrufe, Parameterwerte, etc.
`INFO`	Information über wichtige Ereignisse im Spiel. Jeglicher User-Input wird mit diesem Level geloggt.
`WARN`	Warnungen, wenn etwas passiert, das so nicht geplant war. Das Spiel läuft jedoch weiterhin.
`ERROR`	Fehler, von welchen das System sich wieder erholen kann, wie z.B. Fehler beim Laden/Speichern einer Shape.
`CRITICAL`	Fehler, von welchen das System sich nicht erholen kann und beendet werden muss, z.B. bei einer InterruptedException

6 Testing

Die Funktionalität sollte so gut wie möglich durch Unit-Tests abgedeckt werden. Es macht keinen Sinn die Einbindung ins Game automatisiert zu testen, da viel zu umfangreiche Änderungen notwendig wären. Deswegen werden für die Integration ein paar manuelle Tests definiert, welche regelmässig überprüft werden. Auch die Übertragung der Daten vom Client zum Server wird durch manuelle Test abgedeckt.

6.1 Unit Testing

LoggerCommon

Zur Verifikation der `LogMessage`-Klasse gibt es einen `LogMessageTest`, welcher das wichtigste

Verhalten der Klasse überprüft.

LoggerComponent

Der `BaseLogger` wird durch den `BaseLoggerTest` überprüft. Damit nicht ständig einen TCP-Socket aufgemacht werden muss, verwendet der der Test die Klasse `FakeBaseLogger`, welche von `BaseLogger` abgeleitet wird. Darin wird vorallem die Methode `createSocket` überschrieben und es wird ein `FakeLoggerSocket` erstellt. Ausserdem bietet die abgeleitete Klasse noch ein paar Getters und andere Methoden zur Überprüfung der Daten. Für die Verifikation der anderen Klassen in diesem Modul werden manuelle Tests verwendet, da das Testen einer TCP-Verbindung nicht so trivial ist.

LogWriterAdapter

Der `LogWriterAdapter` hat nur die Methode `void writeLogMessage(LogMessage logMessage)` und diese wird anhand eines JUnit-Tests `LogWriterAdapterTest` getestet. Zuerst wird ein `File`, ein `LogWriterAdapter` und eine `LogMessage` instanziiert. Der `LogMessage` wird ein `LogLevel` und eine Message des Typs `String` übergeben. In einem "PreAssert" mit der Methode `assertEquals(Boolean expected, Boolean actual)` wird geprüft, ob das erstellte File leer ist, da es noch keine `LogMessage` enthalten darf. Danach wird mit der Methode `void writeLogMessage(LogMessage logMessage)` die `LogMessage` in das zuvor erstellte `File` geschrieben. Jetzt kommt der "Assert" wo wieder mithilfe der Methode `assertEquals(Boolean expected, Boolean actual)` geprüft wird, ob das Dokument nun **nicht leer** ist. Am Ende wird das erstellte `File` gelöscht mit der `File`-Methode `delete()`. Der `delete()` kann auskommentiert werden, falls man das Format der im `File` gespeicherten `LogMessage` überprüfen möchte.

StringPersistor

Der `StringPersistor` wird anhand eines JUnit-Tests `StringPersistorTest` getestet. Der Test für die Methode `void setFile()` beginnt mit dem Instanzieren eines `StringPersistor`-Objekts und `File`-Objekts. Das `File` wird über die Methode `setFile` dem `File`-Attribut des `StringPersistor` übergeben. Über die Methode `getFile()` wird in der `assertEquals(Boolean expected, Boolean actual)` geprüft, ob es sich beim Rückgabewert, um dasselbe `File` handelt, das übergeben wurde. Die Methode `void save(Instant instant, LogMessage logMessage)` wird nach ähnlichem Verfahren, wie der LogWriterAdapter getestet (siehe Kapitel Unit Testing > LogWriterAdapter). Die Methode `List<PersistedString> get()` wurde noch nicht getestet, da sie noch nicht vollständig implementiert ist.

6.2 Manual Testing

GameOfLife

Für den Integrationstest der Einbindung in die GameOfLife Applikation wird geprüft, ob die Datei "LogFile.txt" zur Speicherung der Logs auf dem Dateisystem erstellt wurde. Dazu wird zuerst die `main` Methode des `LoggerServer` gestartet. Dann wird die GameOfLife Applikation gestartet. Weiter wird getestet, ob Log-Einträge in "LogFile.txt" vorhanden sind, denn der Aufruf der `init` Methode sollte bereits zu einem Log-Eintrag auf `LogLevel.INFO` mit der Nachricht "Initializing UI..." führen.

LoggerComponent & LoggerServer

Der `LoggerServer` wird vorallem mit dem `DemoLogger` getestet. Dieser schickt vier LogMeldungen mit unterschiedlichen `LogLevels` an den Server. Manuell wird dann überprüft, ob die richtigen Meldungen erhalten wurden. Dieser Test dient hauptsächlich zur Überprüfung der TCP-Verbindung und dem LogMessage-Handling in der Queue. Die Teilkomponenten `StringPersistor` und `LogWriterAdapter` haben ihre eigenen JUnit-Tests (siehe Kapitel Unit Testing > StringPersistor und Unit Testing > LogWriterAdapter).

---

7 Environment

Hier sind die Umgebungsanforderung für unseren MessageLogger aufgelistet.

- Die Logger-Komponente ist mit **Java 1.8.0** realisiert. Es gelten die entsprechenden System-Anforderungen für Java 1.8.0.
- Der Fully-Qualified Class Name der LoggerFactory, die IP Adresse und die Portnummer des Servers müssen in einer Konfigurationsdatei «**config.properties**» vorliegen, um eine beliebige Logger-Komponente eines anderen Teams ohne Anpassungen im Code an das Spiel zu koppeln.
- Eine **Internetverbindung** wird benötigt, um die Nachrichten an den Server zu senden.