Systemspezifikation

Version 2.0.0 (Schlussabgabe)

Gruppe 5 (Patrick Bucher, Pascal Kiser, Fabian Meyer, Sascha Sägesser)

11.05.2018

Inhaltsverzeichnis

1 Systemübersicht

Die Systemarchitektur ist grösstenteils durch den Projektauftrag festgelegt. Dieses ist in der Abbildung Komponentendiagramm Projektauftrag ersichtlich.

Die Anwendung besteht aus zwei Bereichen: Dem Client und dem Server. Das *Game* kommuniziert mittels *Logger*- und *LoggerSetup*-Schnittstelle mit der *LoggerComponent*. Diese schlägt die Brücke zum Server-Bereich über ein TCP/IP-Protokoll, worüber sie mit dem *LoggerServer* kommuniziert. (Dieses Protokoll ist implementierungsspezifisch und bei der Gruppe 5 im Dokument *TCP-Schnittstelle* dokumentiert.) Der *LoggerServer* macht Gebrauch von einer Komponente namens *StringPersistorFile*, mit welcher er über die *StringPersistor*-Schnittstelle kommuniziert.

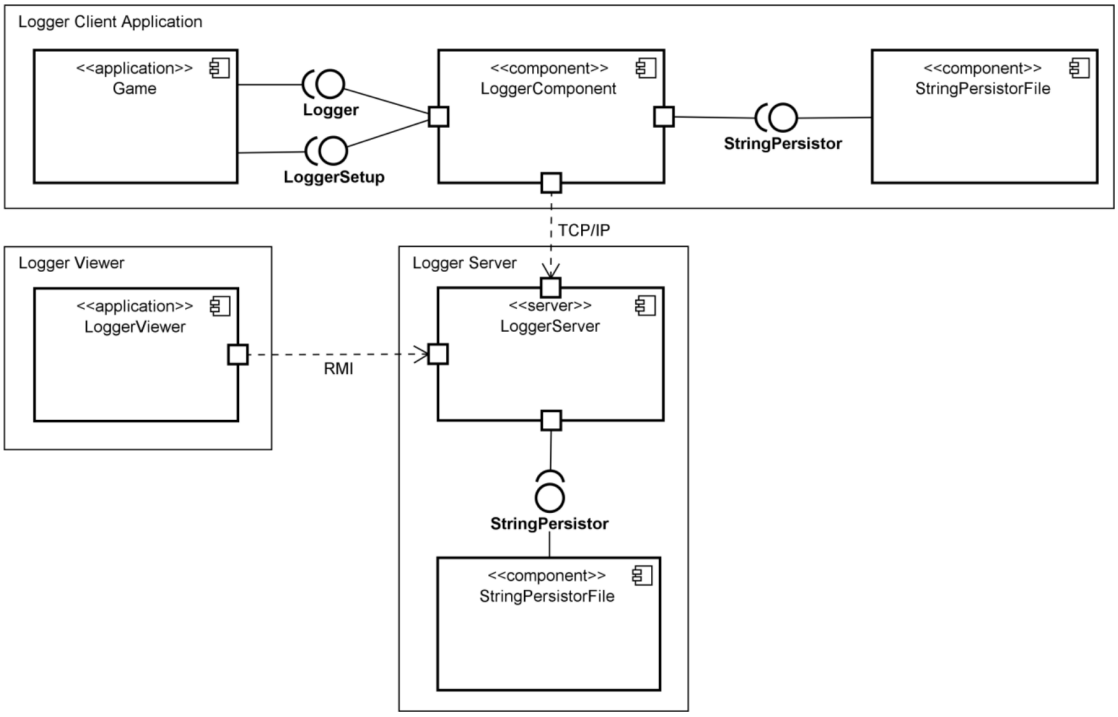

Abbildung 1: Komponentendiagramm Projektauftrag

1.1 Implementierungsspezifische Komponentenarchitektur

Die Abbildung Komponentendiagramm gibt das Komponentendiagramm aus dem Projektauftrag leicht verändert wieder, indem es dem Umstand Rechnung trägt, dass der *LoggerServer* und die Komponente *StringPersistorFile* nicht direkt über die *StringPersistor*-Schnittstelle, sondern per *LogPersistor*-Schnittstelle über den *StringPersistorAdapter* und die *StringPersistor*-Schnittstelle miteinander kommunizieren.

1.2 Kontextdiagramm

Die Abbildung Kontextdiagramm bietet einen abstrakten Überblick über das System und dessen Kontext. Zum System gehört die gesammte Applikation.

Die Schnittstelle zwischen dem System und dem Benutzer stellt z.B. ein Computer dar, auf dem der Benutzer das *Game of Life* spielen und gleichzeitig loggen kann. Zusätzlich gehört die Aufgabenstellung, sprich der *LoggerProjektauftrag*, zum Kontext, da das System auf Basis vom diesem entwickelt wird.

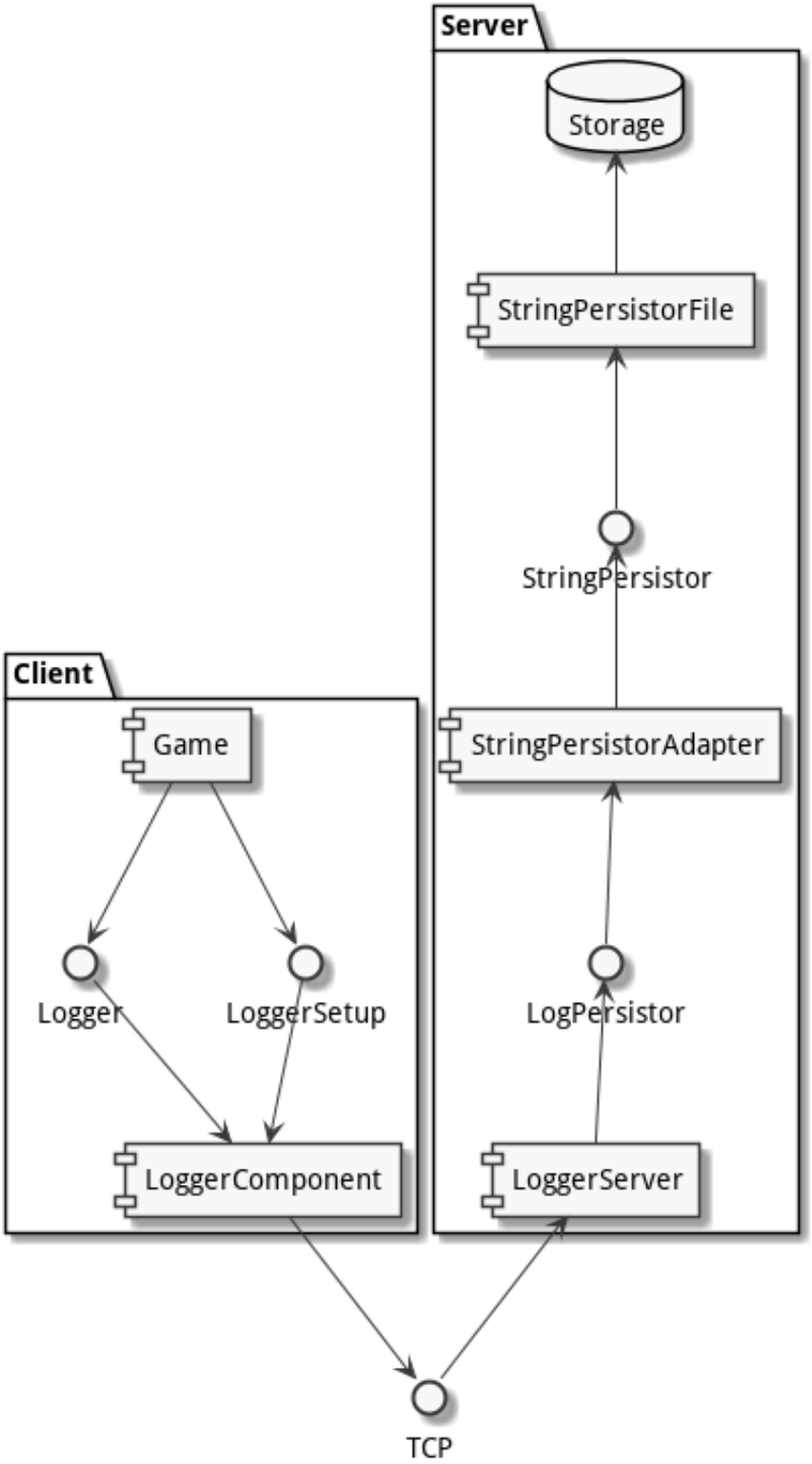

Abbildung 2: Komponentendiagramm

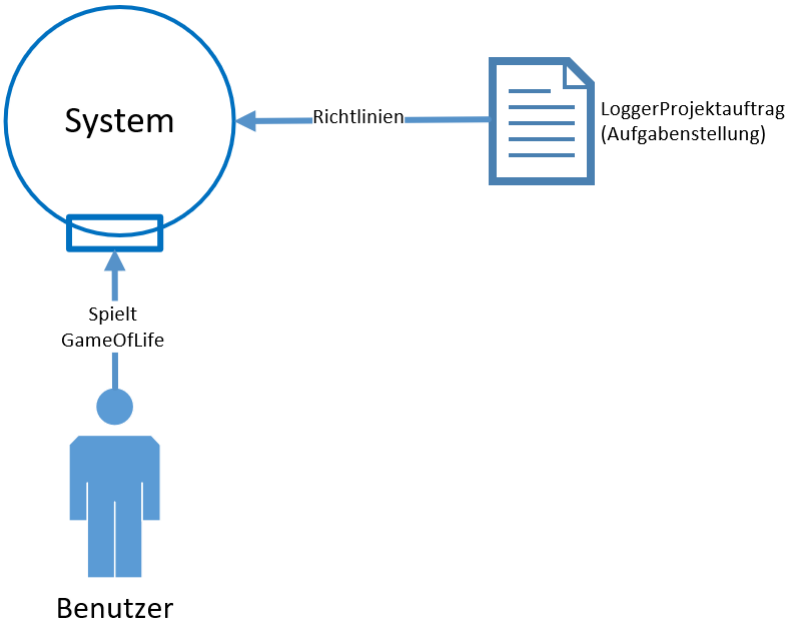

Abbildung 3: Kontextdiagramm

2 Architektur und Designentscheide

Die Architektur wurde weitgehendst vom Auftraggeber vorgegeben. Diese können grösstenteils im Dokument *Projektauftrag* nachgelesen werden bzw. sind bereits eingangs in diesem Dokument aufgeführt.

2.1 Modelle und Sichten

Beim Formulieren der Scrum-Stories (siehe Dokument *Scrum*) wurde die Anwendung von drei Perspektiven aus betrachtet:

- Ein *Anwender* führt die die *Game of Life*-Applikation aus und will seinen Spielstand geloggt wissen.
- Ein *Administrator* führt Konfigurationsarbeiten aus und will Parameter wie Log-Level und Serverkoordinaten einstellen können.
- Ein *Programmierer* will die Software-Komponenten zur Verfügung haben, um damit die Anforderungen von *Anwender* und *Administrator* umsetzen zu können: z.B. eine Logger-Komponente mit entsprechenden Interface, damit der die Logger-Aufrufe in die Anwendung einführen kann.

Da alle Projektmitarbeiter und selbst die Auftraggeber die Anwendung aus allen der genannten Perspektiven betrachten und diese gar von innen her kennen, erübrigt sich eine tiefgehende Diskussion

über Perspektiven.

2.2 Datenstrukturen

Für den Austausch von Log-Meldungen wurde die Datenstrukture `Message` definiert. Hierbei handelt es sich um eine serialisierbare Klasse, als Austauschcontainer für Log-Meldungen zwischen `LoggerComponent` und `LoggerServer` dient. Sie besteht aus den folgenden Attributen:

- `level`: Das Log-Level als `String` (`TRACE`, `DEBUG`, `INFO`, `WARNING`, `ERROR`, `CRITICAL`)
- `creationTimestamp`: Zeitpunkt der Erstellung als `java.time.Instant`
- `serverEntryTimestamp`: Zeitpunkt der Ankunft auf dem Server als `java.time.Instant`
- `source`: Die Quelle der Meldung als `String` im Format `host:port` (z.B. `localhost:1234`)
- `message`: Die eigentliche Log-Meldung als `String`

Die Klasse `Message` implementiert das Interface `LogMessage`, welches getter-Methoden für die genannten Parameter definiert.

Die Klasse `PersistedString` dient zum Abspeichern und späteren Auslesen von Log-Meldungen durch den `StringPersistor`. `Message`-Instanzen können über die Implementierungen des Interfaces `LogMessageFormatter` in `PersistedString`-Instanzen umgewandelt und wieder zurück geparst werden.

Wie diese Datenstrukturen genau von den verschiedenen Komponenten verwendet werden, kann dem Klassendiagramm der `stringpersistor`-Komponente entnommen werden.

TODO: Klassendiagramm erstellen

2.2.1 Anpassungen gegenüber Zwischenabgabe

Seit der Zwischenabgabe haben sich folgende Änderungen ergeben:

- Das (alte) Interface `LogMessage` mit der Implementierung `LogEntry` ist entfallen. Diese wurde als Austauschcontainer zwischen `LoggerServer` und `StringPersistor` verwendet, entsprach aber grösstenteils der gegenwärtigen `Message`-Klasse.
- Die `Message`-Klasse und das (neue) `LogMessage`-Interface wurden vom `loggercommon`-Projekt ins `stringpersistor`-Projekt verschoben. Grund dafür war die Änderung des Projektauftrags, wodurch der `StringPersistor` sowohl client- wie auch serverseitig referenziert werden darf.
  - Vorteil: Die grösstenteils redundante Datenstruktur `LogEntry` konnte zu Gunsten von `Message` entfallen. Es musste auch für den Viewer keine neue Datenstruktur eingeführt werden, da `Message` bereits alle Informationen enthält.
  - Nachteil: Da man aus dem `stringpersistor`-Projekt nicht auf Klassen des `loggercommon`-Projektes zugreifen darf, musste der Formatierungs- und Parsing-Code ebenfalls in das `stringpersistor`-Projekt verschoben wurden. Die Komponente `StringPersis-`

tor wurde mächtiger, das loggercommon-Projekt dadurch hinfällig. Die StringPersistor-Komponente wird zusätzlich als common-Codebasis missbraucht.

Fazit: Durch die Entscheidung zur Vereinfachung und Vereinheitlichung der Datenstrukturen konnte sehr viel Code entfernt werden. Auch die Konvertierungsschritte zwischen den verschiedenen Datenstrukturen entfielen. Das Projekt wurde ingsgesamt schlanker und übersichtlicher, wenn auch die Komponentenstruktur des Auftragsgeber etwas verwässert wurde. (Wobei die Aufteilung in Timestamp und Payload auf der Datenstruktur PersistedString eher Teil des Problems als Teil der Lösung war.)

2.2.2 Struktur der Log-Datei

Die Message-Instanzen werden über die LogMessageFormatter-Implementierungen SimpleFormatter und CurlyFormatter folgendermassen formatiert (Zeilenumbrüche aus Platgründen eingefügt, mit ~ markiert):

SimpleFormatter:

```
2018-05-11T17:20:10.703Z | [TRACE] [2018-05-11T17:20:10.953Z] ~
    [192.168.1.42:52413] Cell at [14;23] died
2018-05-11T17:20:10.727Z | [DEBUG] [2018-05-11T17:20:10.977Z] ~
    [192.168.1.42:52413] New generation
2018-05-11T17:20:10.728Z | [INFO] [2018-05-11T17:20:10.978Z] ~
    [192.168.1.42:52413] Window was resized
2018-05-11T17:20:10.728Z | [WARNING] [2018-05-11T17:20:10.978Z] ~
    [192.168.1.42:52413] Speed 'Hyper' was selected
2018-05-11T17:20:10.728Z | [ERROR] [2018-05-11T17:20:10.978Z] ~
    [192.168.1.42:52413] Connection to server lost
2018-05-11T17:20:10.728Z | [CRITICAL] [2018-05-11T17:20:10.978Z] ~
    [192.168.1.42:52413] Unable to log locally
```

CurlyFormatter:

```
2018-05-11T17:20:10.728Z | {received:2018-05-11T17:20:10.978Z} ~
    {level:TRACE} {source:192.168.1.42:52413} {message:Cell at [14;23] died}
2018-05-11T17:20:10.729Z | {received:2018-05-11T17:20:10.979Z} ~
    {level:DEBUG} {source:192.168.1.42:52413} {message:New generation}
2018-05-11T17:20:10.729Z | {received:2018-05-11T17:20:10.979Z} ~
    {level:INFO} {source:192.168.1.42:52413} {message:Window was resized}
2018-05-11T17:20:10.729Z | {received:2018-05-11T17:20:10.979Z} ~
   {level:WARNING} {source:192.168.1.42:52413} {message:Speed 'Hyper' was selected}
2018-05-11T17:20:10.729Z | {received:2018-05-11T17:20:10.979Z} ~
   {level:ERROR} {source:192.168.1.42:52413} {message:Connection to server lost}
2018-05-11T17:20:10.729Z | {received:2018-05-11T17:20:10.979Z} ~
    {level:CRITICAL} {source:192.168.1.42:52413} {message:Unable to log locally}
```

3 Schnittstellen

Im Rahmen des vorliegenden Projektes wurden mehrere Schnittstellen definiert. Diese wurden einerseits vom Auftraggeber festgelegt (`StringPersistor`) bzw. in den Interface-Kommitees definiert (`Logger` und `LoggerSetup`), wobei der Auftraggeber den entsprechende Rahmen vorgegeben hat; andererseits in den jeweiligen Projektteams ausgearbeitet.

3.1 Externe Schnittstellen

Um einen Austausch der Logger-Komponente mit Implementierungen anderer Gruppen zu ermöglich, wurden folgeden Schnittstellen definiert:

Schnittstelle	Version
`Logger`	1.0.0-SNAPSHOT
`LoggerSetup`	1.0.0-SNAPSHOT
`StringPersistor`	4.0.1

Die Schnittstellen `Logger`, `LoggerSetup` und `StringPersistor` werden hier nicht weiter beschrieben. Dere Dokumentation finden sich auf ILIAS. Von Java 1.8 zur Verfügung gestellte Schnittstellen sind hier nicht aufgeführt.

3.2 Interne Schnittstellen

Neben den externen, von aussen vorgegebenen Schnittstellen enthielt der Projektauftrag auch Anforderungen, welche die Definition weiterer Schnittstellen erforderlich machten. Dies sind:

Schnittstelle	Version
`LogPersistor`	1.0.0-SNAPSHOT
TCP-Protokoll	1.0.0-SNAPSHOT

Die Schnittstelle `LogPersistor` wurde selber definiert. Sie bildet die Grundlage für den `StringPersistorAdapter` und stellt dem `LoggerServer` eine für diesen geeigneten Schnittstelle zum `StringPersistor` dar. Hierzu wurde das Adapter-Pattern (GoF 139) implementiert. Auf das TCP-Protokoll wird im Dokument *TCP-Schnittstelle* näher eingegangen.

4 Implementierung

Auf die Implementierung der Anwendung soll an dieser Stelle nicht weiter eingegangen werden. Stattdessen sei hier auf die JavaDoc und auf das umfassende Klassendiagramm (Dokument *Klassendiagramm*) hingewiesen. Weitere Hinweise zur Implementierung finden sich in den Dokumenten *Testplan* und *TCP-Schnittstelle*.

4.1 Zwischenabgabe

Für die Zwischenabgabe ist weiter zu beachten, dass die gegenwärtige Implementierung *nicht* für die persistente Aufzeichnung von Logmeldungen ausgelegt ist. Die Logdateien werden derzeit im temporären Verzeichnis des jeweiligen Benutzers gehalten. Dies führt dazu, dass bei einem Systemneustart sämtliche Daten verloren gehen. Für die Schlussabgabe soll das System dahingehend erweitert werden, dass das Log-Verzeichnis auf dem Server konfigurierbar ist.

5 Environment-Anforderungen

Zum Ausführen der Anwendung wird client- wie serverseitig die Java SE Runtime Environment 8 benötigt. Tests mit dern Open-Source-Variante *OpenJDK 8* sind problemlos verlaufen.

Zum Kompilieren der Anwendung wird Maven und das HSLU-Nexus-Repository benötigt. Weiter wurde zum bequemen Packen und Ausführen von Client und Server jeweils ein `Makefile` geschrieben, das sich im Wurzelverzeichnis des Projekts `g05-game` (Client) bzw. `g05-logger` (Server) befindet[1].

Zum Ausführen der Anwendung auf zwei verschiedenen Rechnern (Client und Server) wird eine funktionierende Netzwerkverbindung zwischen diesen beiden verlangt. Die beiden Systeme sollten zum gleichen Netzwerk gehören, da eine wechselseitige TCP-Kommunikation über einen Port >1024 oftmals von Firewalls (d.h. an den Netzwerkgrenzen) unterbunden wird.

[1] Ironischerweise ist Maven angetreten um `ant` zu ersetzen, welches wiederum angetreten ist um `make` zu ersetzen. Die Lösung mit dem `Makefile` macht das Zusammensuchen und Ausführen (mit der entsprechenden `-classpath`-Option aber äusserst bequem, da `make` die Vorzüge leichtgewichtigen Dependency-Managements und der Shell kombiniert.