Testplan

Version 2.0.0 (Schlussabgabe)

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

12.05.2018

Inhaltsverzeichnis

1 Feature-Tests

Im Projektauftrag sind auf den Seiten 3 und 4 die Muss-Features für die Zwischen- bzw. Schlussabgabe aufgelistet. Im Folgenden ist aufgeführt, wie die einzelnen geforderten Features getestet werden.

1.1 Logger als Komponente

Der Logger muss als austauschbare Komponente implementiert werden. Im Code des Game-Projekts (`g05-game`) dürfen somit keine Referenzen auf die Klassen vom Projekt `g05-logger/logger-`

component vorhanden sein. Bei frühen Tests wurde die Klasse LoggerComponent direkt aus dem Spiel-Code heraus instanziert. Dies wurde von Maven mit einer entsprechenden Warnung quittiert. Seitdem für die Logger-Erstellung die Klassen LogManager, LoggerComponentSetup und Logging zusammen mit der flexiblen Konfiguration aus config.xml verwendet werden, taucht diese Warnung nicht mehr auf. Da die Implementierung als Komponente anhand eines funktionierenden Komponentenaustauschs demonstriert werden kann, erübrigen sich weitere Tests an dieser Stelle. Diese finden eine Woche nach der Schlussabgabe statt.

1.2 Filterung per Message-Level

Die Message-Levels sind in der Enum LogLevel in aufsteigender Schwere von 0:TRACE bis 5:CRITICAL definiert. Die Logger-Komponente nimmt zwar immer alle Logmeldungen ungeachtet ihres LogLevels entgegen, leitet aber nur diejenigen an den Logger-Server weiter, deren LogLevel mindestens so hoch ist wie das auf dem LoggerSetup konfigurierte. Beispiel: Ist das LogLevel auf dem LoggerSetup mit 3:WARNING gesetzt, werden Meldungen mit dem LogLevel 2:INFO und tiefer nicht geloggt.

Die Filterung per Message-Level wird automatisch im Testfall LoggerComponentTest (Methode testMessageLevelFiltering()) überprüft. Da LoggerComponent seine Nachrichten nicht direkt über einen Socket, sondern über einen ObjectOutputStream verschickt, können diese im Test direkt ausgelesen werden. Dazu erhält der Logger einen ObjectOutputStream, der auf einen PipedOutputStream schreibt. Dieser leitet die Daten an einen PipedInputStream weiter, der von einem ObjectInputStream dekoriert wird. So können Logmeldungen gleich nach dem Verschicken abgeholt werden, wobei Meldungen mit einem zu tiefen LogLevel nicht auftauchen.

1.3 Implementierung Logger und LoggerSetup

Die Implementierung des Logger-Interfaces erfolgt in der Klasse LoggerComponent. Die vier verschiedenen log()-Methoden werden vom LoggerComponentTest abgedeckt. Das Logger-Interface verfügt über zahlreiche Convenience-Methoden (critical(), warning() usw.), welche als default-Methoden direkt im Interface implementiert sind, indem sie eine der vier generischen log()-Methoden (mit LogLevel) aufrufen. Somit deckt der LoggerComponentTest die eigentliche Programmlogik sämtlicher log()-Methoden ab, wenn auch nicht sämtliche auf dem Interface definierten Methoden.

Die Klasse LoggerComponentSetup implementiert das LoggerSetup-Interface und hat die Aufgabe anhand einer LoggerSetupConfiguration eine LoggerComponent-Instanz mit Verbindung zu einem Logger-Server zu erstellen. Um die Erstellung einer funktionierenden LoggerComponent-Instanz zu testen, wird ein Logger-Server benötigt.

Da der Code der logger-server-Komponente nicht von logger-component aus aufgerufen werden kann und soll, muss ein einfacher Logger-Server als Fake nachgestellt werden. Dieser muss in einem eigenen Thread laufen, damit er gleichzeitig mit dem Client (Testfall) ausgeführt werden kann. Der Fake-Server beendet seine Arbeit nach nur einer entgegengenommenen Meldung.

Der erste Testfall `testCreateLogger()` startet einen Fake-Server, erstellt eine `LoggerComponentSetup`-Instanz, sendet dem Server eine Meldung und wartet anschliessend, bis der Server beendet ist. Der Fake-Server prüft, ob eine Meldung hereinkommt.

Der zweite Testfall `testServerSwitch()` macht das gleiche wie der vorherige Testfall, mit dem Unterschied, dass er noch einen zweiten Fake-Server aufstartet und sich nach einer erfolgreich übertragenen Meldung zum zweiten Server verbindet und diesem ebenfalls eine Meldung schickt.

Die Fake-Server verwenden einen zufälligen, freien, vergänglichen Port (> 1024). Dies ist wichtig, da man auf einer Integrationsumgebung nie weiss, welche Ports bereits besetzt sind. Weiter muss nach dem Aufstarten des Server-Threads darauf gewartet werden, dass dieser die `bind`-Operation erfolgreich ausgeführt hat, da sonst keine Verbindungen auf diesen erstellt werden können. Dies sind zwei gute Beispiele für Probleme, die beim Ausführen von Tests aus der Entwicklungsumgebung heraus nicht oder selten, auf einer Integrationsumgebung aber mit höherer Wahrscheinlichkeit auftreten.

1.4 Aufzeichnung durch Logger-Komponente und Logger-Server

Die kausal und verlässliche Aufzeichnung der Log-Ereignisse erfordert das Zusammenspiel sämtlicher im Projekt involvierter Komponenten: von einem Logger-Client über die Logger-Komponente und den Logger-Server bis zum String-Persistor. Die Tests zu den einzelnen Komponenten finden sich in den jeweiligen Abschnitten. Ein kompletter Systemtest, der alle Komponenten abdeckt (und nicht bloss mittels Fakes simuliert), ist nicht implementiert worden. Komplette Systemtests erfolgen ausschliesslich manuell, und zwar mit mehreren physischen Rechnern.

1.5 Austausch der Logger-Komponente

Derzeit (Stand 11. Mai 2018) ist noch keine Logger-Komponente einer anderen Gruppe für Tests verfügbar. Sobald dies der Fall ist, kann die `.jar`-Datei einer anderen Gruppe in ein lokales Verzeichnis kopiert und die Konfiguration (`config.xml`) entsprechend angepasst werden. Dieser Test soll in der Woche nach der Zwischenabgabe stattfinden.

1.6 Mehrere Logger auf einem Server

Ein Server muss die Logmeldungen von mehreren Clients gleichzeitig handhaben können. Für diesen Use-Case gibt es keinen Unit- oder Integrationstest, jedoch einen einfachen Client (Klasse `DemoLoggerClient` im `game`-Projekt), der einhundertmal eine Logmeldung mit fortlaufender Nummer im Abstand von 200 Millisekunden an den Server sendet.

Läuft ein Server, und werden mehrere Instanzen des `DemoLoggerClient` schnell nacheinander aufgestartet, kann man mit dem Aufruf von `tail -f *.log` im Log-Verzeichnis sehen, wie die Logmeldungen abwechselnd von verschiedenen Clients eintreffen.

1.7 Dauerhafte Speicherung

Die dauerhafte Speicherung der Meldungen ist in der Klasse `StringPersistorFile` implementiert. Der Testfall dazu heisst `StringPersistorFileIT`. Dieser testet das Schreiben (`save()`) und anschliessende Einlesen (`get()`) von Logmeldungen. Es werden verschiedene Zeilentrennzeichen getestet (`\n` für Unix und `\r\n` für Windows), indem die entsprechende Laufzeitvariable (`line.separator`) gesetzt wird. Da beim Loggen von Stack-Traces eine Logmeldung mehrere Zeilen umfassen kann, wird auch das Schreiben und Auslesen mehrzeiliger Logmeldungen getestet (`testMultiLineMessages()`).

Dem `StringPersistor` wird über die Methode `setFile(File)` eine Datei zum Schreiben und Einlesen von Logmeldungen angegeben. Dabei kann es passieren, dass auf die angegebene Datei nicht wie gewünscht zugegriffen werden kann. Die folgenden Tests stellen sicher, dass diese Situationen rechtzeitig erkannt und mit einer Exception behandelt wird:

- `testWriteWithoutFile()`: Schreiben, ohne vorher eine Datei angegeben zu haben
- `testReadWithoutFile()`: Lesen, ohne vorher eine Datei angegeben zu haben
- `testUnwritableFile()`: Schreiben auf eine schreibgeschützte Datei
- `testUnreadableFile()`: Lesen von einer lesegeschützten Datei
- `testNotExistantFile()`: Setzen einer nicht existierenden Datei

`StringPersistorFile` verwendet zum Einlesen der Logmeldungen eine Hilfsklasse namens `PersistedStringParser`. Diese liest Logmeldungen in dem Format aus, wie es in der Methode `PersistedString.toString()` implementiert ist. So beginnt eine Logmeldung jeweils mit einem ISO-formatiertem Timestamp. Es folgt ein Leerzeichen, eine Pipe (`|`) und wieder ein Leerzeichen, worauf die eigentliche Logmeldung (`Payload`) folgt.

Der bereits beschriebene Testfall `StringPersistorFileIT` deckt zwar schon einen grossen Teil vom `PersistedStringParser` ab. Dennoch wurde mit `PersistedStringParserTest` ein klassischer Unit-Test für die Klasse `PersistedStringParser` geschrieben. Bei der Umsetzung der `get()`-Methode von `StringPersistorFile`, welche den `PersistedStringParser` verwendet, sollte nämlich zunächst sichergestellt werden, dass die besagte Hilfsklasse soweit funktioniert. So gab es beim Entwickeln der `get()`-Methode Gewissheit darüber, dass etwaige Fehler in derselben und nicht andernorts zu suchen sind, was einem ein hektisches Hin und Her zwischen den einzelnen Klassen erspart.

1.7.1 Qualitätsmerkmale

Die `StringPersistor`-Komponente muss in der Lage sein pro Sekunde 1000 Nachrichten zu schreiben und 500 Nachrichten zu lesen. Dies wird mit dem `StringPersistorBenchmark` getestet, indem mit einem Timeout von 1000 Millisekunden die entsprechende Anzahl Nachrichten geschrieben und gelesen wird. Es wird zusätzlich überprüft[1], ob die Reihenfolge der geschriebenen und gelesenen

[1] Die Assert-Aufrufe finden in einer mit `@After` annotierten Methode statt, sodass der Benchmark nicht von der Überprüfung beeinflusst wird.

Nachrichten die gleiche ist. Dieser Benchmark erhöht zwar nicht die Testabdeckung in Codezeilen, behandelt aber eine mögliche Fehlerquelle.

1.8 `StringPersistor`-Komponente

Die `StringPersistor`-Komponente wird durch die bereits genannten Testfälle (siehe Abschnitt *Dauerhafte Speicherung*) abgedeckt.

1.9 Adapter für `StringPersistor`

Der Logger-Server soll den `StringPersistor` über einen Adapter verwenden. Der Logger-Server nimmt von der Logger-Component Instanzen der Klasse `Message` entgegen. Der `StringPersistor` arbeitet intern mit Instanzen der Klasse `PersistedString`. Für den Austausch der Logmeldungen zwischen Logger-Server und `StringPersistor` wurde das Interface `LogMessage` erstellt, das von der Klasse `Message` implementiert wird, und Methoden definiert um die folgendenden Felder auslesen zu können: `Level` (als `String`, da der Server die Enum `LogLevel` nicht kennt), `CreationTimestamp` und `ServerEntryTimestamp` (beide vom Typ `java.time.Instant`), `Source` (in der Form `host:port`) und `Message` (`String`).

Der `StringPersistorAdapter` wird durch den Testfall `StringPersistorAdapterIT` abgedeckt. In der Testmethode `testAdapter()` wird eine Logmeldung über das Adapter-Interface in eine temporäre Datei geschrieben. Die Logmeldung wird anschliessend aus der Datei ausgelesen und mit der ursprünglichen `LogMessage` verglichen. Dabei wird auch ein grosser Teil der Klasse `StringPersistor` mitgetestet.

1.10 Austauschbares Speicherformat

Für die Speicherstrategien der Logmeldungen gibt die Schnittstelle `LogMessageFormatter` die Methoden `format()` und `parse()` vor. Erstere formatiert eine `LogMessage` zu einem `PersistedString`, letztere parst einen `PersistedString` in eine `LogMessage`. Es gibt zwei Implementierungen: `SimpleFormatter` und `CurlyFormatter`. Da die beiden Implementierungen die gleiche Art von Problemen lösen, können sie durch einen gemeinsamen Testfall abgedeckt werden: Beim `LogMessageFormatterTest` handelt es sich um einen parametrisierten Test, dem die beiden Implementierungen als Parameter mitgegeben werden, und der dann sämtliche Test-Methoden auf beide Implementierungen ausführt.

1.11 Konfiguration

Die Konfiguration der Logging-Komponente (`config.xml`) wurde manuell getestet:

- Das Jar-File wurde über einen absoluten Pfad auf die `loggercomponent`-Jar-Datei im Maven-Repository gesetzt. Obwohl sich bei allen Gruppenmitgliedern der Pfad des Home-Verzeichnisses und somit des Maven-Repositories unterscheiden, konnte der Logger überall gefunden und instanziert werden.
- Das Log-Level wurde von `TRACE` auf `DEBUG` erhöht, sodass nur Logmeldungen ab der entsprechenden Stufe den Server erreichten.
- Der Host wurde für lokale Tests auf `localhost` und für Client/Server-Tests auf eine bestimmte IP-Adresse gesetzt.

1.12 Lokales Logging

Um das lokale Zwischenspeichern (etwa bei Netzwerkunterbrüchen, oder allgemein bei Nicht-Erreichbarkeit des Servers) zu testen, wurde folgendermassen vorgegangen:

1. Auf Host `A` wurde ein Logger-Server aufgestartet.
2. Auf Host `B` wurde der Game-Client gestartet.
    - Es wurden zahlreiche Log-Ereignisse ausgelöst, die entsprechend auf dem Server in der Logdatei auftauchten.
3. Der Logger-Server wurde beendet, währenddem das Spiel weiterlief.
    - Es wurden zahlreiche Log-Ereignisse in eine lokale Datei geschrieben.
4. Der Logger-Server wurde erneut aufgestartet.
    - Es wurde eine neue Log-Datei erstellt.
5. Der Game-Client konnte die Verbindung zum Server wieder aufnehmen.
    - Die zwischenzeitlich lokal gespeicherten Log-Einträge wurden an den Server übermittelt.
    - Die neu erstellte Log-Datei auf dem Server enthielt die Log-Ereignisse, die vormals lokal auf dem Client gespeichert waren. (Der Timestamp der Server-Ankunft wurde jedoch aktualisiert.)

Die lokale Zwischenspeicherung erfolgt aus Gründen der Einfachheit – `File.createTempFile()` erstellt zuverlässig eine schreibbare Datei mit einem eindeutigen Dateinamen – in ein temporäres Verzeichnis. Bei einem Neustart des Betriebssystems gehen die zwischenzeitlich lokal gespeicherten Log-Ereignisse somit verloren. Dies wird in Kauf genommen, da es sich hierbei um eine «Proof-of-Concept»-Implementierung und nicht um ein auslieferbares Produkt handelt.

1.13 Viewer mit RMI-Push

Für den Test des Viewers wurde folgende Testanordnung mit drei Computern verwendet:

1. Auf Host `A` wird als erstes der *Logger-Server* aufgestartet.
2. Auf Host `B` wird als zweites der *Game-Client* aufgestartet. Es werden sogleich einige Ereignisse erzeugt, die auf dem Server mittels `tail -f *.log` zur Kontrolle angezeigt werden.
3. Auf Host `C` wird als drittes der *Logger-Viewer* aufgestartet. Dieser erhält laufend die gleichen Logmeldungen, die auch auf der Ausgabe des *Logger-Servers* (Host `A`) erscheinen.

Damit wäre die Netzwerkfähigkeit des Logger-Viewers getestet. Zur Demonstration, dass mehrere Logger-Viewer pro Logger-Server verwendet werden können, wurde mangels eines vierten Computers ein weiterer *Logger-Viewer* auf Host `A` und `B` gestartet, was beides funktioniert hat.

Bei grossen Mengen an Logmeldungen – verursacht durch den feinsten Log-Level `TRACE`, ein maximiertes *Game of Life*-Fenster und die Geschwindigkeitseinstellung *Hyper* – konnten Instabilitäten und zeitliche Aussetzer bei der Testanordnung beobachtet werden.

Die Übereinstimmung der Ausgaben der verschiedenen Viewer-Instanzen wurde nur grob von Auge überprüft.

2 Testübersicht

Testfall	Art[2]	Testet
`DemoLoggerClient`	TP	Kompletten Anwendungsstack (ohne Viewer)
`LoggerComponentTest`	UT	`LoggerComponent`
`LoggerComponentSetupIT`	IT	`LoggerComponentSetup`, `LoggerComponent`
`LoggerServerIT`	IT	`ConcurrentLoggerServer`, `ConcurrentClientHandler`
`StringPersistorAdapterIT`	IT	`StringPersistorAdapter`, `StringPersistorFile`
`PersistedStringParserTest`	UT	`PersistedStringParser`
`StringPersistorFileIT`	IT	`StringPersistorFile`, `PersistedStringParser`
`StringPersistorBenchmark`	BM	`StringPersistorFile`
`LogMessageFormatterTest`	UT	`SimpleFormatter`, `CurlyFormatter`

2.1 Testabdeckung

Gemäss Jenkins wird gegenwärtig (Stand: 12. Mai 2018) eine Testabdeckung von ca. 50% (`logger`) bzw. 70% (`stringpersistor`) erreicht. Gegenüber der Zwischenabgabe (75% `logger` und 85% `stringpersistor`) ist das ein Rückschritt. Das hat verschiedene Gründe:

1. Für neue Anforderungen wurden teilweise keine neuen Testfälle implementiert:
   - Der Logger-Viewer ist ein reines GUI-Projekt und kann somit nicht auf dem Integrationsserver aufgestartet und automatisch getestet werden.
   - Der Wechsel vom Remote-Logger zum lokalen `StringPersistor` und zurück ist aufwändig zu testen, da ein Server aufgestartet, beendet und wieder neu aufgestartet werden müsste. Für einen aussagekräftige Test sollten auch verschiedene Rechner involviert sein, was auf der Integrationsumgebung nicht möglich ist. Ein entsprechender Test mit Fakes/Mocks wurde aus zeitlichen Gründen nicht in Angriff genommen.
   - Zum automatischen Testen der RMI-Schnittstelle müsste der Port der Registry konfigurierbar sein, damit ein Test überhaupt auf der Integrationsumgebung funktionieren

[2] TP: Testprogramm, UT: Unit-Test, IT: Integrationstest, BM: Benchmark

könnte (zwingende, aber nicht hinreichende Bedingung!). Diese Konfigurationsmöglichkeit wurde wegen Zeitmangels nicht eingabaut. Da die eigentlichen Probleme mit RMI erst bei der physischen Verteilung der Programmteile beginnen, wäre ein solcher Test nur wenig aussagekräftig. Die Mitgabe der Policy-Konfiguration wäre ein weiteres Problem, das aus zeitlichen Gründen nicht angegangen wurde.

2. Die Datenstrukturen wurden gegenüber der Zwischenabgabe vereinfacht. Die durch einen Testfall abgedeckte Klasse `LogEntry` verschwand. Ein Testfall der vereinfachten Datenstruktur `Message` (v.a. getter- und setter-Methoden) erübrigte sich, was etwas mehr nicht abgedeckten Code ergab.

Nicht abgedeckter Code findet sich u.a. in Getter- und Setter-Methoden und in der Ausnahmebehandlung. Manche Exceptions (gerade die `IOException`) können nur schwer und unzuverlässig provoziert werden. Für das `game`-Projekt wurden keine Tests geschrieben, da die eigentlichen Programmlogik nicht Gegenstand des VSK-Projektes ist und in diesem Rahmen auch nicht verändert wurde.

Die gegenwärtige Testabdeckung wird von der Gruppe 5 als knapp ausreichend betrachtet. Für eine grössere Abdeckung müssten wohl Mocking-Frameworks zum Einsatz kommen.