Testplan

Version 1.0.0

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

06.04.2018

Inhaltsverzeichnis

1 Feature-Tests

Im Projektauftrag sind auf Seite 3 die Muss-Features für die Zwischenabgabe 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` zunächst noch 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.

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 `LogLevel`s 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 daher ein einfacher Logger-Server gemockt werden. Dieser muss in einem eigenen Thread ausgeführt werden, damit er gleichzeitig mit dem Client (Testfall) ausgeführt werden kann. Der Mock-Server beendet seine Arbeit nach nur einer entgegengenommenen und entsprechend quittierten Meldung.

Der erste Testfall `testCreateLogger()` startet einen Mock-Server, erstellt eine `LoggerComponentSetup`-Instanz, sendet dem Server eine Meldung und wartet anschliessend, bis der Server beendet ist. Der Mock-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 Mock-Server aufstartet und sich nach einer erfolgreich übertragenen Meldung zum zweiten Server verbindet und diesem ebenfalls eine Meldung schickt.

Die Mock-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 durch Mocking simuliert), ist bisher nicht implementiert worden.

1.5 Austausch der Logger-Komponente

Derzeit (Stand 5. April 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.

1.6 Mehrere Logger auf einem Server

Ein Server muss die Logmeldungen von mehreren Clients gleichzeitig handhaben können. Bei der aktuellen Implementierung wird für jeden Client eine Datei im Temp-Verzeichnis erstellt, die den Hostnamen des Clients und die für den Socket verwendete Portnummer im Dateinamen enthält. Für diesen Use-Case gibt es bisher keinen «echten» Unit- oder Integrationstest, jedoch einen einfachen Client (Klasse `DemoLoggerClient` im `game`-Projekt), der einhundertmal eine einfache 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 Temp-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 `StringPersistorFileTest`. Dieser testet das Schreiben

(`StringPersistorFile.save()`) und anschliessende Einlesen (`StringPersistorFile.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 Klasse `PersistedString.toString()` implementiert ist. (Für die Schlussabgabe soll das Formatieren der Meldungen flexibler per Strategy-Pattern implementiert werden.) So beginnt eine Logmeldung jeweils mit einem ISO-formatiertem Timestamp. Es folgt ein Leerzeichen, eine Pipe (`|`) und wieder ein Leerzeichen, worauf die eigentliche Logmeldung folgt. (Diese enthält wiederum einen Timestamp, was auf dieser Semantikebene aber nicht von Belang ist. Schliesslich geht es hier darum, den Beginn einer neuen Logmeldung feststellen zu können.)

Der bereits beschriebene Testfall `StringPersistorFileTest` 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.

1.8 `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 darum ein neuer Übergabeparameter entwickelt, der sich auf die Aspekte beschränkt, die für Logger-Server und `StringPersistor` wichtig sind: Das Interface `LogMessage`. Dieses wird von der Klasse `LogEntry` implementiert, und definiert Methoden 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`) und `Message` (`String`). Zum komfortablen Erstellen einer `LogEntry`-Instanz wurde das Builder-Pattern implementiert (`LogEntry.Builder`), welches durch den Testfall `LogEntryTest` abgedeckt wird, indem sichergestellt wird, dass die gesetzten Werte (zwingende und optionale) korrekt gesetzt werden.

Der `StringPersistorAdapter` wird durch den Testfall `StringPersistorAdapterTest` 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 dem ursprünglichen `LogEntry` verglichen. Dabei wird auch ein grosser Teil der Klasse `StringPersistor` und der `LogEntry.Builder` mitgetestet.

2 Testübersicht

Testfall	Art[1]	Testet
`DemoLoggerClient`	TP	Kompletten Anwendungsstack
`MessageTest`	UT	`Message`
`LoggerComponentTest`	UT	`LoggerComponent`
`LoggerComponentSetupTest`	IT	`LoggerComponentSetup`, `LoggerComponent`
`LoggerServerTest`	IT	`ConcurrentLoggerServer`, `ConcurrentClientHandler`
`StringPersistorAdapterTest`	IT	`StringPersistorAdapter`, `StringPersistorFile`
`LogEntryTest`	UT	`LogEntry`
`PersistedStringParserTest`	UT	`PersistedStringParser`
`StringPersistorFileTest`	IT	`StringPersistorFile`, `PersistedStringParser`

2.1 Testabdeckung

Gemäss Jenkins wird gegenwärtig (Stand: 5. April 2018) eine Testabdeckung von ca. 75% (`logger`) bzw. 85% (`stringpersistor`) erreicht. 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 werden soll.

Die gegenwärtige Testabdeckung wird von der Gruppe 5 damit als zufriedenstellend betrachtet.

[1] TP: Testprogramm, UT: Unit-Test, IT: Integrationstest