Testplan

Verteilte Systeme und Komponenten

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

03.04.2018

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 die Logger-Erstellung in der Klasse `Logging` gekapselt und flexibel (anhand der Informationen aus `config.xml`) implementiert ist, 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 durch den `LoggerComponentTest` getestet. 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 (oder zumindest «gleichzeitig» im Sinne von *concurrent*, nicht zwingend im Sinne von *parallel*) 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 ihm 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 Verbindung auf diesen erstellt werden kann. 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

TODO: Derzeit (Stand 3. April 2018) ist noch keine Logger-Komponente einer anderen Gruppe verfügbar für Tests. 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 Persistente Speicherung

- `StringPersistorFileTest`
- `PersistedStringParserTest`

1.8 `StringPersistor`-Komponente

- siehe oben

1.9 Adapter für `StringPersistor`

- `StringPersistorAdapterTest`
- `LogEntryTest`

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`

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