Testplan

Verteilte Systeme und Komponenten

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

02.04.2018

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 `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 Unit-Test `LoggerComponentTest` mit der Methode `testMessageLevelFiltering()` überprüft. Da `LoggerComponent` seine Nachrichten nicht direkt über einen Socket sondern über einen `ObjectOutputStream` verschickt, können diese im Test bequem ausgelesen werden. Zu diesem Zweck erhält der Logger einen `ObjectOutputStream`, der auf einen `PipedOutputStream` schreibt. Dieser wiederum leitet die Daten an einen `PipedInputStream` weiter, der von einem `ObjectInputStream` dekoriert wird. So können Logmeldungen direkt nach

dem Verschicken abgeholt werden, wobei Meldungen mit einem zu tiefen `LogLevel` nicht auftauchen sollten.

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.

TODO: `LoggerSetupTest` beschreiben

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

- Versuch: Bearbieten der `config.xml`
- Derzeit noch keine andere Logger-Komponente verfügbar für Tests

1.6 Mehrere Logger auf einem Server

- Testprogramm mehrmals anwerfen
- Auf dem Server mit `tail -f` schauen, ob die Meldungen über verschiedene Hosts/Ports hereinkommen

1.7 Persistente Speicherung

- `StringPersistorFileTest`
- `PersistedStringParserTest`

1.8 `StringPersistor`-Komponente

- siehe oben

1.9 Adapter für `StringPersistor`

- `StringPersistorAdapterTest`
- `LogEntryTest`