Modul VSK 1801

Message-Logger

System-Spezifikation

Änderungsprotokoll

Rev.	Datum	Autor	Bemerkungen	Status
0.1.0	12.03.2018	Projektleiter	1. Entwurf	Fertig
0.1.1	18.03.2018	Projektleiter	Generelle Überarbeitung	Fertig
0.2.0	23.03.2018	Projektleiter	Anpassungen Kapitel Systemspezifikation	Fertig
1.0.0	08.04.2018	Projektleiter	Finalisierung Zwischenabgabe	Fertig
1.1.0	25.04.2018	Entwickler	Beschreibung Strategy, Adapter	Fertig
1.2.0	31.04.2018	Entwickler	Beschreibung System- & Integrationstest	Fertig

Inhaltsverzeichnis

1. Systemübersicht

<tbd> Übersicht Problemstellung und gewählter Lösungsansatz mit Begründung

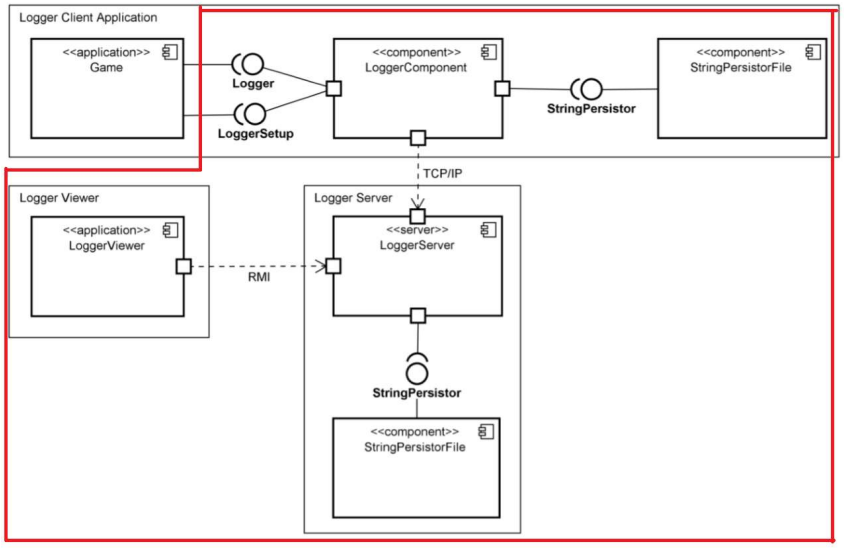


Abbildung 1: Kontextdiagramm

2. Architektur und Designentscheide

2.1. Modell(e) und Sichten
<tbd>

2.1.1. LoggerViewer

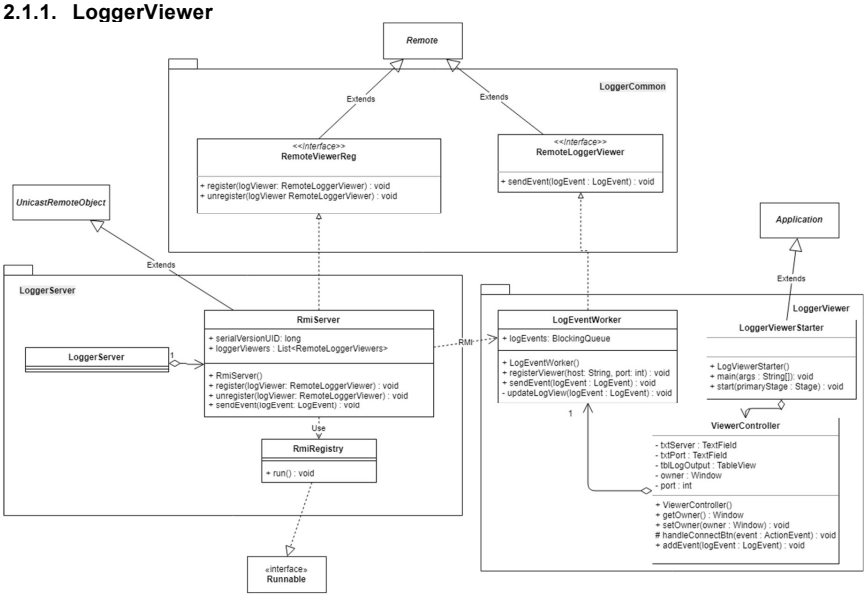


Abbildung 2: Klassendiagramm LoggerViewer

2.1.1.1. LoggerServer
Der LoggerServer wird erweitert, sodass ein RmiServer gestartet wird.

RemoteViewerReg
- Interface, welches vom Handler (RmiServer) implementiert werden muss.
- Stellt eine Methode zur Verfügung, dass sich ein LoggerViewer registrieren kann.
- Zusätzlich besteht eine Methode für die De-Registrierung bei nicht mehr erreichbaren Loggern.

RmiServer
- Startet die RMI Registry.
- LoggerViewer registriert sich bei dieser Klasse.
- Sendet die Log Events an die Viewer.

2.1.1.2. LoggerViewer
LoggerViewer hat ein GUI und registriert sich beim LoggerServer, damit LogEvents gesendet werden.

LoggerViewerStarter
- Startet das JavFFX GUI und den Controller für die View.

ViewerController

- Verwaltet das JavaFX GUI (Button Klicks, Eingaben).
- Erstellt einen LogEventWorker und wartet auf neue LogEvents

LogEventWorker
- Registriert sich beim LoggerServer
- Verwaltet die erhaltenen LogEvents
- Leitet die erhaltenen LogEvents an den Controller weiter.

2.1.1.3. RmiRegistry
Registrierung, damit der RmiServer sein RemoteObject publizieren kann und der LoggerViewer das Objekt finden kann.

RmiRegistry
- Erstellt eine RemoteRegistry

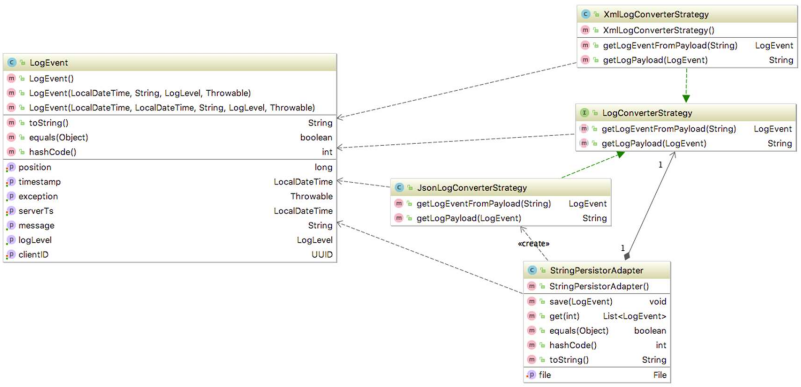
2.1.2. LoggerCommon
Enthält die Interfaces für den Handler und den Callback. Da dies im gleichen Package wie der RMI Server und Client ist, wird keine eigene Codebase benötigt.

2.2. Daten (Mengengerüst & Strukturen)
<tbd>

2.3. Entwurfsentscheide

2.3.1. Logger Interface

2.3.2. LogConverterStrategy
Um eine möglichst hohe Flexibilität in der Auswahl des geeigneten Dateiformats für die Persistierung von LogEvents zu gewährleisten, wurde das Strategy Pattern angewendet.



Der grösste Vorteil in dieser Implementierung, ist die Entkoppelung einer konkreten Implementation im StringPersistorAdapter. Dort wird letztlich nur noch die gewünschte

«Strategie» instanziiert. Die Umwandlung des LogEvent in ein entsprechendes Format wird in den entsprechenden Converter vorgenommen, die das LogConverterStrategy Interface implementieren.

2.3.3. StringPersistorAdapter

Das StringPersistor Interface verlangt als Parameter jeweils ein String für die Zeile in der Datei. Deshalb wurde gemäss dem Adapter Pattern ein Interface definiert (LogPersistor), welches dem StringPersistor Interface ähnlich ist, aber als Parameter ein LogEvent entgegennimmt. Der StringPersistorAdapter implementiert dieses Interface, adaptiert das LogEvent zu einem String um und gibt dieses dem StringPersistor weiter.

Folgendes Klassendiagramm zeigt ein UML Klassendiagramm für den Adapter:

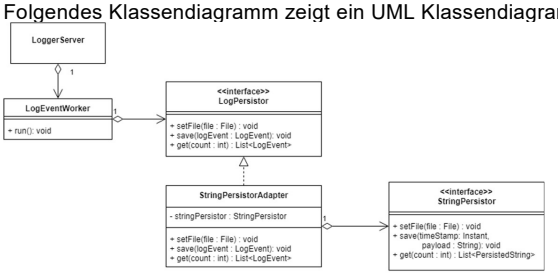


Abbildung 3: Adapter Pattern

Ablauf:

1. LogEventWorker initialisiert ein LogPersistor Objekt.
2. Über dieses Objekt werden die entsprechenden Logs übergeben und vom Adapter in einen String konvertiert.

2.3.4. Diskussionen

2.3.4.1. Uhren Synchronisation

a) Wo könnten logische Uhren zum Einsatz kommen? Begründen Sie in jedem Fall Ihre Antwort:

Die LogEvents müssen kausal korrekt gespeichert werden. Daher könnte man eine logische Uhr bei dem setzen des Zeitstempels auf dem LoggerServer verwenden: Der Zeitstempel auf dem Server muss neuer sein, als der Zeitstempel des Clients.

In unserem Projekt setzen wir keine logischen Uhren ein, da der Client sowie der Server jeweils auf einem Host gestartet werden, bei welchem die Zeit mittels NTP synchronisiert wird und daher korrekt ist.

b) Welchen Mehrwert ergeben die logischen Uhren im Projekt?

Die Kausalität der Inter-Prozess-Kommunikation würde durch eine logische Uhr gewährleistet sein. Es ist sichergestellt, dass Ereignisse in der gewünschten Reihenfolge ablaufen.
Falls z.B. der Server seine Zeit zurücksetzt und dadurch neue eingehende LogEvents einen älteren Zeitstempel als bisherige Events bekommen, führt dies zu einem Fehler in der Reihenfolge der Speicherung. Mit einer logischen Uhr würde dies nicht geschehen.

c) Welche logische Uhr (mit Lamport-Zeitstempel oder Vektor-Zeitstempel) ist sinnvoll, bezüglich des Mehrwerts vs. Aufwand?

Da jeweils nur zwei Prozesse miteinander gleichzeitig kommunizieren würde ein Lamport-Zeitstempel ausreichen. Eine logische Uhr mit Vektor Zeitstempel wäre sinnvoll, wenn Ereignisse von mehr als zwei Prozessen einen Einfluss aufeinander haben.

3. Schnittstellen

3.1. Externe Schnittstellen

Folgende externen Schnittstellen wurden für unser Projekt vorgegeben:

- Logger Interface
- StringPersistor

Logger Interface
Verwendete Version: loggerinterface-1.0.0-20180404.140420-16.jar

Dieses Interface definiert die Schnittstelle zwischen dem Game und der Logger Component.

Interface Spezifikation:
https://elearning.hslu.ch/ilias/goto.php?target=file_3687827_download&client_id=hslu

StringPersistor
Verwendete Version: stringpersistor-api-4.0.1.jar

Dieses Interface definiert, wie die LogEvents als String in eine Datei abgespeichert werden.

Interface Spezifikation:
https://elearning.hslu.ch/ilias/goto.php?target=file_3608366_download&client_id=hslu

3.2. wichtige interne Schnittstellen

Folgende internen Schnittstellen wurden definiert und verwendet:

- LogPersistor
- RMI
  - RemoteLoggerViewer
  - RemoteViewerReg
- LoggerBuffer

3.2.1. LogPersistor

Name	LogPersistor
Beschreibung	Schnittstelle, welche die Persistierung eines LogEvents in eine Datei beschreibt.
Typ	Java Interface

Beschreibung

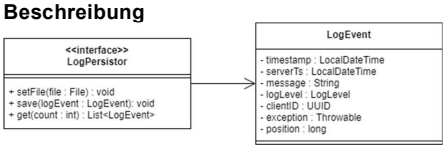


Abbildung 4: LogPersistor UML Diagramm

Der LogPersistor ist die Schnittstelle zwischen dem StringPersistor und dem Logger Server.

Die Schnittstelle ist gemäss dem Adapter Pattern implementiert und ermöglicht, dass der Server direkt ein LogEvent als Parameter übergeben kann. Der LogPersistor hat die Aufgabe, dass LogEvent so zu konvertieren, dass es als String dem StringPersistor übergeben werden kann.
Details für die einzelnen Methoden können der JavaDoc entnommen werden.

Entwurfsentscheidungen
- Der Adapter wurde bewusst so definiert, dass nur der Parameter-Typ von String auf LogEvent geändert werden muss und keine weitere Konfiguration im Code nötig ist.

Voraussetzungen
Die Voraussetzungen für diese Schnittstelle sind analog jener der StringPersistor Schnittstelle und können daher der entsprechenden Spezifikation entnommen werden.

3.2.2. RemoteLoggerViewer

Name	RemoteLoggerViewer
Beschreibung	Schnittstelle, welche definiert, wie der RMI Server ein Event an den RMI Client senden kann.
Typ	Java Interface

Beschreibung

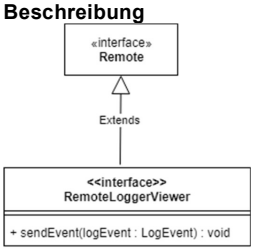


Abbildung 5: RemoteLoggerViewer UML Diagramm

Die Schnittstelle wird für die RMI Kommunikation zwischen LoggerServer und LoggerViewer benötigt. Der LoggerViewer implementiert diese Schnittstelle, um dem RMI Server eine Möglichkeit zu geben, einen «Callback» auf dem RMI Client auszuführen. Dadurch wird ein LogEvent vom RMI Server an den RMI Client übermittelt.

Details für die sendEvent Methode können der JavaDoc entnommen werden.

Entwurfsentscheidungen
- Die Schnittstelle wurde bewusst auf nur eine Methode beschränkt, um die RMI Kommunikation so einfach wie möglich zu gestalten.

Voraussetzungen
- Damit der Zugriff vom RMI Server auf den RMI Client funktioniert, muss sich der RMI Client zuerst beim RMI Server registrieren. Für die Registrierung muss der RMI Client als Objekt des Typs UnicastRemoteObject exportiert und anschliessend registriert werden.

3.2.3. RemoteViewerReg

Name	RemoteViewerReg
Beschreibung	Schnittstelle, welche definiert, welche Methoden der RMI Client auf dem RMI Server aufrufen kann.
Typ	Java Interface

Beschreibung

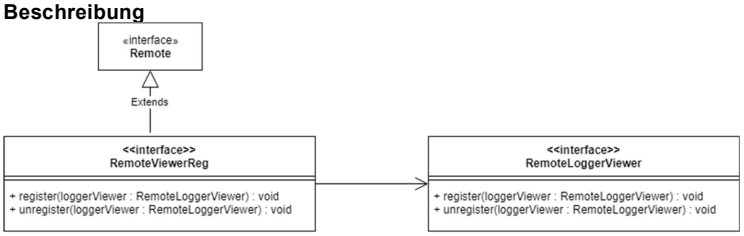


Abbildung 6: RemoteViewerReg UML Diagramm

Die Schnittstelle wird für die RMI Kommunikation zwischen LoggerServer und LoggerViewer benötigt. Der RmiServer implementiert diese Schnittstelle, um dem RMI Client eine Möglichkeit zu geben, sich beim RMI Server zu registrieren. Dadurch wird sichergestellt, dass der RMI Server mittels Push Prinzip LogEvents an den LoggerViewer senden kann.

Diese Schnittstelle muss in Kombination mit der Schnittstelle RemoteLoggerViewer implementiert werden, damit die RMI Kommunikation korrekt funktioniert.

Details für die Methoden können der JavaDoc entnommen werden.

Entwurfsentscheidungen

- Die Schnittstelle bietet die Möglichkeit zur De-Registrierung von LoggerViewers. Diese Methode sollte direkt vom RMI Server verwendet werden, wenn der RMI Client nicht mehr erreichbar ist.

Voraussetzungen

- Keine speziellen Anforderungen

3.2.4. LoggerBuffer

Name	LoggerBuffer
Beschreibung	Schnittstelle, welche das Persistieren von LogEvents erlaubt.
Typ	Java Interface

Beschreibung

<TODO: UML generieren/einfügen>

Die Schnittstelle wird dazu verwendet, LogEvents auf dem Client persistieren zu können, damit diese auch einen allfälligen Neustart des Clients überleben.

Entwurfsentscheidungen

- Die Schnittstelle arbeitet mit dem position-64-bit-Integer des LogEvents, um ihren internen Zustand zu verarbeiten.

Voraussetzungen

- Sofern sie verwendet wird, ist ein Event mit addLogEvent abzuspeichern bevor es an den Server übermittelt wird, und ein allfälliges AcknowledgePacket des Servers ist mit einem Aufruf an setLastPosition wieder an den LoggerBuffer zu melden.

3.2.5. TCP/IP Schnittstelle: Client – Server

- **Art des Message Passings:** ObjectOutputStream/ObjectInputStreams over TCP/IP
- **Verbindungsaufbau:** ausgehend vom Client aus
- **Konfiguration:** Client via LoggerSetup des Logger Interfaces; Server keine Konfiguration möglich
- **Protokoll:** Versand von zwei Klassen (LogEvent und AcknowledgmentPacket) in einem Request-Response-Verfahren
- **Datenformat:** Proprietär; abstrahiert durch die JCL

Verbindung
Der Client baut eine ausgehende Verbindung zum Server via TCP auf. Die Adresse und den Port bezieht der Client dabei vom LoggerSetup des Interfaces mit dem er initialisiert wurde; serverseitig ist keine Konfiguration möglich und der Server hört immer auf Port 10000 auf allen Netzwerkadaptern.
Der Client versucht bei einem allfälligen Verbindungsfehler, die Verbindung zum Server selbstständig wiederaufzunehmen.

Protokoll
Das Protokoll ist sehr simpel gehalten und basiert auf Javas ObjectInputStreams bzw. ObjectOutputStreams. Der eigentliche Datenaustausch findet damit durch ein JCL-proprietäres Format statt. In der aktuellen Implementation schickt der Client lediglich LogEvents, während der Server lediglich AcknowledgmentPackets sendet. Es existiert weder eine Handshake- noch eine Heartbeatmechanik, da diese das Protokoll unnötig verkompliziert hätten und für den geplanten Einsatz des Loggers weder gefordert noch benötigt werden.
<insert Grafik für LogEvent>
<insert Grafik für AcknowledgmentPacket>
Nachdem die TCP Verbindung aufgebaut wurde, dürfen beide Parteien anfangen, nach Belieben Pakete zu versenden. Pakete, welche von der anderen Seite nicht gelesen oder deserialisiert werden können, werden nicht als Protokollfehler gewertet. Auf fehlerhafte Pakete darf via Standard Output, Standard Error oder SelfLog (Teil des Interfaces) hingewiesen werden.
Die Kommunikation erfolgt dadurch, dass der Client ein LogEvent von einem Logger bekommt, gegebenfalls mit einem 64-bit Integer (position) anreichert, und über den ObjectOutputStream an den Server schickt. Der Server liest das Paket mit einem ObjectInputStream, legt es bei sich intern in eine Queue zur Verarbeitung und bestätigt den Empfang (ebenfalls wieder über einen ObjectOutputStream) mit einem Acknowledgment-Paket, welches den empfangenen position-Wert enthält. Der Client kann dann, mit seinem eigenen ObjectInputStream lesen und somit den serverseitigen Empfang des Pakets bestätigen.

Das Protokoll schreibt nicht vor, welche Bedeutung der Wert des position-Feldes hat. Es ist lediglich notwendig, dass der Server den Erhalt des Paketes mit dem Versand eines AcknowledgmentPackets quittiert, welches den entsprechenden position Wert enthält. Die Zustellung des AcknowledgmentPackets basiert auf "best effort". Sollte ein Client nicht in der Lage sein, das Paket zu erhalten, oder die Verbindung abbrechen, so wird das

AcknowledgmentPacket nicht wiederholt. In der aktuellen Implementierung schickt der Client die Position des internen Buffers um damit zu erreichen, dass jedes LogEvent at-least-once an den Server geschickt wurde.
Da als grundlegendes Protokoll TCP verwendet wird, kann davon ausgegangen werden, dass die Reihenfolge der Pakete eingehalten wird und es somit nicht möglich ist, dass eine Bestätigung für ein LogEvent geschickt wird, welches ein nicht-erhaltenes Paket überspringen würde.

Beispiel (nicht-protokollrelevante Felder wurden entfernt, ausser "message"):

```
Client -> Server: LogEvent { message = "Message 1 ", position = 10 }
Server -> Client: AcknowledgmentPacket { position = 10 }
Client -> Server: LogEvent { message = "Message 2", position = 20 }
Client -> Server: LogEvent { message = "Message 3", position = 30 }
Server -> Client: AcknowledgmentPacket { position = 20 }
Client -> Server: LogEvent { message = "Message 4", position = 40 }
Server -> Client: AcknowledgmentPacket { position = 30 }
Server -> Client: AcknowledgmentPacket { position = 40 }
```

3.3. Benutzerschnittstellen

<tbd>

3.x.1. Schnittstelle A

3.x.1.1. Steckbrief

Genauer Name der Schnittstelle, Kurzbeschreibung der Funktionalität, ggf. Autoren und Besitzer (zwischen wem wurde die Schnittstelle ausgehandelt?), ggf. Version

3.x.1.2. Interaktionen

Je nach Schnittstellenart Operationen (z.B. Funktionen, Methoden) oder Datenaustausch (z.B. Nachrichten).

Einschränkungen und Voraussetzungen, Berechtigungen, zeitliche Einschränkungen, parallele Benutzung, Voraussetzungen zur Nutzung

Generelles zur Fehlerbehandlung, mögliche Fehlersituationen als auch deren Behandlung.

Je Interaktion:

- *Beschreibung der Semantik (Fachlichkeit)*
  Diagramm und/oder Beschreibung der fachlichen Abläufe
  Fachliche Bedeutung der Daten
  Nebenwirkungen, Konsequenzen
- *Beschreibung der Technik*
  Methoden/Funktionen, Daten und Datenformate, Gültigkeits- und Plausibilitätsregeln, Codierung, Zeichensätze
- *Fehlerbehandlung*

3.x.1.3. Einstellungen

Kann das Verhalten der Schnittstelle oder der Ressourcen verändert oder konfiguriert werden?

Mögliche Konfigurationsparameter

3.x.1.4. Qualitätsmerkmale

Aussagen über Qualitätsmerkmale, an die Implementierer gebunden sind und auf die sich Nutzer verlassen können.

Welche Qualitätseigenschaften wie Verfügbarkeit, Performance, Sicherheit gelten für diese Schnittstelle? Neudeutsch heisst dieser Teil der Schnittstellenbeschreibung Quality-of-Service (QoS) Requirements.

Mengengerüste Laufzeit Durchsatz / Datenvolumen Verfügbarkeit

3.x.1.5. Entwurfsentscheidungen

Fragestellungen, Einflüsse, Annahmen, Alternativen und Begründungen für Entwurfsentscheidungen im Zusammenhang mit der Schnittstelle, falls angebracht

Welche Gründe haben zum Entwurf dieser Schnittstelle geführt? Welche Alternativen gibt es, und warum wurden diese verworfen?

3.x.1.6. Beispielverwendung

Pseudocode oder Quelltext bei Operationen, Beispieldaten bei Datenformaten

Hinweise und / oder Beispiele zur Benutzung dieser Schnittstelle

4. Environment-Anforderungen

Für den Logger werden folgende Anforderungen gestellt:

Betriebssystem:

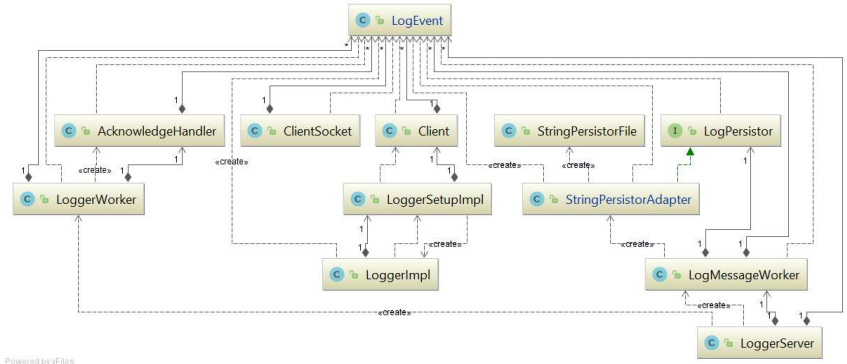
- Alle Betriebssysteme, welche Java verwenden können, werden unterstützt.

Hardware:

- Die Hardware muss die Systemanforderungen für Java erfüllen.
- Es muss ein entsprechendes Ausgabe-Gerät für die Anzeige des GUIs vorhanden sein.

4.1. Systemspezifikation

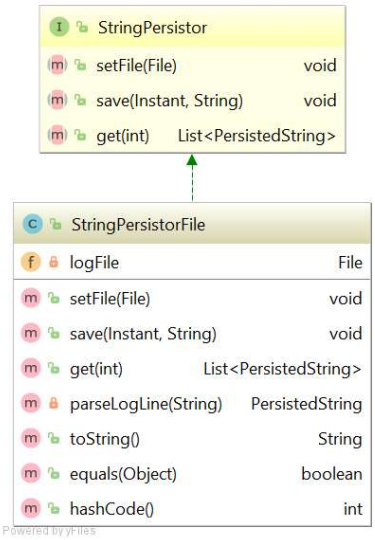
Dieses Kapitel beschreibt den Logger anhand von UML Diagrammen.



4.1.1. StringPersistor

Kommentiert [KPI1]: Diagramm anpassen

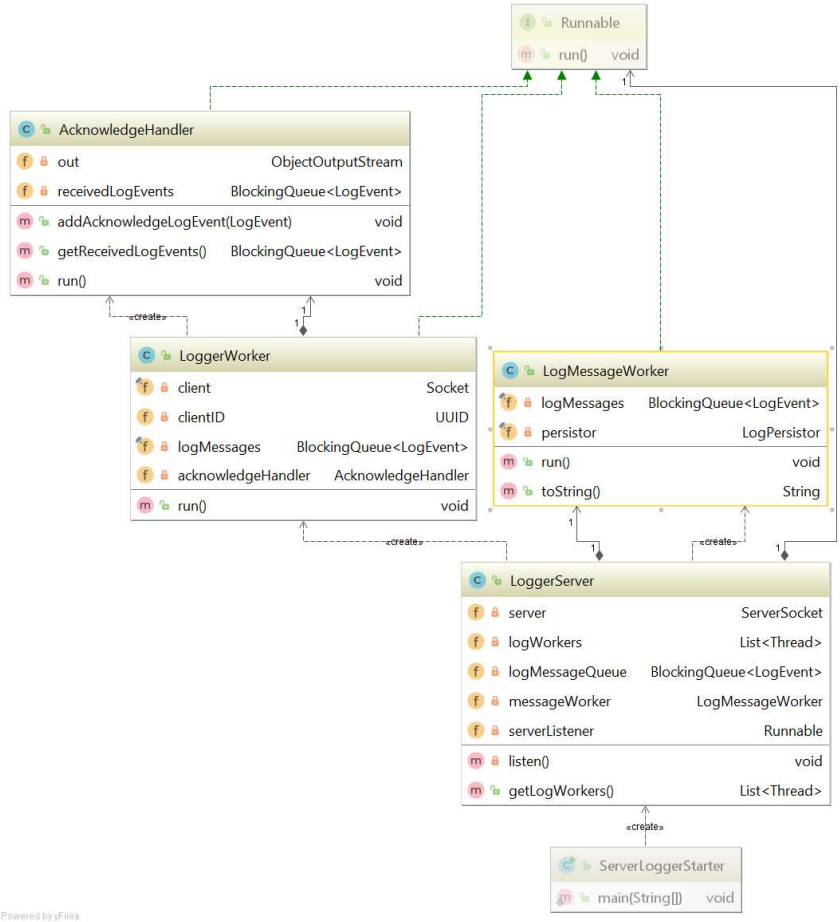
- StringPersistorFile implementiert das Interface «StringPersistor»
- Wird von Server verwendet um Logs zu persistieren.



4.1.2. LoggerServer

- Wartet auf Clients und nimmt LogEvents von diesen entgegen.
- Persistiert die LogEvents in einem File anhand des LogPersistors.
- «LoggerServer» erstellt einen Listener, um auf Clients zu warten.
- Ein weiterer Thread (LogMessageWorker) verarbeitet die erhaltenen Events, persistiert diese und sendet ein AcknowledgePacket an den Client zurück.

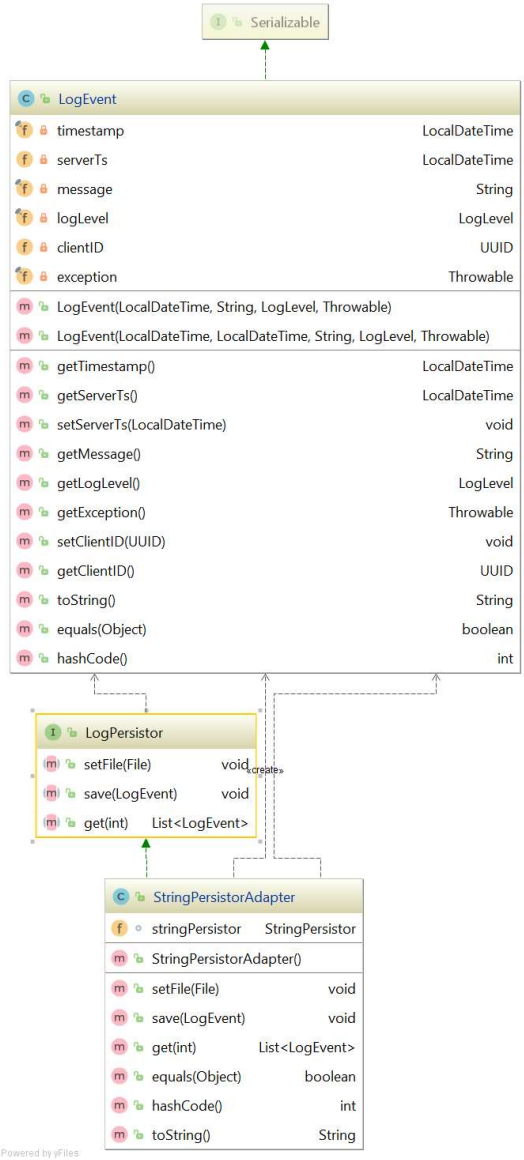
Kommentiert [KPI2]: Beschreibung auf neue Logger Architektur anpassen
Diagramm anpassen



4.1.3. LoggerCommon

Kommentiert [KPI3]: Diagramm anpassen

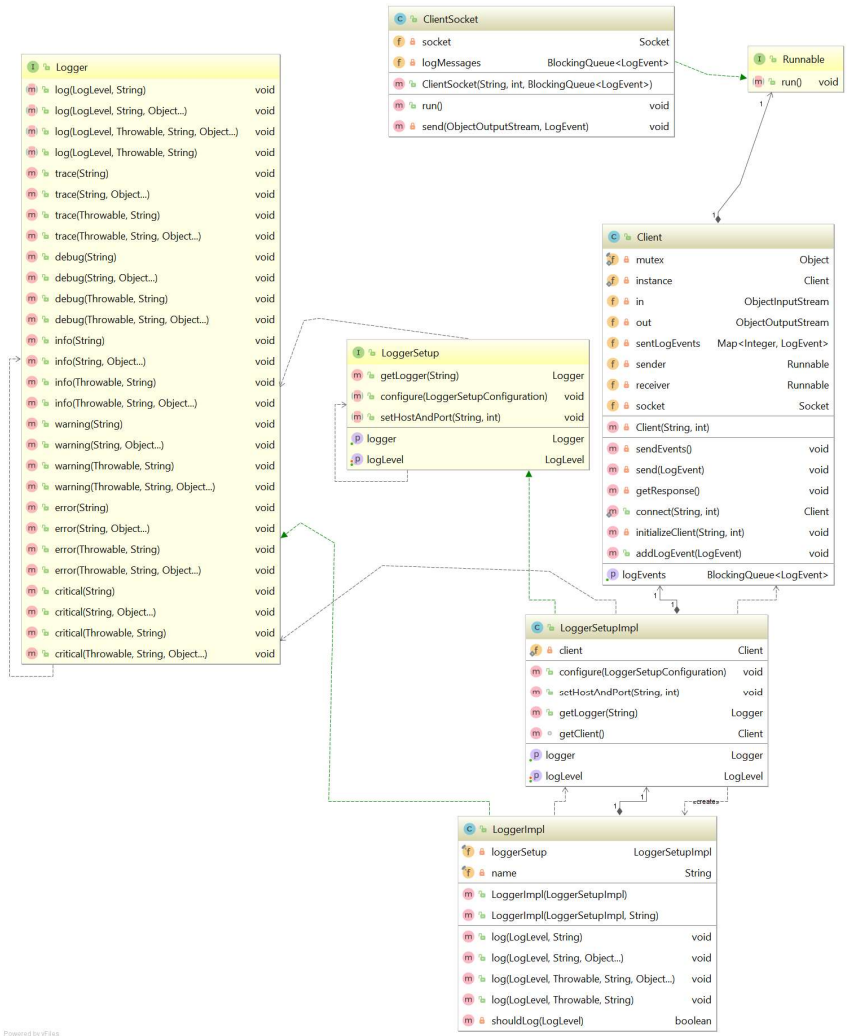
- Elemente, welche von Server und Client verwendet werden.
- LogEvent: Event welches Details über den Log-Eintrag enthält
- LogPersistor: Interface für die Persistierung von LogEvents.
- StringPersistorAdapter: Adapter, welcher LogEvents in das korrekte Format für den StringPersistor bringt.



4.1.4. LoggerComponent

Kommentiert [KPI4]: Diagramm anpassen

- Implementiert das Logger-Interface.
- Schnittstelle zwischen Game und Server.
- Stellt Logger durch ein LoggerSetup zur Verfügung.


ClientSocket
socket Socket
logMessages BlockingQueue<LogEvent>
ClientSocket(String, int, BlockingQueue<LogEvent>)
run() void
send(ObjectOutputStream, LogEvent) void
Runnable
run() void
Logger
log(LogLevel, String) void
log(LogLevel, String, Object...) void
log(LogLevel, Throwable, String, Object...) void
log(LogLevel, Throwable, String) void
trace(String) void
trace(String, Object...) void
trace(Throwable, String) void
trace(Throwable, String, Object...) void
debug(String) void
debug(String, Object...) void
debug(Throwable, String) void
debug(Throwable, String, Object...) void
info(String) void
info(String, Object...) void
info(Throwable, String) void
info(Throwable, String, Object...) void
warning(String) void
warning(String, Object...) void
warning(Throwable, String) void
warning(Throwable, String, Object...) void
error(String) void
error(String, Object...) void
error(Throwable, String) void
error(Throwable, String, Object...) void
critical(String) void
critical(String, Object...) void
critical(Throwable, String) void
critical(Throwable, String, Object...) void
Client
mutex Object
instance Client
in ObjectInputStream
out ObjectOutputStream
sentLogEvents Map<Integer, LogEvent>
sender Runnable
receiver Runnable
socket Socket
Client(String, int)
sendEvents() void
send(LogEvent) void
getResponse() void
connect(String, int) Client
initializeClient(String, int) void
addLogEvent(LogEvent) void
logEvents BlockingQueue<LogEvent>
LoggerSetup
getLogger(String) Logger
configure(LoggerSetupConfiguration) void
setHostAndPort(String, int) void
logger Logger
logLevel LogLevel
LoggerSetupImpl
client Client
configure(LoggerSetupConfiguration) void
setHostAndPort(String, int) void
getLogger(String) Logger
getClient() Client
logger Logger
logLevel LogLevel
«create»
LoggerImpl
loggerSetup LoggerSetupImpl
name String
LoggerImpl(LoggerSetupImpl)
LoggerImpl(LoggerSetupImpl, String)
log(LogLevel, String) void
log(LogLevel, String, Object...) void
log(LogLevel, Throwable, String, Object...) void
log(LogLevel, Throwable, String) void
shouldLog(LogLevel) boolean

5. Test-Architektur

Der Grossteil des Testings erfolgt über automatisierte Tests, welche im Kapitel 5.1 behandelt werden. Zusätzlich werden manuelle Tests durchgeführt mit der Game of Life Applikation, um das korrekte Verhalten des Loggers zu verifizieren.

5.1. Automatisierte Tests

5.1.1. Client

Der Client wird lediglich mit JUnit darauf getestet, dass die Instanzierung und der Versand einer Nachricht ohne Ausnahme funktioniert.

5.1.2. LoggerBuffer

Die Funktionalität wird mit JUnit getestet. Alle Tests erfolgen gegen das echte Dateisystem, wobei die Dateien jedoch vor und nach jedem Test gelöscht werden.

5.1.2.1. incrementalTest

Es werden mehrere LogEvents reingeschrieben und verifiziert, dass für jedes Event setPosition mit einer jeweils höheren Zahl aufgerufen wird.

5.1.2.2. closeAndOpenCase

Es wird ein Buffer erzeugt, Events reingeschrieben und nachher wird der Buffer geschlossen. Danach wird der Buffer mit der gleichen Datei wieder geöffnet und überprüft, ob alle Events herausgelesen werden können. Dies verifiziert, dass wir die persistierten Events auch über Bufferinstanzen hinweg (sprich, Programmneustarts) beibehalten und lesen können.

5.1.2.3. positionTest

Es wird ein Buffer erzeugt, Events reingeschrieben, das erste Event als "gelesen" markiert, und nachher wird der Buffer geschlossen.
Daraufhin werden kontinuierlich Buffer erzeugt, die Events herausgeholt (und verglichen mit den ursprünglich übergebenen), und danach das erste Event als gelesen markiert. Dies wird wiederholt, bis kein ungelesenes Event mehr im Buffer ist.

5.1.3. StringPersistorTest

Die Funktionalität wird mit JUnit getestet. Alle Tests erfolgen gegen das echte Dateisystem.

5.1.3.1. testSaveSuccess

Es wird ein Log Event gespeichert und nachher wieder herausgeholt. Die Meldung wird danach auf Gleichwertigkeit überprüft.

5.1.3.2. testGetSuccess

Es wird ein Log Event gespeichert und nachher verifiziert, dass nur ein Event effektiv gespeichert wurde.

5.1.3.3. testGetSuccess2

Es werden zwei Log Events gespeichert und nachher verifiziert, dass effektiv zwei Events gespeichert wurden.

5.1.3.4. testGetSuccess3

Es wird ein Log Event gespeichert und nachher verifiziert, dass kein Event gelesen wird, sofern 0 Events angefragt werden.

5.1.3.5. testGetSuccess4
Es wird ein Log Event gespeichert und nachher verifiziert, dass dieses Event identisch ist, mit dem gespeicherten.

5.1.3.6. testHashCode
Es wird überprüft, dass zwei Instanzen des Persistors mit dem gleichen File auch den gleichen Hashcode aufweisen.

5.1.3.7. testEquals
Es wird überprüft, dass zwei Instanzen des Persistors mit dem gleichen File identisch sind.

5.1.3.8. testToString
Es wird überprüft, dass zwei Instanzen des Persistors mit dem gleichen File auch den gleichen String ergeben bei einem Aufruf mit toString.

5.1.4. LoggerServerListener
Die Funktionalität wird mit JUnit getestet. Alle Tests erfolgen mit echten Sockets.

5.1.4.1. addAndRemoveNewClientTest
Ein neuer Client wird hinzugefügt und verifiziert, dass er auch tatsächlich bedient wird.

5.1.5. RmiServer
Die Remote Viewers werden gemockt mit Mockito, die Funktionalität an sich mit JUnit verifiziert. Für jeden Test wird bereits ein Viewer an den Server gehängt.

5.1.5.1. registerTestNew
Es wird ein zusätzlicher Viewer drangehängt und verifiziert, dass der Server zwei bedient.

5.1.5.2. registerTestAlreadyExists
Es wird verifiziert, dass ein RemoteViewer nur einmal hinzugefügt wird.

5.1.5.3. unregisterTest
Es wird ein RemoteViewer entfernt und verifiziert, dass keiner mehr am Server ist.

5.1.5.4. sendEventTest
Es wird ein Event an den Server gesendet und verifiziert, dass die Clients dieses erhalten haben.

5.2. Integrationstest

5.3. Systemtest
Für den Systemtest wird der Loggerserver sowie die Game of Life-Applikation inklusive Logger-Client auf eine Azure VM deployed. Zusätzlich können auf einer oder mehreren Entwicklermaschinen die Game of Life-Applikation für die VM konfiguriert werden. Der Logger-Server auf der VM wird gestartet, gefolgt von den Game of Life-Instanzen. Nachdem alle Instanzen gestartet sind und via SelfLog-Funktionalität überprüft wurde, dass keine Fehler aufgetreten sind, werden in jeder Instanz Aktionen getätigt, welche Logevents auslösen und auf dem Server verifiziert, dass diese angekommen und korrekt geloggt wurden.