HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

Runtime-Object-Visualization

Studienarbeit

Abteilung Informatik
Hochschule für Technik Rapperswil

Herbstsemester 2015

Autor(en): Adrian Anthamatten
Jeyanthan Ravindran
Betreuer: Dipl. El. Ing. HTL Thomas Letsch

Inhaltsverzeichnis

Runtime-Object-Visualization

Studenten

- Adrian Anthamatten
- Jeyanthan Ravindran

Einführung

Ein bestehendes Tool kombiniert UML-Klassen- und Objektdiagramme im 3-dimensionalen Raum in eine einzige Sicht: Object-Graph-Visualizer OGV

Die Objekte werden in jenem Tool interaktiv vom Benutzer erzeugt und bearbeitet.

Idee ist nun, von einer beliebigen Java-Applikation die aktuellen Objektdaten zur Laufzeit auszulesen und diese an den Object-Graph-Visualizer zu schicken, welcher dann die Objekt-Daten wiederum entsprechend darstellt.
Damit muss der Benutzer diese Daten nicht mehr 'von Hand' am Object-Graph-Visualizer eingeben.

Aufgabenstellung

1. Es soll eine Evaluation durchgeführt werden um die Varianten zum Auslesen der Objekt-Informationen einer Java-Applikation zur Laufzeit (z.B. JVMTI, AspectJ, etc.) zu diskutieren und einander gegenüber zustellen.

2. Die Realisierung der gewählten Variante zum Auslesen der Objekt-Informationen und eine entsprechende Anbindung zum Object-Graph-Visualizer realisieren.

Unter Berücksichtigung von aktuellen Software-Engineering-Methoden soll ein geeigneter Entwicklungsprozess definiert und darauf basierend das Projekt realisiert werden.

Technologien

- Java
- Enterprise Architect

Generelles

- Die Vorgaben der Abteilung Informatik [1] sind einzuhalten, insbesondere die Anleitung zur Dokumentation [2].
- Die "Generelle Richtlinien für Studien- und Bachelorarbeiten" [3] sind einzuhalten.
- Mit dem CASE-Tool Enterprise Architect ist ein UML-Modell zu führen, welches synchron mit den Programm-Sourcen und der Projekt-Dokumentation ist.
- Ein Java-Entwickler muss mit der Projekt-Dokumentation in die Lage versetzt werden, die Applikation in Betrieb zu nehmen und weiter entwickeln zu können.

Termine

- Montag, 14.09.15 Beginn der Studienarbeit
- Freitag, 18.12.15 12:00 Uhr Abgabe der Studienarbeit

Betreuung

- Betreuer
  Thomas Letsch
  tletsch@hsr.ch
  055 - 22 24 567 (HSR Büro 5.204); 055 - 214 43 50 (Geschäft)
- Besprechungen
  Wöchentliche Besprechung jeweils Montag 17:15 Uhr

Referenzen

[1] www.hsr.ch>HSR-intern>Bachelor-Studiengänge>Informatik>Allgemeine Infos Bachelor- und Studienarbeiten
https://www.hsr.ch/Allgemeine-Infos-Diplom-Bach.4418.0.html
[2] DokuAnleitungBA_SA_140210.pdf
[3] "Generelle Richtlinien für Studien- und Bachelorarbeiten" (v1.7 / 19.08.2015, Thomas Letsch)

Rapperswil, 14.September 2015

Thomas Letsch

Abstract

Ausgangslage

Ziel dieser Arbeit ist das Auslesen von Java-Objektinformationen (z.B. Änderung der Attribute, Initialisierung von Instanzen) einer bestehender Applikation (z.B. .jar-File) zur Laufzeit. Für den Object-Graph-Visualizer (OGV) soll ein Interface implementiert werden, damit dieser später um die Fähigkeit erweitert werden kann, diese Daten darzustellen. Die Verbindung zu diesem Interface soll technologieunabhängig sein, damit die Objektinformationen von verschiedenen Programmiersprachen (neben Java z.B. auch von C#, C++, etc.) im OGV dargestellt werden können.

Vorgehen

In der Evaluation wurden Technologien für das Auslesen von Java-Objektinformationen miteinander verglichen. Bei der Gegenüberstellung dieser Technologien konnte der Java Debug Interface (JDI) die gestellten Anforderungen am besten erfüllen. Für die technologieunabhängige Kommunikation wird JSON-RPC über Socket-Verbindungen verwendet, weil dieses einen sehr simplen Aufbau hat und von vielen Plattformen unterstützt wird. Um die spätere Einbindung des Tools in OGV zu realisieren, wurde zusammen mit dem Auftraggeber ein Interface definiert.

Ergebnis

Entstanden ist ein Runtime-Object-Observer (ROO), welcher eine Java-Applikation (z.B. .jar-File) startet und zur Laufzeit Objektinformationen mittels JDI ausliest. Die ausgelesen Informationen werden mit JSON-RPC einem Serverstub übermittelt. Die RPC-Calls werden anschliessend an ein Interface weitergegeben, dessen Implementierung vom Auftraggeber gemacht wird. Weiter wurde ein Proof of Concept eines ROO in C# entwickelt, um zu demonstrieren, dass das Konzept auf anderen Sprachen/Technologien erweiterbar ist.

ROV Management Summary

Adrian Anthamatten & Jeyanthan Ravindran

18. Dezember 2015

Inhaltsverzeichnis

1 Ausgangslage

1.1 Object-Graph-Visulaizer

Der Object-Graph-Visulaizer (OGV) ist ein bestehendes Tool, welcher in einer vorausgegangenen Studienarbeit erstellt wurde. Dieses Tool kombiniert UML-Klassen- und Objektdiagramme im 3-dimensionalem Raum in eine einzige Schicht. Zurzeit werden im OGV die Objekte interaktiv vom Benutzer erzeugt und bearbeitet. In diesem Projekt soll der OGV erweitert werden.

1.2 Runtime-Object-Observer

Der OGV soll neu in der Lage zur Laufzeit Objektinformationen aus einer Java-Applikation auszulesen. Das von OGV angezeigte Diagramm soll fortlaufend mit den ausgelesenen Objektinformationen aktualisiert werden mit den. Vom Auftraggeber wird eine Schnittstelle angeboten für die Aktualisierung der UML-Klassen- und Objektdiagramme im OGV. Entwickelt werden soll nun eine eigenständige Java-Applikation (Runtime-Object-Observer). Der Runtime-Object-Observer (ROO) soll in der Lage sein eine Java-Applikation auszuführen und zur Laufzeit Objektinformationen auszulesen. Die ausgelesenen Objektinformationen sollen an den OGV weitergeleitet werden. Der OGV stellt dann ein UML-Klassen- oder Objektdiagramm basierend den enthaltenen Daten dar und aktualisiert die Diagramme laufend.

1.3 Kommunikationskanal

Zwischen den ROO und dem OGV soll ein Kommunikationskanal erstellt werden. Durch diesen Kommunikationskanal werden die Objektinformationen von ROO an den OGV gesendet. Der Kommunikationskanal soll so entwickelt werden, sodass dieser nicht abhängig von der verwendeten Programmiersprache (hier Java) ist. Derselbe Kommunikationskanal soll in der Lage sein, Daten von einem ROO zu empfangen, welcher z. B. in C# oder C++ implementiert worden ist. Mit einem solchen Kommunikationskanal wäre gewährleistet den OGV für andere Programmiersprachen erweiterbar zu machen, sodass z. B. Objektinformationen von einem C#-Programm ausgelesen und als Diagramm im OGV dargestellt werden kann.

2 Vorgehen

2.1 Auslesen von Objektinformationen

In der Evaluationsphase wurden mit dem Arbeitgeber die Anforderungen an den ROO bestimmt. Danach wurden mehrere Technologien für das Auslesen von Objektinformationen miteinander verglichen und basierend den Anforderungen ausgewertet. Java Debug Interface (JDI) wurde als am besten passende Technologie für das Auslesen von Objektinformationen ausgewählt.

2.2 Kommunikationskanal

Für die Kommunikation wurde für eine sehr simple und weitverbreitete Technologie ausgewählt: JSON-RPC.

3 Ergebnis

3.1 Auslesen von Objektinformationen

Für das Auslesen der Objektinformationen wird der Pfad der Java-Applikation angegeben, welche inspiziert werden soll. Der ROO startet die entsprechende Java-Applikation und liest die Objektinformationen.

3.2 Kommunikationskanal

Die von einer Java-Applikation ausgelesenen Objekte werden vom ROO in JSON-Format umgewandelt. Die umgewandelten Daten werden dann über den Kommunikationskanal an den OGV gesendet. Der OGV ist in der Lage die Daten, welche in JSON-Format empfangen wurden, zu verstehen und diese entsprechend im Diagramm darzustellen oder zu aktualisieren.

4 Ausblick

4.1 Auslesen von Objektinformationen in andere Programmiersprachen

Als Proof of Concept wurde ein simples C#-Runtime-Object-Observer geschrieben. Dieser sendet Beispieldaten in JSON-Format an den OGV, welcher in der Lage ist die empfangenen Daten zu übersetzten. Das simple C#-Runtime-Object-Observer kann nun durch einen richtigen C#-Runtime-Object-Observer (oder ein ROO in eine beliebige andere Programmiersprache) ersetzt werden, welcher in der Lage ist Objektinformationen von C#-Applikationen (oder einer Applikation in eine beliebige andere Programmiersprache) auszulesen.

Literatur

ROV Technischer Bericht

Adrian Anthamatten & Jeyanthan Ravindran

18. Dezember 2015

Inhaltsverzeichnis

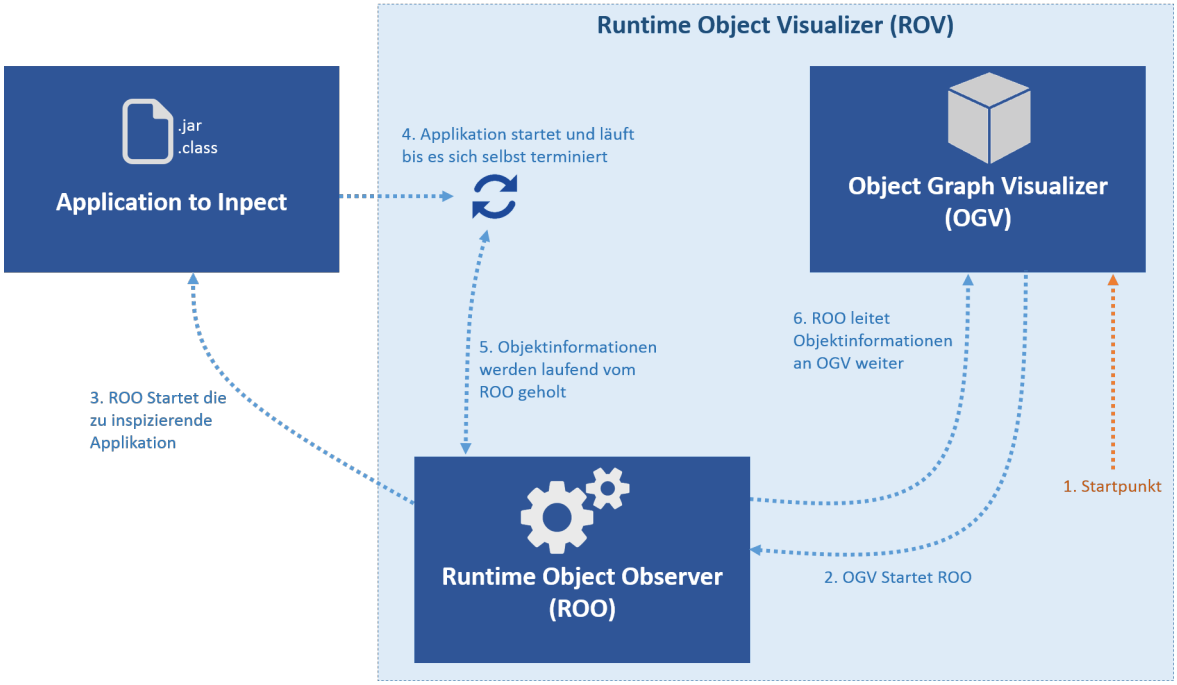
1 Einleitung und Übersicht

1.1 Ausgangslage

Es sollte ein Programm entwickelt werden, welches es dem Object-Graph-Visualizer (OGV) ermöglicht Objektinformationen zur Laufzeit darzustellen. Im Rahmen der Studienarbeit sollte diese Anbindung im speziellen für Java-Applikationen implementiert werden, jedoch mit hinblick auf spätere Erweiterungen in anderen Sprachen oder Technologien.

1.2 Runtime-Object-Observer und Serverstub

Die Aufgabe kann in zwei Teilprobleme unterteilt werden. Zum Einen der Runtime-Object-Observer und zum Anderen den Serverstub, der die Schnittstelle zwischen OGV und dem Runtime-Object-Observer bildet.



1.2.1 Runtime-Object-Observer

Der Runtime-Object-Observer ist eine eigenständige Applikation, welche Objektinformationen zur Laufzeit aus dem laufenden Java-Programm auslesen soll. Dieser Teil soll später auch durch andere Runtime-Object-Observer ersetzt werden können, die für Applikationen in C++, C# oder beliebigen anderen Sprachen auch die Objektinformationen auslesen können.

1.2.2 Serverstub

Dieser Teil der Applikation soll die Schnittstelle zwischen dem OGV und dem Runtime-Object-Observer bilden. Die verwendete Übertragungstechnologie zur Interprozesskommunikation soll technologieunabhängig sein für die beim Runtime-Object-Observer ge-

nannte Erweiterbarkeit. Für die Kommunikation mit dem OGV wird ein Java-Interface definiert, gegen welches implementiert werden kann.

2 Ergebnisse

2.1 Überwachungstechnologien

Um die Objektinformationen aus einer Laufenden Java Virtual Machine auslesen zu können, standen im wesentlichen drei Technologien zur Verfügung: AspectJ, Java Debug Interface (JDI) und Java Virtual Machine Tools Interface (JVMTI). Diese wurden evaluiert und einander gegenüber gestellt. Aus dieser Gegenüberstellung resultierte JDI als Technologie für die weitere Entwlicklung.

2.1.1 Übersicht Technologien

In diesem Abschnitt werden die Technologien kurz vorgestellt.

AspectJ AspectJ ist eine Technologie, mit der man einem .Jar- oder Class-File im Nachhinein noch mit eigenen Prozeduren anreichern kann. Dieser Vorgang nennt sich dort „Weaving“. Es wird also sozusagen der bestehenden Applikation eigener Code „Eingewoben“. Damit kann eine Überwachung realisiert werden, indem vor oder nach einem für den OGV wichtigen Ereignis eigener Code ausgeführt wird.

Java Debug Interface Java Debug Interface ist eine API, welche zur Entwicklung von Debuggern gedacht ist. Man kann sich entweder an eine Laufende Java Virtual Machine (JVM) anhängen oder selbst eine starten. Die Informationen über das zu überwachende Programm werden als Events von der JVM an das Java Debug Interface gesendet. Welche Events dabei gesendet werden sollen, kann man mittels Event-Listener konfigurieren. Dabei können Events wie „MethodEntryEvent“ abgefangen und verarbeitet werden.

Java Virtual Machine Tools Interface Ist vergleichbar mit JDI jedoch bedient es sich der Nativen Methoden der Java Virtual Machine und wird während des Startens der Java Virtual Machine geladen. Dabei kann es über die in der API definierten Methoden Events empfangen und bearbeiten. Es handelt sich dabei um C- oder C++-Code.

2.1.2 Evaluation der Technologien

Für die Evaluation der Technologien wurden zusammen mit dem Auftraggeber Kriterien definiert, um eine der Technologien für die weitere Entwicklung auszuwählen.

Vorgehen Zunächst wurden Kriterien für die Auswahl definiert und diese gewichtet. Es wurden ausserdem „Killerkriterien“ definiert, welche eine die Technologien unbedingt unterstützen mussten, um überhaupt infrage zu kommen. Je nach dem wie gut eine Technologie die Anforderung unterstützte, wurden Punkte vergeben.

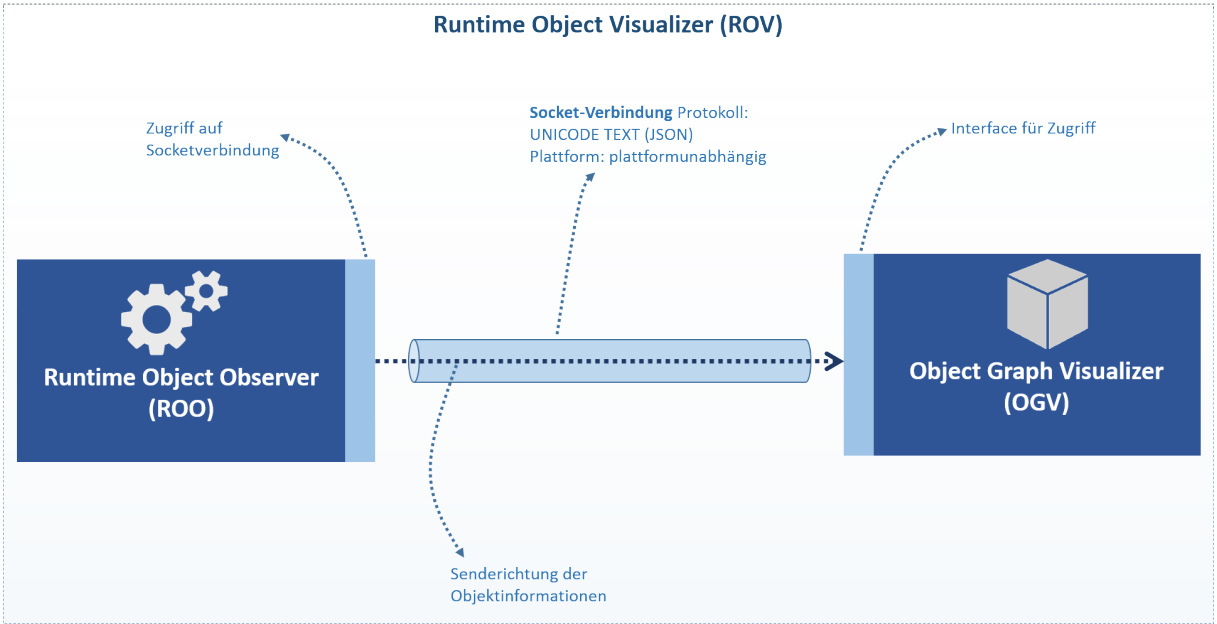
Auswahl Die Evaluation des Java Virtual Machine Tools Interface wurde schon früh gestoppt, obwohl es alle „Killerkriterien“ erfüllte. Ausschlaggebend dafür war vor allem die Tatsache, dass für jede Plattform ein eigenes Executeable ausgeliefert hätte werden müssen. Ausserdem wäre der Entwicklungsaufwand grösser gewesen gegenüber den anderen Lösungen. Da die Performance keine grosse Rolle gespielt hat, wäre der Mehraufwand nicht gerechtfertigt gewesen. AspectJ hatte Probleme bei der Anwendung auf beliebigen Jar- oder Class-Files, da der Weaver jedes Mal spezifisch für eine bestimmte Klasse geschrieben werden musste. Java Debug Interface überzeugte mit einer sehr einfachen Integration und hohen Flexibilität. Es erfüllte damit die Formalen Kriterien am besten.

2.2 Serverstub

Der Serverstub soll die Kommunikation zwischen Runtime-Object-Observer und Object-Graph-Visualizer ermöglichen. Für die Interprozesskommunikation wurde auf Sockets gesetzt und als Übertragungsprotokoll JSON-RPC verwendet. Der Servestub verwendet eine zusammen mit dem Auftraggeber erstelltes Interface zur Kommunikation mit dem Object-Graph-Visualizer.

2.2.1 Interprozesskommunikation

Der Serverstub startet den Runtime-Object-Observer als eigenen Prozess. Für die Kommunikation wird JSON-RPC über Sockets verwendet, damit die Übertragung technologieunabhängig ist. Für JSON-RPC existieren in verschiedenen Sprachen schon APIs und daher ist der Aufwand für die Implementation gering. Als Proof of Concept wurde eine einfache Applikation in C# geschrieben, welche den Serverstub anspricht.



2.2.2 Interface

Für die Kommunikation mit dem Object-Graph-Visualizer wurde ein Interface „DataFeed“ zusammen mit dem Auftraggeber entwickelt. Für die Arbeit wurde ein Test-Interface implementiert, welches die Aufrufe des Runtime-Object-Observers auf die Konsole ausgibt. Der Auftraggeber wird das Interface später noch für den Gebrauch mit dem OGV implementieren.

3 Schlussfolgerungen

3.1 Zusammenfassung

Mit Runtime-Object-Visualization wurde ein Tool entwickelt, welches es ermöglicht, eine Einsicht in eine laufende Applikation zu erhalten. Mit dem Object-Graph-Visualizer als Frontend kann das Entstehen und Verknüpfen von Objekten zur Laufzeit mitverfolgt werden. Die in der Studienarbeit implementierten Komponenten umfassen:

- Runtime-Object-Visualizer für Java
- Serverstub für die Anbindung an den Object-Graph-Visualizer

3.2 Ausblick

3.2.1 Integrierung in Object-Graph-Visualizer

Um Runtime-Object-Visualization verwenden zu können muss noch eine Integrierung in den Object-Graph-Visualizer integriert werden. Dafür muss im Object-Graph-Visualizer das DataFeed-Interface implementiert und der Serverstub integriert werden. Dies wird vom Auftraggeber erledigt.

3.2.2 Runtime Object Observer

Als Erweiterung können Runtime-Object-Observer in verschiedenen Sprachen implementiert werden. Für die Implementation muss lediglich das Interface in JSON-RPC des Serverstubs benutzt werden.

Glossar

ROO	Runtime-Object-Observer Ist ein Teil von ROV, welcher zuständig ist für das auslesen der Objektinformationen von einem gegebenen .jar- oder .class-File und für das gefilterteweiterleiten dieser Informationen an den OGV.
ROV	Runtime-Object-Visualization Ist der Name des Softwares, welcher am Ende des Projekts entwickelt werden sein soll. Es beschreibt den Software, welcher aus dem vorgegebenen Tool OGV und den im Projekt zu implementierenden Tool Runtime-Object-Observer besteht.
OGV	Object-Graph-Visualizer Ist ein vorgegebener Software für die 3D-Darstellung der UML-Klassendiagramme, welche die Infos zu einem Klassenobjekt entgegennehmen und diesen Darstellen kann.
Serverstub	Die Schnittstelle zur SocketVerbindung beim OGV heisst "Serverstub".
Clientstub	Die Schnittstelle zur SocketVerbindung beim ROO heisst "Clientstub".

Erfahrungsbericht Adrian Anthamatten

Es war ein sehr spannendes Projekt. Die Aufgabenstellung fr den ROV war nicht eine alltgliche und ich konnte noch einiges im Bereich Java lernen. Die Zusammenarbeit mit allen Beteiligten hat gut geklappt. Die Dokumente mit Latex zu schreiben war Anfangs zwar mhselig, hat sich im Nachhinein jedoch bezahlt gemacht. Es war einfacher immer alles aktuell zu halten und zu zweit an einem Dokument zu arbeiten war kein Problem. Eher Mhe hatte ich bei dem Schtzen von Aufwnden. Gerade bei JDI waren die Lsungen oft komplizierter, als sie am Anfang aussahen. Auch das Schreiben und der Dokumentation htte etwas rascher und zeitnaher von Statten gehen knnen.

Erfahrungsbericht Jeyanthan Ravindran

Die Studienarbeit war eine sehr gute Praktische Arbeit, um im Studium erworbene Kenntnisse anzuwenden. Zustzlich konnte ich mir auch neue Kenntnisse aneignen und neue Technologien kennenlernen.

Der interssante Teil dieses Projekts war die Architektur dieser Applikation. So musste darauf geachtet werden, dass es keine direkt Abhngigkeit zwischen dem Runtime-Object-Observer und Object-Graph-Visualizer gibt. Nur so konnte sichergestellt werden, dass der Runtime-Object-Visualization in Zukunft auch mit andere Technologien erweitert werden konnte.

In diesem Projekt wagten wir uns auch fr die Dokumentation Latex zu verwenden. Der Einstieg war sehr zeitintensiv und aufwendig. Die Investition hat sich jedoch gelohnt, da wir in diesem Projekt nie Probleme mit zusammenfhren von Dokumenten hatten. Trotz dem grossem Lernaufwand konnten wir schlussendlich davon profitieren, dass wir die Dokumente gleichzeitig bearbeiten und mergen konnten, was bisher mit Word nicht mglich war.

ROV Projektplan

Adrian Anthamatten & Jeyanthan Ravindran

18. Dezember 2015

Inhaltsverzeichnis

1 Projekt Übersicht

In diesem Projekt soll eine Software realisiert werden, welche dem Programm OGV ermöglicht, Objektdiagramme zur Laufzeit zu erstellen.

1.1 Zweck und Ziel

Dieses Projekt wird im Rahmen der Studienarbeit von Adrian Anthamatten und Jeyanthan Ravindran durchgeführt. Ziel ist es, eine Software zu entwickeln, welche es dem OGV erlaubt, Objektdiagramme zur Laufzeit eines überwachten Programms darzustellen. Genauere Angaben sind der Aufgabenstellung des Auftraggebers zu entnehmen [2].

1.2 Lieferumfang

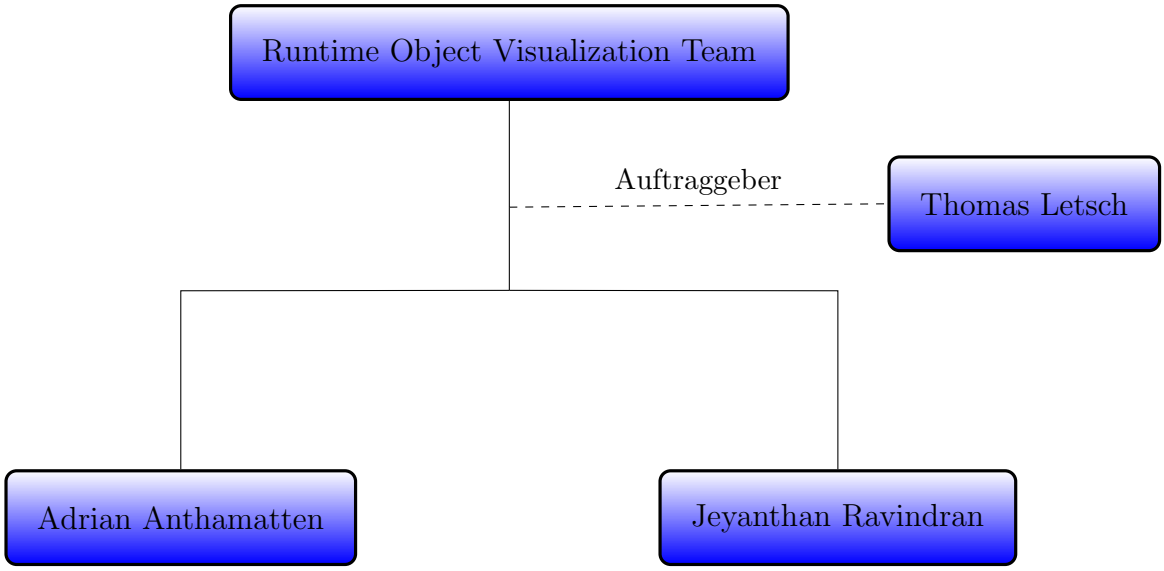
Der Lieferumfang wird die Artefakte gemäss Artefaktdiagramm enthalten. Dieses Richtet sich nach den Vorgaben für Studien- und Bachelorarbeiten an der HSR [1] und des Auftraggebers.

1.3 Annahmen und Einschränkungen

Für die Einbindung in OGV wird gegen ein vom Auftraggeber definierte Schnittstelle programmiert.

2 Projektorganisation

2.1 Organigramm



2.2 Verantwortlichkeiten

Adrian Anthamatten

Buildserver/Git, Code- und Doku Qualitätssicherung, Programmierung

Jeyanthan Ravindran

Code- und Doku Qualitätssicherung, Programmierung

Thomas Letsch

Ist der Auftragsgeber für das Projekt.

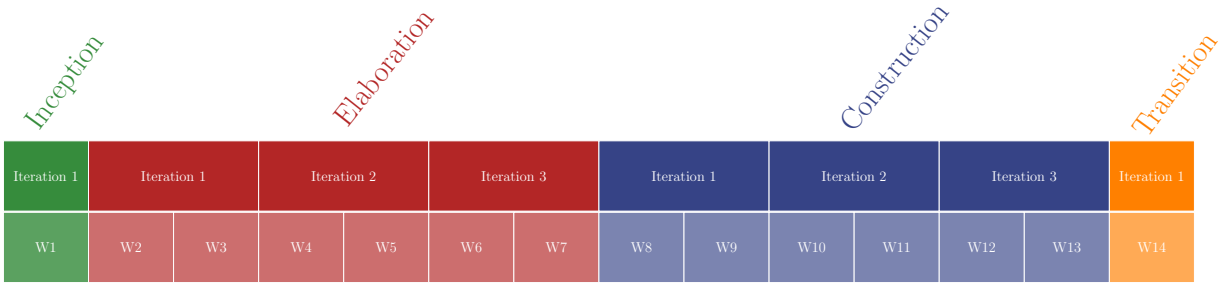
3 Management Abläufe

3.1 Kostenvoranschlag

Dieses Projekt startet und endet zeitlich gleich mit dem Herbstsemester 2015 an der HSR (Hochschule für Technik in Rapperswil), folglich vom 14.09.2015 bis am 18.12.2015. Während dieser Zeit arbeiten 2 Teammitglieder wöchentlich je 18 Stunden am Projekt.

3.2 Zeitliche Planung

Die Planung wird wie folgt gegliedert: 1 Woche für die Phase Inception, 6 Wochen für die Phase Elaboration, 6 Wochen für die Phase Construction und 1 Wochen für die Phase Transition. Jeweils 1-2 Wochen entsprechen einer Iteration. Dadurch ist gewährleistet, dass die Arbeitspakete klein gehalten werden und dass alle Teammitglieder an einem weekly Meeting auf den neuesten Stand gebracht werden können.



3.3 Meilensteine

3.3.1 Projektplan

Ein grober Projektplan ist erstellt, in dem die Meilensteine, die Zuständigkeiten der Teammitglieder und die Risiken des Projekts aufgezählt sind.

3.3.2 Anforderungsspezifikation

Die Anforderungsspezifikation wurde basierend auf den Sitzungen mit dem Auftraggeber erarbeiter. Hier befindet sich die allgemeine Produktbeschreibung, worin auch festgelegt ist was schon vorgegeben ist und was in diesem Projekt entwickelt wird.

3.3.3 Softwarearchitektur

Der Aufbau des Softwares, die Schnittstelle zum gegebenen OGV und die zu verwendenden externe Bibliotheke und Technologien (z.B. zum auslesen der Objekt-Informationen) wurden bestimmt.

3.3.4 End of Elaboration

Basierend der Softwarearchitektur wurde ein Prototyp (Release 1) erstellt, welcher in der Lage ist zur Laufzeit Objektinformationen auszulesen und diese auszugeben. Zusätzlich werden auch Änderungen von Attributen korrekt erkannt und dem Benutzer mitgeteilt.

3.3.5 Release 2

Eine Anbindung zum Virtualizer wurde erstellt. Die Objektinformationen, welche vom Prototyp ausgegeben wurden, werden nun über eine Verbindung in Echtzeit dem Virtualizer übergeben und ausgelesen.

3.3.6 Release 3

Der ORV-Software wurde die Möglichkeit hinzugefügt Filter hinzuzufügen, welche Objekte bestimmter Klassen filtert.

3.3.7 Release 4

Die ORV-Software wurde ausgebessert basierend auf den Korrekturvorschlägen vom Auftraggeber nach dem Review vom Release 3.

3.4 Phasen / Iterationen

3.4.1 Inception

In dieser Phase werden die vorgegebenen Regelungen von den Projektmitgliedern durchgelesen. Es werden basierend dieser Regelungen zuerst alle geforderten Dokumente notiert und eine Artefakt/Zeitdiagramm erstellt.
Zeitaufwand: 36 Personenstunden

3.4.2 Elaboration

In dieser Phase wird die Softwarearchitektur erarbeitet. Ausserdem werden die Anforderungen analysiert. Des weiteren sollen Technologien zum Auslesend der Objektinformationen evaluiert werden. Am Ende steht ein Prototyp bereit, welcher die Architektur wiederspiegelt. Der Projektplan wird in dieser Zeit ebenfalls weiterentwickelt.
Zeitaufwand: 216 Personenstunden

3.4.3 Construction

In dieser Phase wird das Produkt implementiert und getestet.
Zeitaufwand: 216 Personenstunden

3.4.4 Transition

In dieser Phase werden die letzten Tests abgeschlossen und für die Abgabe benötigten Dokumente bereitgestellt.
Zeitaufwand: 36 Personenstunden

4 Risikomanagement

Die Risikoplanung ist im Dokument „Risikomanagement.xlsx“ zu finden. Am Ende jeder Iteration wird dieses Dokument aktualisiert. Die alten Risikoschätzungen werden anhand des Projektverlaufs wenn nötig überarbeitet und neue Risiken werden in die Liste aufgenommen.

5 Infrastruktur

Dies ist eine Auflistung der Tools, welche für die Entwicklung und das Management des Projektes verwendet wurden. Diese Liste wird im Projekt laufend erweitert mit Tools, welche im Verlaufe des Projektsneu hinzugekommen sind.

Tool	Version	Beschreibung	Verweise
Eclipse Mars	?	Java Entwicklungsumgebung	http://www.eclipse.org/
Redmine	?	Projektplanungstool	http://www.redmine.org/
Git		Versionskontrollsystem	http://git-scm.com/
Virtual Server		Host für Redmine	

6 Qualitätsmanagement

6.1 Dokumentation

Die Dokumentationen werden hauptsächlich mit Hilfe von Latex geschrieben. Diese Dokumente werden im Git-Repository gelagert, wodurch das bearbeiten der Dokumente durch mehrere Projektmitglieder einfacher geregelt ist und die Versionierung der Dokumente automatisch geschieht. Die Dokumentation wird fortlaufend ergänzt und nach den Reviews basierend auf den Betreuer-Feedback korrigiert. Damit die Qualität der Dokumente gewährleistet ist, werden die Dokumente vor der Abgabe am Ende des Projekts von allen Projektmitgliedern ein letztes Mal überprüft und verbessert.

6.2 Projektmanagement

Für die Verwaltung und Dokumentierung der Arbeitspakete wird Redmine eingesetzt. Zusätzlich findet auch die Iterationsplanung (jeweils am des vorherigen Iteration) aus-

schliesslich in Redmine statt und hat keinen Einfluss auf die Projektplanung.

6.3 Entwicklung

Der Sourcecode befindet sich auf dem git-Repository der HSR (git.hsr.ch).

6.3.1 Vorgehen

Die Arbeitspakete werden unter den Mitgliedern aufgeteilt. Trotz dieser Aufteilung sollten die Projektmitglieder über den gleichen Wissenstand verfügen. Um alle Projektmitglieder auf den aktuellen Stand von anderen Teammitglieder zu halten gibt es für die Programmierung-Arbeitspaktete Code Reviews (siehe Code Reviews).

6.3.2 Unit Testing

Zur Gewährleistung des korrekten Ablaufs vom Code werden Unit Tests erstellt. Die Tests sollen den gesamten Code abdecken. Bevor man seinen Code eincheckt, müssen alle Tests bestanden sein.

6.3.3 Code Reviews

Beim Code Review wird einerseits überprüft, ob der Code gut und verständlich ist, Styleguides eingehalten wurden und Unit Tests erstellt wurden. Anderseit wird mit den Codereview Wissen verteilt, sodass Projektmitlglieder über die Arbeit anderer Projektmitglieder bescheid wissen. Die Code Reviews werden jeweils im Redmine beim Zeiteintrag gekennzeichnet, damit ersichtlich ist, welche Code-Teile bereits überprüft wurden.

6.3.4 Code Style Guidelines

Bei der Programmierung gelten die Java Coding Conventions von Oracle [3]. Die Variablen und Kommentare werden ausschliesslich in der englischen Sprache geschrieben. Für Benutzer sichtbare Eingabeaufforderungen u. ä. werden in Deutsch beschrieben.

6.4 Testen

6.4.1 Integrationstest

Um den aktuellen Fortschritt und das Zusammenspiel zwischen dem entwickelten Software (ROO) und dem OGV zu testen, werden die Projektmitgliedern in der letzten Construction-Iteration verschiedene Integrationstests durchgeführt. Erst wenn die Tests (basierend auf ein vor dem Test erstelltes Testspezifikation) erfolgreich verlaufen, werden die entsprechenden Funktionen als abgeschlossen betrachtet.

6.4.2 Systemtest

Ein umfangreicher Systemtest wird basierend auf einer Testspezifikation in der letzten Construction-Iteration und am Schluss durchgeführt.

Literatur

[1] Claudia Furrer. *Anleitung: Dokumentation Studien- und Bachelorarbeiten*. HSR Hochschule für Technik Rapperswil, 9 2015. Dokument: DokuAnleitung-BA_SA_150914.pdf.

[2] Thomas Letsch. *Aufgabenstellung Studienarbeit: Runtime-Object-Visualization*. HSR Hochschule für Technik Rapperswil, 9 2015. Dokument: Aufgabenstellung_ROV.pdf.

[3] Sun Microsystems/Oracle. *Java Coding Conventions: Oracle*. Oracle, 11 2015. Link: http://www.oracle.com/technetwork/java/codeconvtoc-136057.html.

ROV Anforderungsspezifikation

Adrian Anthamatten & Jeyanthan Ravindran

18. Dezember 2015

Inhaltsverzeichnis

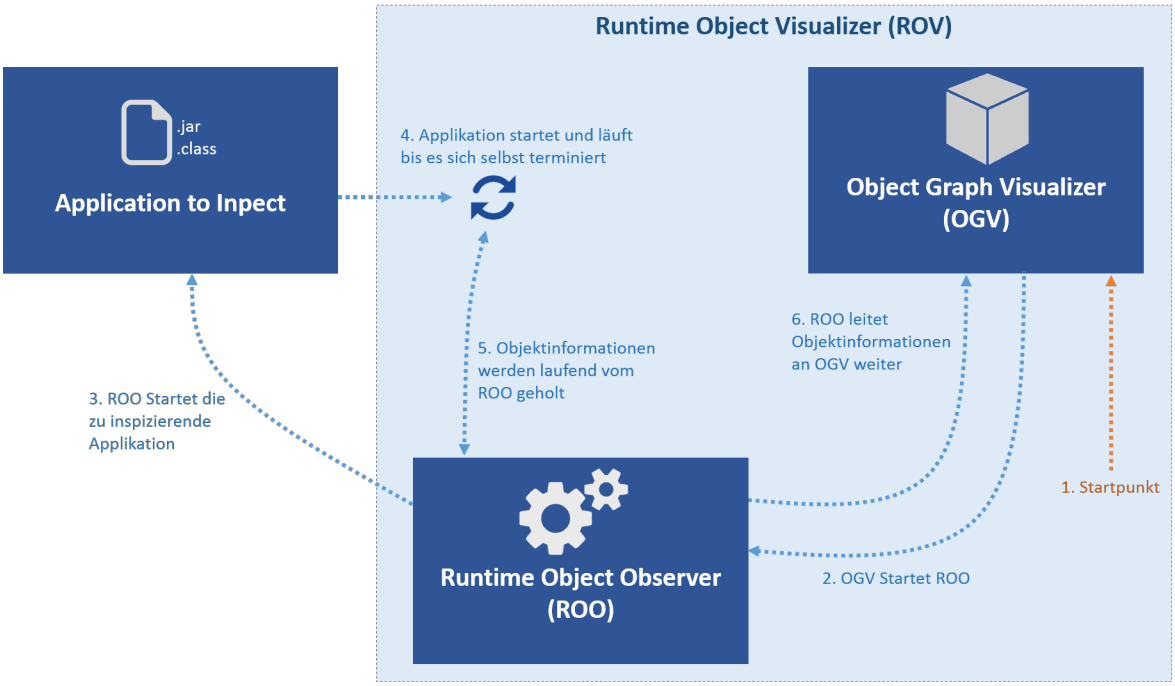
1 Allgemeine Beschreibung

1.1 Ausgangslage

Die detaillierte Aufgabenstellung befindet sich in einem seperaten Dokument mit dem Namen *Aufgabenstellung_ROV.pdf*. Gegeben ist bereits ein existierendes Tool Namens OGV (Object Graph Visualizer), mit den man zurzeit *manuell* Objektdiagramme erstellen kann. Für die Kommunikation mit dem OGV wird vom Auftraggeber eine Schnittstelle definiert.

1.2 Produkt Perspektive und Funktion

Das Ziel dieses Projekts ist es den ein Runtime-Object-Observer zu implementieren und mit diesen eine Verbindung zu Object-Graph-Visualizer zu erstellen. In diesem Bild sind die wichtigsten Schritte beschrieben, wofür der Runtime-Object-Observer verantwortlich ist:



1.2.1 Programm-Vorbereitung

Bevor der ROV gestartet wird muss der Pfad des zu inspzierenden Programms angegeben werden. Desweiteren können Filtereinstellungen vorgenommen werden. Basierend dieser Filtereinstellungen wird dem Benutzer später bestimmte Klassen ausgeblended oder angezeigt.

1.2.2 Programm-Start

Der ROO startet das zu überwachende Programm.

1.2.3 Ausführung des zu inspizierendes Programm

Das zu inspizierende Programm wird ausgeführt bis es sich von selbst terminiert.

1.2.4 Listening

Der Runtime-Object-Observer holt sich alle Informationen von den Instanzen und dessen Attributen vom zu inspizierendem Programm.

1.2.5 Filtern

Die Objektinformationen können vom Benutzer gefiltert werden. Hierzu definiert der Benutzer welche Klassen er beobachten bzw. nicht beobachten möchte.

1.2.6 Visualisieren

Die gefilterten Objektinformationen werden der Ausführung des zu inspizierenden Programms gesammelt.

2 Use Case

2.1 Beschreibung (Brief)

2.1.1 Starten der Überwachung

Die Überwachung der Applikation wird gestartet.

2.1.2 Stoppen der Überwachung

Die Überwachung der Applikation wird gestoppt.

2.1.3 Setzten der Filterung von Klassen für Überwachung

Aus dem OGV-Interface wird eine Liste mit Klassen an ROO geschickt, welche sie überwachen (oder ignorieren) muss. Dabei teilt sie jede Änderung eines Feldes, Aufruf einer Methode oder Instanzierung eines neuen Objekts für die Registrierten Klassen dem Interface mit.

2.2 Beschreibung (Fully Dressed)

2.2.1 Starten der Überwachung

Preconditions: OGV ist gestartet. ROO ist bereit zur Ausführung. Die zu überwachende Applikation ist lauffähig.

Postconditions: ROO ist gestartet. Die zu überwachende Applikation ist gestartet.

Main Success Scenario

1. OGV startet den ROO.
2. ROO initialisiert den Objektfilter.
3. ROO startet die zu überwachende Applikation.

2.2.2 Stoppen der Überwachung

Preconditions: Die Überwachung läuft.

Postconditions: Die Überwachung ist gestoppt.

Main Success Scenario

1. Der OGV stellt die Überwachung ein.
2. ROO beendet die zu überwachende Applikation.
3. ROO beendet sich.

2.2.3 Setzten der Filterung von Klassen für Überwachung

Preconditions: OGV ist gestartet. Zu überwachende Applikation ist Lauffähig.

Postconditions: OGV erhält Meldungen über Änderungen von Feldern, Methodenaufrufen und Instanzierung neuer Klassen.

Main Success Scenario

1. OGV teilt dem ROO mit, welche Klassen es überwachen will.
2. ROO teilt dem OGV Änderungen an den Objekten mit.

3 Anforderungen

In diesem Teil geht es um die funktionalen und nicht funktionalen Anforderungen an den Runtime Object Observer (ROO). Diese Anforderungen wurden gemeinsam mit dem Auftraggeber erarbeitet.

3.1 Überwachen relevanter Daten

Es ist möglich aus dem zu inspizierendem Programm die relevanten Daten auszulesen.

Felder

- Auslesen von private, public, protected und package-protected Felder.
- Auslesen von Initialisierung von Felder.
- Auslesen von Änderungen der Felder.
- Auslesen des ursprünglichen Wertes bei einer Änderung der Felder.
- Auslesen des neu zugewiesenen Wertes bei einer Änderung der Felder.
- Auslesen zu welchem Objekt-Instanz die Felder gehören.

Methodenaufrufe

- Auslesen von private, public, protected und package-protected Methodenaufrufe
- Auslesen von Argumenten, welche dem Methodenaufruf mitgegeben wurde
- Auslesen von der zugehörigen Objekt-Instanz

Statische Variablen

- Auslesen von private, public, protected und package-protected Variablen, welche statisch (static) sind
- Auslesen von Initialisierung von statischen Variablen
- Auslesen von Änderungen der statischen Variablen
- Auslesen des ursprünglichen Wertes bei einer Änderung der statischen Variablen
- Auslesen des neu zugewiesenen Wertes bei einer Änderung der statischen Variablen

Statische Methodenaufrufe

- Auslesen von private, public, protected und package-protected Methodenaufrufe, welche statisch (static) sind
- Auslesen von Argumenten, welche dem Methodenaufruf der statischen Methoden mitgegeben wurde

Konstruktoren

- Auslesen von Aufrufen der Default Konstruktoren (Konstruktor ohne Argumente)
- Auslesen von Aufrufen der spezialisierten Konstruktoren (Konstruktor mit Argumente)
- Auslesen von Argumenten, welche dem spezialisierten Konstruktor mitgegeben wurde
- Auslesen von der zugehörigen Objekt-Instanz

Objekt Zerstörung

- Auslesen wann ein Objekt zerstört wird.

Array-/Collectionsichtbarkeit

- Auslesen von Array- und Collection-Initialisierung
- Auslesen von Neuzuweisung von Array und Collection
- Auslesen von Hinzufügen von Elementen im Array und Collection
- Auslesen von Entfernen von Elementen im Array und Collection
- Auslesen von der zugehörigen Objekt-Instanz bei Änderungen im Array und Collection

Default-Package

- Auslesen Feld-Informationen (siehe Felder), welche sich im Default-Package befinden
- Auslesen von statischen Variablen (siehe statische Variable) im Default-Package
- Auslesen von Methodenaufrufen (siehe Methodenaufrufe) im Default-Package
- Auslesen von statischen Methodenaufrufen (siehe statische Methodenaufrufe) im Default-Package
- Auslesen von Konstruktoren (siehe Konstruktoren) im Default-Package
- Auslesen von Destruktoren (siehe Destruktoren) im Default-Package
- Auslesen von Array und Collections (siehe Array-/Collectionsichtbarkeit), welche sich im Default-Package befinden

3.2 Performance

Bei der Geschwindigkeit wird auf die Referenz-Geschwindigkeit gestützt. Die Referenz-Geschwindigkeit ist die Dauer vom Start der Applikation bis zur Terminierung der Applikation, welche Inspiziert werden soll. Die Referenz-Geschwindigkeit wird gemessen im Interpreter-Modus (d.h. ohne Optimizer). Bei der Inspizierung der Applikation mit dem Runtime-Object-Observer, darf die Geschwindigkeit sich maximal um das 10-Fache der Referenzgeschwindigkeit verlangsamen. Beispiel: Dauert die zu inspizierende Applikation vom Applikationsstart bis zur Terminierung 1 Minute, so muss dieselbe Applikation bei der Inspizierung mit dem ROO innerhalb von 10 Minuten terminieren.

3.3 Integrierung

Es besteht die Möglichkeit eine Applikation im Form von .jar/.class-File zu Überwachen.

3.4 Filterung

Die Technologie bietet die Möglichkeit einer Filterung von z.B. Klassen.

Include-Filter Im Include-Filter kann der ROO-Benutzer die Klassen angeben, wessen Objektinformationen er sich ansehen möchte.

Exclude-Filter Im Exclude-Filter definiert der ROO-Benutzer, welche Klasse nicht inspziert werden sollen.

Kombinationsfilter Ein Filter in dem Exclude und Include möglich ist.

Laufzeitfilter Mit Laufzeitfilter ist die Möglichkeit gemeint, welche es dem Benutzer erlaubt während der Laufzeit Änderungen am Filter vorzunehmen.

3.5 Plattformunabhängigkeit

Portabilität auf verschiedenen Plattformen Die Übertragungstechnologie ist unabhängig von der eingesetzten Technologie zum Auslesen der Objektinformationen. Es soll zum Beispiel möglich sein, einen ROO für C# oder C++ zu schreiben.

Portabilität auf verschiedene Betriebssystem Das auslesen der Objektinformationen kann auf den gängigsten drei Desktop-Betriebssysteme (Windows, Linux und MacOS) ausgeführt werden.

3.6 Testbarkeit

Der Programmcode ist testbar (z.B. mit JUnit-Test).

3.7 Benutzbarkeit

Der Benutzer ist in der Lage basierend der Benutzeranleitung den ROV über einen bestimmten Befehl in der Konsole zu starten.

3.8 Hardware-/Softwareanforderungen

RAM	1GB
Betriebssysteme	Windows 7 oder höher Mac OS X 10.0 oder höher Ubuntu 14.10 oder höher
Java Version	7 oder höher

ROV Softwarearchitekur

Adrian Anthamatten & Jeyanthan Ravindran

18. Dezember 2015

Inhaltsverzeichnis

1 Architektonische Entscheidungen

1.1 Technologieunabhängigkeit

Um zu gewährleisten, dass ROV zukünftig auch andere Technologien unterstützt wird die Verbindung zwischen dem OGV Interface (Serverstub) und dem ROO über Sockets mit JSON-RPC realisiert.

1.1.1 Anforderungen

- Bei Überwachung einer anderen Technologie, soll der Serverstub nicht angepasst werden müssen.

1.1.2 Lösung

Das ROO wird als eigenständige Applikation entwickelt, die über eine Socket-Connection mit dem Serverstub am OGV kommuniziert. Als Übertragungsprotokoll auf Anwendungsebene wird JSON-RPC eingesetzt. Der Serverstub beginnt dabei auf einem freien Port zu horchen und starten den ROO und gibt ihm diesen Port als Parameter mit.

1.1.3 Begründung

Wird der ROO als eigene Applikation realisiert, kann später dem OGV auch unterstützung für andere Sprachen wie C# hinzugefügt werden, indem eine ähnliche Applikation in der jeweiligen Technologie geschrieben wird. Eine Socket-Connection ist dabei die einfachste Form der Interprozesskommunikation. Um nicht ein eigenes Protokoll zu schreiben und zu parsen wurde JSON-RPC eingesetzt. Es ermöglicht ohne grossen Overhead Remote Method Calls zu realisieren und ist, dadurch dass es textbasiert ist, auf allen denkbaren Plattformen einsetzbar. Für viele Sprachen existieren schon JSON-RPC-Libraries oder zumindest JSON-Parser, wodurch die Implementierung für zukünftige ROOs vereinfacht wird.

1.1.4 Ungelöste Probleme

keine

1.1.5 Erwogene Alternativen

keine

1.2 Filtereinstellungen während der Laufzeit

Damit Filter für die zu überwachenden Klassen auch während der Lauftzeit geändert werden kann, wurde bei den RPC-Calls für die Event Übertragung ein Rückgabewert des Typs „Boolean“ eingeführt. Dieser signalisiert dem Serializer, dass der Filter geändert wurde und die eine entsprechende Anpassung vorgenommen werden muss.

1.2.1 Anforderungen

- Der Filter für die Überwachung der Objekte soll auch zur Laufzeit anpassbar sein.

1.2.2 Lösung

Jedem RPC-Call wurde ein boolscher Wert als Rückgabetyp definiert. Dieser signalisiert dem Serializer, dass der Filter im OGV geändert wurde. Der Serializer holt sich mit dem RPC-Call „getFilterList“ anschliessend den neuen Filter. Danach ruft der Serializer die „updateFilter“-Methode des Filter-Interface auf. Genaueres zum Ablauf ist im Kapitel 5.3 beschrieben.

1.2.3 Begründung

Durch den boolschen Rückgabetyp bei jedem RPC-Call kann ein Roundtrip gespart werden.
Durch das Definieren des Filter-Interface konnte die Zyklomatische-Komplexität der „ObjectObserver“-Klasse verringert werden. Ausserdem wird die Implementierung des Serializers mehr vom ObjectObserver entkoppelt.

1.2.4 Ungelöste Probleme

keine

1.2.5 Erwogene Alternativen

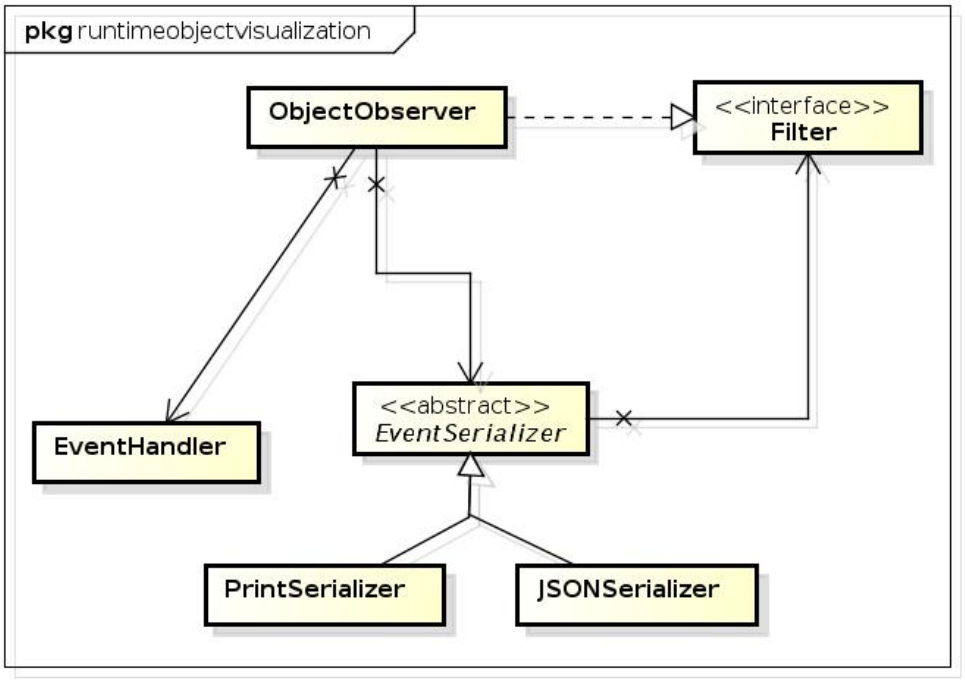
keine

2 Logische Sicht

2.1 Aufbau Runtime Object Observer

Dieser Abschnitt beschreibt grundlegende Komponenten und Funktionsweise des ROO.

2.1.1 Hauptkomponenten



ObjectObserver Der ObjectObserver setzt die Java Virtual Machine (JVM) für die überwachung einer Software auf und startet dann auch die Überwachung über den EventHandler. Er registriert sich selbst beim Serializer um über Änderungen beim Filter benachrichtigt zu werden.

EventHandler Der EventHandler verarbeitet die einzelnen Events, welche auf der JVM der zu überwachenden Applikation auftreten. Er verarbeitet diese und generiert daraus EventDTOs, welcher er über eine ConcurrentLinkedQueue dem ObjectObserver schickt.

MessageSerializer Der Serializer wurde als abstrakte Klasse implementiert und bietet für alle EventDTOs eine abstrakte Methode zur behandlung (Template Method Pattern). Der ObjectObserver registriert sich beim Serializer um über Änderungen bei den Filtern informiert zu werden. Es gibt zwei Implementierungen für den Serializer:

PrintSerializer Der PrintSerializer wird für die Entwlicklung genutzt, um für das Debugging des ROO nicht den Server starten zu müssen und eine direkte Konsolenausgabe der Events zu haben.

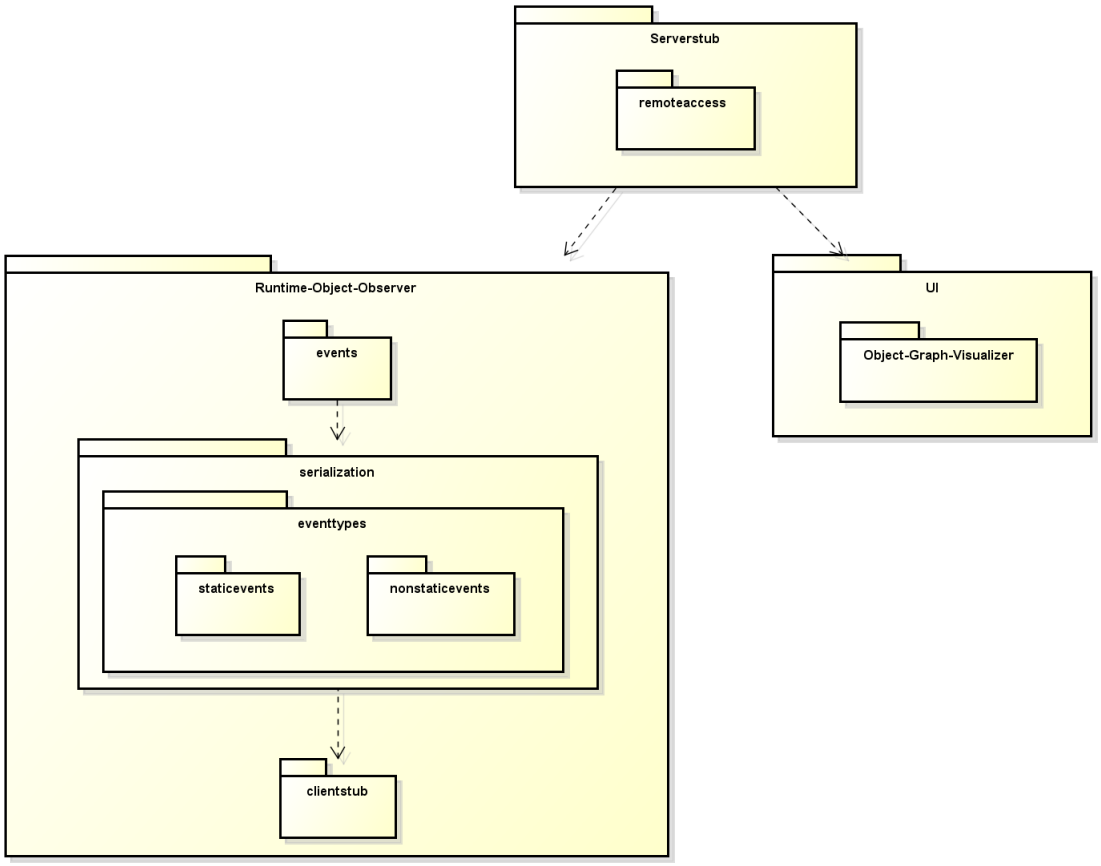
JSONSerializer Der JSONSerializer wird für den späteren produktiven Einsatz benötigt. Er stellt eine Verbindung zum Serverstub her und übermittelt die Events als JSON-RPC.

2.1.2 Events

Die Event-Klassen bilden die Methoden des DataFeeds ab. Sie halten alle Daten, die für den Methodenaufruf nötig ist. Alle Events leiten vom EventDTO-Interface ab. Der EventHandler generiert sie anhand der beobachteten Ereignisse auf der überwachten VM. Danach stellt er sie in die msgQueue damit sie vom EventSerializer verarbeitet werden können.

2.2 Packagediagramm

Hier sind die einzelnen Packages beschrieben:



2.2.1 Runtime-Object-Observer

Der Runtime-Object-Observer beinhaltet die Behandlungen beim Auftrateten der Events (z.B. Method-Call). Im Clientstub wird die Socket-Verbindung mit dem Serverstub gehandlet, worin die verarbeiteten Events und gesendet werden an den Serverstub.

2.2.2 Serverstub

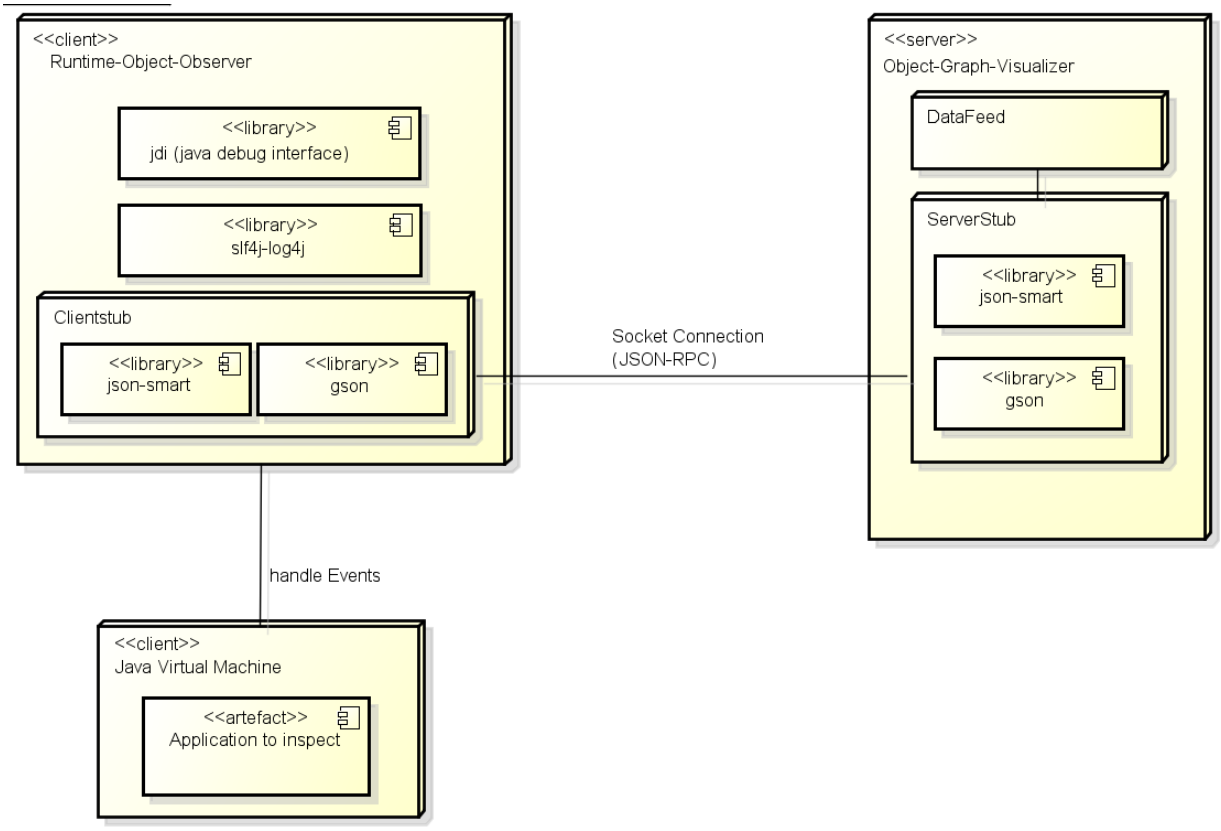
Im Subpackage remoteaccess wird die Schnittstelle zur OGV festgehalten.

2.2.3 UI

Diese Package wird vom Auftraggeber zur verfügung gestellt und beinhaltet den bereits vorhandenen OGV (Object-Graph-Visualizer).

3 Deployment

In diesem Kapitel ist das Deploymentdiagramm dargestellt. Hier ist ersichtlich dass der Runtime-Object-Observer die zu inspizierende Applikation auf einer seperaten Virtuellen Machine ausführt. Die mittels JDI ausgelesenen Events werden dem Runtime-Object-Observer gesendet, welcher diese via Socketverbindung and Object-Graph-Visualizer sendet.



3.1 External Libraries

Im Deployment Diagramm sind die wichtisten verwendeten externen Libraries dargestellt. Eine vollständige Auflistung aller verwendet Libraries ist hier zu entnehmen:

3.1.1 Argumente Parsen

Library: **args4j** [4]
Version: 2.32
Funktion: Parsen von Arugmente, welche beim Starten von ROV mitgegeben werden.

3.1.2 JSON Parsen

Library: **gson** [2]
Version: 2.3.1
Funktion: Konvertierung von Objekten in JSON-Strings und JSON-Strings in Objekten

3.1.3 JSON-RPC

Für die Kommunikation über JSON-RPC waren mehrere Libraries notwendig:
Library: **json-smart** [1]
Version: 1.2
Funktion: Enthält die Basis-Funktionalitäten für die JSON-RPC-Kommunikation

Library: **jsonrpc2-client**
Version: 1.15
Funktion: Enthält die Funktionalitäten für einen Client-seitigen JSON-RPC-Verbindung

Library: **jsonrpc2-server**
Version: 1.11
Funktion: Enthält die Funktionaltäten für einen Server-seitigen JSON-RPC-Verbindung

3.1.4 Logging

Um das Logging zu realisieren wurde SLF4J verwendet:
Library: **slf4j-log4j** [3]
Version: 1.7.13
Funktion: Logging

Library: **slf4j-api**
Version: 1.7.13
Funktion: Logging (slf4j-log4j benötigt diese Library)

Library: **log4j**
Version: 1.2.17
Funktion Logging (slf4j basiert auf log4j)

4 Evaluation Überwachungstechnologien

Für das Auslesen von Objektinformation gibt es verschiedene Technologien. In dieser Evaluationsphase geht es darum sich für die Problemstellung am besten passende Technologie ausfindig zu machen.

4.1 Technologien

In der ersten Phase der Evaluation wurden nach Technologien gesucht, mit der man Objektinformationen auslesen konnte. Die folgenden Technologien wurden schlussendlich gefunden:

- JDI
- AspectJ
- JVMTI
- JavaSnoop

4.2 Recherche und Anforderungen

Im nächsten Schritt wurden Recherchen zu den Technologien gemacht (ohne zu programmieren). Neben den Recherchen wurde mit dem Auftraggeber die Anforderungen erarbeitet (siehe Auswertung), welche die Technologie zu erfüllen hatte. Mit dem Auftraggeber erarbeitete Kriterien wurden in zwei Kategorien unterteilt. Die Kriterien, welche die Technologie zu erfüllen hatte, wurden 'Killerkriterien' genannt. Neben den Killerkriterien wurden weitere optionale Anforderungen für die Technologie bestimmt (Nice-To-Have).

4.3 Aussortierung

Als nächstes war das Ziel von den vier Technologien, jene auszusortieren, welche dir Killerkriterien nicht erfüllen. Hierbei wurde wegen der zeitlich begrenzten Evaluationszeitraums entschieden in der Evaluationsphase auf 3 technologien zu beschränken und die vierte Technologie als Backup (falls die restlichen drei die Killerkriterien nicht erfüllen können) zu behalten.
Als Technologiebackup wurde das JavaSnoop zu vorenthalten, da zu jenem Zeitpunkt über diese Technologie wenigste Vorwissen im Team herrschte. Bei der evaluation konnte dem JVMTI keine weitere Beachtung geschenkt werden, da dieser in C++ implementieren gewesen wäre und somit das Killerkriterium Technologieunabhängigkeit nich erfüllte.

4.4 JDI versus AspectJ

JDI und AspectJ erfüllten im Gegensatz zu JVMTI die ersten Anforderungen. Das Team teilte sich auf, um mit JDI und AspectJ erste Implementationen vorzunehmen. Mit jene Implementationen sollte getestet werden, ob die weiteren Anforderungen von den jeweiligen Technologien umsetztbar wären. Nach dieser Testphase werden die Ergebnisse beider Technologien gegenübergestellt und für die passendere Technologie entschieden.

4.4.1 Gegenüberstellung von JDI und AspectJ

Nach der Testphase wurden die beiden Überwachungstechnologien wurden miteinander verglichen. Für den Vergleich wurde die folgende Auswertungstabelle gemacht. Hierbei wurden zuerst alle Anforderungen des Auftraggebers (siehe Anforderungsspezifikation) in Kategorien unterteilt.
Die einzelnen Anforderungen wurden danach nach Wichtigkeit (8 → Killerkriterium; 3 → Nice-To-Have) bewertet. Um die beiden Technologien zu vergleichen wurden, dann jeweils Bewertungen (zwischen 0 → 'nicht erfüllt' und 10 → 'vollständig erfüllt') der beiden Technologien hinsichtlich der einzelnen Anforderungen abgegeben. Die Begründung für die jeweilige Bewertungen sind im Anhang ersichtlich.

4.4.2 Auswertungstabelle

Anforderungs-kategorie	Anforderung	Anforderungs-gewichtung	Bewertung JDI	Bewertung AspectJ
Überwachung relevanter Daten	Felder	8	10	10
	Methodenaufrufe	3	10	10
	Statische Variablen	8	10	10
	Statische Methodenaufrufe	3	10	10
	Konstruktoren	8	10	10
	Destruktoren	3	5	2
	Array-/Collection-Sichtbarkeit	8	8	5
	Default-Package	8	10	10
Performance	Performance während der Überwachung	8	9	9
Integration	Überwachung von .jar- und .class-Files	8	10	9
Filterung	Include-Filter	8	10	5
	Exclude-Filter	3	10	5
	Laufzeit-Filter	3	10	0
Plattform-unabhängig-keit	Portabilität auf verschiedene Plattformen	3	10	10
	Portabilität auf verschiedene Betriebssysteme	8	10	10
Testbarkeit	Testing mit JUnit	3	10	0
Benutzbarkeit	Ausführung per Konsole	8	10	10
	TOTALE AUSWERTUNG		971	815

5 Prozesse und Threads

5.1 Prozessübersicht

Dieses Kapitel gibt eine Übersicht über alle vorhandenen Prozesse.

5.1.1 Liste der Prozesse

- Runtime Object Observer
- JVM des zu überwachenden Programms
- OGV/Serverstub

5.1.2 Beschreibung der Prozesse

Runtime Object Observer Der ROO wird vom OGV/Serverstub gestartet. Er startet die JVM des zu überwachenden Programms mittels JDI und überwacht dieses. Die Interprozesskomunikation zwischen dem ROO und OGV/Serverstub erfolgt über Sockets.

JVM des zu überwachenden Programms Wird vom ROO gestartet. Führt das Programm aus und übermittelt Events an den ROO.

OGV/Serverstub Startet ROO als eigenen Prozess. Die Interprozesskomunikation zum ROO erfolgt über Sockets.

5.2 Threads

5.2.1 Threads pro Prozess

- Runtime Object Observer
  - ObjectObserver
  - EventHandler
- JVM des zu überwachenden Programms
  - Nur Hauptthread
- OGV/Serverstub
  - Hauptthread
  - RedirectOutput

5.2.2 Beschreibung der Threads

ObjectObserver Startet den EventHandler-Thread. Nimmt die Events aus dem EventHandler entgegen und übergibt sie einem Serializer. Die Schnittstelle zum Austausch der Events ist eine ConcurrentLinkedQueue.

EventHandler Wird vom ObjectObserver gestartet und liest die Events aus der JVM des zu überwachenden Programms aus. Die Events werden in einer ConcurrentLinked-Queue gespeichert. Für weitere Informationen zum Austausch der Filter siehe Kapitel 5.3.

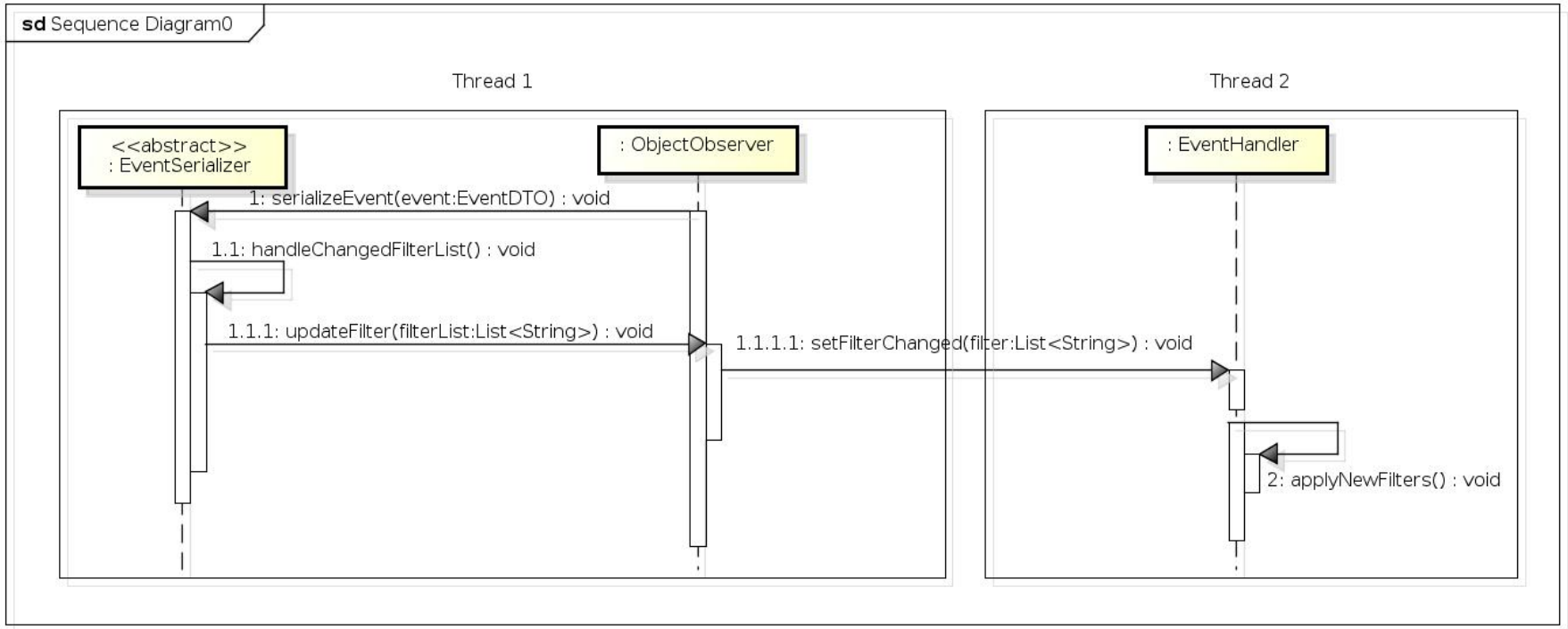
OGV/Serverstub - Hauptthread Startet den RedirectOutput. Horcht auf eingehende Clientverbindungen und gibt eingehende Anfragen an den DataFeed weiter.

OGV/Serverstub - RedirectOutput Leitet den Konsolen-Output des ROO-Prozesses an den Konsolen-Output des Serverstubs weiter.

5.3 Filter Changed

Filter Changed ist die einzige Situation, in der eine Synchronisation zwischen den Thread explizit erforderlich ist. Daher wird dies in diesem Kapitel beschrieben.

5.3.1 Ablauf



1 Der Event wird vom Serializer serialisiert

1.1 Bei der Übertragung wird festgestellt, dass sich die Filterliste geändert hat. Die Methode handleChangedFilter wird aufgerufen.

1.1.1 Über die im Interface Filter definierte Methode updateFilter wird im ObjectObserver der Update ausgelöst.

1.1.1.1 Der Methodenaufruf erfolg Synchronized.

2 Der EventHandler befindet sich in der Run Methode des Runnable-Interface.Anhand eines AtomicBoolean-Wertes „hasFilterChanged“ wird die Methode applyNewFilter ausgeführt. Diese läuft Synchronized

5.3.2 Synchronisierung

Um keine Race Conditions zu haben, wurde auf das Konstrukt der Methoden Synchronisierung von Java und Atomic-Datentypen zurückgegriffen. Folgende Methoden sind mit dem Schlüsselwort Synchronized erstellt:

- setFilterChanged: Setzt den neuen Filter und setzt das Attribut „hasFilterChanged“ auf true
- applyNewFilters: Wendet die neuen Filter an

Das Attribut „hasFilterChanged“ ist ein AtomicBoolean und kann atomar gelesen oder geschrieben werden. Es zeigt dem „EventHandler“ an, dass sich der Filter geändert hat. Es ist wie die Filterliste als volatile markiert, um die Cache-Kohärenz zwischen den Threads zu garantieren.

6 Erweiterungen

In diesem Kapitel sind Erweiterungen für den ROO oder den Serverstub beschrieben, welche analysiert wurden, jedoch noch nicht umgesetzt sind.

6.1 DeleteObject in Java

Dieser Abschnitt beschreibt, wie der ROO um DeleteObject-Fähigkeiten erweitert werden könnte.

6.1.1 Ausgangslage

Da in Java das Freigeben von nicht mehr benötigtem Speicher über Garbage Collection passiert, ist es schwierig herauszufinden, wann ein Objekt wirklich am Ende seines Lebenszyklus steht und entsprechend nicht mehr vom Programm benötigt wird.
Während bei der Erstellung eines Objektes immer der New-Operator aufgerufen wird und dies von JDI überwacht werden kann, ist dies bei Finalizern nur möglich, wenn dieser explizit überschrieben wurde. Und selbst dann, wird das Objekt nicht am frühst möglichen Zeitpunkt finalisiert.

6.1.2 Lösung

Ein Objekt in Java ist am Ende seines Lebenszyklus, wenn kein anderes Objekt aus einem aktiven Scope das Objekt referenziert. Der kleinste Scope der mit JDI einfach überwacht werden kann, ist der Methodenaufruf mittels MethodEntryEvent und MethodExitEvent. Alle Objekte, welche in diesem Scope definiert werden und danach nicht von anderen Objekten auserhalb dieses Scopes referenziert werden, sind nicht länger erreichbar und werden später vom Garbage Collector abgeräumt. Hier setzt diese Lösung an.
Die Klasse „MethodEventHelper“ muss um eine Stackvariable erweiter werden, die Listen von „ObjectReference“ enthält. Für jeden Aufruf der „handleMethodCall“-Methode muss diesem Stack eine Liste hinzugefügt werden. In diesen Listen werden alle Objekte gespeichert, welche im aktuellen Methodenaufruf erstellt werden (jedes Auftreten von „handleNewObject“-Aufrufen).
Bei einem Aufruf der „handleMethodExitEvent“-Methode kann mit Hilfe des Stacks analysiert werden, welche Objekte nach dem Methodenaufruf nicht länger erreichbar sind. Die Klasse „ObjectReference“ von JDI bietet eine Methode „referringObjects“ an, mit welcher überprüft werden kann, welches Objekt noch von anderen Objekten referenziert wird. Mittels diesen Informationen, kann herausgefunden werden, welche Objekte vom Garbage Collector gelöscht werden. Um dies dem ServerStub mitzuteilen, muss ein DeleteObjectEvent in die EventQueue gestellt werden.
Es gibt in diesem Szenario zwei Fälle, in denen ein „DeleteObjectEvent“ in die EventQueue gestellt werden kann:

- Ein Objekt aus der aktuellen Liste (Top of Stack) hat keine referenzierenden Objekte mehr.
- Ein Objekt aus der Liste wird nicht mehr von Objekten ausserhalb der aktuellen Liste referenziert.

7 Bekannte Probleme

7.1 Performance

Trotz der guten Resulte von JDI während der Evaluation, ist die Performance nicht zufriedenstellend.

7.1.1 Beschreibung

In der späten Phase der Entwlicklung fiel die Performance von JDI sehr negativ auf. Statt der aus der Evaluation erwarteten verdopllung bis verdreifachung der Ausführungszeit, wurde das Programm zehn- bis zwanzigfach langsamer als ohne die überwachung durch JDI (nicht im Interpreter-Modus).

7.1.2 Analyse

Die Performance wurde mittels Java VisualVM von Oracle untersucht. Die Applikation wurde mehrere Male gestartet und die Resultate untersucht, um etwaige Unregelmässigkeiten auszuschliessen. Die untersuchung zeigte, dass das Programm sich auffallend lange in der run-Methode des EventHandlers befindet. Dort werden die Events aus der Event Queue der JVM gelesen und zur Weiterverarbeitung an eine andere Methode übergeben. Die meiste in der run-Methode verbrachten Zeit ist der Thread Idle und wartet auf neue Events der Event Queue. Die Resultate variieren auch je nach untersuchtem Programm.

7.1.3 Begründung

Es ist nicht ganz klar, warum die Event Queue lange leer ist und der Thread darum gezwungen wird zu warten. Da auch der Zeitpunkt nicht ganz klar ist, ab wann die Probleme aufgetreten sind, kann auch nicht genau eingegrenz werden, welche Faktoren dazu führen. Ein Unterschied der hier festgestellt wurde und vermutlich zum Problem beiträgt ist, dass nun alle Event-Listener die JVM des zu überwachenden Programms stoppen, bis sie abgearbeitet sind. Diese können auch so konfiguriert werden, dass das Programm während der Behandlung weiterläuft. Das würde jedoch die Konsistenz der Daten gefährden und einige technischer Beschränkungen beim Auslesen der Daten geben. Die technischen Beschränkungen sind besonders bei der Analyse der Methoden zu finden.

8 Anhang

8.1 Technologie-Auswertung - JDI (Java Debung Interface)

Hier werden auf die einzelnen Auswertungen von JDI (siehe Auswertung) eingegangen:

Überwachen relevanter Daten

Felder
✓ OK (Mit JDI besteht die Möglichkeit die Felder zu überwachen)

Methodenaufrufe
✓ OK

Statische Variablen
✓ OK

Statische Methodenaufrufe
✓ OK

Konstruktoren
✓ OK

Destruktoren
✓ In JDI besteht die Möglichkeit Destruktoren zu überwachen, jedoch nur wenn der Destruktor (finalize) in der Klasse überschrieben worden ist.

Array-/Collectionsichtbarkeit
✓ Mit JDI kann man Änderungen im Array nachverfolgen. JDI selbst bietet die Möglichkeit jeden Array-Aufruf (lesend und schreibend) abzufangen. Bei diesen Aufrufen müssten die lesenden Aufrufe herausgefiltert werden.

Default-Package
✓ OK

Performance

✓ OK (Wenn eine Applikation mit JDI inspiziert wird verlangsamt sich die Geschwindigkeit um 100% und ist somit unter den gewünschten 900% bzw. 10 Mal langsamer als die eigendliche Geschwindigkeit der Applikation)

Integrierung

✓ OK

Filterung

Include-Filter
✓ OK

Exclude-Filter
✓ OK

Laufzeitfilter
✓ OK

Plattformunabhängigkeit

Portabilität auf verschiedenen Plattformen
✓ OK

Portabilität auf verschiedene Betriebssystem
✓ OK

Testbarkeit

✓ In JDI können die Inspizierenden klassen mit JUnit-Tests getestet werden.

8.2 Technologie-Auswertung - AspectJ

In diesem Kapitel ist ersichtlich weshalb die AspectJ in der Auswertung (siehe Auswertung) die entsprechenden Wertungen gegeben wurde.

Überwachen relevanter Daten

Felder
✓ OK (Mit AspectJ besteht die Möglichkeit die Felder zu überwachen)

Methodenaufrufe
✓ OK

Statische Variablen
✓ OK

Statische Methodenaufrufe
✓ OK

Konstruktoren
✓ OK

Destruktoren
✗ Auch wenn man die finalize-Methode explizit überwacht, wird dieser beim Aufruf nicht angezeigt. Ob das Problem jedoch beim Garbadge-Collector oder beim AspectJ liegt konnte nicht zweifelsfrei festgestellt werden. Wegen der niedrigen Priorität dieser Funktionalität, wurde nicht genauer auf dieses Problem eingegangen.

Array-/Collectionsichtbarkeit
✓ AspectJ selbst bietet die Möglichkeit Änderungen des Arrays (neues/anderes Array zuweisen) abzufangen. Jedoch gibt es in AspectJ nicht die Möglichkeit Änderungen im Array (z.B. hinzufügen oder entfernen von Elemente) zu erkennen. Hierfür gibt es eine Erweiterung namens ABC-Compiler (AspectBench-Compiler), der in der Lage ist Änderungen im Array (z.B. hinzufügen oder entfernen von Elementen). Vom Entwickler wurde auch ein Beispiel zur Verfügung gestellt. Der Versuch diesen Compiler und das Beispiel auszuführen scheiterte jedoch, wegen einer Fehlermeldung im Jasmine-Kompiler (wird vom ABC-Compiler verwendet). Auf den Fehler konnte vorerst nicht weiter eingegangen werden.

Default-Package
✓ OK

Performance

✓ OK (Wenn eine Applikation mit AspectJ inspiziert wird verlangsamt sich die Geschwindigkeit um 100% und ist somit unter den gewünschten 900% bzw. 10 Mal langsamer als die eigentliche Geschwindigkeit der Applikation)

Integrierung
✓ OK

Filterung
Include-Filter
✗ AspectJ bietet keine Möglichkeit der Filterung mit ëxternenẄerten. Die Filterung kann nur ẗhardcodedïm AspectJ-File selbst vorgenommen werden.

Exclude-Filter
✗ (Begründung die selbe wie bei Include-Filter)

Laufzeitfilter
✗ (Begründung die selbe wie bei Include-Filter)

Plattformunabhängigkeit
Portabilität auf verschiedenen Plattformen
✓ OK

Portabilität auf verschiedene Betriebssystem
✓ OK

Testbarkeit
✗ In AspectJ gibt es keine Möglichkeit JUnit-Tests zu verwenden, da AspectJ nicht wie eine gewöhnliche Java-Klasse aufgebaut ist.

Literatur

[1] Vladimir Dzhuvinov. *JSON-RPC 2.0*, 12 2015. Link: http://software.dzhuvinov.com/json-rpc-2.0.html.

[2] Google. *Gson*. Google, 12 2015. Link: https://github.com/google/gson.

[3] Ceki Gülcü. *SLF4J*, 12 2015. Link: http://www.slf4j.org/.

[4] Kohsuke Kawaguchi. *args4j*, 12 2015. Link: http://args4j.kohsuke.org/.

Risikomanagement

Projekt: Runtime Object Visualization
Erstellt am: 27.09.2015
Autor: Anthamatten Adrian, Ravindran Jeyanthan
Gewichteter Schaden: 3.28

Nr	Titel	Beschreibung	max. Schaden [h]
R1	lokaler Datenverlust	Schaden am Arbeitsstation (Notebook) eines Projektmitglieds führt zu Datenverlust	8
R2	Auslesen der Runtime Object-Informationen komplex	Das auslesen der Runtime Object-Informationen ist komplexer als gedacht/geplant	32
R3	Temporärer Ausfall eines Projektmitglieds	Ein Projektmitglied fällt für mehr als 1 Tag aus und erzielt nicht die gewünschten Fortschritte im Projekt.	32
R3	Array auslesen mit JDI	Es besteht die Möglichkeit eine Änderung im Array über JDI auszulesen. Dies ist jedoch nich eine von JDI vorgegebene Funktion, sondern muss noch ergänzt werden. Hier besteht die Gefahr, dass wegen erhöhter Komplexität das Projekt in Verzug kommen könnte.	24
R4	Killerkriterium (siehe Anforderungsspezifikation - Auswertung) kann nicht implementiert werden	Ein Killerkriterium (z.B. auslesen von Methodenaufrufen in Default-Packages) kann nicht implementiert werden	72
Summe			**40**

Legende

aktuelle Risiken

nicht mehr gültige Risiken

Eintrittswahrscheinlichkeit	Gewichteter Schaden	Vorbeugung	Verhalten beim Eintreten
1%	0.08	Lokal gespeicherte Daten mindestens alle 4 Stunden einchecken	offline erledigte Arbeit (noch nicht synchronisierte) geht verloren und muss nochmals getan werden
10%	3.2	Reserve einplanen, indem 10-20% der verfügbaren Zeit nicht verplant wird	Exerne Hilfe holen oder auf ein anderes Framework verwenden zum auslesen der Object-Informationen
10%	3.2	Reserve einplanen, damit im Falle des Ausfalls der "verbliebene" Projektmitglied die offenen Aufgaben übernehmen kann	Der "verbliebene" Projektmitglied versucht die offenen Aufgaben zu übernehmen oder passt die Projektplanung entsprechend an.
25%	6	Da bei eine Verzug nicht alle vom Arbeitgeber gewünschten Funktionen iimplementiert werden können, mit dem Arbeitgeber definieren, welche Funktionen "optional" sind.	Das Auslesen von Arrayinformationen (CRUD) ist ein Killerkriterium und muss implementiert werden. Dafür werden einzelne optionale Funktionen wegfallen.
2%	1.44	Beim Auswahl der Technologie (Evaluation) überprüfen, ob die Technologie in der Lage ist alle Killerkriterien zu erfüllen. Desweiteren ein Backuptechnologie als Reserve bestimmen.	Entweder die Funktion mit einer Technologie implementieren und diese mit dem bereits vorhanden kombinieren oder die aktuelle Technologie vollständig durch die Backuptechnologie ersetzten.
	3.28		

ROV Testing

Adrian Anthamatten & Jeyanthan Ravindran

18. Dezember 2015

Inhaltsverzeichnis

1 Integrationstests

Die Integrationstests sind hier Dokumentiert. In der 'Ausgangslage' wird ein bestimmter Code-Teil beschrieben, welcher sich in der Java-Applikation befindet, der inspziert werden soll. Die Sektion 'Erwartet' beschreibt in Worten welche Log-Ausgaben erwartet werden. Die Tatsächlich ausgegebene Logs sind im tatsächliche 'Ausgabe' ersichtlich. Zum Schluss wird im 'Testergebnis' beschrieben, ob der Test erfolgreich war oder fehlgeschlagen hat.

1.1 Wert-Änderung - Static Reference

1.1.1 Ausgangslage

Eine statische Referenz Namens 'staticRef' wird erstellt.

```
public class Main {
    private static SimpleClass staticRef = new SimpleClass();
    ...
```

1.1.2 Erwartet

Erwartet wird eine Log-Ausgabe für die Instanziierung der Klasse 'SimpleClass' und für das Zuweisen jener Instanz an 'staticRef'.

1.1.3 Ausgabe

```
DEBUG ClientSocketConnetion:30 - Send json-rpc-request to server: {"method":"newObject","id":"req-1","params":
 {"className":"SimpleClass","packageName":"main","objectId":403},"jsonrpc":"2.0"}
DEBUG ServerStub:79 - Got json-rpc-request from client: {"method":"newObject","id":"req-1","params":
 {"className":"SimpleClass","packageName":"main","objectId":403},"jsonrpc":"2.0"}
INFO  RuntimeObjectRequestHandler:193 - New Object handle...
DEBUG RuntimeObjectRequestHandler:200 - New Object: objectId '403', packageName 'main' className 'SimpleClass'
```

1.1.4 Testergebnis

✓ Test bestanden

1.2 Methodenaufruf - Static

1.2.1 Ausgangslage

Die statische Methode 'main' wird mit dem Argument 'args' aufgerufen.

```
...
public static void main(String[] args) {
...
```

1.2.2 Erwartet

Erwartet wird eine Log-Ausgabe für den Methodenaufruf.

1.2.3 Ausgabe

```
DEBUG ClientSocketConnetion:30 - Send json-rpc-request to server: {"method":"staticMethodCall","id":"req-1",
 "params": {"ownerObjectType":"Main","signature":"[{\"name\":\"args\",\"value\":\"(...)"}]","methodName":"main"},
  "jsonrpc":"2.0"}
DEBUG ServerStub:79 - Got json-rpc-request from client: {"method":"staticMethodCall","id":"req-1","params":
 {"ownerObjectType":"Main","signature":"[{\"name\":\"args\",\"value\":\"(...)"}]","methodName":"main"},
  "jsonrpc":"2.0"}
INFO  RuntimeObjectRequestHandler:77 - static Method Call handle...
DEBUG RuntimeObjectRequestHandler:85 - Static Method call: ownerObjectType 'Main', methodName 'main',
 signatureSize '1'
```

1.2.4 Testergebnis

✓ Test bestanden

1.3 Methodenaufruf - Non-Static

1.3.1 Ausgangslage

Die statische Methode 'methodOne' wird mit dem Argument '1' aufgerufen.

```
...
hw.methodOne(1);
...
```

1.3.2 Erwartet

Erwartet wird eine Log-Ausgabe für den Methodenaufruf.

1.3.3 Ausgabe

```
DEBUG ClientSocketConnetion:30 - Send json-rpc-request to server:
 {"method":"methodCall","id":"req-1","params":{"signature":"[{\"name\":\"var\",\"value\":\"1\"}]",
  "methodName":"methodOne","objectId":408},"jsonrpc":"2.0"}
DEBUG ServerStub:79 - Got json-rpc-request from client: {"method":"methodCall","id":"req-1",
 "params": {"signature":"[{\"name\":\"var\",\"value\":\"1\"}]","methodName":"methodOne",
  "objectId":408},"jsonrpc":"2.0"}
INFO  RuntimeObjectRequestHandler:250 - Method call handle...
DEBUG RuntimeObjectRequestHandler:258 - Method call: objectId '408', methodName 'methodOne'
 nrOfArguments in signature '1'
```

1.3.4 Testergebnis

✓ Test bestanden

1.4 Wert-Änderung - Static Attribute

1.4.1 Ausgangslage

Als Log-Ausgabe wird eine Wertänderung für den statischen Attribut 'staticInt' erwartet.

```
...
SimpleClass.staticInt = 5;
...
```

1.4.2 Erwartet

Erwartet wird eine Log-Ausgabe für den Methodenaufruf.

1.4.3 Ausgabe

```
DEBUG ClientSocketConnetion:30 - Send json-rpc-request to server: {"method":"changedStaticAttribute",
 "id":"req-1","params":{"newValue":"5","ownerObjectType":"main.SimpleClass","attribute":"staticInt"},
 "jsonrpc":"2.0"}
DEBUG ServerStub:79 - Got json-rpc-request from client: {"method":"changedStaticAttribute",
 "id":"req-1","params":{"newValue":"5","ownerObjectType":"main.SimpleClass","attribute":"staticInt"},
 "jsonrpc":"2.0"}
INFO  RuntimeObjectRequestHandler:108 - Changed static attribute handle...
DEBUG RuntimeObjectRequestHandler:115 - Changed attribute: objectId 'main.SimpleClass',
 attribute 'staticInt', newValue '5'
```

1.4.4 Testergebnis

✓ Test bestanden

1.5 Wert-Änderung - String

1.5.1 Ausgangslage

Wertänderung für String sollen nicht als Referenz-Änderung, sondern Attribut-Änderung angezeigt werden.

```
...
public String name = "Heijyo";
...
```

1.5.2 Erwartet

Erwartet wird eine Log-Ausgabe für eine Attribut-Änderung.

1.5.3 Ausgabe

```
DEBUG ClientSocketConnetion:30 - Send json-rpc-request to server: {"method":"changedAttribute",
 "id":"req-1","params":{"newValue":"\"Heijyo\"","attribute":"name","objectId":403},"jsonrpc":"2.0"}
DEBUG ServerStub:79 - Got json-rpc-request from client: {"method":"changedAttribute","id":"req-1",
 "params":{"newValue":"\"Heijyo\"","attribute":"name","objectId":403},"jsonrpc":"2.0"}
INFO  RuntimeObjectRequestHandler:220 - Change attribute handle...
DEBUG RuntimeObjectRequestHandler:227 - Changed attribute: objectId '403', attribute 'name' newValue '"Heijyo"'
```

1.5.4 Testergebnis

✓ Test bestanden

1.6 Wert-Änderung - Static String

1.6.1 Ausgangslage

Wertänderung für statischen String sollen nicht als Referenz-Änderung, sondern als statische Attribut-Änderung angezeigt werden.

```
...
public static String staticString = new String("s");
...
```

1.6.2 Erwartet

Erwartet wird eine Log-Ausgabe für eine statische Attribut-Änderung.

1.6.3 Ausgabe

```
DEBUG ClientSocketConnetion:30 - Send json-rpc-request to server: {"method":"changedStaticAttribute",
 "id":"req-1","params":{"newValue":"\"s\"","ownerObjectType":"main.SimpleClass","attribute":"staticString"},
 "jsonrpc":"2.0"}
DEBUG ServerStub:79 - Got json-rpc-request from client: {"method":"changedStaticAttribute","id":"req-1",
 "params":{"newValue":"\"s\"","ownerObjectType":"main.SimpleClass","attribute":"staticString"},"jsonrpc":"2.0"}
INFO  RuntimeObjectRequestHandler:108 - Changed static attribute handle...
DEBUG RuntimeObjectRequestHandler:115 - Changed attribute: objectId 'main.SimpleClass', attribute 'staticString',
 newValue '"s"'
```

1.6.4 Testergebnis

✓ Test bestanden

1.7 Werte-Änderung - Primitiver Array

1.7.1 Ausgangslage

Die Initialisierung eines primitiven Arrays wird auch als Werteänderung aufgefasst.

```
...
public int[] primitiveArray = {2, 4, 8};
...
```

1.7.2 Erwartet

Erwartet wird eine Log-Ausgabe für die Werte-Änderung im Array, da die Werte 2, 4 und 8 neu hinzugefügt werden.

1.7.3 Ausgabe

Die Ausgabe wurde hier aus Platzgründen auf das Hinzufügen des Wertes 2 (auf Index 0) abgekürzt:

```
DEBUG ClientSocketConnetion:30 - Send json-rpc-request to server: {"method":"changedArrayAttribute",
 "id":"req-1","params":{"newValue":"2","index":0,"attribute":"primitiveArray","objectId":409},"jsonrpc":"2.0"}
DEBUG ServerStub:79 - Got json-rpc-request from client: {"method":"changedArrayAttribute","id":"req-1",
 "params":{"newValue":"2","index":0,"attribute":"primitiveArray","objectId":409},"jsonrpc":"2.0"}
INFO  RuntimeObjectRequestHandler:173 - Change array attribute handle...
DEBUG RuntimeObjectRequestHandler:181 - Changed array attribute: objectId '409', index '0',
 attribute 'primitiveArray', newValue '2'
...
```

1.7.4 Testergebnis

✓ Test bestanden

1.8 Werte-Änderung - Array-Reference

1.8.1 Ausgangslage

Die Initialisierung eines Arrays-Reference wird auch als Werteänderung aufgefasst. Desweiteren wird ein String-Array als Array-Reference angesehen (nicht als primitiver Array).

```
...
public String[] referenceArray = {"kal", "pant", "hu"};
...
```

1.8.2 Erwartet

Erwartet wird eine Log-Ausgabe für die Werte-Änderung im Array, da neue String-Werte hinzugefügt werden.

1.8.3 Ausgabe

Die Ausgabe wurde hier aus Platzgründen auf das Hinzufügen des Wertes 'hu' (auf Index 2) abgekürzt:

```
...
DEBUG ClientSocketConnetion:30 - Send json-rpc-request to server: {"method":"changedArrayAttribute","id":"req-1",
 "params":{"newValue":"\"hu\"","index":2,"attribute":"referenceArray","objectId":410},"jsonrpc":"2.0"}
DEBUG ServerStub:79 - Got json-rpc-request from client: {"method":"changedArrayAttribute","id":"req-1",
 "params":{"newValue":"\"hu\"","index":2,"attribute":"referenceArray","objectId":410},"jsonrpc":"2.0"}
INFO  RuntimeObjectRequestHandler:173 - Change array attribute handle...
DEBUG RuntimeObjectRequestHandler:181 - Changed array attribute: objectId '410', index '2',
 attribute 'referenceArray', newValue '"hu"'
```

1.8.4 Testergebnis

✓ Test bestanden

1.9 Werte-Änderung - statischer primitiver Array

1.9.1 Ausgangslage

Die Initialisierung eines statischen primitiven Arrays wird als Werteänderung aufgefasst.

```
...
public static int[] staticPrimitiveArray = {1, 1, 2, 3};
...
```

1.9.2 Erwartet

Erwartet wird eine Log-Ausgabe für die Werte-Änderung im Array, da neue Werte hinzugefügt werden.

1.9.3 Ausgabe

Die Ausgabe wurde hier aus Platzgründen auf das Hinzufügen des Wertes '2' (auf Index 2) abgekürzt:

```
...
DEBUG ClientSocketConnetion:30 - Send json-rpc-request to server: {"method":"changedStaticArrayAttribute",
 "id":"req-1","params":{"newValue":"2","ownerObjectType":"main.SimpleClass","index":2,
 "attribute":"staticPrimitiveArray"},"jsonrpc":"2.0"}
DEBUG ServerStub:79 - Got json-rpc-request from client: {"method":"changedStaticArrayAttribute","id":"req-1",
 "params":{"newValue":"2","ownerObjectType":"main.SimpleClass","index":2,"attribute":"staticPrimitiveArray"},
 "jsonrpc":"2.0"}
INFO  RuntimeObjectRequestHandler:140 - Changed static array attribute handle...
DEBUG RuntimeObjectRequestHandler:148 - Changed static array attribute: objectId 'main.SimpleClass', index '2',
 attribute 'staticPrimitiveArray', newValue '2'
...
```

1.9.4 Testergebnis

✓ Test bestanden

1.10 Werte-Änderung - statischer Array-Referenz

1.10.1 Ausgangslage

Die Initialisierung eines statischen Array-Referenz wird als Werteänderung aufgefasst.

```
...
public static String[] staticReferenceRawArray = {"p", "pu", "pan"};
...
```

1.10.2 Erwartet

Erwartet wird eine Log-Ausgabe für die Werte-Änderung im Array, da neue String-Werte hinzugefügt werden.

1.10.3 Ausgabe

Die Ausgabe wurde hier aus Platzgründen auf das Hinzufügen des Wertes 'pu' (auf Index 1) abgekürzt:

```
...
DEBUG ClientSocketConnetion:30 - Send json-rpc-request to server: {"method":"changedStaticArrayAttribute",
 "id":"req-1","params":{"newValue":"\"pu\"","ownerObjectType":"main.SimpleClass","index":1,
 "attribute":"staticReferenceRawArray"},"jsonrpc":"2.0"}
DEBUG ServerStub:79 - Got json-rpc-request from client: {"method":"changedStaticArrayAttribute","id":"req-1",
 "params":{"newValue":"\"pu\"","ownerObjectType":"main.SimpleClass","index":1,
 "attribute":"staticReferenceRawArray"},"jsonrpc":"2.0"}
INFO  RuntimeObjectRequestHandler:140 - Changed static array attribute handle...
DEBUG RuntimeObjectRequestHandler:148 - Changed static array attribute: objectId 'main.SimpleClass', index '1',
 attribute 'staticReferenceRawArray' newValue '"pu"'
...
```

1.10.4 Testergebnis

✓ Test bestanden

Literatur

ROV Produktstatus

Adrian Anthamatten & Jeyanthan Ravindran

18. Dezember 2015

Inhaltsverzeichnis

1 Produktstatus

Dieses Dokument gibt einen Überblick über den aktuellen Status der Entwicklung von Runtime Object Visualization.

1.1 Funktionen und Anforderungen

Die detaillierten Beschreibungen der jeweiligen Funktionen sind im Softwarearchitektur ersichtlich.

1.1.1 Überwachen relevanter Daten

Release 2	Release 3	Release 4	Funktion	Kommentar
✓	✓	✓	Felder überwachen	
✓	✓	✓	Methodenaufrufen überwachen	
✗	✓	✓	Statische Variablen	
✗	✓	✓	Statische Methodenaufrufe überwachen	
✓	✓	✓	Konstruktoren überwachen	
✗	✗	✗	Destruktoren überwachen	
✗	✗	✓	Referenz-Arrays überwachen	Überwachung von Arrays, welche als Elemente Referenze enthalten
✗	✓	✓	Primitive-Arrays überwachen	Überwachung von Arrays, welche als Elemente mit primitiven Datentypen enthalten
✗	✗	✓	Collection-Klassen überwachen	Listen überwachen, welche Collection implementieren
✓	✓	✓	Default-Package überwachen	
✓	✓	✓	Filtern	Filter
✗	✓	✓	Filtern	Laufzeit-Filter
✓	✓	✓	Socketverbindung mit Plattformunabhängigen Protokoll	Wurde mit JSON realisiert
✗	✓	✓	Free-Port-Detection	Falls der Start-Port besetzt ist, wird automatisch der nächste freie Port ausgewählt

1.1.2 weitere Anforderungen

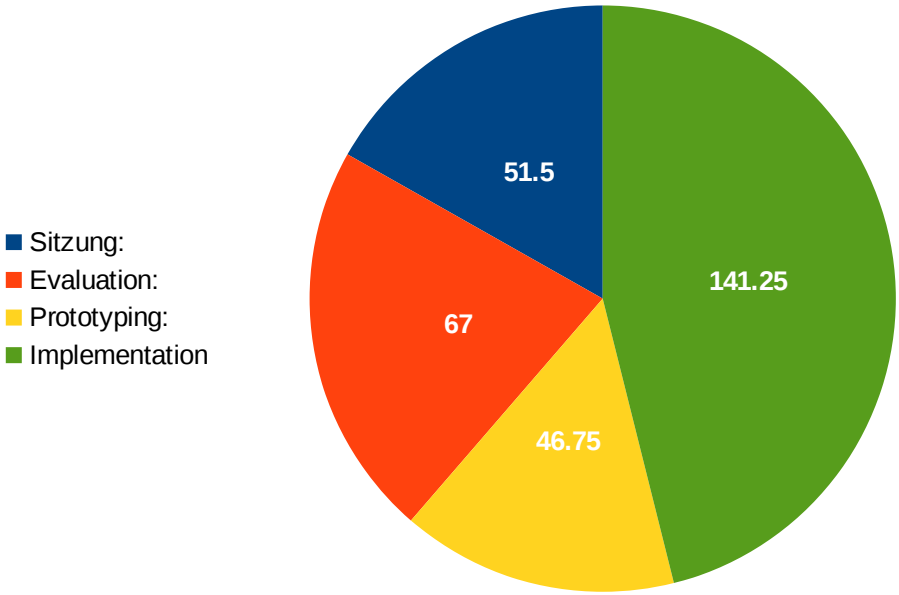
Die detaillierten Beschreibungen der jeweiligen Funktionen sind im Softwarearchitektur ersichtlich.

Release 2	**Release 3**	**Release 4**	**Funktion**	**Kommentar**
✗	✗	✗	Performance	
✓	✓	✓	Portabilität auf verschiedene Plattformen	
✓	✓	✓	Portabilität auf verschiedene Betriebssysteme	
✗	✗	✓	Serverstub Auto-Shutdown	

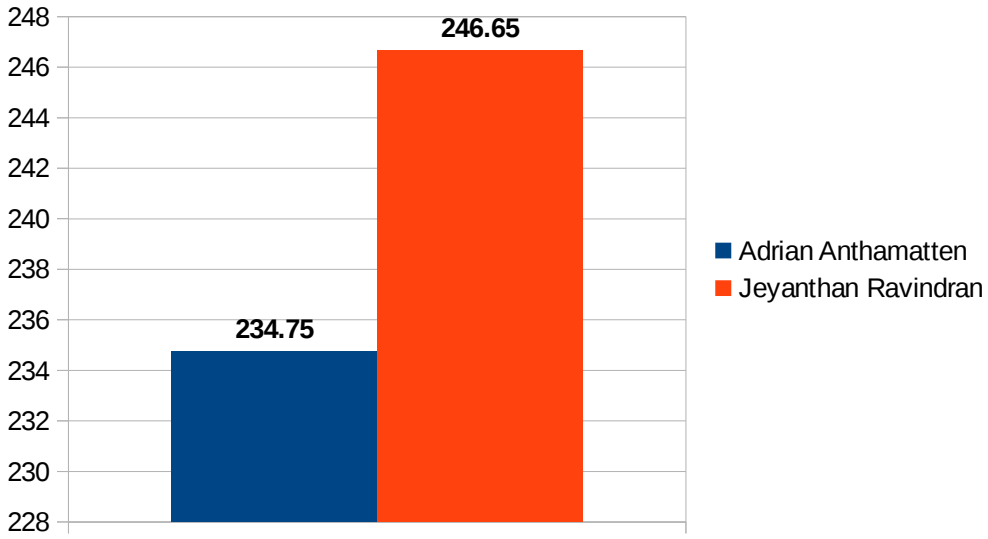
Literatur

Zeitauswertung

Zeit in Kategorien



Zeit pro Mitglied



Zeit gesamthaft:	481.4