

Benutzerhandbuch ShuttleTycoon



Gruppe 04
Softwaretechnikpraktikum 2003
Universität Paderborn

Impressum

Projektleiter:	Stefan Feldkord
Stellvertretender Projektleiter:	Jochen Meissner
Qualitätsbeauftragter:	Thilo Sülberg
Reengineering-Beauftragter:	Dennis Hannwacker
Pflichtenheft-Beauftragter:	Arkadius Gawrych
(Re-)Design-Beauftragter:	Marcus Dürksen
Implementierungs-Beauftragter:	Henrik Niehaus
Test-Beauftragter:	Bernd Niklas Klein
Präsentation:	Jochen Meissner Thilo Sülberg
Web-Auftritt:	Alexander Gebel Axel Vincenz
Betreuer:	Stefan Sauer

Inhaltsverzeichnis

1	Einleitung	1
2	Gesamtarchitektur	3
3	Shuttlesteuerung	4
3.1	ShuttleTycoon	4
3.2	MessageReceivedQueue	4
3.3	MessageSentQueue	4
3.4	Strategy	4
3.5	Analysis	4
3.6	Planner	6
3.7	Plan / PlanElem	6
3.8	WaitList	6
3.9	Evaluation	6
3.9.1	Evaluator	6
3.9.2	Bewertung eines Auftrages anhand von Planvorschlägen	7
3.10	Broker	7
3.11	Negotiator	7
3.12	PathFinder	9
3.13	Banker	9
3.14	PluginData	9
4	Integration eines neuen Auftragsverhandlungsmoduls	10
5	Plugin	12
5.1	Beschreibung der neuen Plugin-Umgebung	12
5.2	Datentransfer zum Plugin	15
6	Rekorder	18
6.1	Aufbau und Funktionsweise	18
6.2	Datenformat	21
6.3	Zustände des Rekorders	22
7	Analysemodul	24
7.1	Datenstruktur	24
7.2	Daten der Shuttles	24
7.3	Das Streckennetz	26
7.4	Gesamtstruktur	26
7.4.1	Analyse_Gui	26
7.4.2	RMIClient	28

7.4.3	ConnectDialog	28
7.4.4	RML_Converter	28
7.4.5	Converter	28
7.4.6	DataVault	28

1 Einleitung

Im Rahmen des diesjährigen Softwaretechnik-Praktikums an der Universität Paderborn sollte die bestehende Simulationsumgebung für ein neuartiges Transportsystem erweitert und verbessert werden. Dazu war zunächst eine intelligente Shuttlesteuerung zu entwickeln, die die folgenden wesentliche Anforderungen erfüllen sollte:

- Der Shuttle muss sich innerhalb eines bekannten Schienennetzes autonom bewegen können. Dies beinhaltet die selbständige Wegfindung, das Einhalten von Wartungsintervallen sowie der Umgang mit auftretenden Streckenausfällen.
- Die Bearbeitung mehrerer, zeitgleicher Aufträge muss gewährleistet sein. Dazu muss der Shuttle diese logistisch planen, Angebote kalkulieren und verschiedenen Abrechnungsverfahren beherrschen.
- Der Shuttle selbst soll modular aufgebaut sein, um eine leichte Erweiterbarkeit mit neuen Modulen zu gewährleisten.
- Das Ziel ist schließlich, hiermit eine möglichst gewinnbringende Shuttle-Strategie umzusetzen.

Zu diesem Shuttle sollte ein Plugin entwickelt werden, das die bestehende Visualisierung erweitert. Damit sollten Informationen über den internen Status des Shuttles an die Benutzer weitergegeben werden.

Weiterhin war während dieses Projekts eine Analyseumgebung zu realisieren, die die Daten mehrerer verschiedener Shuttles sowie der Simulation selbst sammelt und statistisch auswertet. Die daraus resultierenden Ergebnisse sollten graphisch dargestellt werden. Zudem sollte es den Benutzern möglich sein, diese Statistiken miteinander zu kombinieren.

Als letztes war noch ein Rekorder-Modul zu entwickeln. Dieses sollte die Möglichkeiten bieten, Simulationen online an mehrere Klienten weiterzuleiten sowie diese aufzuzeichnen, um sie später offline wiederzugeben zu können. Außerdem sollten jeweils die Funktionen *Pause* und *Schneller Vorlauf* realisiert werden.

Dieses Dokument zeigt nun auf, wie diese Anforderungen von unserer Projektgruppe umgesetzt wurden. Es basiert auf dem Sollkonzept des Pflichtenheftes und dokumentiert die wesentlichen Entwurfsentscheidungen.

Hierzu wollen wir im folgenden Kapitel 2 zunächst einen Einblick geben, wie die von uns entwickelten Komponenten in das bestehende System integriert wurden.

In Kapitel 3 werden wir dann auf die genauere Realisierung der Shuttlesteuerung eingehen. Dazu werden wir die einzelnen Module des ShuttleTycoon näher erklären sowie deren Zusammenhänge deutlich machen.

Kapitel 4 beschreibt anschließend, wie der ShuttleTycoon um weitere Module erweitert werden kann. Dies haben wir dort beispielhaft für die Integration eines neuen Verhandlungsmoduls erklärt.

In Kapitel 5 werden wir dann das Plugin vorstellen. Hier gibt es insbesondere ein paar Neuerungen gegenüber dem Sollkonzept des Pflichtenheftes, auf die wir ausführlich eingehen werden.

In Kapitel 6 beschreiben wir unsere Realisierung des Rekorder. Dabei haben wir neben der Implementierung der oben genannten Funktionen besonderes Augenmerk darauf gelegt, dass auch sehr lange laufenden Simulationen problemlos aufgezeichnet und wiedergegeben werden können.

Den Schluss bildet Kapitel 7, das das Analyse-Modul beschreibt. Auch hier haben wir bei der Umsetzung darauf geachtet, den Benutzern ein funktionales und leicht zu bedienendes Werkzeug an die Hand zu geben, dass allen ihren Erwartungen gerecht wird.

2 Gesamtarchitektur

Die Architektur des gesamten Projekts besteht aus verschiedenen Komponenten, von denen jede für sich genommen eine voll funktionsfähige Einheit darstellt. Abbildung 1 zeigt wie die Komponenten mit dem vorhandenen System bestehend aus **Kernel** und **Visualisation** sowie untereinander in Zusammenhang stehen.

Der Hauptteil der Kommunikation läuft dabei über die Schnittstelle **RMIVisualisationInterface**, die von allen Klienten implementiert werden muss, um Daten von der Simulation erhalten zu können. Im einzelnen sind dies die Visualisierung selbst sowie die von uns entwickelten Komponenten **Analyse** und **Recorder**. Der **Recorder** nutzt das Interface jedoch nicht nur, um selbst Daten vom **Kernel** zu empfangen, sondern versendet hierüber ebenfalls entweder weitergeleitete oder aufgezeichnete Daten.

Die Klasse **ShuttleTycoon** als Herzstück der Shuttlesteuerungskomponente **Agent** kommuniziert direkt mit dem **Kernel**, indem sie die Klasse **ShuttleAgentBase** spezialisiert und so ihre Funktionen zum Versenden von Nachrichten erbt. Dies gilt gleichermaßen für das **ShuttleTycoonPlugin** als Teil der **Plugin**-Komponente, das die zu **Visualisation** gehörende Klasse **PluginBase** spezialisiert.

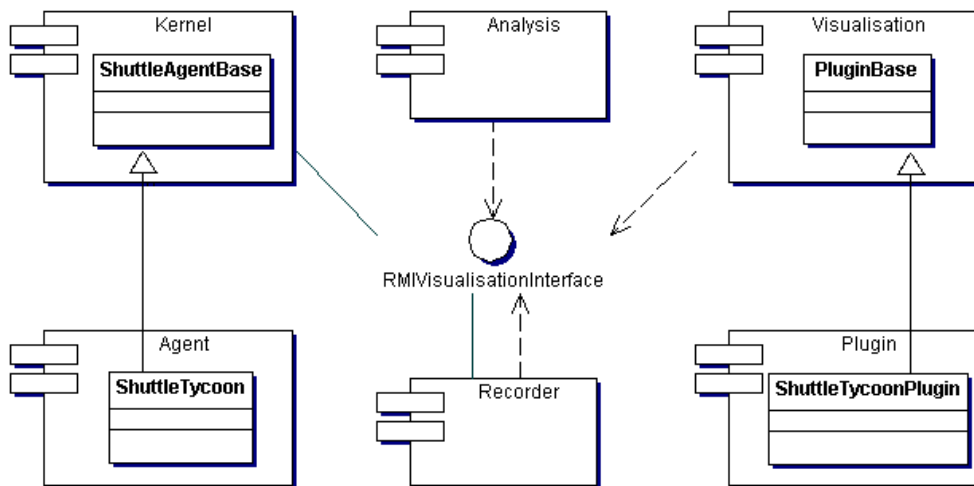


Abbildung 1: Gesamtarchitektur des Projekts

3 Shuttlesteuerung

Dieses Kapitel beschreibt die Klassen, die zusammen für die Steuerung des ShuttleTycoon verantwortlich sind. Abbildung 2 gibt einen Überblick, wie die Klassen im einzelnen zusammenhängen.

3.1 ShuttleTycoon

Die Klasse `ShuttleTycoon` erweitert `ShuttleAgentBase` und stellt damit die Schnittstelle zwischen dem Simulationskern (`Kernel`) und den Modulen des Shuttles dar. Sie übernimmt die Verteilung der ankommenden Nachrichten an die Module des Shuttles sowie das Versenden von Nachrichten von diesen Modulen an den Simulationskern.

3.2 MessageReceivedQueue

Die Klasse `MessageReceivedQueue` speichert und priorisiert ankommende Nachrichten und stellt diese der Klasse `ShuttleTycoon` zur späteren Verteilung an die Module zur Verfügung.

3.3 MessageSentQueue

Die Klasse `MessageSentQueue` sammelt und priorisiert Nachrichten, die die Module an den Simulationskern senden wollen, und stellt diese der Klasse `ShuttleTycoon` zur endgültigen Versendung an den Simulationskern zur Verfügung.

3.4 Strategy

Die Klasse `Strategy` lädt Strategieparameter aus Konfigurationsdateien (`.cfg`) und stellt diese anderen Modulen zur Verfügung und kann leicht um neue Parameter erweitert werden. Lesen Sie mehr zu den Strategieparametern im Kapitel 2 des Benutzerhandbuchs zum ShuttleTycoon.

3.5 Analysis

Die Klasse `Analysis` analysiert Daten und stellt die Ergebnisse dieser Auswertungen anderen Modulen zur Verfügung. Sie kann leicht um neue statistische Werte erweitert werden.

CT

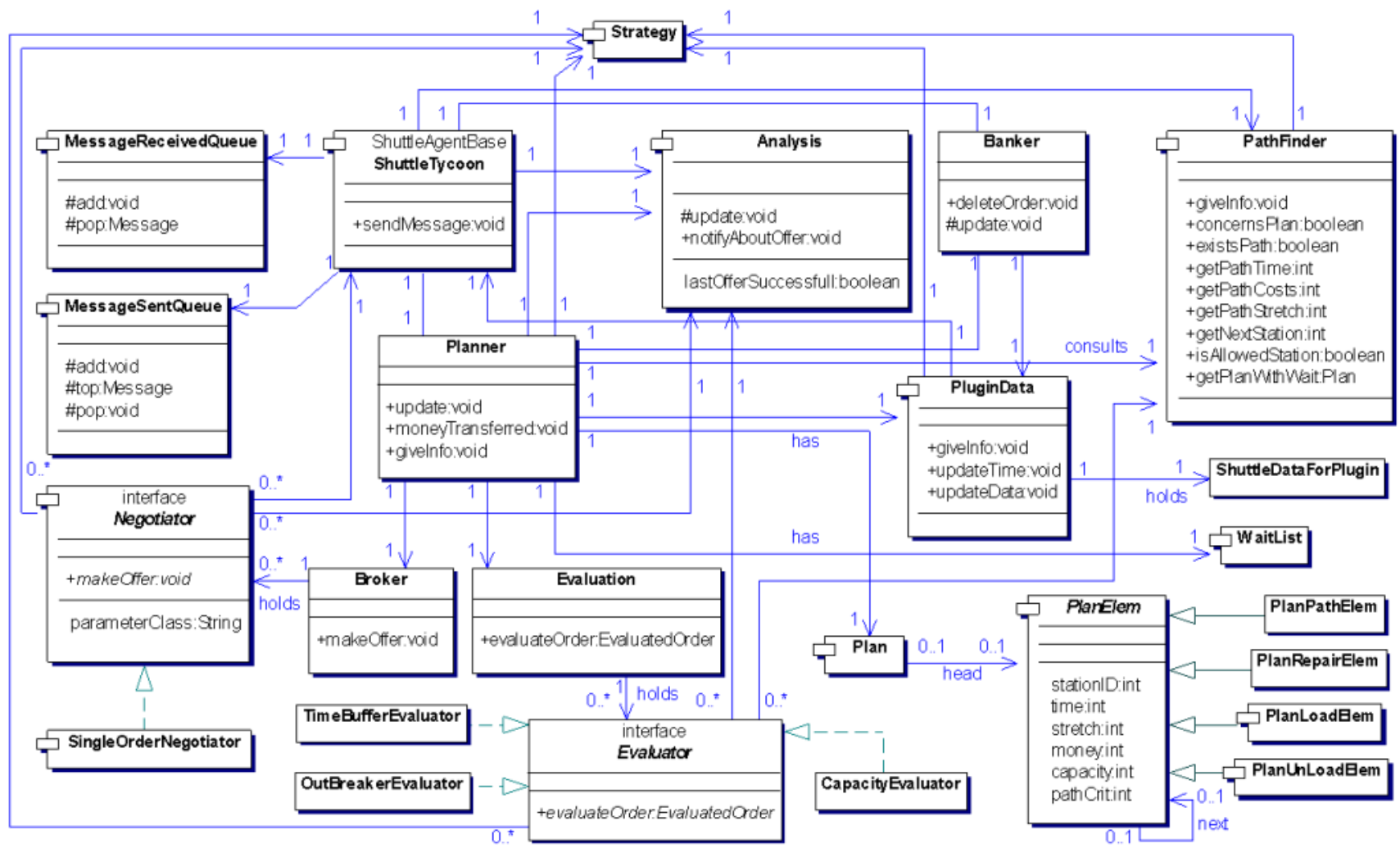


Abbildung 2: Struktur der Shuttlesteuerung

3.6 Planner

Der **Planner** ist das „Gehirn“ des ShuttleTycoon. Er verwaltet den Plan und arbeitet diesen Punkt für Punkt ab. Für verfügbare Aufträge berechnet er mithilfe des **PathFinder** Planvorschläge, lässt diese über die **Evaluation** von den Evaluatoren bewerten und gibt den **Broker** den Auftrag an ein Auftragsverhandlungsmodul (**Negotiator**), ein Angebot abzugeben (siehe auch Kapitel 3.9.2).

Mithilfe des **PathFinder** kann er auch feststellen, ob Streckenausfälle den Fahrplan betreffen und gegebenenfalls Umwege berechnen oder Haltepunkte setzen, die dann in der **WaitList** gespeichert werden.

Über geplante bzw. gerade ausgeführte Aktionen informiert der **Planner** **PluginData**.

3.7 Plan / PlanElem

Der **Plan** (**Plan**) besteht aus einzelnen Planelementen (**PlanElem**). Planelemente stellen auszuführende Aktionen dar, wie z.B. das Be- und Entladen oder die Wartung eines Shuttles. Zu jedem Planelement ist festgehalten, an welcher Station die jeweilige Aktion durchzuführen ist, sowie Informationen über die zurückgelegte Strecke und die Kapazität des Shuttles nach Ausführung der jeweiligen Aktion. Außerdem ist festgehalten, zu welchem Zeitpunkt die geplante Aktion voraussichtlich ausgeführt sein wird.

3.8 WaitList

Die **WaitList** ist eine Liste von Haltepunkten. Haltepunkte sind Stationen, an denen der Shuttle warten muss, bis die Verbindung, die er von dieser Station aus befahren möchte, wieder befahrbar ist.

3.9 Evaluation

Die Klasse **Evaluation** lädt und verwaltet die Evaluatoren (**Evaluator**). Wenn sie vom **Planner** den Auftrag erhält, einen Planvorschlag für einen Auftrag zu bewerten, schleust sie den Planvorschlag durch alle von ihr geladenen Evaluatoren, lässt diese den Planvorschlag bewerten und gibt das Gesamtergebnis dieser Bewertungen an den **Planner** zurück.

3.9.1 Evaluator

Ein **Evaluator** bewertet einen Planvorschlag hinsichtlich eines einzelnen Kriteriums.

Der `TimeBufferEvaluator` untersucht den Planvorschlag hinsichtlich des Zeitpolsters zwischen dem Entladen des Auftrages und dem Verstreichen der zugehörigen Deadline.

Der `CapacityEvaluator` bewertet die Auslastung des Shuttles, und der `OutBreakerEvaluator` bevorzugt Planvorschläge, die den Shuttle näher zur gewünschten Zusammenhangskomponente bringen.

3.9.2 Bewertung eines Auftrages anhand von Planvorschlägen

Der wesentliche Ablauf, wie der `ShuttleTycoon` einen Auftrag anhand von Planvorschlägen bewertet, ist im Sequenzdiagramm in Abbildung 3 dargestellt.

Kommt eine Nachricht über die Verfügbarkeit eines Auftrages im `ShuttleTycoon` an, so wird diese Nachricht mittels der `update`-Methode an den `Planner` weitergegeben. Dieser erstellt dann für den Auftrag mithilfe der Methode `layoutOrder` Planvorschläge für diesen Auftrag und lässt diese von der `Evaluation` bewerten.

Die `Evaluation` delegiert die Bewertung von Planvorschläge an die einzelnen Evaluatoren, die von der `Evaluation` geladen und verwaltet werden. Jeder `Evaluator` bewertet den Planvorschlag hinsichtlich eines einzelnen Kriteriums. Das Gesamtergebnis der Bewertung wird von der `Evaluation` an den `Planner` zurückgegeben.

Der `Planner` weist dann den `Broker` an, ein Angebot für den bewerteten Planvorschlag abzugeben. Der `Broker` verwaltet die von ihm geladenen `Negotiator` und delegiert dann die Angebotsabgabe an den `Negotiator`, der für die Angebotsabgabe zuständig ist. In diesem Fall ist das der `SingleOrderNegotiator`, der für Planvorschläge einzelner Aufträge zuständig ist.

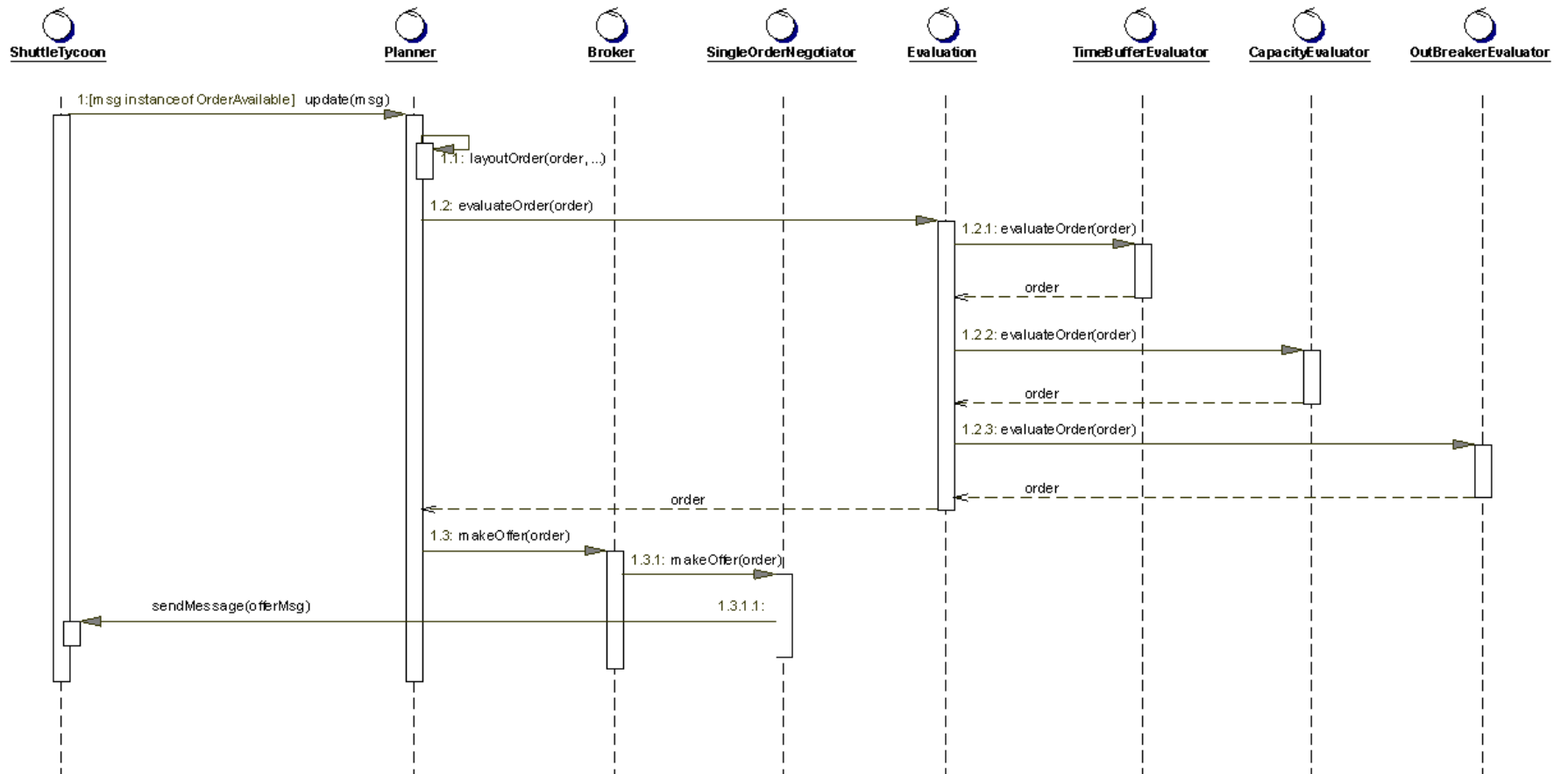
Der `SingleOrderNegotiator` kalkuliert das Angebot und sendet es über `ShuttleTycoon` an den `Kernel`.

3.10 Broker

Der `Broker` lädt und verwaltet die `Negotiator`. Wenn er vom `Planner` den Auftrag erhält, für ein Objekt ein Angebot abzugeben, sucht er den für diesen Objekttyp zuständigen `Negotiator` heraus und weist diesen an, ein Angebot zu machen.

3.11 Negotiator

Ein `Negotiator` macht für einen von ihm festgelegten Objekttyp Angebote. Der `SingleOrderNegotiator` macht Angebote für Planvorschläge einzelner Aufträge.



∞

Abbildung 3: Bewertung eines Auftrags

3.12 PathFinder

Der **PathFinder** übernimmt die Datenhaltung des Streckennetzes sowie die Wegberechnung in diesem Netz. Er kennt zu jedem Zeitpunkt alle befahrba- ren Wege, deren Eigenschaften (wie z.B. Fahrzeit, Streckenlänge, etc.) sowie alle ausgefallenen Strecken und deren Ausfalldauer. Er hat auch alle erreich- baren Zusammenhangskomponenten erfasst und eine davon als gewünscht markiert. Das Kriterium nach dem die gewünschte Zusammenhangskompo- nente ausgewählt wird ist standardmäßig die Größe.

3.13 Banker

Der **Banker** übernimmt die Abrechnung erfolgreich ausgeführter Aufträge. Er kann nach Ausführung des Auftrages autonom das Kreditkartenverfahren, so- wie das Rechnungs- und Mahnverfahren abwickeln. Über den aktuellen Stand der Abrechnung von Aufträgen informiert der **Banker PluginData**.

3.14 PluginData

PluginData verwaltet in **ShuttleDataForPlugin** die Daten, die an das Plugin geschickt werden sollen, und verschickt diese in bestimmten Zeitabständen als kodierten String über eine **ShuttleAgentStatusMessage** (siehe Abschnitt 5.2).

4 Integration eines neuen Auftragsverhandlungsmoduls

Die Architektur des ShuttleTycoon ist von Grund auf modular gehalten und so leicht um neue Funktionen und Module erweiterbar. Die Auftragsverhandlung macht hier keine Ausnahme. Jedes Auftragsverhandlungsmodul muss das Interface **Negotiator** (siehe Abbildung 4) implementieren. Die Methode `init` dient hierbei zur Initialisierung des Moduls und zur Übergabe aller benötigten Referenzen. Über die Referenz auf **ShuttleTycoon** kann der **Negotiator** Nachrichten an den Simulationskern schicken lassen.

Die Analyse stellt ihm alle ausgewerteten Daten zur Verfügung, und über **Strategy** hat er Zugriff auf alle Strategieparameter (vergleiche Abbildung 2). Um dem **Broker** mitzuteilen, für welchen Objekttyp er Angebote tätigen möchte, existiert die Methode `getParameterClass`. Soll der **Negotiator** ein Angebot abgeben, wird die Methode `makeOffer` aufgerufen und das Objekt, für das ein Angebot getätigt werden soll, als Parameter übergeben.

Alle **Negotiator** werden vom **Broker** geladen und verwaltet. Welche **Negotiator** vom **Broker** geladen werden, steht in der Datei `defbroker.cfg`, die eine Liste aller zu ladenden **Negotiator** enthält.

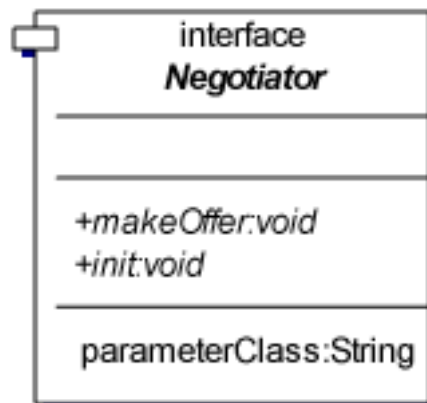


Abbildung 4: Das Negotiator Interface

Um den Shuttle um die Funktionalität des *Last-Minute-Angebots* zu erweitern, sind die folgenden Schritte auszuführen:

1. Erstellung einer neuen Klasse `LastMinuteNegotiator`, die `Negotiator` implementiert.
2. Die Methode `getParameterClass` so implementieren, dass sie mitteilt, dass der neue `Negotiator` einen Plan als Grundlage für die Angebotsabgabe benötigt.
3. Die Methode `makeOffer()` so implementieren, dass sie einen ihr übergebenen Plan auf freie Kapazitäten untersucht, und diese dem Simulationskern mitteilt.
4. Einen entsprechenden Eintrag in der Datei `defbroker.cfg` hinzufügen.

Nach diesen wenigen Handgriffen ist das neue Auftragsverhandlungsmodul für den `Planner` verfügbar.

5 Plugin

Das Visualisierungs-Plugin haben wir gegenüber der Version im Grobentwurf des Pflichtenheftes stark erweitert. Zu den dort bereits erwähnten tabellarischen Darstellungen des aktuellen Shuttlestatus sowie der Auftrags- und Wegeplanung sind neue Funktionalitäten hinzugetreten. Die Navigation über Karteikartenreiter wurde beibehalten.

5.1 Beschreibung der neuen Plugin-Umgebung

Das Plugin besitzt in der neuen Version zwei Modi: Einen Daten-Modus (Data) und einen graphisch-visuellen Modus (Visual).

Der in Abbildung 5 dargestellte Daten-Modus stellt dabei eine verbesserte Version der bereits festgelegten Funktionen dar. Dabei wurde zunächst Wert darauf gelegt, dass die wichtigsten Shuttle-Daten wie Kontostand und aktuelle Beladung immer in einer übersichtlichen Form in der oberen Hälfte des Plugins zu sehen sind. Hierzu haben wir uns intuitiv verständlicher Statusbalken bedient, die nun zum Teil eine reine Zahlendarstellung ersetzt. Auch werden die aktuellen Aktionen des Shuttles mittels geeigneter, leicht verständlicher Symbole dargestellt, was ebenfalls einen leichtere Erfassung des aktuellen Geschehens durch die Benutzer unterstützt.

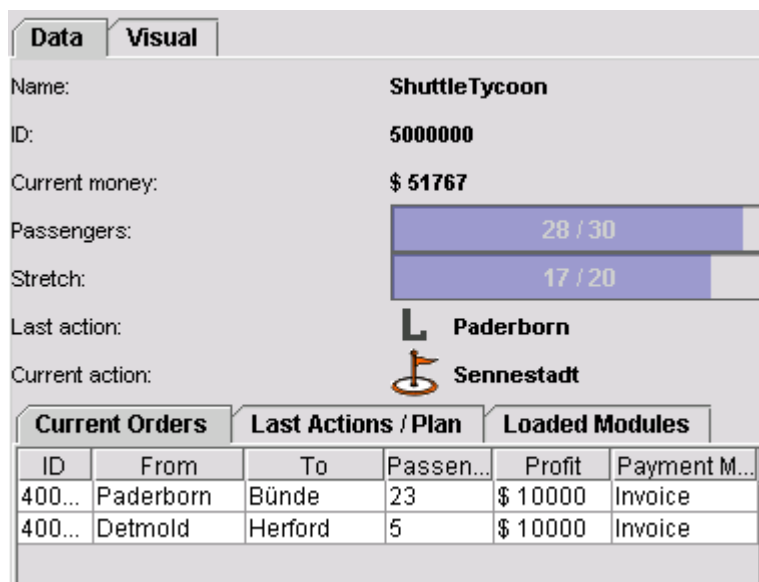


Abbildung 5: Daten-Modus – In der unteren Hälfte sind die aktuellen Aufträge zu sehen.

Darunter befinden sich die schon bekannten Karteikarten für die aktuellen Aufträge und die Streckenplanung sowie zusätzlich eine weitere Karteikarte, die die vom Shuttle zur Laufzeit geladenen Module anzeigt; so wird den Benutzern ein kleiner Einblick in die Shuttle-Strategie gegeben.

Die beiden aus dem Sollkonzept bekannten Panels haben sich geringfügig geändert. So werden nicht mehr alle Aufträge angezeigt, sondern nur noch die aktuell in Bearbeitung befindlichen. Dies ist darin begründet, dass diese Liste bei langen Simulationen enorm wächst und die dazu benötigten Daten (aufgrund der Architektur der umschließenden Visualisierung) fortlaufend gesendet werden müssen, so dass dies wiederum die RMI-Kommunikation unverhältnismäßig belasten würde. Selbiges gilt für die Darstellung der Streckenplanung: Hier beschränken wir uns nun auf die Anzeige der letzten 15 Aktionen sowie des vollständigen Plans für die Abarbeitung der Aufträge. Diese beiden zeitlich und logisch unterschiedlichen Datensätze befinden sich nun auch in verschiedenen Tabellen, jedoch auf der selben Tafel (siehe Abbildung 6).

Current Orders			Last Actions / Plan			Loaded Modules		
Last Actions:								
Action	Station	Details						
Repaired at	Bünde							
Loaded at	Bünde	Order 4000058 to Detmold						
Loaded at	Enger	Order 4000057 to Dringen...						
Repaired at	Enger							
Loaded at	Enger	Order 4000060 to Bad Salz...						
Unloaded at	Bad Salzuflen	Order 4000060 from Enger						
Loaded at	Lage	Order 4000061 to Dringen...						
Loaded at	Lage	Order 4000064 to Paderborn						
Plan:								
Action	Station	Details						
Go to	Detmold							
Unload at	Detmold	Order 4000058 from Bünde						
Repair at	Detmold							
Unload at	Dringenberg	Order 4000061 from Lage						
Unload at	Dringenberg	Order 4000057 from Enger						
Unload at	Paderborn	Order 4000064 from Lage						
Repair at	Paderborn							

Abbildung 6: Anzeige für Aktionen – In der oberen Tabelle werden die letzten 15 Aktionen angezeigt, in der unteren die als nächstes geplanten.

Kommen wir nun zur wesentlichen Neuerung gegenüber dem Grobentwurf – dem visuellen Modus. Wie in Abbildung 7 zu erkennen ist, unterscheidet sich dieser deutlich von der einfachen, tabellarischen Darstellung des Daten-Modus. Auch hier herrscht eine Zweiteilung, wobei die obere Hälfte wiederum dem Überblick dient, zusätzlich aber auch die Navigation für die untere Hälfte in sich birgt. Hier werden dann Details zu den ausgewählten Objekten dargestellt. Das Panel dieses Modus ist in sich konfigurabel. So lässt sich der mittlere Trennbalken beliebig verstellen. Dabei passen sich die angezeigten

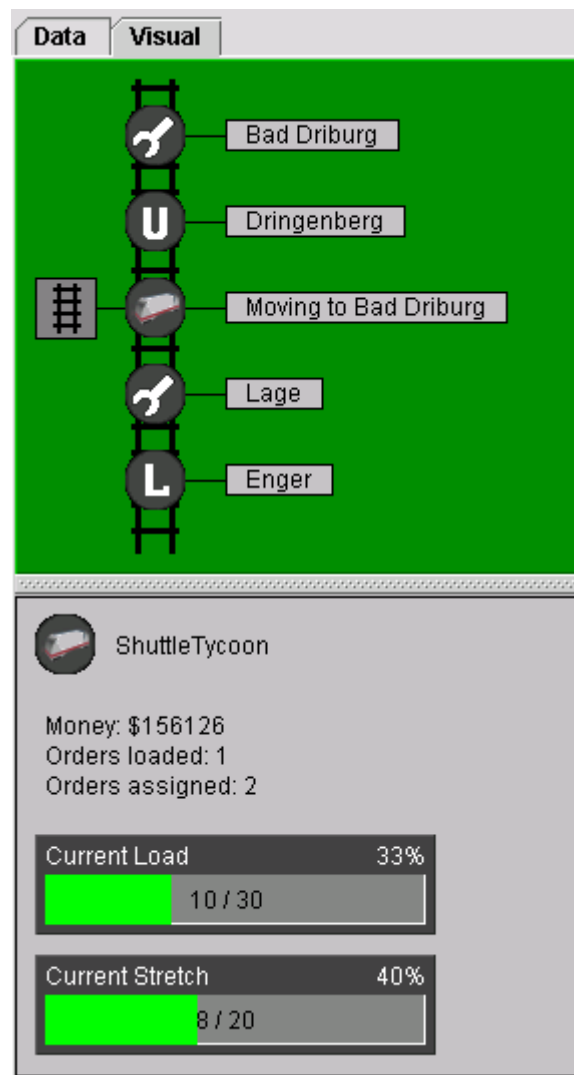


Abbildung 7: Visueller Modus – In der unteren Hälfte ist die Anzeige bei selektiertem Shuttle zu erkennen.

Daten der Benutzersicht an, so dass es den Benutzern z.B. möglich ist, eine weitaus längere Liste der Aktionen in der oberen Hälfte anzeigen zu lassen. Die Navigation gestaltet sich dabei höchst intuitiv durch anklicken der einzelnen Symbole, die mit den bereits im Daten-Modus verwendeten korrespondieren. Rechts neben den Aktionen wird dabei der Bahnhof angezeigt, in dem diese abgewickelt wird, sowie links davon eventuell zusätzlich auftretenden Ereignisse wie Strafen. Klickt man auf ein solches Symbol, etwa auf auf das Shuttle selbst, so werden alle verfügbaren Daten angezeigt. Ähnliche gestaltete Informationen sind auch für die einzelnen Aktionen abrufbar. Zu einem Load- bzw. Unload-Symbol erhält man so Details zum zugehörigen Auftrag, zu einem Repair-Symbol werden die Kosten und die Dauer der Wartung angezeigt.

5.2 Datentransfer zum Plugin

Um Daten vom Shuttle an das Plugin zu senden, wird die `ShuttleAgent-StatusMessage` benutzt. Da mithilfe dieser Nachricht lediglich Strings übermittelt werden können, sind die Daten, die an das Plugin gesendet werden, kodiert. Um den String so klein wie möglich zu halten, wird eine platzsparende Kodierung der Daten vorgenommen, die im Gegensatz zu den Standard-Serialisierungstechniken dem Problem angepasst ist und so eine höhere Komprimierung der Daten ermöglicht.

Die Klasse `PluginData`, die die Daten für das Plugin in `ShuttleDataForPlugin` verwaltet, versendet diese Daten in festgelegten Zeitabständen. Dazu ruft sie die `toString`-Methode von `ShuttleDataForPlugin` auf, um alle dort abgelegten Daten in einen String zu kodieren.

Die Kodierung wird wie folgt vorgenommen:

Der Java-Datentyp *Character* besteht aus 16 bit. Das ergibt 65536 verschiedene Zeichen. Davon werden einige als Trennsymbole reserviert, um die einzelnen logischen Abschnitte innerhalb der Daten abzubilden, wie z.B. die zugewiesenen Aufträge von den geplanten Aktionen, sowie innerhalb dieser Abschnitte weitere Aufteilungen wie z.B. zwischen den einzelnen Aufträgen bis runter zu den einzelnen Werten eines Auftrages. Auf Indentifikatoren der einzelnen Abschnitte kann dabei verzichtet werden, da die Reihenfolge und Anzahl der einzelnen Abschnitte fest ist und immer eingehalten wird.

In der untersten Aufteilungsstufe gibt es nur noch die Grundwerte der einzelnen abgebildeten Objekte. Diese Grundwerte sind entweder Zeichenketten oder ganze Zahlen.

Bei der Kodierung von Zahlen wird zusätzlich zu den 3 Zeichen, die als Trennsymbole reserviert sind, ein weiteres Zeichen als Minuszeichen reserviert. Die

ganzen Zahlen werden dann zur Basis 65531 dargestellt. Negative Zahlen erhalten dabei ein führendes Minuszeichen.

Bei der Kodierung von Zeichenketten werden die 26 Buchstaben des Alphabets als Großbuchstaben auf die Zahlen von 1 bis 26 und die Kleinbuchstaben auf 27 bis 52 abgebildet. Von 53 bis 99 wird eine dynamische Zuteilung vorgenommen, die sich nach dem Bedarf der aktuell zu kodierenden Daten richtet. Diese Kodierungstabelle wird ans Ende des Strings angehängt und bei der Dekodierung zuerst ausgelesen.

Durch dieses Vorgehen lassen sich jetzt zwei Zeichen in einem kodieren, indem man die ersten zwei Ziffern des Zeichencodes zur Kodierung des ersten Zeichen benutzt und die zwei folgenden Ziffern für die Kodierung des anderen. Die 0 steht dabei für ein nicht vorhandenes Zeichen. Da die Anzahl der verwendbaren Zeichen in einem Datensatz durch dieses Vorgehen offensichtlich eingeschränkt wird, kann es vorkommen, dass eine Kodierung nicht mehr möglich ist. Dazu wird vor jeder Zeichenkette ein Zeichen gehängt, das anzeigt, ob die folgende Zeichenkette kodiert oder unkodiert ist.

Dieser als String kodierte Datensatz wird dann als `ShuttleAgentStatusMessage` über das `ShuttleTycoon` an den Kernel gesendet. Dieser leitet die Nachricht als `RemoteShuttleStatusString` an `Visualisation` weiter, die sie ihrerseits an das `ShuttleTycoonPlugin` weiterreicht.

Das `ShuttleTycoonPlugin` erzeugt dann ein neues `ShuttleDataForPlugin`-Objekt mit dem Konstruktor, der als einzigen Parameter den kodierten String benötigt. Der String wird im Konstruktor wieder dekodiert und das neue Objekt mit den daraus gewonnenen Daten initialisiert. So wird eine 1:1 Kopie des Objektes erzeugt, von dem der String ursprünglich erzeugt wurde (siehe Abbildung 8).

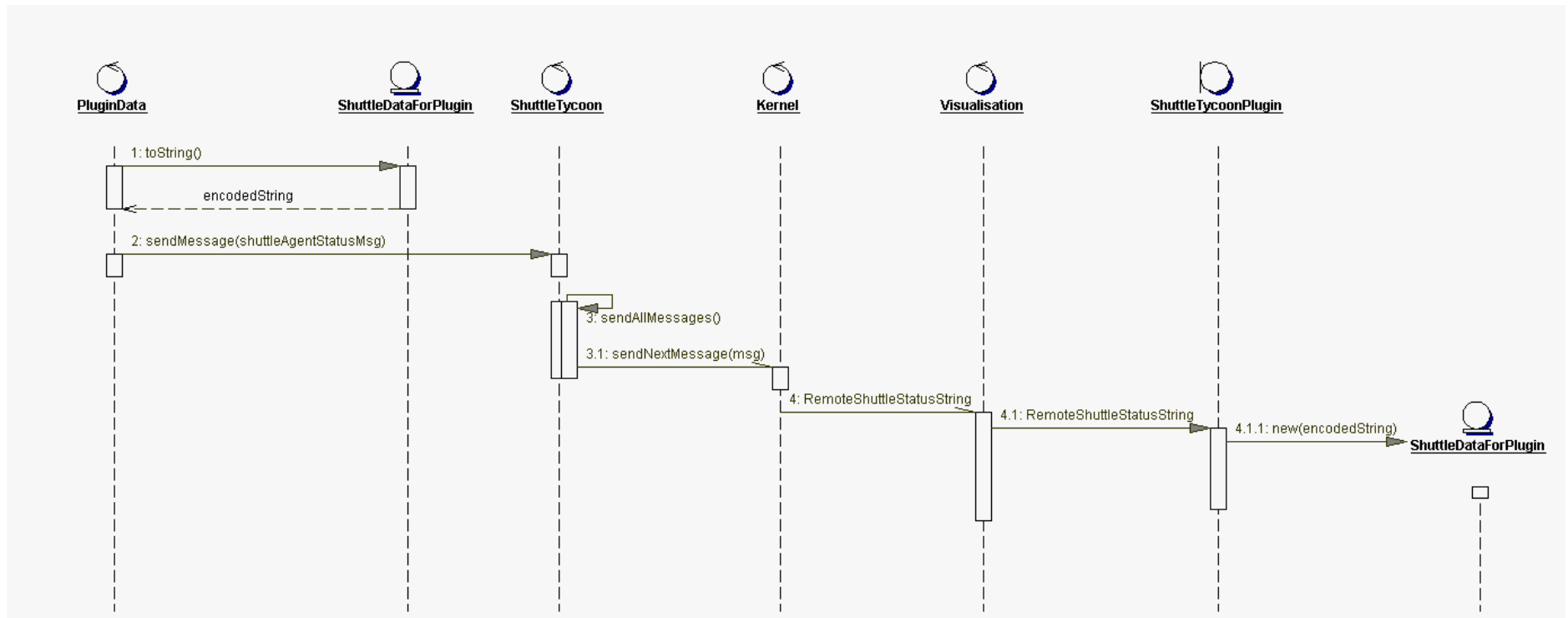


Abbildung 8: Kommunikationsfluss zwischen der Shuttlesteuerung und dem Plugin

6 Rekorder

Damit eine Simulation auch später noch einmal visualisiert werden kann, wurde der Rekorder entwickelt. Er kann zwischen den Simulationskern und die Visualisierung bzw. Analyse geschaltet werden und eine laufende Simulation aufzeichnen und zu einem späteren Zeitpunkt erneut wiedergeben. Der Rekorder setzt sich aus sieben wesentlichen Klasse zusammen, wie in Abbildung 9 zu erkennen ist.

6.1 Aufbau und Funktionsweise

Die Klasse `Main` ist für das Starten des Rekorders zuständig. Sie enthält die `main()`-Methode und initialisiert den Rekorder nach dem Starten. Die Verbindung zum Simulationskern wird über den `RMIClient` hergestellt. `RMIClient` spezialisiert dazu `VisualisationClientBase`. Über die `init()`-Methode wird die Topologie und über die `update()`-Methode werden die Nachrichten (`Message`, `RemoteObj`) empfangen. Die empfangenen Daten reicht der `RMIClient` an den `Converter` weiter.

Die Klasse `Converter` ist für jegliches Konvertieren von Daten zuständig. Sie wandelt die empfangenen Nachrichten so um, dass sie auf der Festplatte gespeichert werden können, bzw. vertauscht die gespeicherten Daten wieder in Nachrichten, die dann an zu dem Rekorder verbundene Klienten geschickt werden können. Außerdem misst der `Converter` die Zeitabstände zwischen eingehenden Nachrichten. So kann man später die Simulation 1:1 wieder abspielen.

Die einkommenden Daten wandelt der Rekorder um und reicht sie dann an den `FileWriter` weiter. Wenn sich der Rekorder im Online-Modus befindet, werden die Daten gleichzeitig an den `RMIServer` weitergereicht, welcher sie an die Clients schickt. Der `FileWriter` ist nur für das Speichern der Daten zuständig. Die Vorgehensweise dabei wird weiter unten genauer erklärt.

Bis jetzt wurden die Klassen zum Empfangen von Daten vorgestellt. Als nächstes werden die Klassen zum Senden von Daten vorgestellt.

Zum Senden der Daten werden die Daten zunächst durch den `FileReader` ausgelesen. Der `FileReader` ist, wie der Name schon sagt, das Pendant zum `FileWriter`. Die ausgelesenen Daten werden, wie oben schon erwähnt, an den `Converter` gereicht, welcher sie in Nachrichten umwandelt. Dieser wiederum verpackt die fertige Nachricht zusammen mit dem beim Empfang gemessenen Zeitabstand zur vorherigen Nachricht in ein Wrapperobjekt (`ObjectTimeContainer`) und reicht dieses Objekt zum `RMIServer` weiter.

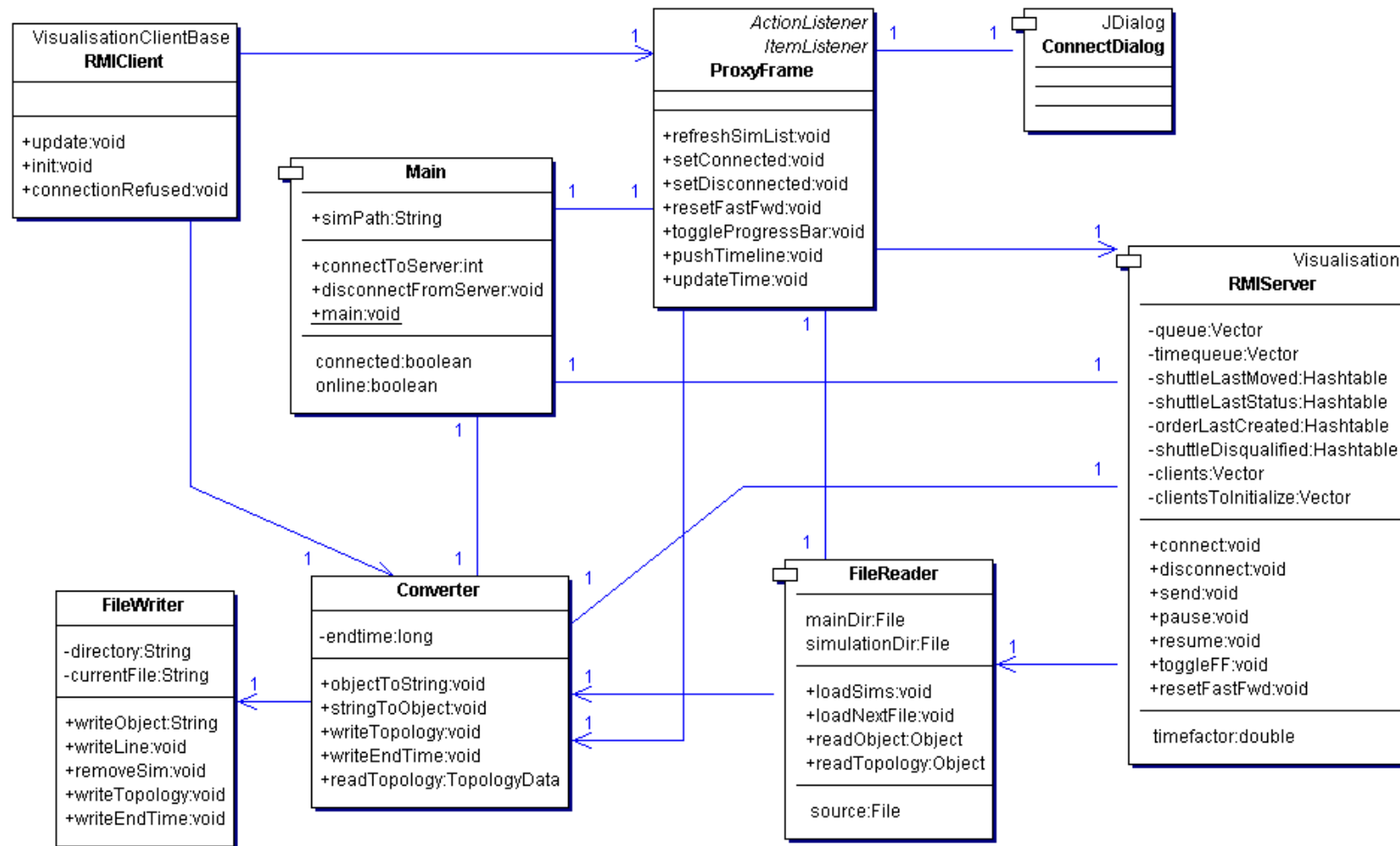


Abbildung 9: Klassendiagramm des Rekorders

Der `RMIServer` stellt die Schnittstelle für die Klienten dar. Er verwaltet die verbundenen Klienten und kümmert sich um das korrekte Absenden der Nachrichten. Die `ObjectTimeContainer`-Objekte, die der `RMIServer` vom `Converter` bekommt, packt der `RMIServer` wieder aus und stellt die Nachricht in die `queue`, den Zeitabstand in die `timeQueue`.

Zum Versenden der Nachrichten holt sich der `RMIServer` einen Zeitabstand aus der `timeQueue` und wartet den Zeitabstand ab. Danach holt er sich die entsprechende Nachricht (die Reihenfolge ist in beiden Warteschlangen gleich) aus der `queue` und sendet sie dann an alle verbundenen Klienten. Wenn sich der Rekorder im schnellen Vorlauf befindet, so halbiert er hier einfach den Zeitabstand. So ist der schnelle Vorlauf für beide Modi (Online/Offline) möglich.

Der `RMIServer` ermöglicht das Verbinden mehrerer Clients zu jedem beliebigen Zeitpunkt. Der reibungslose Ablauf wird durch verschiedene Mechanismen gewährleistet. Es gibt mehrere Bedingungen, die der `RMIServer` beachten muss, wenn sich ein Klient verbindet.

Es besteht die Möglichkeit, dass sich ein Klient verbindet, bevor der Rekorder selbst Daten empfangen hat. Dann können der Klient noch nicht mit der Topologie initialisiert werden. Deshalb gibt es zwei verschiedene Listen, in die verbindende Klienten eingereiht werden. Wenn der Rekorder noch keine Topologie empfangen hat, kommen alle verbindenden Klienten in die Liste `clientsToInitialize`. Die zweite Liste `clients` enthält Klienten, die bereits initialisiert wurden. Der `RMIServer` schickt Nachrichten nur zu Klienten in dieser Liste. Sobald die Topologie im Rekorder vorliegt, werden die Klienten in `clientsToInitialize` initialisiert, indem ihnen die Topologie zugesendet wird. Gleichzeitig werden die Klienten aus `clientsToInitialize` gelöscht und in `clients` aufgenommen.

Was passiert, wenn die Simulation bereits abgespielt wird und sich ein Klient verbinden möchte? Diesem Klient fehlen in diesem Fall die bereits gesendeten Nachrichten. Der Rekorder löst dieses Problem dadurch, dass er genau wie der Simulationskern beim Versenden von Nachrichten prüft, welche Art von Nachricht er verschickt. Wichtige Nachrichten speichert er zwischen, so dass er immer einen relativ aktuellen Stand hat. Dazu dienen die vier `Hashtables` `shuttleLastMoved`, `shuttleLastStatus`, `orderLastCreate` und `shuttleDisqualified`. Wenn sich jetzt ein Klient verbindet, so wird ihm zuerst dieser zwischengespeicherte Zustand, welcher übrigens auch die Topologie und die Simulationskonstanten `GameConstants` enthält, übermittelt. Danach kann der neu verbundene Klient wie alle anderen auch behandelt werden. Er wird also in `clients` aufgenommen.

Zum Bedienen des Rekorders gibt es die Benutzungsschnittstelle `ProxyFrame`. Über diese GUI kann der Benutzer alle gewünschten Aktionen ausführen.

6.2 Datenformat

Die Simulationen werden in einem Simulationsverzeichnis gespeichert. Beim Starten des Rekorders wird das Simulationsverzeichnis als Parameter erwartet. Für jede Simulation enthält das Simulationsverzeichnis wiederum ein Unterverzeichnis, welches die Simulationsdaten einer speziellen Simulation enthält. Der Name des Unterverzeichnisses muss ein vorgegebenes Schema einhalten, damit der Rekorder das Verzeichnis als Simulation erkennt. Der Name muss folgende Struktur haben:

`<Tag>p<Monat>p<Jahr>-<Stunde>c<Minute>c<Sekunde>`

Man sieht, dass sich der Name aus dem Startzeitpunkt der Simulation und einigen Trennzeichen zusammensetzt. Das Verzeichnis enthält folgende Dateien:

- `order`,
- `endtime`,
- `topology`,
- `obj_*`,
- Dateien, die das gleiche Namensformat wie das Verzeichnis haben.

Die Datei `endtime` enthält die komplette Simulationsdauer. Diese wird benötigt, um im Offline-Modus den Fortschrittsbalken anzeigen zu können.

Die Datei `topology` enthält die Topologie, die wir ganz am Anfang der Simulation vom Simulationskern erhalten und die wir zum Initialisieren an die Klienten schicken. Die Dateien `obj_*` sind die eigentlichen Nachrichten. Die einkommenden Nachrichten werden einfach als Objekte abgespeichert. Das ist möglich, da sie alle das Interface `Serializable` implementieren.

Die Dateien mit einem Zeitpunkt als Namen enthalten 1000 Zeilen. Eine Zeile enthält folgende Informationen:

- Art der Nachricht,
- Simulationszeit,
- Zeitabstand zur vorherigen Nachricht,
- Dateiname der zugehörigen Nachricht (`obj_*`-Datei).

Durch diese Dateien wird also die Reihenfolge der Nachrichten festgelegt, und es werden weitere Informationen gespeichert. Die Größe von 1000 Zeilen ist so gewählt, um die Ladezeiten der Dateien möglichst kurz zu halten und Systemgrenzen (z.B. 2GB pro Datei o.ä.) zu umgehen. Der Rekorder kann so Monate lange Simulationen aufzeichnen und wird nur durch die Hardware eingeschränkt. Die Reihenfolge dieser Dateien wird durch den Inhalt der Datei `order` festgelegt.

6.3 Zustände des Rekorders

Das in Abbildung 10 dargestellte Zustandsdiagramm beschreibt die verschiedenen Zustände, in die der Rekorder über die GUI versetzt werden kann. Der Rekorder lässt sich über `connect` mit einem Simulationskern verbinden und mit `disconnect` wieder trennen und zurücksetzen. Je nach Wahl der Quelle befindet sich der Rekorder im Online-Modus, in dem er von einem Simulationskern empfangene Daten weiterleitet, oder im Offline-Modus, um aufgezeichnete Daten abzuspielen. Die Wiedergabe kann über `play` erfolgen und danach pausiert, wiederaufgenommen und gestoppt werden. Zudem ist es möglich die Wiedergabe im schnellen Vorlauf erfolgen zu lassen. Schließlich lässt sich der Rekorder natürlich jederzeit beenden.

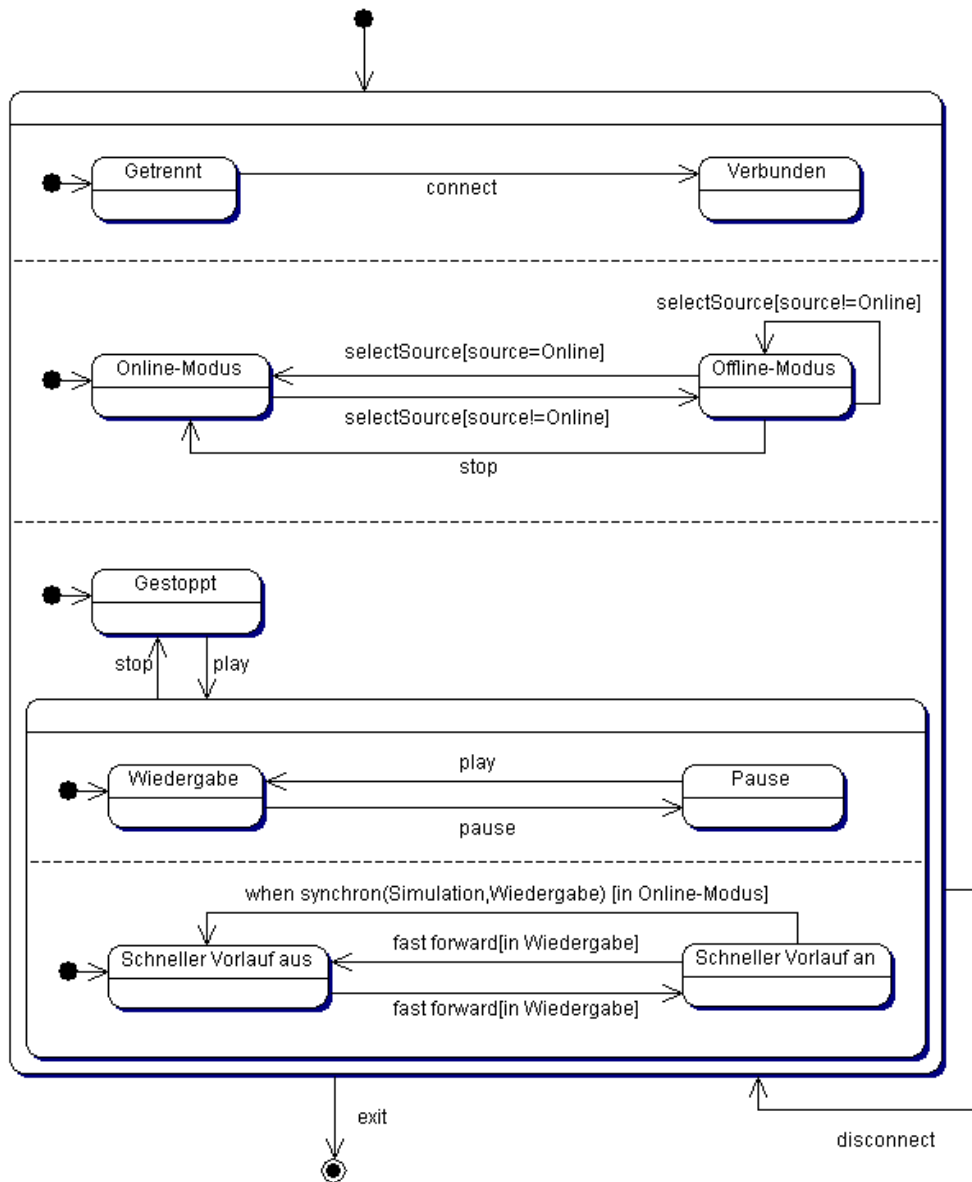


Abbildung 10: Zustände des Rekorders

7 Analysemodul

Das Analysemodul verfügt über alle geforderten Funktionen und wird im Folgenden näher erläutert. Dazu werden wir zunächst auf die verwendete Datenstruktur und anschließend auf die Gesamtstruktur eingehen.

7.1 Datenstruktur

Um seinen Aufgaben gerecht werden zu können, verfügt das Analysemodul über eine spezialisierte Datenstruktur, die in Abbildung ?? zu erkennen ist und im folgenden erläutert wird.

Die Klasse `DataVault` dient als umfassendes Archiv für alle zu erfassenden Daten. Das geschieht in zweifacher Form. Zum einen werden zu jedem Shuttle alle verfügbaren Informationen gespeichert, zum anderen wird das Streckennetz erfaßt, analysiert und auch hier werden alle relevanten Daten gespeichert.

`DataVault` stellt ebenfalls eine Reihe von Methoden zur Verfügung, um aus den wenigen direkt vom Simulationskern verfügbaren Informationen weitere sinnvolle Datensätze zu konstruieren (z.B. Durchschnittliche Auslastung, durchschnittlicher Gewinn, Kosten, Gesamteinnahmen, etc.) oder aus bereits vorhandenen Datensätzen vorgefertigte Objekte vom Typ `Vector` zu schaffen, die dann schnell von der Klasse `Converter` in Diagramme konvertiert werden können.

7.2 Daten der Shuttles

Die Hashtable `allShuttles` enthält für jedes Shuttle einen Eintrag mit der ShuttleID als Schlüssel und einem Objekt vom Typ `ShuttleData`, als Datenspeicher. Sobald ein Shuttle als solches erkannt wird, startet eine Initialisierungsmethode, um die Speicherung von weiteren Daten zu ermöglichen.

Durch die Nutzung einer dynamischen Datenstruktur kann das Analysemodul auf jedes neu hinzukommende Shuttle einzeln reagieren und benötigt keine eigene Initialisierung.

Die Klasse `ShuttleData` besteht aus der Hashtable `myShuttleData`, die mehrere Vektoren enthält. Diese Vektoren halten `StreamObject`-Instanzen, die aus einem Zeit-Eintrag und einem Wert-Eintrag bestehen (beides Werte vom Typ `int`). Neue Informationen werden an den existierenden Vektor angehängt, so dass ein vollständiger Verlauf des Wertes vom Beginn der Aufzeichnung der Analyse besteht.

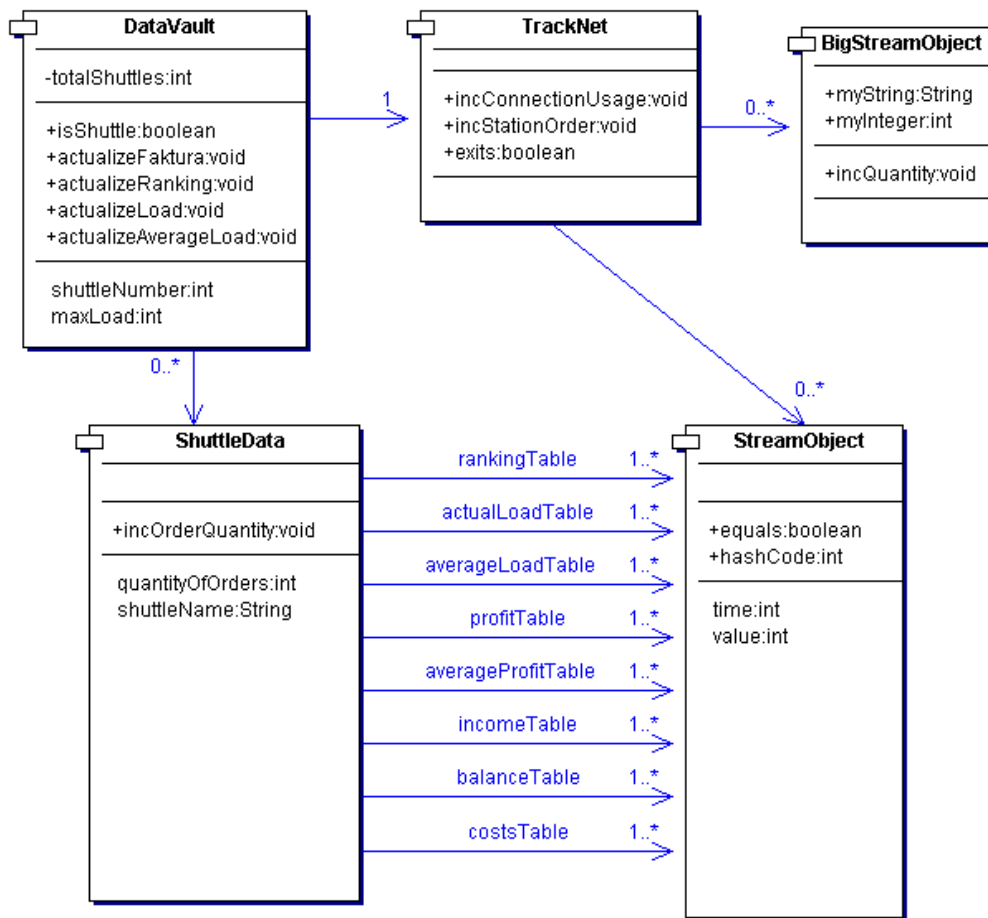


Abbildung 11: Datenstruktur der Analyse

Vektoren existieren zu folgenden Informationen:

- Ranking
- Overall Costs
- Average Profit
- Overall Profit
- Overall Gain
- Average Load
- Actual Load
- Account Balance

7.3 Das Streckennetz

Das Streckennetz wird in `DataVault` in der Variablen `myNet` als Objekt vom Typ `TrackNet` gespeichert. Alle Stationen werden ausgelesen und separat unter ihrer ID mit Name und Anzahl der bisherigen Aufträge, in der die jeweilige Station als Startbahnhof des Auftrages fungiert, gespeichert. Zudem werden alle Verbindungen berechnet und mit ihrer bisherigen Streckenauslastung festgehalten.

`TrackNet` nutzt dazu `BigStreamObject`-Objekte, die es ermöglichen, einen String und einen `int`-Wert zu speichern. Für jeden Bahnhof existiert eine separate `BigStreamObject`-Instanz.

7.4 Gesamtstruktur

Die Analyse besteht aus 13 Klassen plus der Displayengine der Visualisierung, die weitestgehend unverändert übernommen wurde. Die folgenden Abschnitte beschreiben die sieben wichtigsten Klassen. Die übrigen Klassen sind Hilfsklassen, auf die wir hier nicht näher eingehen wollen (siehe Abbildung 12).

7.4.1 Analyse_Gui

Die zentrale Benutzerschnittstelle ist in der Klasse `Analyse_Gui` definiert und wird über die Klasse `Main` aufgerufen. `Analyse_Gui` erstellt von allen benötigten Klassen Instanzen. Sie bietet alle notwendigen Methoden, um die sichtbaren Attribute zu verändern, und gestattet dem Benutzer Zugriff auf die gesammelten Daten sowie die Auswahl der Analysefunktionen.

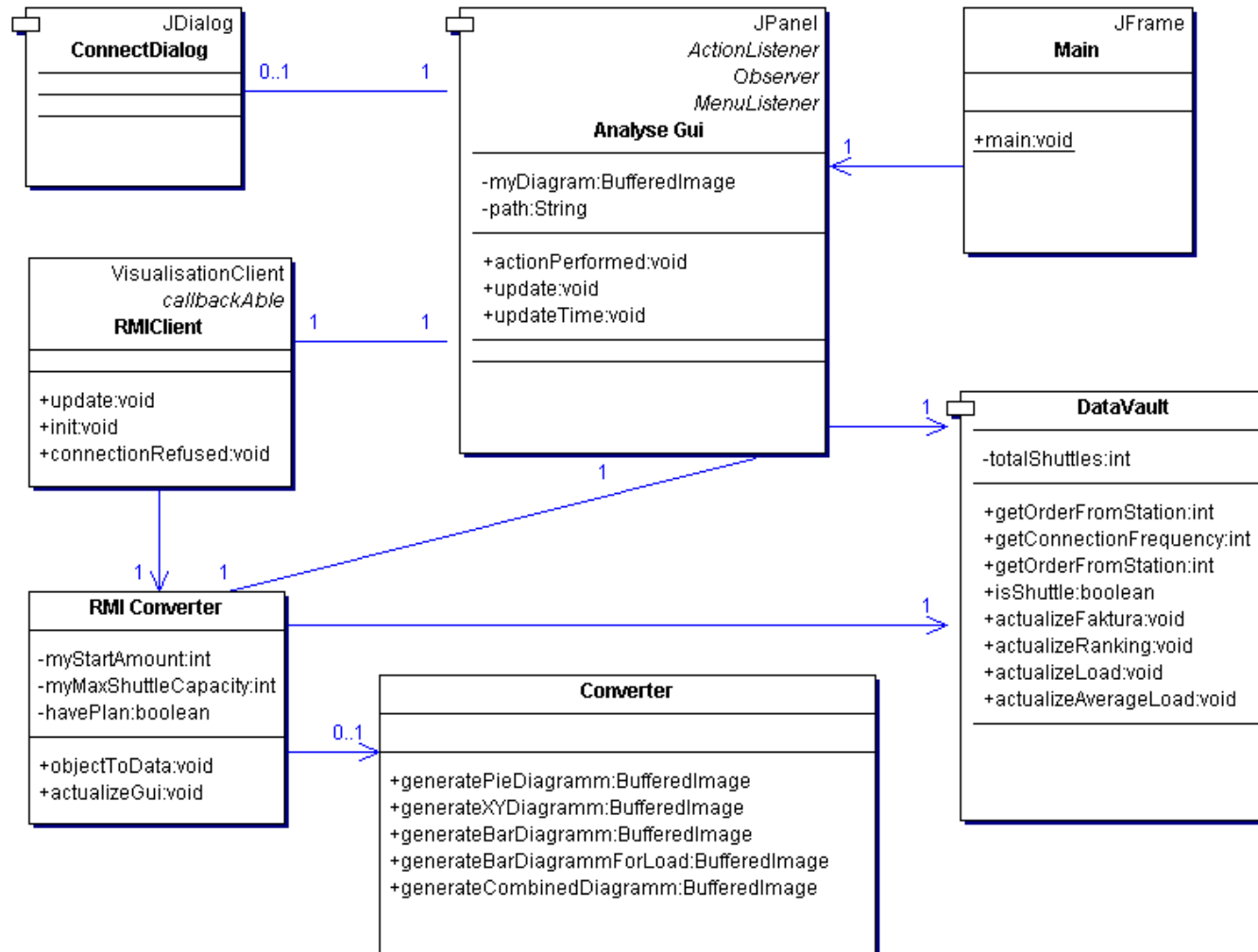


Abbildung 12: Gesamtstruktur der Analyse

7.4.2 RMIClient

Der `RMIClient` der Analyse ist der gleiche, der auch in der Visualisierung oder dem Rekorder (siehe Kapitel 6) verwendet wird. Er verbindet sich entweder direkt zur Simulation oder aber über den Rekorder.

7.4.3 ConnectDialog

Die Klasse `ConnectDialog` gibt dem Nutzer die Möglichkeit, die Quelle der zu analysierenden Daten auszuwählen bzw. den Server zu bestimmen, mit dem sich `RMIClient` verbindet.

7.4.4 RMLConverter

Die Klasse `RMLConverter` ist die Schnittstelle zwischen ankommenden Daten und der Darstellung durch die Analyse. Sie speichert alle neu hinzugekommenen Daten in der Datenstruktur der Analyse und aktualisiert die Werte, die im Analyse-Fenster angezeigt werden.

Dazu wird jedes vom `RMIClient` erfaßte `RemoteObject` an den `RMLConverter` weitergeleitet, der dann die Unterklasse von `RemotObject` ausliest und die darin enthaltenen Informationen weiterverarbeitet. Unter Zuhilfenahme des Converter erzeugt sie Statistik-Objekte vom Typ `BufferedImage`, die von der Analyse visualisiert werden sollen, und leitet sie an die `Analyse_Gui` weiter.

7.4.5 Converter

Der `Converter` ist die Schnittstelle zwischen dem Analysemodul und dem `JFreeChart`-Paket, das zur Berechnung der Graphiken genutzt wird. Er stellt Methoden zur Verfügung, mit denen alle in Kapitel 6 des Handbuchs erläuterten Diagramme als `BufferedImage` erstellt werden können.

7.4.6 DataVault

Die Klasse `DataVault` wurde bereits in Abschnitt 7.1 näher erläutert und ist hier stellvertretend für die komplexe Datenstruktur aufgeführt.