Serviceorientiertes Framework für ein Satellitenmissionssimulationssystem

Diplomarbeit

Institut für Luft- und Raumfahrttechnik
Fakultät V – Verkehrs- und Maschinensysteme
Technische Universität Berlin

Marco Berlin
Matrikelnummer: 204888

Berlin, 9. August 2008

Eidesstattliche Erklärung

„Ich versichere hiermit an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.“

Berlin, 9. August 2008

Unterschrift Marco Berlin

Aufgabenstellung

Serviceorientiertes Framework für ein Satellitenmissionssimulationssystem

Im Rahmen von mehreren Studien- und Diplomarbeiten wird am Institut für Luft- und Raumfahrttechnik der Technischen Universität Berlin das Satellite Mission Analysis and Simulation System (SMASS) entwickelt. SMASS ist ein System zur Simulation von Satelliten innerhalb einer Mission. Zurzeit enthält SMASS einen Server und vier Berechnungsmodule. Diese decken die Orbit-, die Thermalhaushalt-, die Energiehaushalt- und die Magnetfeldberechnung für die Simulation eines Satelliten ab. Nach der Analyse des aktuellen Systems im Rahmen eines Praktikums wurde ein Optimierungsbedarf hinsichtlich der Modularisierung von SMASS erkannt.

Im Rahmen dieser Diplomarbeit soll nun eine Basis für die weitere Entwicklung von SMASS geschaffen werden. Dazu muss ein Konzept für den Kern von SMASS erstellt werden, welches die folgenden Aufgaben übernimmt: Koordination der Kommunikation zwischen den Berechnungsmodulen sowie die Datenhaltung von Simulationsergebnissen und Konfigurationsdaten. Durch geeignete Schnittstellen ist das System so zu beschreiben, dass eine einfache Einbindung von bereits bestehenden und noch zu entwickelnden Berechnungsmodulen ermöglicht wird. Zur Umsetzung dieses Konzeptes soll auf das Prinzip der Objektorientierung zurückgegriffen werden.

Die Diplomarbeit wird im Rahmen der Zusammenarbeit der TU Berlin mit der Einrichtung SISTEC des DLRs im Rahmen des Projektes Virtueller Satellit durchgeführt. Die Aufgabenstellung lässt sich in vier Teilaufgaben unterteilen:

1. Modellierung und Implementierung des neuen Frameworks von SMASS
2. Erstellung von Schnittstellenkomponenten einschließlich ihrer Spezifikation und jeweils einem Programmierbeispiel
3. Erstellung/Anpassung von mindestens zwei Modulen
4. Demonstration des neuen Kerns mit Hilfe der Module

Nicht zur Aufgabenstellung gehört die Umsetzung und Ausarbeitung einer Datenbank für die Simulations- und Konfigurationsdaten, wohl aber die Bereitstellung einer Schnittstelle zur Datenverarbeitung. Ebenfalls nicht zur Aufgabenstellung gehört die Umsetzung der grafischen Oberflächen und Module, für welche aber eine Schnittstelle zur Verfügung gestellt wird.

Betreuer DLR
Axel Berres
Rutherfordstrasse 2
12489 Berlin
E-Mail: Axel.Berres@dlr.de

Betreuer TU Berlin
Dr.-Ing. Hakan Kayal
Marchstrasse 12
10587 Berlin
E-Mail: Hakan.Kayal@TU-Berlin.de

Inhaltsverzeichnis

Abbildungsverzeichnis iv

Tabellenverzeichnis vi

Glossar viii

Abkürzungsverzeichnis xii

1. Einleitung 1

2. Problemstellung 2
2.1. Aufgabe von SMASS . . . 2
2.2. Ausgangszustand von SMASS . . . 4
2.3. Ziel dieser Arbeit . . . 7

3. Aufbau und Funktion 9
3.1. „Simulation“ . . . 17
3.1.1. Struktur einer Simulation . . . 17
3.1.2. Verhalten einer Simulation . . . 19
3.1.3. Laufzeitverhalten - „schwache Echtzeit“ . . . 24
3.1.4. Datenaustausch zwischen Services . . . 25
3.2. „Utilities“ . . . 27
3.2.1. Netzwerkverbindung . . . 27
3.2.2. Nachrichtenformat . . . 28
3.2.3. Verarbeitung der Nachrichten . . . 31
3.2.4. Fazit . . . 33
3.3. „Services“ . . . 35
3.3.1. Serviceschnittstellen . . . 35
3.3.2. Konkrete Serviceklassen am Beispiel von ORB001 . . . 36
3.3.3. Bibliothek für die Services . . . 38
3.3.4. Fazit . . . 38
3.4. „Data Handling“ . . . 40
3.4.1. Laden/Speichern der Konfigurationsdaten . . . 40
3.4.2. XML-Konfigurationsdatei . . . 42
3.4.3. Laden/Speichern der Simulationsergebnisse . . . 48
3.4.4. Speicherung der Simulationsergebnisse in Textdateien . . . 49
3.4.5. Fazit . . . 51

3.5. „Frontend“ . . . 52
3.5.1. Struktur und Verhalten . . . 52
3.5.2. Fazit . . . 58

4. Bedienung **59**
4.1. Installation . . . 59
4.2. Konfigurierung . . . 61
4.3. Durchführen einer Simulation . . . 61
4.4. Quelltext ändern . . . 65

5. Einbinden eines neuen Services **66**
5.1. Schritt 1 - Kopieren der Vorlage und Umbenennen des Ordners . . . 66
5.2. Schritt 2 - Umbenennen der Klassen zur Schnittstellenimplementierung . . . 67
5.3. Schritt 3 - Bearbeiten des Eintrittspunkts . . . 67
5.4. Schritt 4 - Implementieren der Servicefunktionalität . . . 68
5.5. Schritt 5 - Umbenennen und Bearbeiten der *Compile**-Dateien . . . 69
5.6. Schritt 6 - Umbenennen und Bearbeiten der *Start**-Dateien . . . 70
5.7. Schritt 7 - Eintragen des Services in die Konfigurationsdatei . . . 70

6. Tests **73**
6.1. Funktionstest mit ORB001 . . . 73
6.2. Test des Datenaustauschs zwischen ORB001 und PWR000 . . . 75
6.3. Test der „schwachen Echtzeit“ . . . 76
6.4. Mehrere parallele Simulationen . . . 80
6.5. Betriebssystemunabhängigkeit und verteiltes Simulieren . . . 84

7. Zusammenfassung und Ergebnisse **85**

8. Ausblick **89**

Literaturverzeichnis **91**

A. Verwendete Software **I**
A.1. Enterprise Architect . . . I
A.2. Eclipse . . . I

B. Vorgehensweise **III**
B.1. Modellierung . . . III
B.1.1. Verwendete Modellierungssprache . . . III
B.1.2. Verwendetes Modellierungswerkzeug . . . III
B.1.3. Verwendete UML-Elemente . . . IV
B.2. Umsetzung . . . VI

C. Patterns **VII**
C.1. Observer-Pattern . . . VII

C.1.1. Struktur des Observer-Patterns VII
C.1.2. Das Observer-Pattern in Java VIII
C.2. MVC-Pattern VIII

D. Inhalt der CD **X**

Abbildungsverzeichnis

2.1. Anwendung von SMASS 3
2.2. Datenaustausch bei SMASS 5

3.1. Schichten des SMASS-Frameworks 12
3.2. Hierarchie des Modells von SMASS 17
3.3. Interfaces der „Simulation“-Schicht 18
3.4. Ablauf einer Simulation (business logic) 19
3.5. Konfigurierung der Satellitensimulationen 21
3.6. Aktivitäten innerhalb von *run simulation* 22
3.7. Berechnung eines Simulationsschrittes 23
3.8. „schwache Echtzeit“ 24
3.9. Datenaustausch bei einseitiger Abhängigkeit 26
3.10. Datenaustausch bei gegenseitiger Abhängigkeit 26
3.11. Struktur des *communication*-Pakets 27
3.12. Nachrichtenklassen 28
3.13. *ServerMessageHandler* und *ServiceMessageHandler* 31
3.14. Zustände des Servers 32
3.15. Zustände eines Clients 33
3.16. Schnittstellen für die Services 35
3.17. Struktur eines Services 36
3.18. Realisierung der Schnittstellen 37
3.19. Struktur des *configuration* Pakets 40
3.20. Klassen des *xmlconfiguration*-Pakets 41
3.21. Struktur des *storage*-Pakets 48
3.22. Ordner- und Dateistruktur der Ergebnisse 50
3.23. Struktur der „Frontend“-Schicht 52
3.24. Umsetzung des Observer-Patterns 53
3.25. Klassen für die Ansichten der Konsolenoberfläche 55
3.26. Klassen für die Manipulation des Modells 56
3.27. MVC-Pattern am Beispiel des Triples aus *SMASS* 57

4.1. Systemvariable in Windows XP setzen 60
4.2. Hauptmenü nach dem Start von SMASS 62
4.3. Simulationsmenü 62
4.4. Satellitensimulationsmenü 63
4.5. Hauptmenü nachdem eine Simulation instanziiert wurde 64
4.6. Menü zur Editierung von Parameterwerten 64

5.1. Ordnerstruktur in *NewJavaService* 67

6.1. Relativer Fehler zwischen Daten des originalen und des angepassten Services 74
6.2. Datenaustausch zwischen ORB001 und PWR000 76
6.3. Test der „schwachen Echtzeit“ - Satellit 1 78
6.4. Test der „schwachen Echtzeit“ - Satellit 2 79
6.5. relativer Fehler der Schrittweite in Simulation 1 82
6.6. Vergleich zweier Solarzellenkonzepte 83

B.1. UML Klassen und Interfaces IV
B.2. UML Pakete V
B.3. UML Zustandsdiagramm V
B.4. UML Aktivitätsdiagramm VI

C.1. Struktur des Beobachtermusters [Vog02] VIII
C.2. Struktur des MVC VIII

Tabellenverzeichnis

2.1. Anforderungen an SMASS - Teil 1 4
2.2. Anforderungen an SMASS - Teil 2 6
2.3. Anforderungen an SMASS - Teil 3 8

3.1. Notation von Diagrammelementen 10
3.2. Aufgaben und Schichten von SMASS 11
3.3. Zustände der Modellklassen 19
3.4. Struktur *SMASSFormatMessage* 29
3.5. benutzte Token 29
3.6. Nachrichtennamen 30
3.7. Datenteil einer Nachricht 30
3.8. Methoden des Interfaces *IService* 35
3.9. Dateien der Bibliothek für die Services 38
3.10. Eigenschaften des Wurzelelementes 43
3.11. Eigenschaften des *simulation*-Elementes 43
3.12. Eigenschaften der *parameter*-Elemente einer Simulation 44
3.13. Eigenschaften des *satelliteSimulation*-Elementes 45
3.14. Eigenschaften des *parameter*-Elementes einer Satellitensimulation 45
3.15. Eigenschaften des *service*-Elementes 46
3.16. Eigenschaften der *parameter*-Elemente eines Services 46
3.17. Eigenschaften der *parameter*-Elemente für die Ergebnisse eines Services . 47
3.18. Eigenschaften der *parameter*-Elemente für die von anderen Services benötigten Werte eines Services 47
3.19. Speicherungsstruktur der Ergebnisse beeinflussende Konfigurationsdaten 50
3.20. Speicherungsstruktur der Ergebnisse beeinflussende Daten 50
3.21. Ansichten der Konsolenoberfläche 54

4.1. Kompilierungsdateien für SMASS 65

5.1. Schritte zur Erstellung eines neuen Services 66
5.2. Inhalt der Ordnerstruktur in *NewJavaService* 67

6.1. Eingangsdaten für den Funktionstest mit ORB001 73
6.2. Eingangsdaten für den Test zum Datenaustausch 75
6.3. Eingangsdaten für den Test zur „schwachen Echtzeit“ 77
6.4. Eingangsdaten für Simulation 1 80
6.5. Eingangsdaten für Simulation 2 81

6.6. Tests zur Betriebssystemunabhängigkeit und verteilten Simulation 84

Glossar

A

abstrakt Als abstrakt werden Methoden bezeichnet, welche nicht vollständig sind und erst von einer abgeleiteten Klasse implementiert werden. Klassen, welche eine abstrakte Methode besitzen, werden ebenfalls abstrakt genannt. Solche Klassen und Methoden werden in Java mit dem Schlüsselwort *abstract* gekennzeichnet [Mül04b].

Attribut Ein Attribut ist eine Eigenschaft eines Objektes. Im Bereich der UML ist ein Attribut ein strukturelles Merkmal einer Klasse und damit auch von Objekten, die durch Instanziieren von dieser Klasse gebildet werden. Im Zusammenhang mit der Auszeichnungssprache XML ist ein Attribut ein Merkmal, welches die Tags genauer beschreibt.

C

Client Der Client ist grundsätzlich ein Programm, welches über ein Netzwerk den Dienst eines Servers anfordert [RSSW03]. Im Rahmen dieser Arbeit ist die Aufgabenverteilung zwischen Server und Client allerdings differenzierter zu sehen. Der Client ist ein Programm, welches eine Verbindung zu einem Server aufnimmt und Nachrichten mit dem Server austauscht [Ber07b]. Allerdings ist er im Rahmen von SMASS nur passiver Natur und reagiert lediglich auf eingehende Nachrichten.

Compiler Ist ein „Programm, das die Befehle einer Programmiersprache in eine für den Computer lesbare Maschinensprache überträgt“ [JH06].

F

Framework Bezeichnet in der Informatik eine Struktur eines Programms und wird häufig im Zusammenhang mit Objektorientierung und Klassen verwendet.

H

Header Metadaten, die am Anfang einer Datei oder eines Datenblocks stehen, werden in der Informationstechnik als Header bezeichnet.

I

Interface Interfaces sind ähnlich aufgebaut wie Klassen. Sie enthalten allerdings nur Prototypen von Klassen und als Attribute sind nur Konstanten gestattet. Interfaces werden von Klassen implementiert und können nicht instanziiert werden. Sie dienen als Art Wunschzettel oder Schnittstelle für implementierende Klassen [Mül04b].

J

Java Java ist eine Programmiersprache.

K

Klasse Klassen sind die Bausteine eines Programms. Sie beschreiben Methoden und Eigenschaften eines Objekts [Mül04b].

Konstruktor Eine spezielle Methode/Prozedur, welche Variablen erzeugt, wird im Bereich der Informatik als Konstruktor bezeichnet. Durch den Aufruf des Konstruktors ist diese Variable in einem definierten Anfangszustand.

M

Metadaten Daten, die Information über andere Daten beinhalten, werden als Metadaten bezeichnet. Metadaten liegen also eine Ebene über den eigentlichen Daten.

Methode Eine Methode ist eine Prozedur, welche mit einem bestimmten Objekt verknüpft ist [Mül04b].

Modellierung Die Modellierung ist ein oft grafisches Verfahren zur Darstellung eines Programms, seiner Struktur und inneren Abläufe. Es wird häufig im Bereich der Softwareentwicklung eingesetzt.

Modellierungssprache Modellierungssprachen stellen Anforderungen, Struktur und innere Abläufe einer Software dar und helfen Spezifikationen in grafischer Form verständlich zu machen. Zusätzlich können aus den so modellierten Strukturen Quellcodefragmente generiert werden. Beispiele sind UML und SysML.

Modellierungswerkzeug Ein Modellierungswerkzeug ist eine Software, welche die Erstellung eines Modells ermöglicht.

O

Objektorientierung Die Objektorientierung ist ein Prinzip in der Informatik, bei dem versucht wird, die zu bearbeitenden Daten aufgrund ihrer Eigenschaften und Operationen einzuordnen. Ein solches Objekt wird dann in Form von Klassen dargestellt.

Open Source Open Source ist der Fachbegriff für frei verfügbare Software. Dabei ist nicht nur das Programm an sich verfügbar, sondern auch der Quellcode in einer von Menschen lesbaren Form. Dieser Quellcode darf bearbeitet werden und das Produkt dann weiter genutzt und verteilt werden.

P

Paket Als Pakete werden Gruppierungen von Klassen, welche eine Beziehung zueinander haben, bezeichnet. Dabei können Pakete selbst auch wieder Unterpakete besitzen. In Java werden Paketnamen klein geschrieben und durch einen Punkt getrennt [Mül04b].

Pattern In der Informatik wird der Begriff Pattern als Bezeichnung für Entwurfsmuster verwendet. „Entwurfsmuster lösen bekannte und wiederkehrende Entwurfsprobleme. Sie fassen Design- und Architekturwissen in kompakter und wieder verwertbarer Form zusammen“ [ES04].

Persistenz „Persistenz ist die nachhaltige Speicherung von Daten oder Objekten auf nichtflüchtigen Medien“ [Bog99].

S

Server Ein Server ist i.A. ein Programm, das auf einem Rechner läuft und einen bestimmten Dienst anbietet, der über das Netzwerk von anderen Rechnern oder Programmen genutzt werden kann [RSSW03]. Im Rahmen dieser Arbeit weicht die Definition etwas ab. Der Server ist der aktive Part, welcher einen Client anspricht. Somit ist ein Server eine Software, die mit einem oder mehreren Clients kommuniziert und diesen Zugang zu weiteren Clients verschafft [Ber07b].

Service Im Rahmen dieser Diplomarbeit wird unter einem Service ein Berechnungsmodul für ein Subsystem eines Satelliten verstanden. Dies kann z.B. das Orbitmodell sein.

Socket „Sockets stellen aus Programmiersicht die Softwareschnittstelle zu einem Netzwerk dar“ [KY02].

String Ein String ist in der Informatik eine Kette von Zeichen aus einem definierten Zeichensatz. In vielen Programmiersprachen ist String auch ein Datentyp.

T

Tag In der Informatik wird Tag als Begriff für einen Auszeichner genutzt. In den sogenannten Auszeichnungssprachen (z.B. XML) werden sie häufig als benannte Klammern für die Erzeugung der Baumstruktur eingesetzt. Sie treten in der Regel dort immer paarweise mit einem öffnenden und einem schließenden Element auf. Gekennzeichnet sind sie durch die „kleiner als“ und „größer als“ Zeichen, wobei der schließende Tag noch durch einen führenden Schrägstrich markiert ist.

Thread „Allgemein gesprochen, versteht man unter einem Thread eine Folge von Anweisungen, die unabhängig von anderen Threads nebenläufig ausgeführt werden können“[RSSW03].

V

Validierung In der EDV wird das Prüfen von Daten bezüglich definierter Strukturvorgaben mit einem Programm als validieren oder Validierung bezeichnet [JH06].

Vererbung Die Vererbung bezeichnet in der Informatik ein Konzept zum Aufbau einer neuen Klasse mit Hilfe von bereits bestehenden Klassen [Ber07a].

X

XML Die XML ist eine Sprache aus der EDV, mit der die Struktur von Dokumenten beschrieben wird [SSEHW+06].

Abkürzungsverzeichnis

A

API Application Programming Interface.

ASCII American Standard Code for Information Interchange.

C

CD Compact Disc.

D

DLR Deutsches Zentrum für Luft- und Raumfahrt in der Helmholtz-Gemeinschaft.

E

EA Enterprise Architect.

EDV Elektronische Datenverarbeitung.

EPM Earth Pointing Mode.

G

GMT Greenwich Mean Time.

I

IDE Integrated Development Environment.

ILR Institut für Luft- und Raumfahrttechnik.

IP Internet Protocol.

J

JDK Java Development Kit.

JRE Java Runtime Environment.

L

LEO Lower Earth Orbit.

R

RTF Rich Text Format.

S

SI Système International d'Unités.

SISTEC Simulations- und Softwaretechnik.

SMASS Satellite Mission Analysis and Simulation System.

SPM Sun Pointing Mode.

SSH Secure Shell.

SysML Systems Modeling Language.

T

TCP Transmission Control Protocol.

TU Technische Universität.

U

UML Unified Modeling Language.

UR User Requirement.

X

XML Extensible Markup Language.

XSD XML Schema Definition.

1. Einleitung

In der heutigen Zeit werden Systeme immer komplexer. Sie sollen immer mehr können und dabei immer kleiner werden. Die Entwicklung von diesen komplexen Systemen ist deshalb sehr zeit- und kostenintensiv. Um in einer möglichst frühen Phase eines Projektes, in der noch keine Hardware existiert, die Leistungsfähigkeit des geplanten Systems zu überprüfen, wird es simuliert [CK06]. Die Simulation dient der Analyse insbesondere von dynamischen Systemen und deren Verhalten. Es werden Schlüsselparameter untersucht und mit den Zielvorstellungen verglichen.

Im Bereich der Raumfahrt verhält es sich ähnlich. Auch Satelliten haben sich in den letzten 51 Jahren stark weiterentwickelt. Der erste Satellit Sputnik 1 startete 1957 und sendete ein Kurzwellensignal in Form eines Pieptons aus. Die heutigen Satelliten haben dagegen sehr breit gefächerte Aufgaben, angefangen von der Erdbeobachtung über die Kommunikation und Navigation bis hin zur Forschung [RNB88].

Da Satelliten zumeist Einzelanfertigungen sind, dauert die Entwicklung eines Satelliten ca. fünf Jahre [HL88]. In den Phasen eines Raumfahrtprojektes werden immer wieder Simulationen durchgeführt. Es werden alternative Konzepte untersucht und bestehende Konzepte aufgrund der Simulationsergebnisse optimiert. Der Fokus liegt bei diesen Simulationen auf Schlüsseldisziplinen, wie Energie, Orbit oder Thermal [WL91].

Für die Simulation von Systemen muss zunächst ein Modell erstellt werden. An diesem Modell werden dann Experimente in Form von Simulationen durchgeführt, mit deren Ergebnissen wieder Rückschlüsse auf das reale System gezogen werden können. Da ein Modell eines Satelliten als ganzes sehr schwer zu erstellen ist, werden die Schlüsseldisziplinen in eigene kleine Modelle ausgelagert, welche später zu einem Satelliten zusammengefügt werden. Für jedes Teilmodell gibt es einen oder mehrere Spezialisten, die sich nur mit dieser einer Disziplin beschäftigen. So existiert dann z.B. ein Orbit- oder ein Energiemodell für die Simulation.

Ein großer Vorteil einer Simulation ist, dass die Hardware nicht zwingend zur Verfügung stehen muss. Es können Experimente virtuell durchgeführt werden, ohne das ein Satellit vorhanden ist. Dieser Vorteil ist auch einer der Hauptgründe, warum die TU-Berlin seit dem Jahr 2004 ein eigenständiges System zur Simulation von Satellitenmissionen (SMASS) aufbaut. Mit diesem System ist es möglich, die Lehre an der Universität noch effektiver zu gestalten. Die Studenten bekommen durch die Simulationen ein Bild davon, wie die Schlüsseldisziplinen miteinander zusammenhängen und welchen Einfluss die einzelnen Parameter haben. Sie können mit SMASS eigene Satelliten entwerfen und verschiedene Konzepte miteinander vergleichen.

Mit dieser Arbeit soll SMASS nun strukturell weiterentwickelt werden.

2. Problemstellung

2.1. Aufgabe von SMASS

SMASS zielt auf eine Simulation aller Aspekte einer Satellitenmission ab. Dies beinhaltet nicht nur alle Subsysteme eines Satellitenentwurfs und deren Zusammenspiel, sondern auch die verschiedenen Missionsparameter. Zu den Missionsparametern zählen die Satellitenkonfiguration, die Orbits und die verschiedenen Modi eines Satelliten. Die Missionsparameter können sich während einer Mission ändern. [BK04]

Als Beispiel hierfür seien folgende Szenarien aufgelistet:

- Änderung der Satellitenkonfiguration durch Ausklappen von Antennen oder Solarzellen
- Änderung des Orbit durch Zünden des Antriebs
- Änderung des Modus des Satelliten vom Erdbeobachtungsmodus (EPM) hin zum Modus für die Ausrichtung der Solarzellen auf die Sonne (SPM) zur Energiegewinnung

Derartige Änderungen der Missionsparameter werden in den Simulationen herkömmlicher Designphilosophien zum Entwurf von Kleinsatelliten oft nur bedingt berücksichtigt. Ebenso werden die Subsysteme in der Entwurfphase von Satellitenprojekten meistens nur mit statischen Annahmen und elementaren Abhängigkeiten simuliert. Dadurch geht ein Teil der gegenseitigen Beeinflussung von Subsystemen eines Satelliten und deren Auswirkungen auf die Mission verloren.

SMASS soll diese Lücke nun schließen und eine Simulation ermöglichen, welche die Änderung der Missionsparameter während der Mission berücksichtigt und somit das gesamte Missionsszenario simuliert. Auch die Erprobung der Zusammenarbeit der einzelnen Disziplinen der Simulation eines Satelliten in Form von Modulen und deren mögliche gegenseitige Beeinflussung soll durch SMASS gewährleistet werden. Mit Hilfe von SMASS soll ein Entwicklungsteam zeit- und kosteneffektiv zu einem ausgewogenen Systementwurf für einen Satelliten gelangen. Es ist im Gegensatz dazu nicht das Ziel, mit SMASS jedes Subsystem des Satellitenentwurfs bis ins kleinste Detail genau zu simulieren. [BK04]

Der mit SMASS mögliche Entwicklungsprozess wird in Abbildung 2.1 dargestellt. Zunächst entwirft das Team ein System und die zugehörige Mission. Danach wird dieses

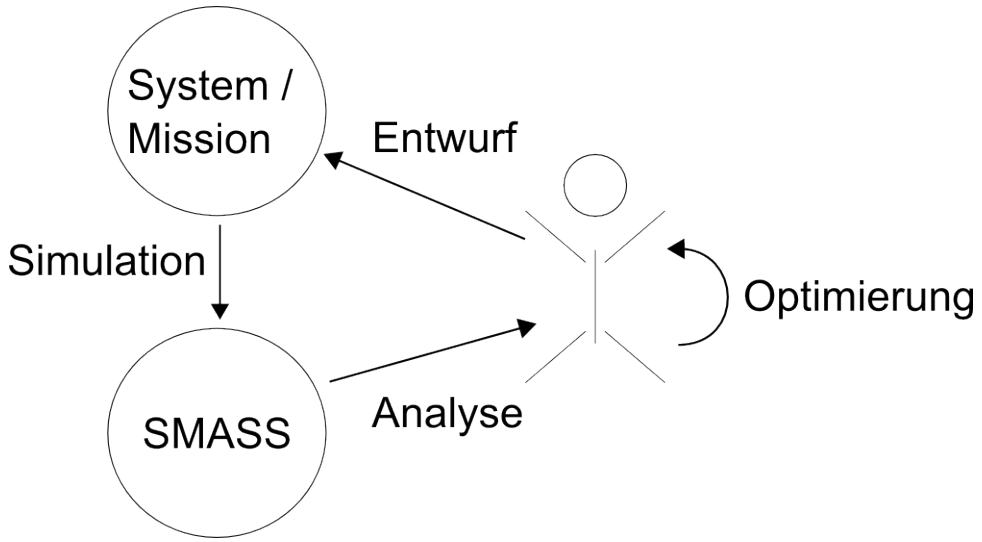


Abbildung 2.1.: Anwendung von SMASS

System zusammen mit der Mission als Input für SMASS verwendet, um eine erste Simulation durchzuführen. Die Ergebnisse von SMASS werden anschließend von den Mitgliedern des Entwurfprojektes ausgewertet und analysiert. Daraufhin erfolgt ein erster Iterationsschritt für das System und die Mission, welcher dann wieder als Input für SMASS dient. Auf diese Art und Weise wird das Projekt vorangetrieben.

Weitere geforderte Eigenschaften von SMASS sind, dass die Module auf unterschiedlichen Computern, Betriebssystemen und Microcontrollern laufen können. Es soll also eine Hardware- und Softwareunabhängigkeit geschaffen werden, sodass es sogar für den Kern und die anderen Module von SMASS egal ist, in welcher Programmiersprache das jeweilige Modul geschrieben wurde. Dies impliziert auch, dass die verschiedenen Module auf voneinander räumlich getrennten Computern laufen können und ein Modul auch durch die entsprechende Hardwarekomponente ausgetauscht werden kann. Um dies zu gewährleisten, wurde bereits der Vorschlag gemacht, dass Nachrichtenformat auf ASCII basierend zu gestalten. [BK04]

SMASS wird vorwiegend im universitären Bereich eingesetzt werden. Hierbei soll SMASS in verschiedenen Lehrveranstaltungen, an der TU-Berlin wären mögliche Veranstaltungen z.B. Raumfahrtsystementwurf und Satellitenentwurf, eingebunden werden. Dabei wird die Aufgabe des Entwurfs an die Studenten weitergeleitet. Diese können dann innerhalb der Übungen zu den Vorlesungen einen Satelliten selber entwerfen, eine zugehörige Mission entwickeln und mit SMASS beides zusammen simulieren. Bei dieser Anwendung kommt bereits zum Tragen, das SMASS den Entwurfsprozess erheblich beschleunigt. Denn in diesen Lehrveranstaltungen soll ein erster grober Entwurf eines Satelliten mit all seinen Subsystemen getätigt und eine dazu passende Mission entworfen werden, was in der Raumfahrt sonst mehrere Jahre dauert. Weiterhin soll SMASS den Studenten und Leitern der Lehrveranstaltungen eine Visualisierungsmöglichkeit bieten. Es soll der Einfluss von Veränderungen der Parameter innerhalb der Subsysteme oder

der Mission dargestellt werden.

Aus diesen Aufgaben von SMASS lassen sich nun folgende Anforderungen (User Requirements) für die Weiterentwicklung von SMASS ableiten:

UR	1.0	Simulation von Missionen mit unterschiedlichen Modi des Satelliten müssen gewährleistet werden.
UR	2.0	Simulation von Satellitenentwürfen als ganzes, nicht nur auf der Ebene von Subsystemen, müssen gewährleistet werden.
UR	3.0	Verschiedene Disziplinen müssen aufeinander Einfluss nehmen können.
UR	4.0	SMASS muss eine Möglichkeit zur Repräsentation von Simulationsdaten beinhalten.
UR	5.0	SMASS muss für die Analyse die Simulationsergebnisse abspeichern können.
UR	6.0	Es muss eine zentrale Stelle für die Konfigurationsdaten der Module und damit für das Modell der Simulation geben.
UR	7.0	Eine Simulation muss steuerbar sein.
UR	8.0	SMASS muss mit Modulen auf anderen Computern kommunizieren können.
UR	9.0	SMASS soll Hardware-unabhängig sein.
UR	10.0	SMASS soll Software-unabhängig sein.
UR	11.0	SMASS soll die Einbindung von Hardware in die Simulation ermöglichen.

Tabelle 2.1.: Anforderungen an SMASS - Teil 1

2.2. Ausgangszustand von SMASS

In diesem Abschnitt soll der Ausgangszustand von SMASS am Anfang dieser Arbeit kurz beschrieben werden.

Historisch bedingt wurden zunächst die Berechnungsmodule entwickelt. Zu diesem Zeitpunkt gab es allerdings noch keine Richtlinien und Spezifikationen bezüglich der Kommunikation der Module untereinander und mit dem noch zu entwickelnden Server. Es war lediglich das Nachrichtenformat in Form einer ASCII Nachricht und die Kommunikation mittels TCP/IP festgelegt. Eine Kommunikation zwischen den Modulen oder mit einer zentralen Stelle war zu diesem Zeitpunkt noch nicht möglich, da der Server erst später entwickelt wurde. Dadurch ist in manchen Modulen noch Funktionalität aus den anderen Modulen integriert. So ist z.B. in allen am Anfang dieser Arbeit existierenden

Modulen ein Orbitmodell integriert. Dies war bis zum Zeitpunkt der Entwicklung des ersten Servers und damit der Möglichkeit der Kommunikation untereinander auch nötig, um die Module überhaupt zu testen.

Aufgrund dieser doch recht isolierten Entwicklung der einzelnen Berechnungsmodule in den Studienarbeiten waren zum Anfang der Entwicklung des Servers [Pfe07] die von den Modulen erwarteten Eingangsdaten und die produzierten Ausgabedaten nicht unter den einzelnen Modulen kompatibel. Deshalb wurde sich mit einem Konverter beholfen (siehe Abbildung 2.2). Dieser Konverter hat Teile der Ausgabedaten des Bahn- und die Ausgabedaten des Thermalmoduls in für die anderen Module verständliche Formate umgewandelt.

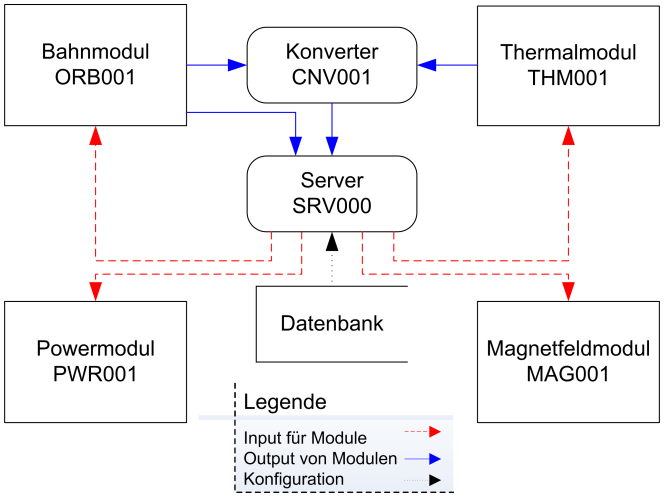


Abbildung 2.2.: Datenaustausch bei SMASS

Während meines Praktikums bei SISTEC am DLR war es meine Aufgabe, SMASS zu analysieren und ein Handbuch dazu anzufertigen. In diesem Handbuch sollte die Eignung von SMASS für das Projekt „Virtueller Satellit“ im DLR herausgestellt werden. Aufgrund der Kompetenzen von SISTEC, welche in der Simulations- und Softwaretechnik liegen, stellte sich ein Optimierungsbedarf hinsichtlich der Struktur von SMASS heraus. Die erste Version des Servers ist nicht objektorientiert implementiert worden und daher sehr schlecht les- und nachvollziehbar. Aus der prozeduralen Programmierung der ersten Version des Servers ergeben sich weitere Nachteile.

Durch die nicht vorhandene Modularität sind keine Schnittstellen für die Datenverarbeitung, die Kommunikation und die grafische Oberfläche von SMASS angelegt worden. Dies geschah im Bereich der Kommunikation sicherlich deshalb, da die einzelnen Module sich bis zur Entwicklung des Servers jeweils selber um ihren Teil der Kommunikation gekümmert haben. Dieser Ansatz birgt allerdings den Nachteil, dass jeder Entwickler eines Moduls immer wieder vor dem Problem steht, die Kommunikation zu gewährleisten. In dem vorliegenden SMASS System existieren vier verschiedene Implementierungen des Teils der Kommunikation, welcher auf Seiten der Module nötig ist.

Jeder Modulentwickler erstellt also eine eigene Implementierung der Kommunikation. Fehlt hier allerdings das nötige Wissen über die Kommunikation, ist die richtige Anwendung und Implementierung gefährdet. Im schlimmsten Fall führt es dazu, dass zu dem Modul keine Verbindung aufgebaut werden kann. Außerdem ist der Entwickler der Module dadurch von seiner eigentlichen Arbeit, der Berechnung der jeweiligen Disziplin, abgelenkt. Dies erhöht den Zeitaufwand der Entwicklung einzelner Module. Im Bereich der Datenverarbeitung soll in Zukunft eine Datenbank angebunden werden. Aber auch hierfür existiert leider keine Schnittstelle in SMASS. Analog dazu verhält es sich mit der Benutzeroberfläche. Hier wäre es ebenfalls sehr schwierig eine andere Oberfläche, z.B. ein Web-Frontend, einzubinden.

Neben den nicht vorhandenen Schnittstellen, was die Flexibilität und damit Erweiterbarkeit des Gesamtsystems einschränkt, ist als weiterer Nachteil von SMASS die schlechte Wartbarkeit zu nennen. Diese liegt vor allem begründet in der Lesbarkeit und der Vermischung von Funktionalitäten innerhalb des prozeduralen Quelltextes. Die Vermischung von Funktionalitäten führt zu einem erhöhten Zeit- und Kostenaufwand, wenn in einer Funktionalität eine Änderung vorgenommen werden soll. Diese Änderung hat dann in prozeduralen Systemen oft Auswirkungen auf die ganze Anwendung.

Durch die Entwicklung des Servers wurde SMASS zum ersten Mal als Gesamtsystem in einen ausführbaren Zustand gebracht. Bei der Entwicklung des Servers hat Pfeiff ([Pfe07] S.49ff) mehrere Probleme mit den Modulen identifiziert. Ein großer Teil dieser Probleme ist auf eine nicht einheitliche Kommunikation und auf die bereits angesprochene mehrfache Implementierung des Teils der Kommunikation, welcher auf Seite der Module stattfindet, zurückzuführen. Diese Probleme zu beheben, war allerdings nicht die Aufgabe der ersten Version des Servers. Vielmehr kann die Aufgabe aus heutiger Sicht grob so zusammengefasst werden, dass durch die Entwicklung des Servers die Module zum Laufen gebracht werden sollten. Es wurde versucht mit den Unzulänglichkeiten der Module umzugehen, ohne jedoch diese zu beheben.

Aus dem Stand von SMASS vor dieser Arbeit ergeben sich die folgenden weiteren Anforderungen für die Weiterentwicklung von SMASS:

UR	12.0	Die Wartbarkeit von SMASS muss sichergestellt werden.
UR	13.0	SMASS soll einfach erweiterbar sein.
UR	14.0	SMASS muss Schnittstellen für die Datenverarbeitung und die grafische Oberfläche bereitstellen.
UR	15.0	SMASS soll ein Framework als Vorlage für die Kommunikation auf Seiten der Module beinhalten.

Tabelle 2.2.: Anforderungen an SMASS - Teil 2

2.3. Ziel dieser Arbeit

Diese Arbeit soll nach der Analyse von SMASS und der Herausstellung der Anforderungen das System strukturell weiterentwickeln. Dies ist allerdings aus den in Abschnitt 2.2 erläuterten Punkten nicht auf Basis des bereits existierenden Servers möglich. Deshalb soll im Rahmen dieser Arbeit ein Framework geschaffen werden. Dieses Framework soll sich durch die verstärkte Verwendung der Objektorientierung und damit verbunden durch eine höhere Modularität auszeichnen. Ein objektorientierter Ansatz ermöglicht das Beherrschen von immer komplexer werdenden Systemen. SMASS ist ein solches komplexes System. Durch die Objektorientierung erhält der Entwickler die Möglichkeit zu abstrahieren und somit komplexe Systeme zu strukturieren. Die Strukturierung oder auch Modularität wird dabei durch die Trennung von Funktionalitäten erreicht.

Das Konzept der Modularität ist Voraussetzung für die Zukunftstauglichkeit von SMASS und gilt als wichtiges Ziel dieser Arbeit. Durch die Modularität können Softwareschnittstellen definiert werden, welche den Kern des Systems von den anderen Schichten, wie z.B. der Datenverarbeitung oder der Benutzeroberfläche, klar abgrenzen. Es können dadurch einzelne Schichten ausgetauscht werden, ohne dass es einer aufwendigen Bearbeitung des Kerns von SMASS bedarf. Die Funktionalitäten werden also klar getrennt.

Auch innerhalb der einzelnen Schichten von SMASS soll das Prinzip der Objektorientierung gelten, wodurch sich zugleich die Wartbarkeit des Systems erhöht. Es können innerhalb der Schichten Änderungen gemacht werden, die ihrerseits keinen direkten Einfluss auf die anderen Schichten von SMASS haben. Dadurch wird es auch möglich, das System relativ einfach zu erweitern. Es könnte z.B. das Nachrichtenformat vom SMASS-Format auf eine XML-Nachricht umgestellt werden.

Durch die Modularisierung wird ebenfalls die Kapselung der Kommunikation möglich. Folglich wird die doppelte Implementierung der Kommunikation in den Berechnungsmodulen überflüssig. Es wird nur noch eine Implementierung für die Kommunikation pro Programmiersprache geben. Die Module werden dadurch eher zu einer Art Service, der eine reine Dienstleistung in Form eines Berechnungsschrittes der jeweiligen Disziplin anbietet. Ein Entwickler eines Services muss sich somit keine Gedanken mehr um die Infrastruktur der Kommunikation machen, sondern kann sich voll auf die Entwicklung seiner Berechnungsalgorithmen konzentrieren. Dementsprechend entsteht ein erheblicher Zeitvorteil und eine Verringerung des Testaufwands bei der Entwicklung der Module. Ein weiterer Vorteil der Kapselung der Kommunikation ist die erhöhte Zuverlässigkeit des Systems. Es müssen nur die vorgegebenen Standardbauteile der Kommunikation zu einem ausgereiften Zustand gebracht werden. Diese werden dann nur noch von den Services benutzt.

Die Auslagerung der Kommunikation, wodurch die Module zu den Services werden, ist auch aufgrund der Arbeitsweise der bereits entwickelten Module gerechtfertigt. Diese sind vollkommen passiv und reagieren nur, wenn sie Nachrichten vom Server erhalten. Aufgrund dieses so genannten „polling“-Modus, welcher ein zyklisches Abfragen des Status beschreibt, sieht der funktionale Teil der Kommunikation auf Seiten der Services für

alle identisch aus. Das einzige, was zwischen den Services unterschiedlich sein kann, ist die Anzahl und Art der übermittelten Parameter. Die Parameter sind aber Bestandteil einer Nachricht. Werden die Nachrichten geschickt modelliert, kann auch dieser Teil der Kommunikation für alle Services gleich gestaltet werden.

Neben den erwähnten Anforderungen aus Kapitel 2.1 und 2.2 ergeben sich noch folgende Anforderungen an SMASS:

UR	16.0	Die Funktionalitäten in SMASS sollen getrennt werden.
UR	17.0	Die Benutzung von SMASS soll vereinfacht werden.

Tabelle 2.3.: Anforderungen an SMASS - Teil 3

3. Aufbau und Funktion

Dieses Kapitel beschreibt die Architektur des SMASS-Frameworks und wie diese Architektur entstanden ist. Anhand von UML-Diagrammen wird die Struktur und Funktion erläutert. Es soll ein Überblick und ein grundsätzliches Systemverständnis von dem weiterentwickelten SMASS-System vermittelt werden.

Um die folgenden Ausführungen und Diagramme besser nachvollziehen zu können, soll hier noch kurz auf die verwendeten Konzepte der Softwareentwicklung und deren Notation eingegangen werden.

Ein Mittel der Strukturierung in der Softwareentwicklung sind Schichten. Dabei werden einzelne Teile des Systems einer bestimmten Schicht zugeordnet. Diese Schichten werden mit einer Paketstruktur auf Quelltextebene realisiert. Die Pakete enthalten weitere Unterpakete, Klassen und Interfaces (siehe Anhang B), welche dann die Aufgabe der jeweiligen Schicht gewährleisten.

Eine Schichten-/Paketstruktur hat ganz allgemein Vor- und Nachteile, welche von Müller ([Mül04a], S.11) und Fleischer ([Fle06], S.44f) im Wesentlichen herausgestellt wurden.

Vorteile:

- Wiederverwendung in unterschiedlichen Kontexten bei geeigneter Wahl des Interfaces
- Reduzierung des Aufwands für Implementierung bei Wiederverwendung
- Reduzierung der Fehler bei Wiederverwendung
- Austausch einzelner Implementierungen einer Schicht durch geringe Abhängigkeiten zwischen den Schichten
- Änderung der Nutzung einer Schnittstelle zieht nur Änderungen in der Nutzerschicht nach sich, Anbieterschicht bleibt unangetastet
- Wahrung der Übersichtlichkeit komplexer Systeme
- Einfache Modellierung von Abhängigkeiten durch Interfaces
- Implementierungen innerhalb der Pakete durch Interfaces voneinander unabhängig
- Gegenseitige Benutzbarkeit der Pakete durch Übergabe der Objekte des zu benutzenden Typs

- Einfache Erweiterung des Systems durch Vererbung
- Direkte Umsetzung in objektorientierte Programmiersprache

Nachteile:

- Einbußen in der Performance durch Datentransport
- Finden der richtigen Anzahl der Schichten oft kompliziert

In den folgenden Erläuterungen wurden verschiedene Arten von Diagrammen für die Darstellung des SMASS-Frameworks verwendet. Dies sind vorwiegend UML-Diagramme. Im Bereich der Struktur wurden Paket- (*pkg*) und Klassendiagramme (*class*) verwendet. Für die Darstellung von Verhalten wurden Zustands- (*stm*) und Aktivitätsdiagramme (*act*) benutzt. Weitere Erläuterungen zur UML und deren Diagramme finden sich im Anhang B.

Die Schichten werden im weiteren Textverlauf in Anführungszeichen gesetzt (z.B. „Data Handling“). Die Elemente der Diagramme werden im Text immer kursiv hervorgehoben. Bei der Namensgebung der Elemente wurde folgende Notation verwendet:

Element	**Bezeichnung**	**Beispiel**
Pakete	am Anfang eine Folge von Großbuchstaben möglich, ansonsten kleine Buchstaben	*frontend*, *ORB001-implementation*
Klassen	beginnen mit Großbuchstaben; bei zusammengesetzten Bezeichnungen beginnt jedes neue Wort mit einem Großbuchstaben	*ORB001Implementation*, *SatelliteSimulation*
abstrakte Klassen	großes „A“ am Anfang und dann der Klassenname	*AServiceApplication*, *AView*
Interfaces	großes „I“ am Anfang und dann der Klassenname	*IService*, *IConfigurableObject*

Tabelle 3.1.: Notation von Diagrammelementen

Für das SMASS-Framework musste zunächst ein Konzept erarbeitet werden, damit das System am Ende tatsächlich flexibel und erweiterbar ist und damit die Anforderungen aus Kapitel 2 erfüllt werden. Die grundlegende Idee hinter der Objektorientierung ist die Kapselung der Funktionalitäten. Dafür müssen im ersten Schritt die nötigen Funktionalitäten identifiziert werden. Im zweiten Schritt muss dann untersucht werden, wie diese Funktionalitäten sinnvoll getrennt werden können. Dabei soll eine möglichst maximale Modularität des Systems erreicht werden, ohne die Funktionalität einzuschränken.

SMASS ist grundsätzlich eine Server/Client-Anwendung. Der Server und der Client kommunizieren dabei mittels Nachrichten, welche über eine TCP/IP-Verbindung verschickt werden. Dabei ist der Server der aktive Part in der Kommunikation.

Die Steuerung der Simulation der Satelliten wird ebenfalls vom Server übernommen. Dazu zählt neben der Ausführung auch die Verwaltung von Simulationen, da es mit SMASS nun möglich ist, mehrere Simulationen parallel zu betreiben.

Eine weitere Aufgabe des Servers ist es, die Daten einer Simulation zu verarbeiten. Dies umfasst zum einen das Laden und Speichern der Konfigurationsdaten und zum anderen das Speichern der Ergebnisse einer Simulation zur späteren Auswertung.

Die letzte Aufgabe des Servers ist es, dem Benutzer eine Möglichkeit zum Eingreifen in den Simulationsablauf zu geben. Dazu ist es nötig, eine Repräsentation der Simulation zu gewährleisten und die Benutzerwünsche entgegen zu nehmen.

Die Clients von SMASS müssen zunächst einmal erfolgreich vom Benutzer beim Server angemeldet werden. Anschließend reagieren sie nur noch auf eingehende Nachrichten. Ihre Aufgabe besteht in der Berechnung der jeweiligen Disziplin eines Satelliten. Zusätzlich muss eine Möglichkeit gegeben werden, die Clients zu konfigurieren.

Um die Trennung dieser Funktionalitäten zu gewährleisten, wurde nun für jede Aufgabe eine entsprechende Schicht im SMASS-Framework angelegt. Folgende Tabelle fasst die Aufgaben und die zugeordneten Schichten zusammen:

Aufgabe	**Schicht**
Kommunikation	„Utilities“
Simulationsausführung und -verwaltung	„Simulation“
Datenverarbeitung	„Data Handling“
Benutzeroberfläche	„Frontend“
Berechnungen der Fachdisziplinen	„Services“

Tabelle 3.2.: Aufgaben und Schichten von SMASS

Die konkreten Schichten von SMASS, an denen einige ausgewählte Vor- und Nachteile und die Erfüllung der Anforderungen erläutert werden, sind in der Abbildung 3.1 dargestellt. Dabei wurden die englischen Begriffe verwendet, um die Analogie zur Implementierung zu wahren. Auf die Funktionen der Bestandteile der Pakete wird in den folgenden Unterkapiteln ab 3.1 eingegangen.

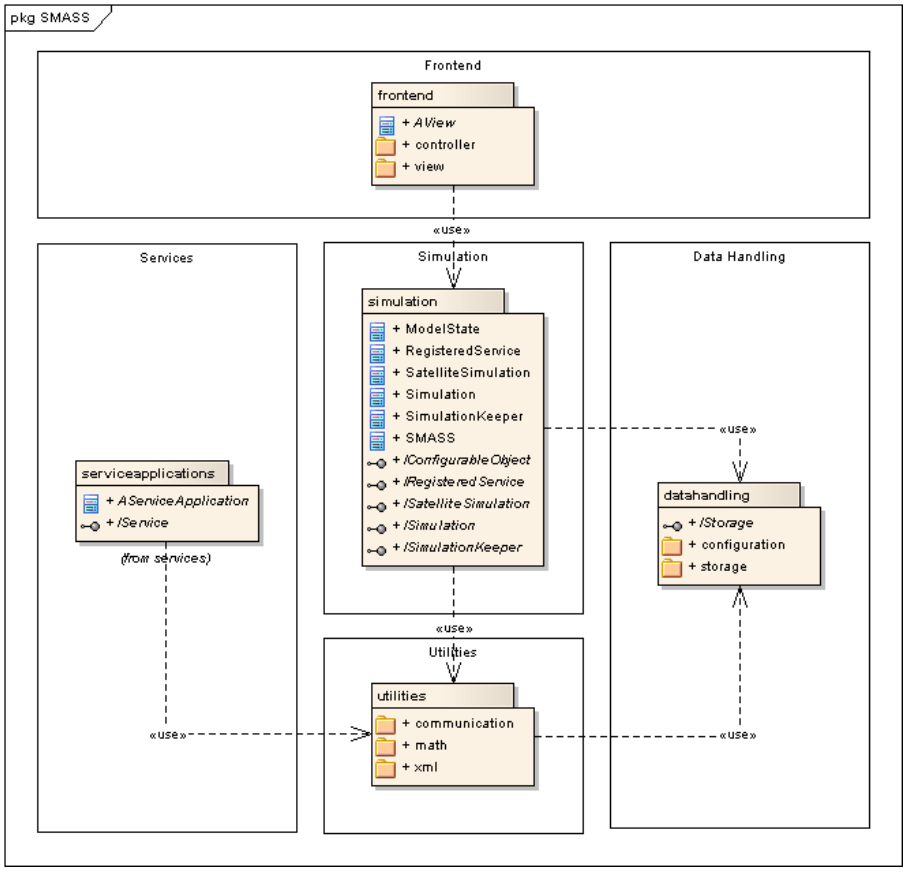


Abbildung 3.1.: Schichten des SMASS-Frameworks

„Simulation“

„Simulation“ ist die zentrale Schicht in dem SMASS-Framework. Diese Schicht stellt die Laufzeitrepräsentation der Simulationen dar. Sie beinhaltet die Haltung der Daten zur Laufzeit und erfüllt damit Anforderung UR 6.0. Darüber hinaus ist auch die Ausführungslogik (business logic) in dieser Schicht hinterlegt. Die Ausführungslogik übernimmt das Anlegen und Verwalten von Simulationen und die Steuerung einzelner Simulationen, was in UR 7.0 gefordert wird. Die Laufzeitrepräsentation der Simulationen mit der Ausfürungslogik wird im weiteren Verlauf auch als Modell bezeichnet.

Um eine Simulation durchzuführen müssen zunächst einmal alle Bestandteile der Simulation konfiguriert werden. Für das Laden der Konfigurationsdaten benutzt die „Simulation“-Schicht das *configuration*-Paket aus der „Data Handling“-Schicht. Nach der Konfiguration kann eine Simulation gestartet werden. Die Ergebnisse der Berechnungen in den Services fragt die Schicht dann über Nachrichten ab. Zum Versenden und Empfangen der Nachrichten benutzt die „Simulation“-Schicht das *communication*-Paket aus der „Utilities“-Schicht. Für den Zugriff von anderen Schichten auf „Simulation“ sind Interfaces realisiert (siehe Abbildung 3.1), wodurch die Abhängigkeit von anderen Schichten minimiert wurde.

„Utilities“

In der Schicht „Utilities“ ist der Bereich der Kommunikation innerhalb des Paketes *communication* untergebracht (siehe 3.2). Zum Bereich der Kommunikation gehört der Aufbau einer Verbindung, wie z.B. einer TCP/IP-Verbindung, zur Registrierung der Clients und zum Nachrichtenaustausch. Dies ermöglicht die Kommunikation mit den Clients, was Grundlage für die Erfüllung von UR 8.0 ist.

Weiterhin gehört das Nachrichtenhandling zur Kommunikation. Hierbei wird unterschieden zwischen dem Server und den Clients. Auf Seiten des Servers zählen das Verschicken der Nachrichten an die Clients und das Verarbeiten der Antworten der Clients zum Nachrichtenhandling. Die Ergebnisse der Berechnung eines Simulationsschrittes durch die Services, welche sich in den Antwortnachrichten der Clients befinden, werden durch die Nutzung der Schicht „Data Handling“ gespeichert.

Auf der Seite der Clients gehören zum Nachrichtenhandling das Anfragen und Setzen der aktuellen Konfiguration der Services sowie das Auslösen der Berechnung in den Services. Dafür nutzt das *communication*-Paket das Interface *IService* aus der Schicht „Services“.

Mit den oben genannten Punkten setzt das *communication*-Paket konsequent die Anforderung nach der einheitlichen Kommunikation für alle Services (UR 15.0) um.

Neben der Kommunikation ist in dieser Schicht auch das *math*-Paket dargestellt. In diesem Paket sollen mathematische Implementierungen z.B. für Näherungsverfahren wie das Newton- oder Runge-Kutta-Verfahren hinterlegt werden. Diese können dann von den Services benutzt werden.

Das *xml*-Paket in der „Utilities“-Schicht beinhaltet allgemeine Funktionen zum Auslesen und schreiben einer XML-Datei, welche z.B. bei der Konfigurierung verwendet werden.

„Services“

Die dritte Schicht heißt „Services“ und beinhaltet die Schnittstelle für die Services und deren konkrete Implementierungen. Aus Gründen der Übersichtlichkeit ist in Abbildung 3.1 nur die Schnittstelle der Services dargestellt.

Die Idee der Services entstand aus der Analyse der Berechnungsmodule von SMASS, welche jeweils eine eigene Implementierung der Kommunikation beinhalteten. Die Kommunikation wird im Framework ausgelagert, wodurch die Berechnungsmodule zu Services und damit unabhängiger von den anderen Funktionalitäten, wie z.B. der Kommunikation, werden. Dadurch kann die Entwicklung der Services auf die Berechnungsroutinen konzentriert werden und sie bieten so eine Dienstleistung in Form der Berechnung des nächsten Simulationsschrittes an. Diese Dienstleistung wird von der „Utilities“-Schicht genutzt. Durch die eigene Schicht für die Services wird auch die Serviceorientierung des SMASS-Frameworks hervorgehoben.

Bei den Services besteht aber, wie Pfeiff ([Pfe07] S.7) bereits feststellte, die Schwierigkeit, dass ein Großteil noch entwickelt werden muss. Deshalb wurde für die Services ein standardisiertes Interface geschaffen (siehe Kapitel 3.3). Dieses Interface deckt die Funktionalitäten eines Services ab und dient damit als Schnittstelle für andere Schichten. Die Entwickler zukünftiger Services realisieren nur die in Abbildung 3.1 gezeigte Schnittstelle aus dem Paket *serviceapplications*. Ein Service hat dann nur noch die Aufgabe sich zu konfigurieren, die aktuelle Konfiguration auszugeben und einen Simulationsschritt zu berechnen.

„Data Handling“

Für die Verarbeitung von Daten ist die „Data Handling“-Schicht zuständig. Diese umfasst zum einen das Speichern der Ergebnisse der Simulation zur späteren Auswertung (UR 5.0) und zum anderen das Laden und Speichern der Konfigurationsdaten (UR 6.0). Beide Funktionalitäten sind noch weiter gekapselt in jeweils einem Paket dieser Schicht (siehe Kapitel 3.4).

Die Datenhaltung ist nicht in der „Utilities“-Schicht angeordnet worden, da hinter den Daten noch ein für SMASS spezifisches Datenmodell steckt. Sie ist also kein Werkzeug, welches ohne Modifikationen in anderen Anwendungen eingesetzt werden kann.

Damit die Flexibilität und Erweiterbarkeit von SMASS gewahrt bleibt, wurde der ganze Bereich Datenhaltung in eine eigene Schicht verlegt. Dies ist laut Rausch ([Rau06] S.20) im Bereich der Softwarearchitektur durchaus üblich. Dank der Einordnung in einer eigenen Schicht und der Entwicklung einer geeigneten Schnittstelle ist es möglich, die komplette Datenhaltung auszutauschen. Diese Austauschmöglichkeit ist insbesondere auch dadurch wichtig, da es nicht Teil dieser Arbeit war, eine ausgereifte Datenverarbeitung zu entwickeln. Mit dem leicht zu realisierenden Austausch der Datenverarbeitung ist auch die Anforderung nach einfacher Erweiterbarkeit (UR 13.0) in diesem Bereich erfüllt.

Damit diese Schicht ausgetauscht werden kann, wurde für den Bereich der Speicherung der Ergebnisse (Paket *storage*) das Interface *IStorage* angelegt. So kann die für die Demonstration umgesetzte Variante mit Textdateien durch eine beliebige Datenbankanwendung ersetzt werden. *IStorage* und die Interfaces für die Konfiguration innerhalb des *configuration*-Pakets bilden zusammen die Schnittstelle für die Datenverarbeitung, wodurch auch UR 14.0 erfüllt ist.

„Frontend“

Die letzte Schicht ist die „Frontend“-Schicht. Der „Frontend“-Schicht wird die Benutzeroberfläche und damit auch die Darstellung der Daten zugeordnet (UR 4.0). Um an die darzustellenden Daten zu gelangen, benutzt die „Frontend“-Schicht die „Simulation“-Schicht. Darüber hinaus manipuliert sie diese Daten, sollte der Benutzer dies wünschen. Somit ist auch der fehlende Teil der Benutzerinteraktion aus der Anforderung der Steuerbarkeit einer Simulation (UR 7.0) gewährleistet.

Die „Frontend“-Schicht ist ebenfalls aus dem Gedanken der Austauschbarkeit entstanden. Ähnlich wie bei der „Data Handling“-Schicht ist es auch hier nicht Aufgabe dieser Arbeit gewesen, eine ausgereifte Benutzeroberfläche zu schaffen. Deshalb muss ein Wechsel der kompletten Schicht durch eine andere Implementierung möglich sein. Es könnte z.B. anstelle des jetzt implementierten Konsolen-Frontends ein Web-Frontend für die Steuerung der Simulationen übers Internet verwendet werden. Dies ist mit der angelegten abstrakten Klasse *AView* und dem *controller*-Paket gewährleistet. Das *controller*-Paket übernimmt dabei die Manipulation des Modells, was aus den Klassen der „Simulation“-Schicht besteht. Dieses Paket könnte ohne Veränderung von einem neuen Frontend benutzt werden.

Durch die Schnittstelle ist in diesem Bereich zum einen die Erweiterbarkeit gewährleistet (UR 13.0) und zum anderen auch die in UR 14.0 geforderte Schnittstelle für die grafische Oberfläche geschaffen worden.

Fazit

Zusammenfassend ist festzustellen, dass die Vorteile für die Schichten-/Paketstruktur überwiegen. Zunächst aber soll der Umgang mit den beiden Nachteilen beschrieben werden, bevor auf einzelne Vorteile anhand von Beispielen eingegangen wird.

An der Benutzerinteraktion und der damit verbundenen Manipulation der Daten durch die Schicht „Frontend“ kann die schwächere Performance von Schichten nachvollzogen werden. Von der Benutzeroberfläche werden die Benutzereingaben entgegengenommen, wie z.B. das Setzen der Schrittweite für die Simulation. Diese Information wird weiter geleitet an das Modell („Simulation“) und von dort an die Datenhaltung übergeben (siehe Abbildung 3.1). Die Daten werden also durch drei Schichten transportiert, bevor sie an ihrem Ziel, z.B. eine Datei, ankommen. Allerdings ist die schwächere Performance in diesem konkreten Anwendungsfall als nicht so gravierend einzuschätzen, da die einzelnen Services auf andere Computer ausgelagert werden können und somit im optimalen Fall die Rechenleistung eines Computers nur für die Aufgaben des Servers zur Verfügung steht.

Schwierig war die Wahl der richtigen Anzahl der Schichten, was der zweite Nachteil der Schichtenstruktur ist. Werden zu viele Schichten definiert, wird das Gesamtsystem unübersichtlich und es steigt der Mehraufwand für die Implementierung. Werden jedoch zu wenige Schichten definiert, kommen die Vorteile des Schichtenmodells nicht zur Geltung. Deshalb wurde ein Kompromiß gefunden, welcher sich teilweise an den in der Softwareentwicklung allgemein gültigen Schichten wie Datenhaltung und Benutzeroberfläche orientiert.

Ein erster Vorteil kann am Beispiel der Kommunikation erläutert werden. Oben wurde als mögliche Verbindung die TCP/IP-Verbindung genannt. Es kann aber auch jede beliebige andere Verbindung an deren Stelle treten, sollte TCP/IP nicht kompatibel zu der bestehenden Hard- bzw. Softwarelösung sein. Dazu müssten lediglich die Implementierungen in dem Paket *communication* geändert oder ausgetauscht werden, ohne das

irgendwo anders im SMASS-Framework etwas bearbeitet werden muss. Damit ist auch die geforderte Hard- und Softwareunabhängigkeit gegeben (UR 9.0 und UR 10.0).

Die Erweiterbarkeit des SMASS-Frameworks zeigt sich auch durch folgende Gedanken. Es wäre neben den *math*-Paket in der „Utilities"-Schicht auch ein Paket *constants* denkbar, welches Naturkonstanten beinhaltet, die dann wiederum von den Services genutzt werden können. Damit würden identische Genauigkeiten und Einheiten verwendet werden. Eine weitere mögliche Erweiterung wäre das Anlegen eines Paketes für einheitliche Fehlermeldungen innerhalb von SMASS.

Mit der Aufspaltung der Clients in die Bereiche Kommunikation und Services wird die Entwicklung weiterer Services vereinfacht (UR 13.0) und somit auch die Benutzerfreundlichkeit gesteigert (UR 17.0). Darüber hinaus kann mit der Auslagerung der Kommunikation auch die Einbindung von Hardwaremodulen anstelle der Services vereinfacht werden (UR 11.0). Es müssen nur noch die Werte aus dem Hardwaremodul ausgelesen werden, was in der konkreten Serviceimplementierung realisiert werden kann.

Weitere Vorteile können an der „Frontend"-Schicht sehr gut erläutert werden. Es handelt sich hierbei um die Vorteile der vereinfachten Modellierung von Abhängigkeiten und die damit im Zusammenhang stehende Unabhängigkeit von den konkreten Implementierungen innerhalb der Pakete. Ein Beispiel für diese Vorteile stellt die abstrakte Klasse *AView* aus der Schicht der Benutzeroberfläche („Frontend") dar. Diese Schnittstelle wird von der Klasse *SMASS* aus der Schicht „Simulation" benutzt. Für das *simulation*-Paket ist es gleichgültig, wie die von *AView* ableitende Klasse, welche sich im Paket *frontend.view* befindet (Anmerkung: Klasse ist in Abbildung 3.1 nicht zu sehen), aussieht. Es benutzt lediglich die in der abstrakten Klasse festgelegten Methoden. Auch die Implementierung, welche die Schnittstelle zur Verfügung stellt, ist unabhängig von der nutzenden Implementierung, da sie nur die Verbindung umsetzt. Am Beispiel von *AView* bedeutet diese Tatsache, dass bei der Implementierung der Klasse, welche von *AView* ableitet, die Nutzung der Schnittstelle (in diesem Beispiel also die Klasse *SMASS*) außen vor gelassen werden kann.

Außerdem spricht für die Strukturierung aus Abbildung 3.1, dass bereits ein wichtiger Teil der Ziele und Anforderungen (siehe Kapitel 2) erfüllt werden kann. Die Struktur ermöglicht nun eine einfache Erweiterbarkeit (UR 13.0) durch die Trennung der Funktionalitäten (UR 16.0) in die Schichten und Pakete. Die Pakete erlauben eine weitere Gliederung der vielen Komponenten von SMASS. Es können jetzt einzelne Komponenten ausgetauscht werden, da die Implementierungen der Komponenten durch die Einführung von Interfaces voneinander unabhängiger sind. Dadurch erhöht sich auch die Flexibilität und Wartbarkeit des Systems (UR 12.0). Darüber hinaus wird durch die Struktur bereits angedeutet, dass auch die Kommunikation aus dem Verantwortungsbereich der Services (was bisher die Berechnungsmodule waren) rausgezogen worden ist und sich jetzt in der Schicht „Utilities" befindet (UR 15.0).

3.1. „Simulation“

3.1.1. Struktur einer Simulation

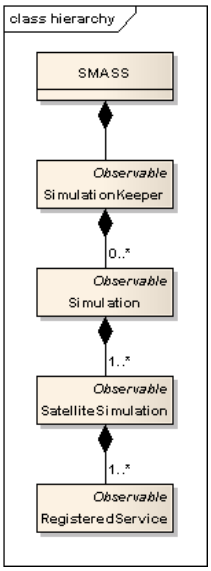


Abbildung 3.2.: Hierarchie des Modells von SMASS
(Attribute und Methoden ausgeblendet)

Die „Simulation“-Schicht beinhaltet das Modell von SMASS. Dieses Modell ist hierarchisch aufgebaut(siehe Abbildung 3.2). An oberster Stelle steht die Klasse *SMASS*, welche den Einstiegspunkt darstellt. In dieser Klasse wird die Hauptansicht der Benutzeroberfläche mit dem zugehörigen Controller zur Manipulation des Modells gestartet.

Innerhalb von *SMASS* wird außerdem genau eine Instanz der *SimulationKeeper*-Klasse erstellt. Diese Klasse ist für die Verwaltung der einzelnen Simulation zuständig, welche parallel laufen können. Mit den Methoden dieser Klasse werden Simulationen erstellt, konfiguriert, gestartet, pausiert, gestoppt oder deren Laufzeitrepräsentation wieder gelöscht.

Die Simulationen, welche im *SimulationKeeper* erstellt werden, sind durch die *Simulation*-Klasse repräsentiert. Diese Klasse übernimmt die Steuerung einer Simulation mit den konkreten Methoden zum Konfigurieren, Starten, Pausieren und Stoppen einer Simulation. Zur Steuerung gehört auch die Berechnung des nächsten Simulationszeitpunkts und die Option zur Gewährleistung einer Mindestzeit bis zum nächsten Simulationsschritt („schwache Echtzeit“), welche in 3.1.3 erläutert wird. Ist eine Simulation gestartet, läuft sie in einem eigenständigen Thread ab.

Zusätzlich zu der Steuerung ist auch die Verwaltung der Satelliten innerhalb der *Simulation*-Klasse untergebracht. Es können hier Satelliten hinzugefügt und wieder entfernt werden. Eine Simulation hat dabei immer mindestens einen Satelliten unter sich. Durch die Trennung von Simulationen und Satelliten ist es möglich, Satellitenschwärme in einer

Simulation zu simulieren. Es könnte so in Zukunft auch leicht eine Kommunikation unter den Satelliten implementiert werden.

Die Satelliten werden durch die Klasse *SatelliteSimulation* repräsentiert. Sie verwaltet die einzelnen Services, welche zu dem Satelliten gehören. Dazu sind hier Methoden zum Verbinden und Trennen eines Services implementiert. Einer *SatelliteSimulation* ist demnach ein ganz konkreter Port der umgesetzten TCP/IP-Verbindung zugeordnet. Über diesen Port werden die Nachrichten an die Services verschickt. Die Initiierung des Verschickens der Nachrichten erfolgt auch durch die *SatelliteSimulation*-Klasse.

Die unterste Ebene des Modells bilden die angemeldeten Services. Die zugehörige Klasse *RegisteredService* beinhaltet Methoden zum Abfragen/Ändern der Konfigurationsdaten der Services und zum Ausgeben der benötigten Daten für die „STATUS“-Nachricht.

In den „STATUS“-Daten können sich auch Werte von anderen Services befinden. So benötigt z.B. eine Anwendung zur Berechnung des Energiehaushaltes die Information, ob sich der Satellit in der Sonne oder im Schatten befindet. Wie dieser Datenaustausch stattfindet wird in 3.1.4 gezeigt.

Neben den genannten hierarchisch angeordneten Modellklassen existiert in der „Simulation“-Schicht zusätzlich die Klasse *ModelStatus*. Diese Klasse stellt die möglichen Zustände dar, welche eine Modellklasse (z.B. *Simulation*) i.A. annehmen kann.

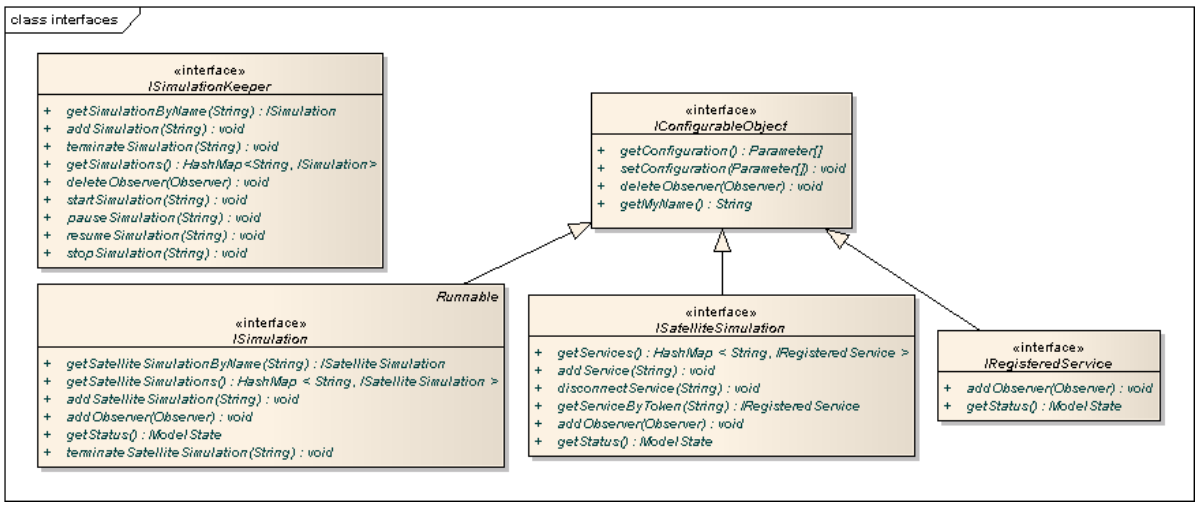


Abbildung 3.3.: Interfaces der „Simulation“-Schicht

Die „Simulation“-Schicht stellt außerdem eine Reihe von Interfaces bereit, welche von anderen Schichten benutzt werden können (Abbildung 3.3).

Es existiert für jede der vier Modellklassen mindestens ein Interface. Diese Interfaces werden von der Schicht „Frontend“ verwendet. Die Schicht fragt die Daten der Modellklassen über diese Interfaces ab und stellt sie dar. Darüber hinaus leitet die „Frontend“-Schicht Benutzerinteraktion über diese Interfaces an die Modellklassen weiter.

Die Interfaces für die Klassen *Simulation*, *SatelliteSimulation* und *RegisteredService* leiten dabei noch von einem allgemeinerem Interface *IConfigurableObject* ab. Dieses Interface wird innerhalb der „Frontend"-Schicht benutzt, um die jeweilige Modellklasse zu konfigurieren. Das Interface der *SimulationKeeper*-Klasse leitet nicht von dem allgemeineren Interface ab, da diese Klasse nicht konfiguriert werden muss.

3.1.2. Verhalten einer Simulation

In dem Unterkapitel 3.1.1 wurde die Klasse *ModelStatus* erwähnt. Diese fasst die Zustände einer Modellklasse wie z.B. *Simulation* zusammen. Die Bedeutungen der einzelnen Zustände sind in der folgenden Tabelle aufgelistet:

Zustand	Bedeutung
LISTED	in der Konfigurationsdatei aufgeführt
INSTANTIATED	Modellklasse und untergeordnete Modellklassen instanziiert
RUNNING	Modell wird gerade simuliert
PAUSED	Simulation des Modells wurde pausiert

Tabelle 3.3.: Zustände der Modellklassen

Mit dem folgenden Aktivitätsdiagramm zur Geschäftslogik einer Simulation wird die Anwendung der Zustände aus Tabelle 3.3 beschrieben.

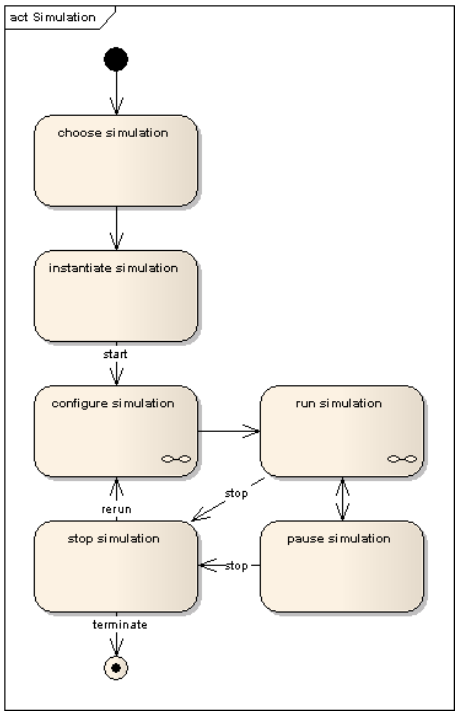


Abbildung 3.4.: Ablauf einer Simulation (business logic)

Um eine Simulation durchzuführen, muss SMASS zunächst gestartet werden. Dadurch werden dem Benutzer alle zur Verfügung stehenden Simulationen angezeigt.

In dem Block *choose simulation* wählt der Benutzer eine bestimmte Simulation aus. Diese Simulationen befinden sich in dem LISTED-Zustand.

Anschließend muss der Benutzer die Simulation erstellen, was durch den Block *instantiate simulation* in Abbildung 3.4 dargestellt wird. Er muss die Simulation von dem LISTED- in den INSTANTIATED-Zustand überführen. Dazu gehört die Auswahl der Satelliten, welche bei der Simulation benutzt werden sollen. Es muss mindestens ein Satellit erstellt werden und ebenfalls in den INSTANTIATED-Zustand gelangen. Eine Satellitensimulation wechselt zu INSTANTIATED, wenn mindestens ein Service angemeldet ist. Die zu benutzenden Services wählt der Benutzer aus einer Liste. Hat der Benutzer mindestens einen Service eines Satelliten gestartet, ist die Simulation instanziiert.

Nach dem Instanziieren einer Simulation kann der Benutzer die ausgewählte Simulation starten. Dabei wird die Simulation zunächst konfiguriert (*configure simulation*). Anschließend wird in dem Block *run Simulation* der erste Berechnungszeitpunkt ermittelt, wodurch sich die Simulation im Status RUNNING befindet. Eine ausführlichere Beschreibung der beiden Blöcke *configure simulation* und *run simulation* folgt in den nächsten Abschnitten.

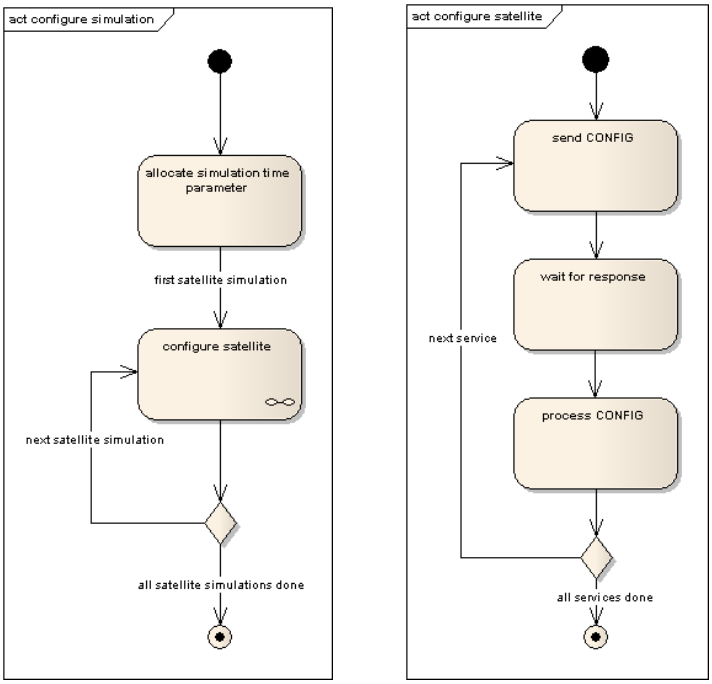
Von einer laufenden Simulation aus hat der Benutzer zwei Einflussmöglichkeiten. Er kann die Simulation pausieren oder stoppen.

Das Pausieren einer Simulation veranlasst diese nach dem Abarbeiten eines bereits angefangenen Simulationsschrittes in den PAUSED-Zustand zu wechseln. Der Benutzer kann im Anschluss an den PAUSED-Zustand die Simulation fortsetzen oder ganz stoppen.

Der Unterschied zwischen dem Pausieren und dem Stoppen einer Simulation liegt darin, dass beim Stoppen die Werte der Simulationszeitsteuerung wieder auf die Anfangswerte zurückgesetzt werden. Durch das Stoppen einer Simulation nimmt diese wieder den Zustand INSTANTIATED an. Das heißt, alle Services bleiben angemeldet und die Simulation kann von neuem gestartet werden. Ist die Stoppzeit einer Simulation erreicht, wechselt sie ebenfalls in den INSTANTIATED-Status.

Die Laufzeitrepräsentation einer Simulation kann von jedem Status aus gelöscht werden, was in Abbildung 3.4 „terminate“ bezeichnet wird. Dies bedeutet, dass alle Services abgemeldet werden und die Instanz der *Simulation*-Klasse aus dem *SimulationKeeper* ausgetragen wird. Eine solche Simulation wechselt wieder in den Status LISTED und muss neu instantiiert werden. Um die Übersichtlichkeit zu wahren ist in Abbildung 3.4 nur eine Variante für das Löschen der Laufzeitrepräsentation einer Simulation dargestellt.

Konfigurierung einer Simulation (*configure simulation*)



(a) Schleife über alle Satellitensimulationen

(b) Schleife über alle Services (*configure Services*)

Abbildung 3.5.: Konfigurierung der Satellitensimulationen

In Abbildung 3.5 ist der Ablauf zur Konfigurierung einer Simulation dargestellt. Mit der ersten Aktivität werden die Simulationsparameter, wie z.B. die Startzeit oder die Schrittweite, aus der Konfigurationsdatei gelesen und in den Speicher geladen. In dem folgenden Block *configure satellite* werden alle Services einer Satellitensimulation mittels Nachrichten konfiguriert, was in Teil (b) der Abbildung 3.5 dargestellt ist. An dieser Stelle sei schon einmal vorweg genommen, dass eine Nachricht zum Konfigurieren eines Services CONFIG heißt (siehe Kapitel 3.2.2).

Zunächst wird eine Konfigurationsnachricht an den ersten Service verschickt. Anschließend wartet die Satellitensimulation auf die Antwort. In der Antwort teilt der Service mit, ob die Konfigurierung erfolgreich war. Diese Antwort wird im nächsten Block *process CONFIG* verarbeitet. Sollte die Konfiguration nicht erfolgreich gewesen sein, wird dies dem Benutzer mitgeteilt.

Nachdem der erste Service konfiguriert wurde, wird überprüft, ob noch weitere Services konfiguriert werden müssen. Wenn ja, beginnt der Prozess von vorne und es wird wieder eine Konfigurationsnachricht verschickt.

Sind alle Services konfiguriert, wird der Block *configure satellite* in Abbildung 3.5(b) verlassen und diese Satellitensimulation ist konfiguriert. Nun wird überprüft, ob weitere

Satellitensimulationen konfiguriert werden müssen. In diesem Fall werden wieder die Aktivitäten aus *configure satellite* durchgeführt. Falls bereits alle Satellitensimulationen konfiguriert sind, wird der Block *configure simulation* aus Abbildung 3.5(a) verlassen.

run Simulation

In diesem Abschnitt sollen die Vorgänge innerhalb des Blockes *run Simulation* aus Abbildung 3.4 noch näher beschrieben werden.

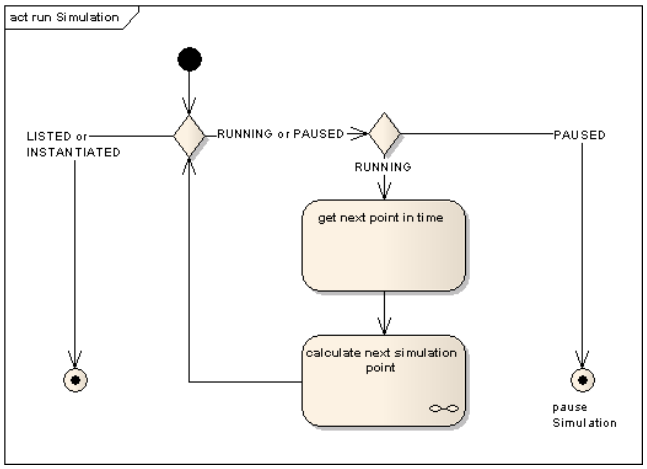


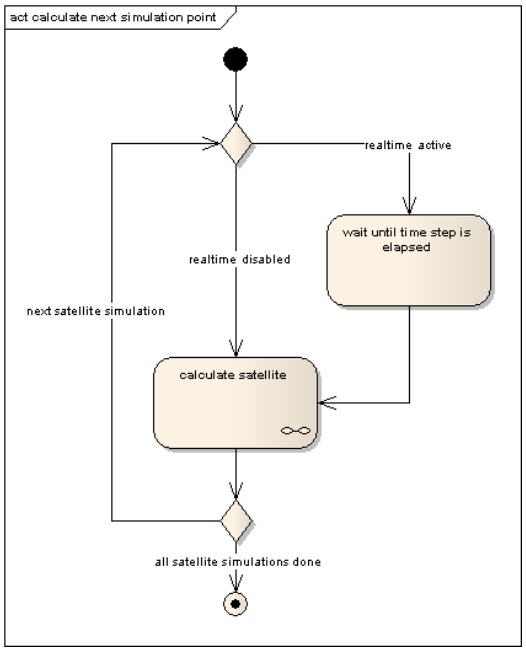
Abbildung 3.6.: Aktivitäten innerhalb von *run simulation*

Als erstes wird in Abbildung 3.6 überprüft, in welchem Zustand sich die Simulation befindet. Ist die Simulation im Zustand LISTED oder INSTANTIATED, hat der Benutzer die Laufzeitumgebung der Simulation gelöscht oder die Konfigurierung der Simulation war nicht erfolgreich. In diesem Fall wird der Block *run simulation* wieder verlassen.

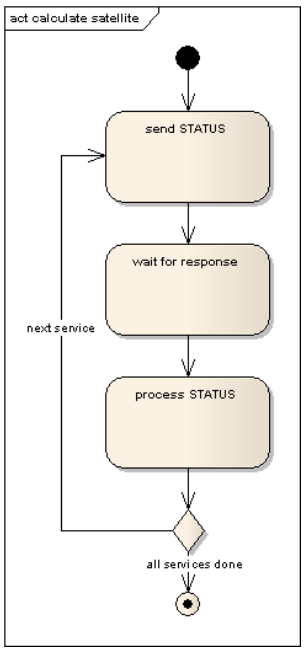
Ist die Simulation ordnungsgemäß konfiguriert worden und noch nicht vom Benutzer beendet, wird überprüft, ob die Simulation im Zustand RUNNING oder PAUSED ist. Im PAUSED-Zustand, wird der Block *run simulation* zum Pausieren der Simulation verlassen.

Nur wenn die Simulation im Zustand RUNNING ist, wird der nächste Simulationszeitpunkt im Block *get next point in time* berechnet. Die Simulationszeitsteuerung wird also zentral von der *Simulation*-Klasse durchgeführt. Sollte in diesem Block festgestellt werden, dass der berechnete Simulationszeitpunkt bereits der Letzte ist, wird der Zustand einer Simulation auf INSTANTIATED gesetzt.

Der berechnete Simulationszeitpunkt wird innerhalb des Blockes *calculate next simulation point* an die Satellitensimulationen übermittelt. Die Aktivitäten innerhalb dieses Blockes werden in der folgenden Abbildung gezeigt.



(a) Schleife über alle Satellitensimulationen



(b) Schleife über alle Services (*calculate services*)

Abbildung 3.7.: Berechnung eines Simulationsschrittes

Die Berechnung eines Simulationsschritts läuft analog zur Konfigurierung ab. Der einzige Unterschied ist die Überprüfung auf die „schwache Echtzeit"-Option, welche in Abbildung 3.7(a) gezeigt ist. Ist die Option aktiviert, wird in dem Block *wait until time step is elapsed* der Zeitstempel der letzten Berechnung abgefragt und mit der aktuellen Zeit verglichen. Ist die Schrittweite noch nicht erreicht, wird solange gewartet, bis die Schrittweite verstrichen ist. Nähere Erläuterungen zur „schwachen Echtzeit" folgen im Kapitel 3.1.3.

Ist die Echtzeitoption deaktiviert oder wurde im Fall der aktivierten Echtzeitoption die verbleibende Zeit gewartet, werden als nächstes die Aktivitäten für das Berechnen des Simulationsschrittes ausgeführt (*calculate satellite*). Diese Aktivitäten sind im Teil (b) der Abbildung 3.7 dargestellt und bis auf den Namen der Nachricht identisch mit den Aktivitäten aus dem Block *configure satellite* aus Abbildung 3.5(b).

An dieser Stelle sei lediglich noch erwähnt, dass durch die Übermittlung des Simulationszeitpunkts an eine Satellitensimulation diese ebenfalls für den Zeitraum der Berechnung eines Simulationsschrittes in den Status RUNNING wechselt. Die Berechnung eines Simulationsschrittes innerhalb der Satellitensimulation ist mit dem Verlassen des Blockes *calculate satellite* beendet. Dies hat zur Folge, dass die Satellitensimulation bis zum nächsten Simulationsschritt den Status PAUSED inne hat.

Nachdem eine Satellitensimulation den Simulationsschritt berechnet hat, wird nun wieder überprüft, ob noch weitere Satelliten mit einbezogen werden müssen. Gegebenenfalls beginnt der Ablauf von vorne mit der Überprüfung der Option zur „schwachen

Echtzeit“.

Sind alle Satellitensimulationen für diesen Simulationsschritt beendet, wird der Block *calculate next simulation point* aus Abbildung 3.6 verlassen und es wird wieder der Status der Simulation überprüft.

3.1.3. Laufzeitverhalten - „schwache Echtzeit“

In diesem Abschnitt soll kurz auf die zuvor bereits erwähnte „schwache Echtzeit“ eingegangen werden.

Es soll mit SMASS die Möglichkeit gegeben werden, dass anstelle von den Services auch Hardwaremodule eingebunden werden können (UR 11.0). Hardwaremodule haben aber die Eigenschaft, dass sie gewisse Latenzzeiten haben. Um sie dennoch in SMASS einzubeziehen, ist es notwendig, dass eine Mindestzeit zwischen zwei Simulationsschritten gewährleistet werden kann. Dies wurde mit der Option für die „schwache Echtzeit“ umgesetzt.

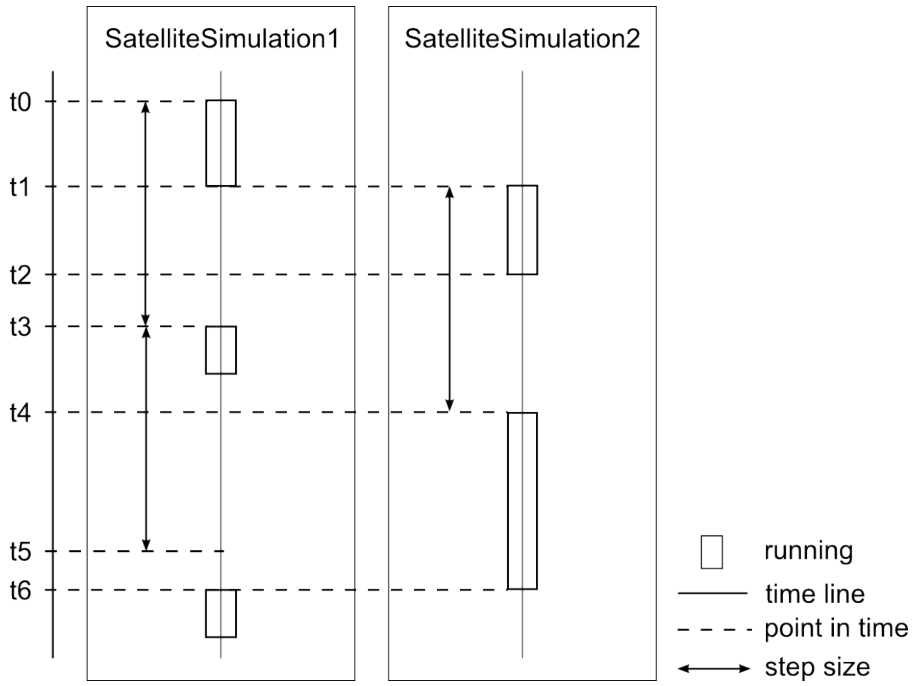


Abbildung 3.8.: „schwache Echtzeit“

In Abbildung 3.8 ist ein Beispielszenario für die Umsetzung mit zwei Satellitensimulationen zu sehen. Die erste Satellitensimulation erhält zum Zeitpunkt $t0$ den ersten Simulationszeitpunkt zur Berechnung. Zum Zeitpunkt $t1$ ist die Berechnung innerhalb der ersten Satellitensimulation abgeschlossen und die zweite Satellitensimulation bekommt den Simulationszeitpunkt. Diese benötigt bis $t2$, um die Berechnung abzuschließen.

Nun wird innerhalb der *Simulation*-Klasse, welche die Instanzen von *SatelliteSimulation* verwaltet, der Zeitstempel für die erste Satellitensimulation ausgewertet. Da $t2 - t0 \leq step\ size$ ist, wird zunächst noch gewartet, bis die Schrittweite zum Zeitpunkt $t3$ erreicht ist. Dann wird der zweite Simulationszeitpunkt der ersten Satellitensimulation übermittelt.

Da für diesen Simulationszeitpunkt die Berechnung innerhalb der ersten Satellitensimulation sehr viel kürzer ist, ist diese Satellitensimulation fertig, bevor die Schrittweite auch für die zweite Satellitensimulation erreicht ist. Demzufolge wartet die *Simulation*-Klasse nun wieder bis $t4$, bevor sie den zweiten Simulationszeitpunkt an die zweite Satellitensimulation übermittelt.

Die zweite Satellitensimulation benötigt im Gegensatz zur ersten Satellitensimulation nun aber sehr viel länger als im ersten Schritt. Dadurch ist die Schrittweite für die erste Satellitensimulation zum Zeitpunkt $t6$ bereits überschritten. Es wird also sofort der nächste Simulationszeitpunkt übermittelt. Spätestens ab diesem Punkt simuliert das System damit nicht mehr in Echtzeit, weshalb hier die Formulierung „schwache Echtzeit“ bzw. Mindestzeit verwendet wird.

Das Vermeiden solcher Verzögerungen ist Aufgabe des Benutzers. Er hat dafür zwei Möglichkeiten. Zum einen kann er einfach die Schrittweite erhöhen, sodass über den gesamten Simulationsverlauf die Intervalle eingehalten werden. Zum anderen könnte er versuchen, die Berechnung zu beschleunigen, indem er die Hardware des bremsenden Elementes optimiert.

Die Option zur Simulierung in „schwacher Echtzeit“ ist auch für den universitären Betrieb sinnvoll. In der Realität kommen die Daten von den Satelliten auch nur in bestimmten Intervallen zur Erde. Dies könnte somit auch simuliert werden, sofern der Zeitschritt groß genug gewählt wurde und es nicht zu Verzögerungen wie im Beispiel aus Abbildung 3.8 kommt.

3.1.4. Datenaustausch zwischen Services

Der Datenaustausch zwischen den Services ermöglicht die Berücksichtigung der gegenseitigen Einflüsse von Satellitensubsystemen. Damit die Services immer die aktuellesten Daten erhalten, wurde eine Simulationsreihenfolge für die Services eingeführt, welche in der Konfigurationsdatei (siehe 3.4.2) hinterlegt wird. Durch diese Reihenfolge kann bei einseitiger Abhängigkeit der Services gewährleistet werden, dass der anbietende Service den aktuellen Simulationsschritt bereits berechnet hat bevor der nutzende Service diesen berechnet. Ein Beispiel für einseitige Abhängigkeiten zwischen Services wurde in dem Kapitel 3.1.1 angesprochen. Es handelt sich um die Orbit- und Energiehaushaltberechnung. Letztere benötigt die Information, ob sich der Satellit im Erdschatten befindet und ist damit vom Orbitservice abhängig. Die Festlegung einer Simulationsreihenfolge ist in diesem Fall sehr einfach. Zunächst muss das Orbitmodell die Daten berechnen und anschließend kann das Modell für den Energiehaushalt diese Daten benutzen. Dieses Beispiel ist in der folgenden Abbildung dargestellt:

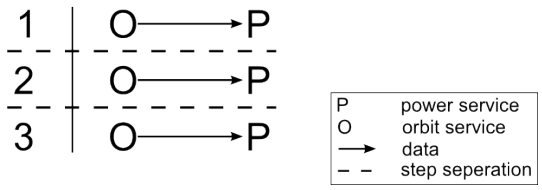


Abbildung 3.9.: Datenaustausch bei einseitiger Abhängigkeit

Da sich einige Services auch gegenseitig beeinflussen können, wie z.B. die Energieversorgung und der Thermalhaushalt, können nicht beide Services die Daten aus dem aktuellen Zeitschritt benutzen. In diesem Fall bestimmt die Simulationsreihenfolge die Aktualität der Daten. Wird der Thermalhaushalt vor der Energieversorgung berechnet, würde der Datenfluss zwischen den beiden Services wie folgt aussehen:

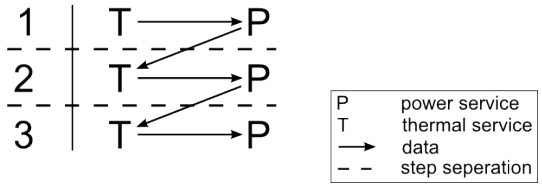


Abbildung 3.10.: Datenaustausch bei gegenseitiger Abhängigkeit

In diesem Beispiel berechnet zunächst der Thermalservice die Daten. Diese Daten benutzt der Service zum Energiehaushalt (P). Im zweiten Simulationsschritt benötigt der Thermalservice dann die Daten des Energiehaushaltes. Da dieser Service aber erst nach dem Thermalservice die aktuellen Daten berechnet, benutzt der Thermalservice die Daten vom ersten Simulationsschritt des Services zur Energieversorgung. Der Thermalservice bezieht seine Daten also immer vom vorherigen Simulationsschritt, während der Service zur Berechnung des Energiehaushaltes immer auf die Daten vom aktuellen Simulationsschritt zurückgreifen kann. Würde die Simulationsreihenfolge vertauscht werden, würde hingegen das Thermalmodell immer auf die Daten des aktuellen Simulationsschrittes vom Service zum Energiehaushalt zurückgreifen können.

Grundsätzlich werden immer die aktuellsten Daten beim Austausch zwischen den Services verwendet. Ob diese Daten aber vom aktuellen Simulationsschritt, oder vom vorherigen Simulationsschritt sind, hängt von der Simulationsreihenfolge ab. Die Festlegung der Simulationsreihenfolge der Services erfolgt in der Konfigurationsdatei (siehe 3.4.2), da nur der Benutzer entscheiden kann, welcher Service besonders genaue Daten benötigt.

3.2. „Utilities“

Die „Utilities“-Schicht beinhaltet mehrere Hilfspakete. Diese stehen allgemein allen Klassen zur Verfügung. Zu den Hilfspaketen zählen *communication* für die Kommunikation, *math* für mathematische Ausdrücke und *xml* für das Be- und Verarbeiten einer XML-Datei.

Es wären an dieser Stelle aber auch weitere Pakete denkbar. So könnte z.B. ein Paket *constants* für Naturkonstanten oder ein Paket *error* für allgemeine Fehlermeldungen angelegt werden.

In den folgenden Abschnitten soll nun der wichtigste Teil der „Utilities“-Schicht, die Kommunikation, beschrieben werden. In der Einleitung des Kapitels 3 wurde bereits erwähnt, dass die Kommunikation aus dem Bereich der Services ausgelagert wurde und sich innerhalb des Paketes *communication* befindet (siehe Abbildung 3.1). Es wurde auch schon herausgestellt, dass die Kommunikation über eine Netzwerkverbindung realisert wird, über welche schließlich die Nachrichten verschickt werden. Die Verarbeitung dieser Nachrichten ist dabei server- und clientspezifisch.

Somit ergeben sich drei Funktionaltitäten, die von dem Kommunikationspaket umgesetzt werden.

- Aufbau einer Netzwerkverbindung zwischen Client und Server
- Definition eines Nachrichtenformates
- Verarbeitung der Nachrichten

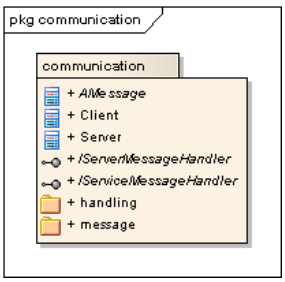


Abbildung 3.11.: Struktur des *communication*-Pakets

Diese drei Funktionalitäten finden sich auch in der Struktur des *communication*-Pakets wieder, welche in Abbildung 3.11 gezeigt ist.

3.2.1. Netzwerkverbindung

Der Aufbau einer Netzwerkverbindung und die Kommunikation über diese Netzwerkverbindung wird von den Klassen *Client* und *Server* übernommen. Diese Klassen stellen

Methoden zum Verschicken und Empfangen einer Nachricht zur Verfügung. Die *Server*-Klasse bietet darüber hinaus noch die Möglichkeiten den Socket eines kontaktierenden Klienten auszugeben und den geöffneten Kommunikationskanal der Netzwerkverbindung zu schließen.

3.2.2. Nachrichtenformat

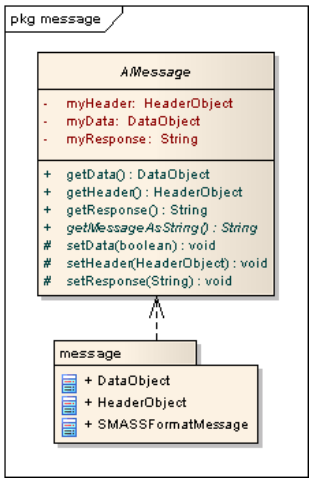


Abbildung 3.12.: Nachrichtenklassen

Die Definition des Nachrichtenformats erfolgt über die Klassen in dem Unterpaket *message* und die zugehörige Schnittstelle *AMessage* (Abbildung 3.12). Von der abstrakten Klasse *AMessage* leiten alle konkreten Nachrichtenklassen ab, die sich in dem *message*-Paket befinden. Somit ist die Einführung weiterer Nachrichtentypen in Zukunft gewährleistet. Es wäre hier z.B. die Einführung einer XML-Nachricht denkbar.

In der Schnittstelle *AMessage* ist bereits festgelegt, dass eine Nachricht aus einem Header (*HeaderObject*) und einem Datenteil besteht. Der Datenteil kann dabei als *DataObject* oder als *String* in Form eines Textteils gestaltet werden, welcher nur für Bestätigungsmeldungen genutzt wird. Die Unterteilung in Header- und Datenteil hat den Hintergrund, dass der Header grundsätzlich bei allen Nachrichten gleich aufgebaut ist, während der Datenteil nicht immer vorhanden ist und auch in der Dimension, also der Anzahl der Elemente, variiert. Außerdem bietet diese Trennung den Vorteil, dass ein Header auch umgestaltet werden kann. Es können ohne Probleme die Reihenfolge von den einzelnen Elementen des Headers mittels definierter Konstanten geändert werden oder einzelne Elemente dem Header hinzugefügt werden. An diesen beiden Beispielen kann die durch die Objektorientierung gesteigerte Modularität und die damit erreichte Flexibilität von SMASS gut nachvollzogen werden.

SMASSFormatMessage

Die einzige bisher implementierte konkrete Nachrichtenklasse ist *SMASSFormatMessage*. Dies ist ein historisch gewachsenes ASCII basiertes Nachrichtenformat und hat sich im Vergleich zur Arbeit von Pfeiff [Pfe07] nur an einer Stelle verändert. Die ursprünglich vorgesehene Prüfsumme ist im Rahmen dieser Arbeit weggefallen. Sie hat keinen Mehrnutzen an Sicherheit gebracht, da bereits durch die Transportschicht der TCP/IP-Verbindung Prüfsummen für die Datenpakete integriert werden. Eine Nachricht hat demzufolge jetzt folgende Struktur:

source	target	message ID	message Name	data 0	...	data n

Tabelle 3.4.: Struktur *SMASSFormatMessage*

Die ersten vier Felder gehören dabei zum Header und sind innerhalb der *HeaderObject*-Klasse aus Abbildung 3.12 definiert. Diese Felder sind jeweils sechs Zeichen lang und durch ein Komma getrennt. Die dem Header folgenden Felder gehören zum Datenteil. Diese können je nach Art der Nachricht von beliebiger Anzahl und beliebig groß sein. Die Felder des Datenteils werden ebenfalls durch ein Komma getrennt. Sollten Nachkommastellen eines Parameters benutzt werden, so ist dafür ein Punkt zu verwenden.

Die Felder *source* und *target* beinhalten sogenannte Tokens, welche aus drei Buchstaben für den Server/Service und drei Zahlen für die Versionsnummer bestehen. Folgende Tokens werden bereits benutzt:

Token	**Beschreibung**
WHOSIT	target Token für einen bislang unbekannten Service
SRV001	Server Version 1
ORB001	Orbitmodell Version 1
PWR000	Powermodell Version 0

Tabelle 3.5.: benutzte Token

Das Feld zur Nachrichtennummer (*message ID*) beinhaltet eine fortlaufende sechsstellige natürliche Zahl. Sollte die Zahl 999999 erreicht werden, beginnt sie wieder bei eins.

Im letzten Feld des Headers sind die Nachrichtennamen untergebracht. Diese Namen wurden im Gegensatz zur Arbeit von Pfeiff [Pfe07] limitiert. Nach der Analyse von SMASS stellte sich heraus, dass die komplette Funktionalität der ursprünglich neun vorgesehenen Nachrichten auf vier Nachrichten reduziert werden kann, wenn die Kommunikation, die Datenhaltung und die Darstellung von Daten aus den Services ausgelagert wird. Folgende Varianten von Nachrichten sind noch möglich:

Name	Beschreibung
WELCOM	Willkommensnachricht
CONFIG	Nachricht zum Konfigurieren der Services
GETCNF	Nachricht zum Abfragen der Konfiguration
STATUS	Nachricht zum Berechnen eines Zeitschrittes

Tabelle 3.6.: Nachrichtennamen

In Abhängigkeit von der Nachrichtenart und von der Quelle der Nachricht sieht der Datenteil wie folgt aus:

Name	Server	Client
WELCOM	Welcome to the Satellite Simulation Server of the TU-Berlin!	leerer *String*
CONFIG	*DataObject* mit Konfigurationsdaten des Services	*String* mit Bestätigung (OK oder Fehlermeldung)
GETCNF	leerer *String*	*DataObject* mit aktueller Konfiguration des Services
STATUS	*DataObject* mit Inputdaten für Berechnung	*DataObject* mit Ergebnissen der Berechnung

Tabelle 3.7.: Datenteil einer Nachricht

3.2.3. Verarbeitung der Nachrichten

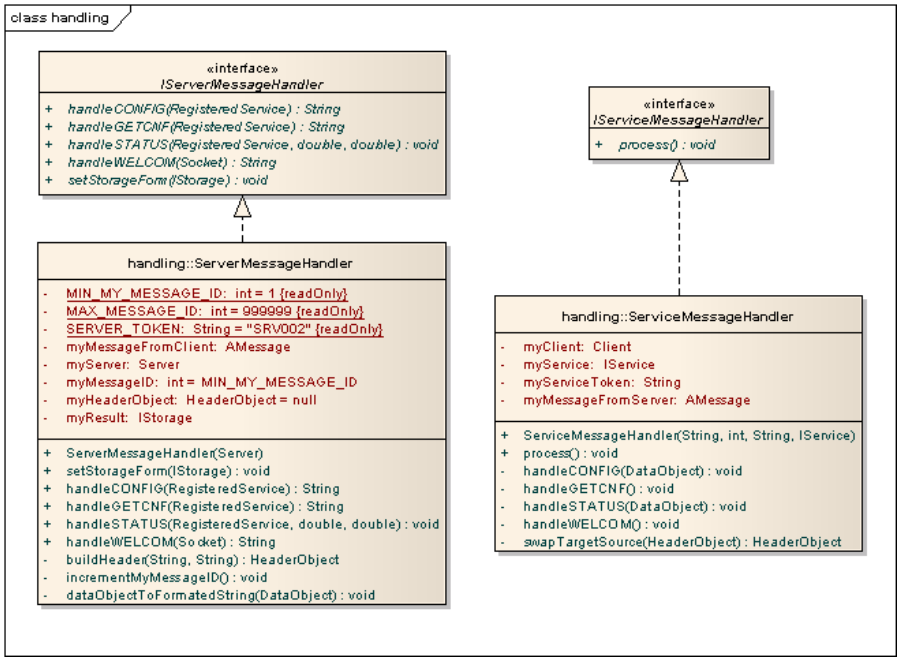


Abbildung 3.13.: *ServerMessageHandler* und *ServiceMessageHandler*

Die letzte der drei Funktionalitäten aus dem *communication* Paket ist die Verarbeitung einer Nachricht. Diese ist bei den Services der Clients etwas anders gelagert als beim Server. Demzufolge wurde auch hier wieder getrennt und für jeden Bereich eine Klasse angelegt. Beide Klassen sind in dem Unterpaket *handling* zusammengefasst. Sowohl *ServerMessageHandler* als auch *ServiceMessageHandler* haben jeweils pro Nachrichtenart eine Methode zur Bearbeitung (siehe Abbildung 3.13). Allerdings sehen die Parameter der Methoden anders aus, weshalb hier keine abstrakte Elternklasse eingeführt wurde.

Die *ServerMessageHandler*-Klasse ist der aktive Part in der Kommunikation. Sie setzt die jeweiligen Nachrichten in den *handle*-Methoden zusammen und verschickt diese dann über die *Server*-Instanz *myServer* an die Services. Darüber hinaus speichert die Klasse auch die Ergebnisse eines jeden Simulationsschrittes innerhalb der *handleSTATUS*-Methode.

Die Nachrichtenverarbeitung auf Seiten der Services ist sehr viel einfacher, da die Services der passive Part sind. Sie warten nur auf eingehende Nachrichten mit Hilfe der *Client*-Instanz *myClient* und reagieren dann darauf. Dabei wird nur die Quelle und das Ziel einer Nachricht im Header getauscht und die jeweilige Aktion in den Services ausgelöst.

Beide *MessageHandler*-Klassen implementieren jeweils ein Interface, welches die Benutzung dieser Klassen außerhalb des *handling*-Paketes ermöglicht. Es wurde also auch im kleinen Maßstab versucht, die Funktionalitäten bestmöglich zu kapseln und damit unabhängiger von anderen Implementierungen zu machen.

Zum Schluss soll nun anhand der Abbildungen 3.14 und 3.15 die Abläufe der Kommunikation auf der Seite des Servers und des Clients erläutert werden.

Server

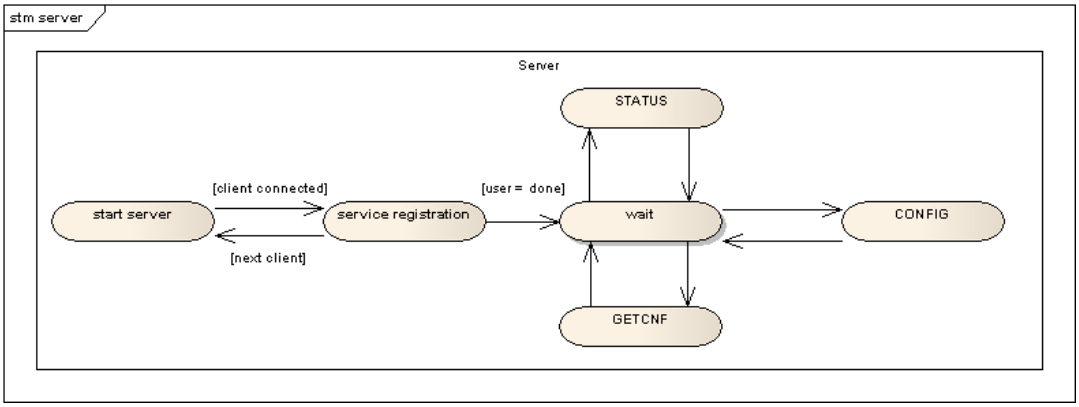


Abbildung 3.14.: Zustände des Servers

Das grundsätzliche Kommunikationsschema auf Seiten des Servers für eine Satellitensimulation ist in der Abbildung 3.14 dargestellt. Zunächst wird der Server gestartet. Nach dem Starten wartet der Server auf die Clients und baut eine Verbindung zu ihnen auf. Dazu legt der Server einen Server-Socket an dem vorher in der Konfigurationsdatei festgelegten Port an. Dann wartet der Server, bis ein Client ihn kontaktiert. Wenn der Server die Anfrage der Clients akzeptiert hat, legt er an einem freien Port einen weiteren Socket an, über den dann die Kommunikation abgewickelt wird. Hierbei werden auch die Ein- und Ausgabeströme angelegt.

Ist eine Verbindung zu einem Client aufgebaut, wechselt der Server in den nächsten Zustand zur Registrierung des Services dieses Clients. Um einen Service zu registrieren, verschickt der Server die WELCOM Nachricht mit dem *target*-Token WHOSIT an den bislang unbekannten Service. Danach wartet er auf die Antwort, in der als *source*-Token im Header der WELCOM-Nachricht der Service-Token steht. Dieser wird dann in der zugehörigen Instanz von *RegisteredService* gespeichert. Somit ist dieser Service registriert.

Signalisiert der Benutzer anschließend, dass alle von ihm für diese Satellitensimulation vorgesehenen Services registriert sind, wechselt der Zustand des Servers von der Serviceregistrierung zu einem Wartezustand. Aus diesem Zustand wird je nach Benutzerwunsch in den Zustand *STATUS* zum Berechnen eines Simulationsschrittes, *CONFIG* zum Konfigurieren eines Services oder *GETCNF* zum Abfragen der aktuellen Konfiguration eines Services gewechselt.

Client

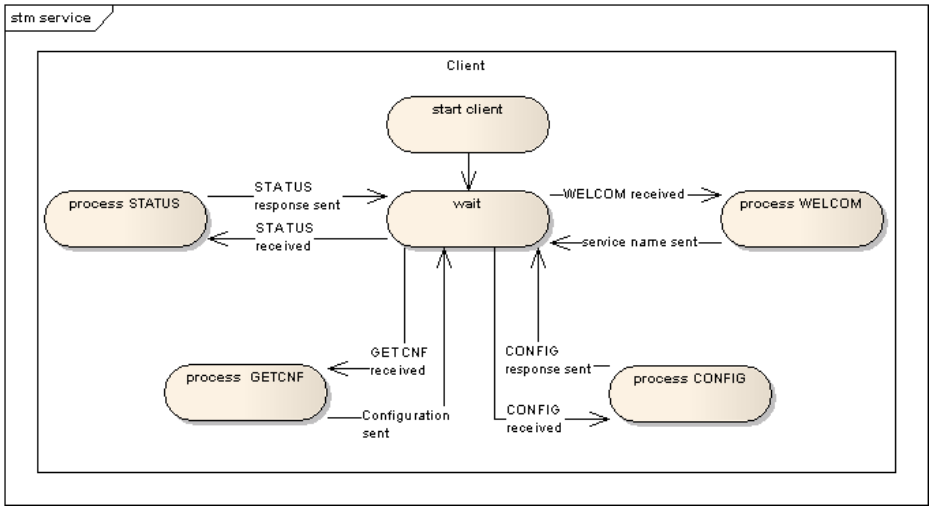


Abbildung 3.15.: Zustände eines Clients

In Abbildung 3.15 ist die Kommunikation aus der Sicht eines Clients gezeigt. Zunächst wird ein Client gestartet und baut damit die Verbindung zum Server auf. Dazu versucht der Client über die IP-Adresse und den Port den Server zu kontaktieren und erzeugt einen Socket. Hat der Server die Anfrage akzeptiert, werden auch hier die Ein- und Ausgabeströme geöffnet. Damit ist dann die Netzwerkverbindung aufgebaut.

Von diesem Zustand wechselt der Client in einen *wait*-Zustand, indem er auf einkommende Nachrichten wartet. Wenn eine Nachricht eingetroffen ist, wird diese Nachricht mit der jeweils passenden Methode verarbeitet. Aus diesen vier gezeigten Verarbeitungszuständen (*process WELCOM*, *process CONFIG*, *process STATUS* und *process GETCNF*) wechselt der Service, mittels des Versendens der Antwort auf die Anfrage, wieder in den *wait*-Zustand.

3.2.4. Fazit

Abschließend stellt sich heraus, das mit dem Konzept für die Kommunikation erneut ein großer Teil der Anforderungen (UR) aus Kapitel 2 umgesetzt wird. Als wichtigster Punkt wird die Kommunikation zwischen Server und Service standardisiert (UR 15.0).

Darüber hinaus wird mit den gezeigten Klassen das reine Senden/Empfangen über TCP/IP unabhängig davon gemacht, was exakt gesendet wird. Durch die Ausgliederung der Behandlung der Nachrichten und der Einführung einer abstrakten Datenklasse für Nachrichten (*AMessage*), ist die Behandlung von Nachrichten ebenfalls unabhängig von dem konkreten Nachrichtenformat. Die Einführung der abstrakten Klasse *AMessage* bewirkt außerdem eine Trennung von Nachrichteninfrastruktur in Form des Headers und den Daten.

Durch die Verwendung des TCP/IP-Standards und der Client/Server Architektur ist auch die Kommunikation zwischen mehreren Rechnern möglich (UR 8.0). Somit können die Berechnungen der Services auch auf andere Computer ausgelagert werden. Der TCP/IP-Standard ist außerdem sehr weit verbreitet. Er wird von nahezu jedem Betriebssystem auf Softwareebene unterstützt und es gibt unzählige Hardwarelösungen, welche diesen Standard umsetzen. Somit trägt die Verwendung von TCP/IP auch zur Software- und Hardwareunabhängigkeit bei (UR 9.0 und UR 10.0).

Letztendlich wurde auch in dieser Schicht durch den hohen Grad der Modularität die Wart- und Erweiterbarkeit sichergestellt. Dies wurde bereits kurz anhand des Beispiels des Headers einer Nachricht erwähnt. Auch im Bereich des Nachrichtenformates ist eine Erweiterung denkbar. Am Anfang des Kapitels 3.2.2 wurde bereits von XML-Nachrichten gesprochen. Im Folgenden ist ein Beispiel gezeigt, wie eine solche XML-Nachricht aussehen könnte.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <xmlMessage>
3         <header source="SRV001" target="ORB001" messageID="2" messageName="STATUS"/>
4         <data>
5                 <parameter name="time" unit="JD" value="2454655.5"/>
6         </data>
7 </xmlMessage>
```

Dieses Beispiel einer STATUS-Nachricht zeigt, dass mit einer XML-Nachricht in Zukunft die Nachrichten noch lesbarer gemacht werden können.

Der Datenteil, welcher bei der *SMASSFormatMessage* eine Zahlenkolonne ist, bekommt innerhalb der XML-Nachricht noch weitere Attribute. So besteht der Datenteil in Zeile vier bis sechs aus Parametern. Diese Parameter haben neben dem eigentlichen Wert zusätzlich einen Namen und eine Einheit.

3.3. „Services“

Die Schicht „Services“ wurde eingeführt, um diese von den übrigen Funktionalitäten, wie z.B. der Kommunikation zu trennen. Die Services sind eigenständige Applikationen, welche Teile des SMASS-Frameworks benutzen. Dies sind zum einen Klassen für die Kommunikation aus der „Utilities“-Schicht, zum anderen sind das zwei Schnittstellen für die Services, welche die Kompatibilität zu dem Framework gewährleisten.

Die benötigten Klassen werden in der Bibliothek *Service.jar* zusammengefasst, welche der Entwickler einer neuen Anwendung in sein Projekt einbinden muss. Seine Aufgabe ist es nur noch, die beiden Schnittstellen mit eigenen Klassen zu realisieren.

3.3.1. Serviceschnittstellen

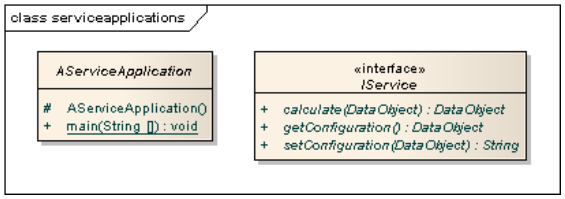


Abbildung 3.16.: Schnittstellen für die Services

Die Schnittstellen für die Services befinden sich im Paket *serviceapplications* und spiegeln die beiden identifizierten Funktionalitäten im Bereich der Services wieder.

Die erste Funktionalität ist das Starten eines Services. Die abstrakte Klasse *AServiceApplication* stellt die Basisklasse für den Einstiegspunkt eines Services dar. Außerdem sind hier ausführliche Kommentare und eine auskommentierte Vorlage für die konkrete Serviceklasse, welche von *AServiceApplication* ableitet, hinterlegt. Diese Vorlage soll den zukünftigen Serviceentwicklern als Hilfestellung dienen.

Als zweite Funktionalität wurde das Anbieten von Methoden identifiziert, die zum Reagieren auf Nachrichten benötigt werden. Diese Methoden sind innerhalb des Interfaces *IService* aus Abbildung 3.16 zusammengefasst. Sie ergaben sich aus der Analyse der bestehenden Berechnungsmodule. Es wurde untersucht, welche Methoden tatsächlich für andere Schichten von Belang sind. Auf folgende Methoden eines Services muss demnach von außen zugegriffen werden können:

Name	**Aufgabe**
calculate	Simulationsschritt berechnen
getConfiguration	aktuelle Konfiguration zurückgeben
setConfiguration	Konfiguration setzen

Tabelle 3.8.: Methoden des Interfaces *IService*

Diese drei Methoden sind auch diejenigen, welche der Entwickler eines Services implementieren muss. Sie werden vom *ServiceMessageHandler* genutzt, um nach der Auswertung des Nachrichtentyps die jeweils angebrachte Methode des Services aufzurufen. Im Falle einer STATUS-Nachricht würde der *ServiceMessageHandler* z.B. die *calculate*-Methode des Services aufrufen.

3.3.2. Konkrete Serviceklassen am Beispiel von ORB001

Die konkreten Implementierungen der Services sind in separaten Paketen untergebracht. Als Bezeichnung wurde eine Kombination aus dem Service-Token und dem englischen Wort „implementation“ gewählt. Ein Paket z.B. für den Service zur Berechnung des Orbits heißt demnach laut der in Tabelle 3.1 festgelegten Konventionen z.B. *ORB001-implementation*.

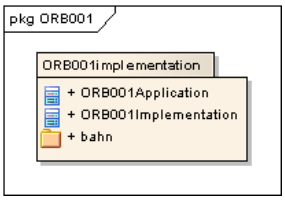


Abbildung 3.17.: Struktur eines Services

In Abbildung 3.17 ist dieses Paket für den Orbitservice dargestellt. Innerhalb eines Servicepakets befinden sich zwei Klassen, deren Bezeichnung ebenfalls mit dem Service-Token beginnt. Das eine ist die *<Service-Token>Application*-Klasse, welche von der abstrakte Klasse *AServiceApplication* ableitet. Das andere ist die Klasse *<Service-Token>Implementation*, welche das Interface *IService* realisiert. Diese beiden Klassen stellen das Minimum an Code dar, was ein Serviceentwickler schreiben muss.

Es können sich je nach Service natürlich auch noch weitere Klassen und Pakete innerhalb von *<Service-Token>implementation* befinden. Dies sind dann aber Klassen und Pakete, welche für die Umsetzung des jeweiligen Services zuständig sind. Bei der Orbitapplikation aus Abbildung 3.17 ist dies auch der Fall. Dort ist ein weiteres Unterpaket namens *bahn* gezeigt. Dieses Paket beinhaltet die an das Framework angepassten Klassen aus dem Bahnmodell, welches von Torsten Giese entwickelt wurde ([Gie05]).

In der folgenden Abbildung wird die Schnittstelle und deren Umsetzung anhand des Services zur Orbitberechnung gezeigt.

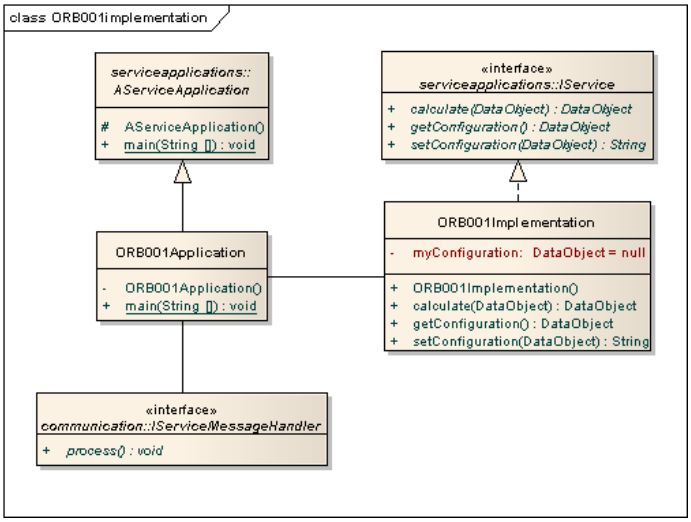


Abbildung 3.18.: Realisierung der Schnittstellen
(Teil der Attribute in *ORB001Implementation* ausgeblendet)

Das zugehörige Paket *ORB001implementation* beinhaltet unter anderem die zwei bereits angesprochenen Klassen *ORB001Implementation* und *ORB001Application*. Die drei weiteren Klassen aus Abbildung 3.18 sind Teil der Bibliothek *Service.jar*, die von jedem Service eingebunden werden muss.

Die *Implementation*-Klasse ist zuständig für die Realisierung des Interfaces *IService* aus dem Schnittstellenpaket *serviceapplications*. Dadurch wird ein Service zu einer Dienstleistung, welche die in Tabelle 3.8 aufgeführten Methoden zur Verfügung stellt. Die Funktionalität des Services wird innerhalb der Methode *calculate* implementiert. Im Beispiel besteht die Funktionalität aus der Berechnung des Orbits. Die anderen beiden Methoden sind die Umsetzung der Konfigurierung des Services (*setConfiguration(DataObject)*) und des Abfragens der aktuellen Konfiguration (*getConfiguration()*).

ORB001Application leitet von der abstrakten Klasse *AServiceApplication* ab. In dieser abstrakten Klasse ist eine *main*-Methode vorgesehen, welche für das Starten des jeweiligen Services zuständig ist. Dadurch wird festgelegt, dass jeder Service mindestens eine Klasse haben muss, welche eine solche *main*-Methode beinhaltet.

Innerhalb der *main*-Methode wird eine Instanz von *ORB001Implementation* und *ServiceMessageHandler* angelegt. Letztere ist allerdings vom Typ *IServiceMessageHandler*, da hier ein Interface für die Kapselung vorgesehen wurde (siehe 3.2.3).

Die Instanz der Klasse *ORB001Implementation* wird dem Konstruktor von *ServiceMessageHandler* übergeben. Dieser erwartet ein Objekt vom Typ *IService*. Durch die Übergabe des Objektes ist der *ServiceMessageHandler* in der Lage, auf die eingehenden Nachrichten zu reagieren und die passende Methode aus Tabelle 3.8 aufzurufen. Die Übergabe des Objektes ist an dieser Stelle auch wieder einer der am Anfang des Kapitels erwähnten Vorteile der Schichten-/Paketstruktur. Durch die Übergabe wird die Benutzbarkeit zwischen der „Utilities“- und der „Services“-Schicht gewährleistet.

3.3.3. Bibliothek für die Services

Die Bibliothek für die Services wurde für die Programmiersprache Java bereits erstellt (*Service.jar*). Um auch Services in anderen Programmiersprachen zu integrieren, muss die Bibliothek für diese Programmiersprachen noch erstellt werden. Die in Tabelle 3.9 aufgeführten Dateien müssen in die jeweilige Programmiersprache übersetzt und in der Bibliotehk zusammengefasst werden.

Ordner	Datei	Beschreibung
services|serviceapplications	*AServiceApplication*	Schnittstelle für den Eintrittspunkt in die Serviceapplikation
	IService	Schnittstelle für die Methoden eines Services
utilities|communication	*AMessage*	Schnittstelle für eine Nachricht
	Client	Klasse für den Aufbau der TCP/IP-Verbindung und das Verschicken/-senden von Daten über TCP/IP
	IServiceMessageHandler	Schnittstelle für den *ServiceMessageHandler*
utilities|communication|handling	*ServiceMessageHandler*	Klasse zum Verarbeiten der Nachrichten für die Services
utilities|communication|message	*DataObject*	Klasse für den Datenteil einer Nachricht
	HeaderObject	Klasse für den Header einer Nachricht
	SMASSFormatMessage	Klasse für eine konkrete Nachricht im SMASS-Format

Tabelle 3.9.: Dateien der Bibliothek für die Services

3.3.4. Fazit

Jeder Service bietet eine Funktionalität in Form eines Umweltmodells (z.B. Orbit) oder eines Satellitensubsystems (z.B. Energieversorgung) an. Die Services an sich sind eigenständige Dienstleistungsanwendungen. Dadurch wird SMASS zu einem verteilten System. Jeder Service kann auf eine andere Hardware ausgelagert werden. Durch die Verteilung der Services ist dieses System sehr gut skalierbar, was eine optimale Performance erlaubt. Besonders wichtig ist dieser Fakt bei einer Simulation mit der Berücksichtigung der „schwachen Echtzeit“-Option.

Die eingeführte Schnittstelle gewährleistet die Benutzung der Services unabhängig davon, wie die Services konkret implementiert sind. Die konkreten Services auf der anderen

Seite müssen nur die Schnittstelle realisieren. Damit ist also nicht nur SMASS unabhängig von den Implementierungen der Services, sondern auch umgedreht, die Services sind unabhängig von den Implementierungen des Frameworks von SMASS.

Außerdem hat dieses Kapitel gezeigt, dass Services durch die Kapselung von der Kommunikation, der Datenverarbeitung und dem Frontend getrennt sind. Ein Entwickler muss sich nur noch Gedanken machen, wie er die drei Methoden aus dem Interface *IService* umsetzt und eine Klasse für den Eintrittspunkt in die Service-Applikation bereitstellt. Die Entwicklung von neuen Services wurde dadurch erheblich vereinfacht.

Als weitere Funktionalität für Services sind auch verschiedene Ausgabeservices denkbar. Diese sollten als Token „OUT“ in Verbindung mit einer eindeutigen Zahlenkombination bestehend aus drei Ziffern benutzen. In deren *calculate*-Methode würden diese Services nichts direkt berechnen, sondern die Ansicht mit den nächsten Simulationswerten aktualisieren. Sie würden also nur Daten von anderen Services benutzen und keine Daten selber generieren. Ansonsten werden sie genauso behandelt, wie jeder andere Service auch.

3.4. „Data Handling“

Die Schicht „Data Handling“ setzt die Funktionalitäten Laden/Speichern der Konfigurationsdaten und Speichern der Simulationsergebnisse um. Die Schicht ist somit das Bindeglied von der Persistenzschicht zum SMASS-Framework. Für das Laden/Speichern der Konfiguration von Simulationen wurde das Paket *configuration* und für die Speicherung der Simulationsergebnisse das Paket *storage* angelegt. Dadurch ist auch der Einsatz jeweils unterschiedlicher Persistenzen für die Konfigurationsdaten und Ergebnisse möglich.

Ziel der Einführung von „Data Handling“ ist die Kapselung der Datenverarbeitung. So wird gewährleistet, dass dieser Bereich einfach ausgetauscht werden kann.

3.4.1. Laden/Speichern der Konfigurationsdaten

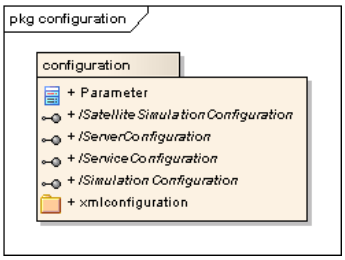


Abbildung 3.19.: Struktur des *configuration* Pakets

Das *configuration*-Paket ist für das Laden und Speichern der Konfigurationsdaten des Servers aus „Utilities“ und der Modellklassen aus „Simulation“ zuständig. Für die Daten wurde ein allgemeiner Datentyp *Parameter* angelegt. Neben dem eigentlichen Wert können Daten vom *Parameter*-Typ auch noch einen Namen und eine Einheit besitzen. So können die Daten durch die „Frontend“-Schicht besser identifiziert und dargestellt werden.

In *configuration* befinden sich auch die Interfaces für den Zugriff auf die Konfigurationsdaten. Sie werden von den Modellklassen (z.B. *Simulation*) und der *Server*-Klasse genutzt. Die Interfaces müssen von der konkreten Implementierung für die Konfiguration realisiert werden. Der Vorteil der Einführung der Interfaces liegt in der Unabhängigkeit der Verarbeitung der Konfigurationsdaten von der Art, wie diese Daten zur Verfügung stehen.

Als ein Beispiel für die Konfigurationsdaten wurde in dieser Arbeit eine XML-Datei angelegt, deren Inhalt in Kapitel 3.4.2 erläutert wird. Für die Verarbeitung der Daten aus der XML-Datei wurde das Unterpaket *xmlconfiguration* innerhalb von *configuration* erstellt.

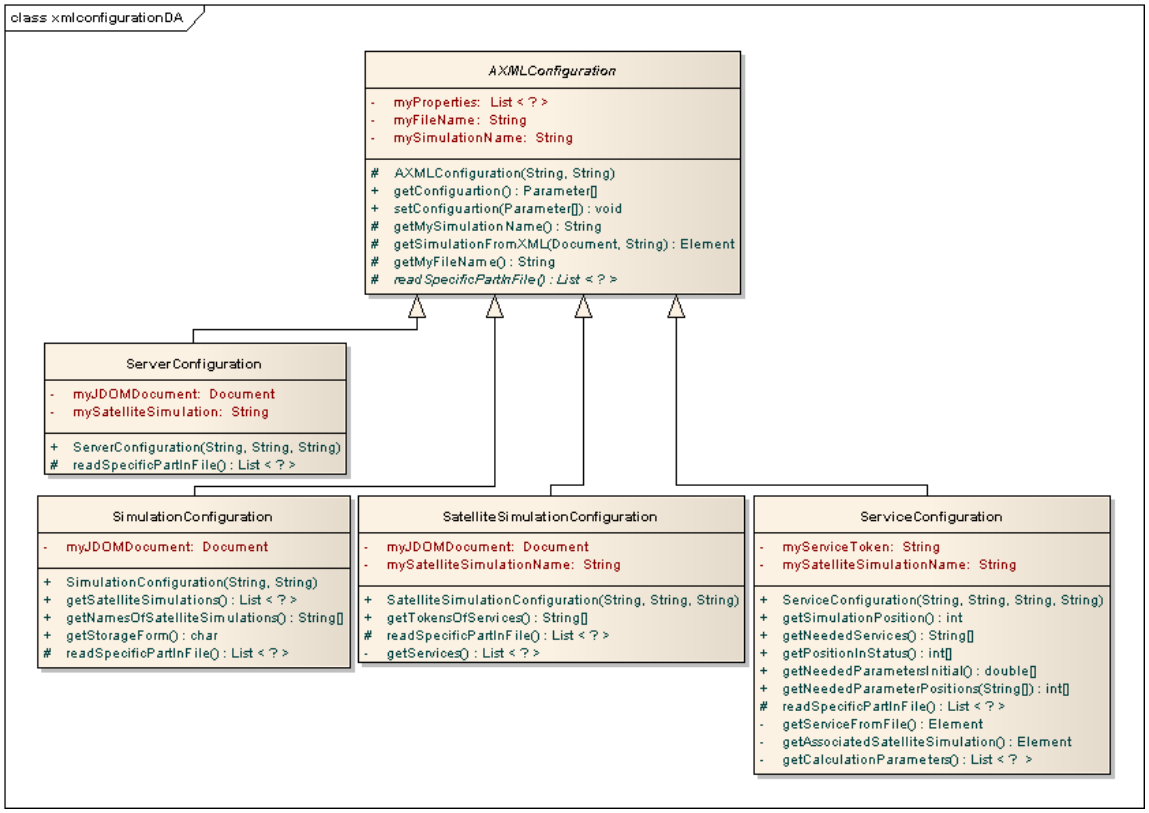


Abbildung 3.20.: Klassen des *xmlconfiguration*-Pakets

Dieses Paket ist in Abbildung 3.20 dargestellt. Es gibt hier eine abstrakte Klasse für eine allgemeine XML-Konfiguration, von der die anderen speziellen Konfigurationsklassen ableiten. Diese abstrakte Klasse entstand dadurch, dass ein Teil der Funktionalität für alle zu konfigurierenden Elemente bei Benutzung der XML-Datei gleich aussieht. Alle müssen sogenannte „Getter“ und „Setter“ für die Konfigurationsdaten besitzen. Diese wurden in der abstrakten Klasse mit den Methoden *getConfiguration()* und *setConfiguration(Parameter[])* modelliert.

Außerdem stellte sich heraus, dass alle Konfigurationsobjekte immer bis zu einem bestimmten Punkt die Datei auslesen, welches durch die konkrete Methode *getSimulationFromXML(Document,String)* ausgedrückt wird. Darüber hinaus liest jedes Konfigurationsobjekt noch einen spezifischen Teil der Datei aus. Der Fakt wurde mit der abstrakten Methode *readSpecificPartInFile()* berücksichtigt. Diese abstrakte Methode der Klasse *AXMLConfiguration* muss von den konkreten Klassen (z.B. *ServiceConfiguration*) jeweils implementiert werden.

Die übrigen Methoden in *AXMLConfiguration* sehen für alle Klassen gleich aus und wurden folglich ebenso herausgezogen. Damit wurden an dieser Stelle doppelte Code Passagen vermieden. Diese Passagen wären sonst in jeder konkreten Klasse zu finden.

Neben den Methoden der abstrakten Klasse implementieren die konkreten Klassen auch die in Abbildung 3.19 gezeigten Interfaces. Diese Interfaces müssten auch von einer

möglichen Datenbankanwendung realisiert werden. Die Klassen dazu könnten in einem Unterpaket *databaseconfiguration* angelegt werden.

3.4.2. XML-Konfigurationsdatei

Die Konfigurationsdatei stellt die Persistenzschicht für die Daten zur Konfigurierung dar. Durch die Einführung einer Datei wird die Forderung nach einer zentralen Stelle für die Konfigurationsdaten (UR 6.0) umgesetzt. Sie dient auch als Synchronisierungsmedium für die Konfigurationsdaten. Das bedeutet, wenn zur Laufzeit von SMASS ein Teil neu konfiguriert wird, werden diese Daten in die Datei geschrieben. Anschließend liest der betreffende Part die neuen Daten aus der Datei und ist damit konfiguriert.

Im Folgenden soll kurz der Aufbau der Konfigurationsdatei dargestellt werden. Die Konfigurationsdatei ist in einem XML-Format hinterlegt. Dieses Format eignet sich besonders gut für strukturierte Inhalte. In einer XML-Datei sind Elemente aufgelistet, welche ihrerseits noch Attribute haben können. Ein Element ist gekennzeichnet durch einen öffnenden Tag (Auszeichner) und einen schließenden Tag. Die Elemente in einer XML-Datei sind in einer Baumstruktur angeordnet.

Als Atribute enthalten die Elemente nicht nur die Zahlenwerte, sondern auch weitere Informationen, wie z.B. kurze Erläuterungen, Namen oder auch Einheiten. Dies führt dazu, dass die Konfigurationsdatei leicht verständlich und lesbar ist. Folglich ist die Datei auch leicht editierbar und vereinfacht damit die Benutzung (UR 17.0).

Neben den Attributen kann jedes Element weitere untergeordnete Elemente besitzen, welche Kindelemente genannt werden. Die Struktur der Elemente in der Konfigurationsdatei beruht auf der Beziehung der einzelnen Teile einer Simulation zueinander und unterstreicht so den modularen Aufbau von SMASS.

Die Einteilung der Elemente sieht wie folgt aus:

```
...
<SMASS>
   <simulation ...>
      ...
      <satelliteSimulation ...>
         ...
         <service ...>
            ...
         </service>
         ...
      </satelliteSimulation>
      ...
   </simulation>
   ...
</SMASS>
```

Ein weiterer Vorteil von XML-Dateien liegt in der Validierung. Für XML-Dateien können Schemas (XSDs) erstellt werden, welche die Struktur der Datei vorgeben. In den Schemas können die Anzahl und die Art der Elemente, deren Attribute und der Inhalt der Elemente vorgegeben werden. Wird dann eine XML-Datei erstellt, die diesem

Schema folgen soll, kann überprfüt werden, ob die Eingaben korrekt sind. Ein solches Schema wurde auch für die Konfigurationsdatei angelegt.

Wurzelelement und Header

Das Wurzelelement ist ein Pflichtelement, welches immer genau einmal in einer XML-Datei existieren muss. Dieses Element hat in der Konfigurationsdatei den Namen *SMASS* und zwei Attribute (siehe Tabelle 3.10). Die beiden Attribute müssen gesetzt werden, damit das Schema *smass_config_schema.xsd* verwendet werden kann.

Element		**Attribut**	
Name	**Anzahl**	**Name**	**Wert**
SMASS	1	*xmlns:xsi*	„http://www.w3.org/2001/XMLSchema-instance“
		xsi:noNamespaceSchemaLocation	„smass_config_schema.xsd“

Tabelle 3.10.: Eigenschaften des Wurzelelementes

Der Header der XML-Datei beinhaltet die Versionsnummer des verwendeten XML-Standards und die Kodierungsform.

Ein Beispiel für den Code des Headers und des *SMASS*-Elementes sieht wie folgt aus:

```
<?xml version="1.0" encoding="UTF-8"?>
<SMASS xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:noNamespaceSchemaLocation="smass_config_schema.xsd">
    ...
</SMASS>
```

simulation-Element

Damit eine Konfigurationsdatei für das Framework gültig ist, muss das Wurzelelement *SMASS* mindestens ein Kindelement vom Typ *simulation* besitzen. Das *simulation*-Element besitzt drei Attribute, welche in der folgenden Tabelle zusammengefasst sind:

Element		**Attribut**	
Name	**Anzahl**	**Name**	**Wert**
simulation	min. 1	*name*	eindeutiger Name
		storageForm	„text“
		stepsPerFile	natürliche Zahl > 0

Tabelle 3.11.: Eigenschaften des *simulation*-Elementes

Das erste Attribut ist der Name der Simulation. Dieser muss eindeutig sein, da er als Identifizierungsmerkmal dient. Es darf keine zweite Simulation mit dem gleichen Namen in der Konfigurationsdatei geben.

Als zweites Attribut kann die Art der Speicherung der Ergebnisse für diese Simulation angegeben werden. Da momentan nur die Speicherung in Textdateien umgesetzt ist, muss der Eintrag in diesem Feld „text“ lauten.

Die Anzahl der Simulationsschritte, deren Ergebnisse pro Datei gespeichert werden sollen, wird mit dem letzten Attribut angegeben. Damit kann der Benutzer die Größe der Dateien für die Ergebnisse beinflussen. Dies hat den Hintergrund, dass manche Betriebssysteme und Editoren nur Dateien bis zu einer bestimmten Größe verarbeiten können.

Eine Simulation muss zusätzlich zu den Attributen noch vier Parameter und mindestens ein Element vom Typ *satelliteSimulation* beinhalten. Letzteres wird im nächsten Abschnitt erläutert. Die Parameter und ihre Attribute sind in der folgenden Tabelle zusammengefasst:

Element		**Attribut**	
Name	**Anzahl**	**Name**	**Wert**
parameter	4	*name*	„startTime“; „stopTime“; „stepSize“; „realTimeModus“
		unit	„JD“; „JD“; „s“; -
		value	Gleitpunktzahl; Gleitpunktzahl; Gleitpunktzahl $>$ 0; „yes“ oder „no“

Tabelle 3.12.: Eigenschaften der *parameter*-Elemente einer Simulation

Die ersten beiden Parameter definieren den Start- und den Endzeitpunkt einer Simulation. Diese beiden Parameter sind jeweils im Julianischen Datum anzugeben. Der dritte Parameter beinhaltet die Schrittweite, welche in Sekunden angegeben wird. Als letztes muss noch die Option zur „schwachen Echtzeit“ angegeben werden. Hier sind mögliche Eingaben „yes“ oder „no“.

Ein gültiges Codefragment für eine Simulation würde demnach wie folgt aussehen:

```
...
   <simulation name="testSimulation" storageForm="text" stepsPerFile="5000">
      <parameter name="startTime" unit="JD" value="2454666.0">
      </parameter>
      <parameter name="stopTime" unit="JD" value="2454668.0">
      </parameter>
      <parameter name="stepSize" unit="s" value="5">
      </parameter>
      <parameter name="realTimeModus" unit="" value="no">
      </parameter>
      <satelliteSimulation ...>
         ...
      </satelliteSimulation>
      ...
   </simulation>
...
```

satelliteSimulation-Element

Das eine Simulation mindestens einen Satelliten beinhalten muss, spiegelt sich auch in der Konfigurationsdatei wieder. In der Datei muss das *simulation*-Element mindestens

ein Kindelement vom Typ *satelliteSimulation* enthalten, damit es gültig ist. Das *satelliteSimulation*-Element besitzt als Attribut nur den Namen der Satellitensimulation (siehe Tabelle 3.13). Dieser muss innerhalb des zugehörigen *simulation*-Elementes wieder eindeutig sein, da auch der Name der Satellitensimulation für die Identifizierung in der „Data Handling“-Schicht verwendet wird.

Element		**Attribut**	
Name	**Anzahl**	**Name**	**Wert**
satelliteSimulation	min. 1	*name*	eindeutiger Name

Tabelle 3.13.: Eigenschaften des *satelliteSimulation*-Elementes

Neben dem einen Attribut gehört zu diesem Element auch genau ein Parameter. Dieser ist in der folgenden Tabelle dargestellt:

Element		**Attribut**	
Name	**Anzahl**	**Name**	**Wert**
parameter	1	*name*	„port“
		unit	-
		value	natürliche Zahl (0 bis 65535), muss eindeutig sein

Tabelle 3.14.: Eigenschaften des *parameter*-Elementes einer Satellitensimulation

Der einzige Parameter, der zu einer Satellitensimulaion gehört, ist die Portnummer. Diese Nummer muss eindeutig sein, damit klar ist, welcher Service zu welchem Satelliten gehört, da für jede Satellitensimulation ein TCP/IP-Server angelegt wird.

Damit ein *satelliteSimulation*-Element gültig ist, muss es zusätzlich mindestens einen Service enthalten. Das *service*-Element wird im nächsten Abschnitt beschrieben. Ein zulässiges Fragment aus der Konfigurationsdatei für die Satellitensimulation könnte wie folgt aussehen:

```
...
    ...
        <satelliteSimulation name="testSatelliteSimulation">
            <parameter name="port" unit="" value="4444">
            </parameter>
            <service ...>
            </service>
             ...
        </satelliteSimulation>
    ...
...
```

service-Element

Das letzte Element betrifft die Services. Ein Element vom Typ *service* hat drei Attribute. Das erste Attribut ist der Name des Services und das zweite Attribut ist sein Token. Dieser Token muss innerhalb einer Satellitensimulation wieder eindeutig sein,

damit es zwischen den Services keine Verwechslungen geben kann. Das dritte Attribut ist die Position des Services in der Simulationsreihenfolge. Damit kann der Benutzer die Aktualität der von anderen Services benötigten Daten beeinflussen. Diese hängt davon ab, ob der Benutzer den Service in der Simulationsreihenfolge vor oder hinter dem Service mit den benötigten Daten positioniert. Die Attribute sind in der folgende Tabelle zusammengefasst:

Element		**Attribut**	
Name	**Anzahl**	**Name**	**Wert**
service	min. 1	*name*	Name des Services
		serviceToken	eindeutiger Token
		simulationPosition	natürliche Zahl >0

Tabelle 3.15.: Eigenschaften des *service*-Elementes

Ein Service hat außerdem eine beliebige Anzahl von Parametern. Diese Parameter definieren die Werte, welche mit der „CONFIG“-Nachricht an die Services verschickt werden. Die Reihenfolge der Parameter in der Konfigurationsdatei bestimmt auch die Reihenfolge in der „CONFIG“-Nachricht. Die Parameter für die Services haben die folgenden Attribute:

Element		**Attribut**	
Name	**Anzahl**	**Name**	**Wert**
parameter	beliebig	*name*	eindeutiger Name
		symbol	beliebiges Formelzeichen
		unit	Einheit des Parameters, wenn vorhanden
		value	beliebige Gleitkommazahl oder Zeichenfolge

Tabelle 3.16.: Eigenschaften der *parameter*-Elemente eines Services

Der Name der Parameter muss innerhalb des *service*-Elementes wieder eindeutig sein. Zusätzlich zu den bekannten Attributen *name*, *unit* und *value* ist noch ein Attribut *symbol* eingeführt worden, um die Parameter noch genauer beschreiben zu können.

Neben den Parametern aus Tabelle 3.16 gehört zu einem Service auch ein *result*- und ein *calculate*-Tag. Diese beiden Tags beinhalten ebenfalls Parameter.

In *result* sind die Ergebnisse des Services aus der „STATUS“-Nachricht aufgeführt. Diese Form von Parametern hat vier Attribute.

Das erste Attribut ist ein Name, welcher wieder eindeutig sein muss. Er dient zur Identifizierung innerhalb des Services.

Das zweite Attribut gibt die Position in der „STATUS“-Nachricht des Services an. Die Position beschreibt die Stelle im Datenteil der Nachricht, beginnend bei eins.

Im dritten Attribut wird die Einheit des Ergebnisses festgelegt. Damit wird ein Konverter überflüssig. Jeder Service, der Daten von anderen Services benutzt, muss diese in Zukunft selber intern in ein für ihn geeignetes Format wandeln.

Alle Attribute der Parameter aus dem *result*-Tag sind in der folgenden Tabelle zusammengefasst:

Element		**Attribut**	
Name	**Anzahl**	**Name**	**Wert**
parameter	beliebig	*name*	eindeutiger Name
		position	natürliche Zahl > 0
		symbol	beliebiges Formelzeichen
		unit	Einheit des Parameters, wenn vorhanden

Tabelle 3.17.: Eigenschaften der *parameter*-Elemente für die Ergebnisse eines Services

Im *calculate*-Tag kann ein Serviceentwickler die Daten angeben, welche von anderen Services benötigt werden. Die Daten werden ebenfalls als Parameter bezeichnet. Mit Hilfe der Attribute *name* und *fromService* ist es möglich, den Parameter innerhalb der Datei mit den Ergebnissen zu finden. Dazu muss der Name aber exakt dem Namen des Parameters in dem anbietenden Service entsprechen.

Es kann aber auch vorkommen, dass in der Ergebnisdatei Datei die geforderten Daten nicht zu finden sind. Ursache dafür könnte sein, dass der anbietende Service noch gar nicht entwickelt ist oder in dieser Simulation nicht berücksichtigt wird. Und selbst wenn er angemeldet ist, könnte es sein, dass aufgrund der Simulationsreihenfolge oder der Tatsache, dass es der erste Simulationsschritt ist, noch keine Daten in den Ergebnisdateien gespeichert sind. Damit die Simulation trotzdem durchgeführt werden kann ist noch ein Attribut namens *initialValue* eingeführt worden. Dieser Wert wird für den Fall, dass die geforderten Daten nicht vorhanden sind, geladen.

Die Position des Wertes in der „STATUS“-Nachricht des anfordernden Services muss ebenfalls bestimmt werden. Für diese Angabe existiert das Attribut *position*.

Die folgende Tabelle listet noch mal alle Attribute für die Parameter aus *calculate* auf:

Element		**Attribut**	
Name	**Anzahl**	**Name**	**Wert**
parameter	beliebig	*name*	Name des Parameters wie im anbietenden Service
		position	natürliche Zahl > 0
		fromService	Token des anbietenden Services
		initialValue	beliebige Gleitkommazahl oder Zeichenfolge
		unit	Einheit des Parameters, wenn vorhanden

Tabelle 3.18.: Eigenschaften der *parameter*-Elemente für die von anderen Services benötigten Werte eines Services

Zum Abschluss soll ein gültiges XML-Codebeispiel das *service*-Element zusammenfassen.

```
...
    ...
        ...
            <service name="Power" serviceToken="PWR000" simulationPosition="2">
                <parameter name="capacity_of_batteries" symbol="C" unit="Ah" value="4">
                </parameter>
                ...
                <result>
                    <parameter name="capacity_of_batteries" position="1" symbol="C" unit=
                        "Ah">
                    </parameter>
                    ...
                </result>
                <calculate>
                    <parameter name="eclipse" position="1" fromService="ORB001"
                        initialValue="1" unit="">
                    </parameter>
                    ...
                </calculate>
            </service>
        ...
    ...
...
```

3.4.3. Laden/Speichern der Simulationsergebnisse

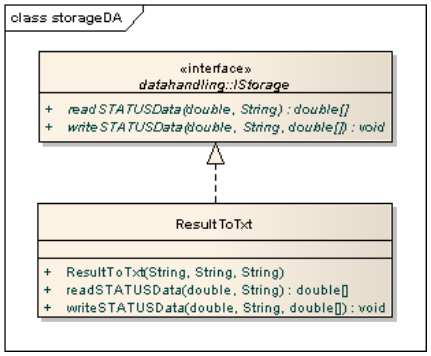


Abbildung 3.21.: Struktur des *storage*-Pakets
(Attribute und Teil der Methoden in *ResultToTxt* ausgeblendet)

In Abbildung 3.21 ist nun das zweite Paket aus der Schicht „Data Handling“ dargestellt. Dieses Paket ist zuständig für das Laden und die Speicherung der Simulationsergebnisse, was wiederum eine eigenständige Funtkionalität ist. Es wurde ein Interface (*IStorage*) geschaffen, über welches der *ServerMessageHandler* die Daten der „STATUS“-Nachrichten speichern kann.

Es ist aber auch möglich, Daten von einem bereits gespeicherten Zeitpunkt über die Methode *readSTATUSData(double,String)* des Interfaces *IStorage* wieder auszulesen. Dies resultiert aus der Anforderung UR 3.0 aus Kapitel 2. Dort wurde gefordert, dass die Services sich gegenseitig beeinflussen können. Mit der Möglichkeit des Einlesens von Simulationsdaten ist die Anforderung umgesetzt worden. Diese Daten können als Eingangswerte für den nächsten Simulationsschritt eines Services verwendet werden.

In dieser Arbeit werden die Ergebnisse in Textdateien gespeichert. Die zugehörige Verzeichnisstruktur und die Darstellung der Daten innerhalb der Dateien werden in Kapitel 3.4.4 erläutert.

Für die Speicherung in Textdateien wurde die Klasse *ResultToTxt* erstellt, welche die beiden Methoden des Interfaces realisiert. Darüber hinaus besitzt diese Klasse weitere Attribute und Methoden, welche lediglich das Schreiben und Lesen der Ergebnisse umsetzen. Diese sind in 3.21 nicht gezeigt, um die Darstellung nicht zu überladen.

Neben der Speicherung der Ergebnisse in Textdateien könnte in Zukunft auch eine Variante für die Speicherung in XML-Dateien oder in einer Datenbank hinzugefügt werden. Diese Klassen müssten dann auch analog zu *ResultToTxt* das Interface *IStorage* aus Abbildung 3.21 realisieren und so ebenfalls die Möglichkeiten zum Speichern und zum Lesen von Simulationsergebnissen gewährleisten.

3.4.4. Speicherung der Simulationsergebnisse in Textdateien

Wie in Kapitel 3.4.3 bereits angedeutet, werden die Ergebnisse in Textdateien gespeichert. Die Textdateien bilden die Persistenzschicht im Bereich der Ergebnisse und erfüllen damit die Forderung nach der Speicherung der Simulationsergebnisse (UR 5.0).

Die Dateien für die Ergebnisse werden in einer ähnlichen Struktur, wie die Modellklassen angelegt. Zunächst wird ein Ordner in dem Verzeichnis erstellt, wo sich auch die Konfigurationsdatei befindet. Der Ordner erhält den Namen der verwendeten Simulation. In diesem Ordner wird ein Unterordner mit einem Zeitstempel als Namen angelegt. Dieser Zeitstempel beschreibt den Zeitpunkt, in dem der Benutzer die Simulation gestartet hat. Es wird auch jedes Mal ein neuer Ordner mit einem neuen Zeitstempel angelegt, wenn eine Simulation wiederholt wird.

In dem Ordner mit dem Zeitstempel befinden sich noch weitere Unterordner. Diese sind nach den Namen der zu der Simulation gehörenden Satellitensimulationen benannt. In diesen Ordnern liegen die Textdateien mit den Ergebnissen. Deren Name ergibt sich jeweils aus dem ersten und dem letzten Simulationszeitpunkt und der Endung „.txt“. Innerhalb der Dateien wird pro Service und Simulationspunkt eine neue Zeile geschrieben.

Diese Ordnerstruktur soll kurz anhand eines Beispiels gezeigt werden. In der folgenden Tabelle sind die nötigen Informationen aus der Konfigurationsdatei dargestellt, welche die Struktur und die Bezeichnungen beeinflussen.

Bezeichnung	Wert
Name der Simulation	testSimulation
Name der ersten Satellitensimulation	testSatelliteSimulation
Anzahl der Services	2
Start der Simulationszeit	0.0 JD
Ende der Simulationszeit	1.0 JD
Schrittweite für die Simulation	43200 s
Anzahl der Simulationspunkte pro Datei	2

Tabelle 3.19.: Speicherungsstruktur der Ergebnisse beeinflussende Konfigurationsdaten

Zusätzlich zu den in Tabelle 3.19 aufgeführten Daten ist der Pfad für die Konfigurationsdatei und der Zeitpunkt, wann der Benutzer die Simulation gestartet hat, wichtig. Folgende Werte sollen hier als Beispiel dienen:

Bezeichnung	Wert
Pfad der Konfigurationsdatei	$\ldots\backslash data\backslash$
Zeitpunkt des Simulationsstarts	20 July 2008, 18:01:58

Tabelle 3.20.: Speicherungsstruktur der Ergebnisse beeinflussende Daten

Damit ergibt sich die folgende Ordnerstruktur für die Dateien mit den Ergebnissen:

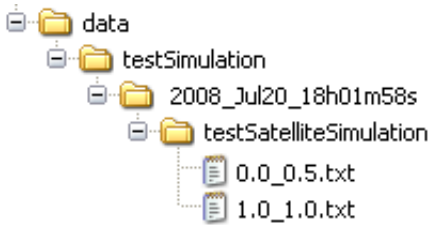


Abbildung 3.22.: Ordner- und Dateistruktur der Ergebnisse

Es werden in diesem Beispiel zwei Dateien angelegt. Insgesamt gibt es aber drei Simulationsschritte, wobei in dem Bespiel maximal zwei pro Datei gespeichert werden sollen. Die erste Datei beinhaltet zwei Simulationsschritte verteilt auf vier Zeilen, während die zweite Datei nur einen Simulationsschritt mit zwei Zeilen beinhaltet. Der Inhalt der zweiten Datei sieht wie folgt aus:

```
1 1.0;ORB001;1.0;6540962.627499752;1702698.773925063;1821160.2869944293; ...
2 1.0;PWR000;0.0;
```

Zunächst wird am Anfang jeder Zeile der betreffende Simulationszeitpunkt eingetragen. Anschließend folgt der Service-Token. Danach kommen die Ergebnisse des Services für diesen Simulationsschritt. Alle Felder sind dabei durch Semikolons getrennt. Dies erleichtert die weitere Verarbeitung in anderen Anwendungen, wie z.B. Excel. In der zweiten Zeile sind die Ergebnisse des zweiten Services dargestellt.

Für den Zeilenwechsel wird automatisch die Systemeigenschaft für die Trennung von Zeilen verwendet. Kodiert werden die Daten ebenfalls mit der im System hinterlegten Standardkodierung. Durch die Abfrage der Linientrennung und Kodierung von den Systemeigenschaften ist die Speicherung der Ergebnisse plattformunabhängig und unter verschiedenen Betriebssystemen einsetzbar (UR 9.0/10.0).

3.4.5. Fazit

Zunächst wird mit der Schicht „Data Handling“ die Anforderung UR 3.0 umgesetzt. In dieser Anforderung wurde ein gegenseitiger Einfluß der einzelnen Disziplinen formuliert, welcher durch das Lesen von Simulationsergebnissen gewährleistet werden kann.

Darüber hinaus wird mit dem Paket *storage* die grundsätzliche Speicherung von Simulationsergebnissen umgesetzt (UR 5.0). Diese Daten werden zur späteren Auswertung genutzt.

Das Paket *configuration* aus der „Data Handling“-Schicht trägt zu UR 6.0 bei, wo eine zentrale Stelle für die Konfigurationsdaten und das Modell gefordert wird. Die Funktionalitäten aus *configuration* ermöglichen den Zugriff auf die eingeführte Konfigurationsdatei. In dieser Datei werden zentral die Daten für die Konfigurierung verwaltet.

Durch die Kapselung der gesamten Datenverarbeitung ist es möglich diese sehr leicht auszutauschen. Die in dieser Arbeit zu Test- und Vorführungszwecken implementierte Datenverarbeitung auf Dateibasis kann durch höherwertige Systeme ersetzt werden. Dazu müssen die vier Interfaces aus Abbildung 3.19 für die Konfigurierung und das Interface *IStorage* für die Speicherung von Simulationsergebnissen implementiert werden. Mit diesen Interfaces ist auch die Schnittstelle zur Datenverarbeitung (UR 14.0) klar definiert.

3.5. „Frontend“

Die Schicht „Frontend“ setzt die Benutzeroberfläche um. Sie ermöglicht die Interaktion des Benutzers mit SMASS. Als Interaktionen sind das Anlegen und Konfigurieren der Bestandteile von Simulationen und das Steuern von Simulationen möglich.

Um diese Interaktionen zu gewährleisten, stellt die „Frontend“-Schicht die Daten einer Simulation dar. Zu diesen Daten zählen die Konfiguration und die Zustände einer Simulation.

Darüber hinaus muss die Oberfläche auch die Wünsche der Benutzer entgegennehmen und an die Simulation weiterleiten. Die Schicht „Frontend“ beeinflusst demnach auch die Simulationen.

Somit beinhaltet „Frontend“ zwei Funktionalitäten, das Darstellen und die Manipulation der Daten. Für die reine Darstellung der Daten muss nur lesend auf die Daten zugegriffen werden. Im Gegensatz dazu ist zur Manipulation der Daten auch ein schreibender Zugriff nötig.

Um die beiden Funktionalitäten sauber zu trennen und eine möglichst große Flexibilität bezüglich der Schnittstelle für die Benutzerinteraktion zu gewährleisten ([ES04] S.55), wurde hier das Entwurfsmuster (Pattern) „Model View Controller“ (MVC) verwendet. Außerdem wurde für den Austausch von Daten zwischen dem Modell in Form der Klassen aus der „Simulation“-Schicht und der Benutzeroberfläche auf das Observer-Pattern zurückgegriffen. Damit kann gewährleistet werden, dass die dargestellten Daten immer aktuell sind. Beide Entwurfsmuster sind im Anhang C beschrieben.

Um die vollen Möglichkeiten der beiden angesprochenen Patterns zu nutzen, ist es sinnvoll, die Benutzeroberfläche in einem eigenen Thread laufen zu lassen. Dementsprechend wurde festgelegt, dass mit dem Starten der Benutzeroberfläche auch ein neuer Thread gestartet wird.

3.5.1. Struktur und Verhalten

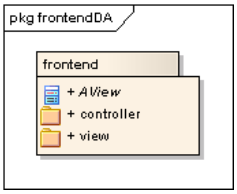


Abbildung 3.23.: Struktur der „Frontend“-Schicht

Die Trennung der beiden Funktionalitäten wurde in der in Abbildung 3.23 gezeigten Struktur umgesetzt. „Frontend“ enthält das Paket *controller* mit den Klassen zur Manipulation des Modells und das Paket *view* für die Darstellung der Daten. Für dieses Paket ist in Abbildung 3.23 auch die Schnittstelle in Form der Klasse *AView* dargestellt, welche im folgenden Abschnitt zum Observer-Pattern erläutert wird.

Umsetzung des Observer-Patterns

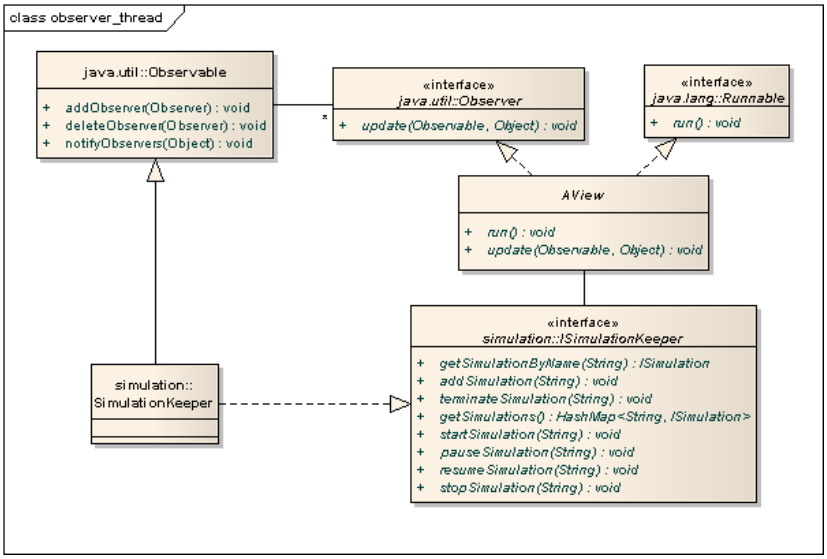


Abbildung 3.24.: Umsetzung des Observer-Patterns
(Attribute und Methoden in *Observable*, *SimulationKeeper* und *ISimulationKeeper* teilweise ausgeblendet)

AView beinhaltet zwei Methoden, welche von jeder konkreten Benutzeroberfläche durch Vererbung umgesetzt werden. Die Methode *run()* sorgt dafür, dass die Benutzeroberfläche in einem eigenen Thread läuft. Zur Umsetzung des Observer-Patterns muss eine Benutzeroberfläche aber auch die Methode *update(Observable,Object)* bereitstellen. Diese Methode dient der Übermittlung der Information, dass sich das von der Benutzeroberfläche beobachtete Modell geändert hat. Der Aufruf der Methode wird durch die Modellklassen initiiert. In Abbildung 3.24 ist die Klasse *SimulationKeeper* als Beispiel für eine Modellklasse dargestellt worden.

Die Modellklassen leiten von der Klasse *Observable* ab. Durch die Vererbung steht den Modellklassen unter anderem die Methode *notifyObservers(Object)* zur Verfügung. Diese Methode wird immer dann aufgerufen, wenn sich der Zustand des Modells geändert hat. Innerhalb von *notifyObservers(Object)* wird die *update(Observable,Object)*-Methode aller Beobachter (*Observer*) aufgerufen.

Die Beobachter können an ein Objekt vom Typ *Observable* angehangen bzw. wieder gelöst werden. Dazu stehen die beiden Methoden *addObserver(Observer)* bzw. *delete(Observer)* aus der Klasse *Observable* bereit. An den Übergabeparametern der beiden Methoden zeigt sich, dass ein Beobachter vom Typ *Observer* sein muss. Um dies zu gewährleisten, implementiert *AView* dieses Interface.

Der Beobachter *AView* aus 3.24 wiederum hat Zugriff auf das Modell, was im Beispiel die *SimulationKeeper*-Klasse ist. Diese Klasse wurde noch gekapselt, so dass der Beobachter nur Zugriff auf das zugehörige Interface des Modells hat. Mit den Methoden aus dem Beispielinterface *ISimulationKeeper* wird dann die Ansicht für den Benutzer aktualisert.

Um das SMASS-Framework zu testen, wurde zusätzlich noch eine Konsolenoberfläche implementiert. Diese ermöglicht die Überprüfung der grundlegenden Funktionen von SMASS und besteht aus vier Ansichten, die in folgender Tabelle erläutert werden.

Ansicht	**Beschreibung**	**mögliche Interaktionen**
MainView	Darstellung des Hauptmenüs	• Anlegen einer Simulation • Auswählen einer Simulation • Zur Änderung der Simulationsparameter zum *ParameterView* wechseln • Steuern einer Simulation • Löschen der Laufzeitrepräsentation einer Simulation • Zum *SimulationView* wechseln • Ansicht aktualisieren • SMASS beenden
SimulationView	Darstellung der Simulation	• Anlegen einer Satellitensimulation • Auswählen einer Satellitensimulation • Löschen der Laufzeitrepräsentation einer Satellitensimulation • Zum *SatelliteSimulationView* wechseln • Ansicht verlassen
SatelliteSimulationView	Darstellung einer Satellitensimulation	• Verbinden eines Services • Auswählen eines Services • Trennen der Verbindung zu einem Service • Zur Änderung der Servicekonfiguration zum *ParameterView* wechseln • Ansicht verlassen
ParameterView	Darstellung von Konfigurationsparametern	• Auswählen eines Parameters • Editieren eines Parameters • Ansicht verlassen

Tabelle 3.21.: Ansichten der Konsolenoberfläche

Alle vier Ansichten aus Tabelle 3.21 haben Gemeinsamkeiten in Form von Attibuten und Methoden, welche in einer abstrakten Klasse *ACommandLineView* zusammengefasst sind. Zugriff auf diese Methoden erhalten die Ansichten durch Ableitung von dieser abstrakten Klasse. *ACommandLineView* selber leitet von der allgemeinen Klasse *AView* ab.

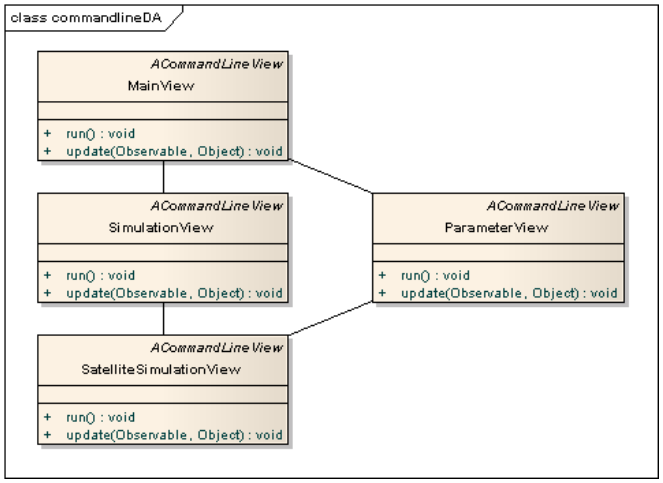


Abbildung 3.25.: Klassen für die Ansichten der Konsolenoberfläche
(Attribute und einige Methoden ausgeblendet)

Die Ansichten sind ähnlich wie die Modellklassen hierarchisch aufgebaut (siehe Abbildung 3.25). An oberster Stelle steht der *MainView*. Diese Ansicht kann den *SimulationView* starten. Ist dieser gestartet, läuft der *MainView* im Hintergrund weiter. Er ist lediglich deaktiviert. Analog verhält es sich mit dem *SatelliteSimulationView*. Dieser wird vom *SimulationView* gestartet, dessen Thread trotzdem im Hintergrund weiter läuft.

Der *ParameterView* kann sowohl vom *MainView*, als auch vom *SatelliteSimulationView* aus gestartet werden. Auch dabei wird der startende Thread nicht beendet, sondern läuft im Hintergrund weiter. Mit dem *ParameterView* können zum einen die Simulationsparameter geändert werden, wenn dieser aus dem *MainView* heraus gestartet wird. Zum anderen können die Konfigurationsdaten der Services verändert werden, wenn die Ansicht aus dem *SatelliteSimulationView* gestartet wird.

Durch den hierarchischen Aufbau und die konsequente Trennung der Ansichten in einzelne Threads bewirkt die Interaktion „Ansicht verlassen“ aus Tabelle 3.21 immer den Wechsel zu der vorherigen Ansicht. Als kleines Beispiel soll das Paar *MainView* und *ParameterView* dienen. Innerhalb des *MainView*s möchte der Benutzer z.B. die Simulationsparameter ändern. Dadurch wird der *ParameterView* in einem eigenen Thread gestartet. Nachdem der Benutzer die gewünschten Änderungen vorgenommen hat, wechselt er mit der Interaktion „Ansicht verlassen“ wieder in den *MainView*. Der Thread für den *ParameterView* wird beendet und der *MainView* ist wieder aktiviert.

Unsetzung des MVC-Patterns

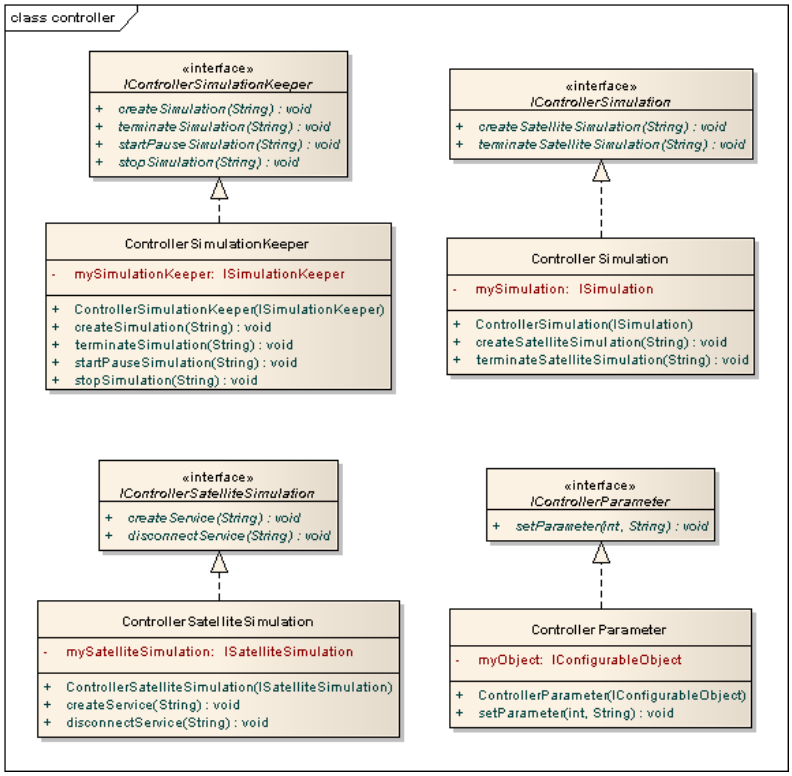


Abbildung 3.26.: Klassen für die Manipulation des Modells

Um die durch das MVC-Pattern gewährleistete Flexibilität zu erhalten, wurden die in Abbildung 3.26 gezeigten Klassen und Interfaces für die Manipulation des Modells aus der „Simulation“-Schicht angelegt. Diese befinden sich in dem Unterpaket *controller*. Es existiert für jede Modellklasse, welche weitere Modellklassen instanziieren kann, jeweils ein Controller. Zusätzlich wurde noch ein *ControllerParameter* eingeführt, der für die Konfigurierung einer Modellklasse zuständig ist. Alle vier Controller implementieren jeweils ein Interface. Damit ist die Verwendung durch das *view*-Paket gewährleistet.

Für jede Ansicht der Benutzeroberfläche wird nun ein Triple aus einer Modellklasse, einem Controller und einer Klasse aus dem *view*-Paket angelegt. Auch im Einstiegspunkt in der Klasse *SMASS* aus der „Simulation“-Schicht wird so ein Triple angelegt. Anhand dieses Triples mit der Hauptansicht (*MainView*) und der folgenden Abbildung 3.27 wird die Umsetzung des MVC-Patterns erläutert.

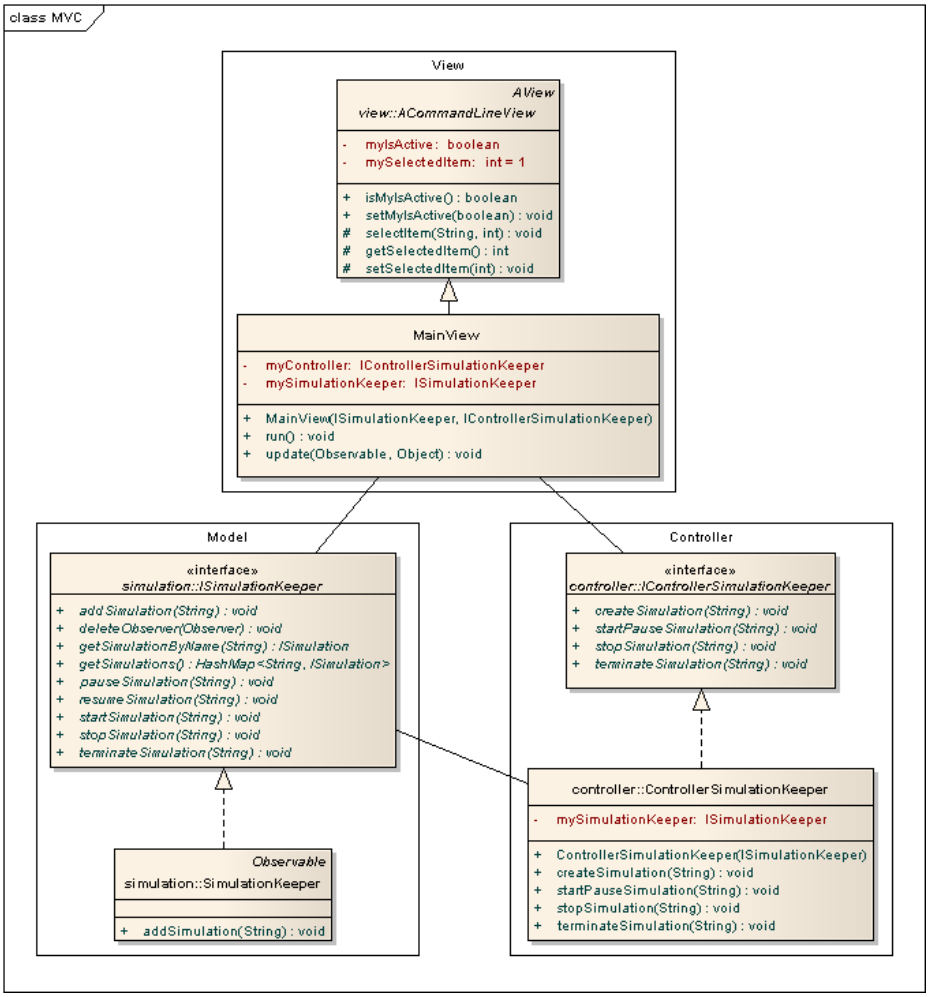


Abbildung 3.27.: MVC-Pattern am Beispiel des Triples aus *SMASS*
(Attribute und Methoden in *ACommandLineView*, *MainView* und *SimulationKeeper* teilweise ausgeblendet)

Innerhalb der *SMASS*-Klasse wird zunächst eine *SimulationKeeper*-Instanz vom Typ *ISimulationKeeper* angelegt. Diese Instanz ist das Modell in dem MVC-Beispiel. Die Instanz der Modellklasse wird anschließend dem Konstruktor des zugehörigen *ControllerSimulationKeeper* übergeben. Somit ist auch eine Instanz des Controllers angelegt. Die Controllerinstanz ist im Beispiel vom Typ *IControllerSimulationKeeper* und wird zusammen mit der Instanz vom Modell dem Konstruktor der Ansicht *MainView* übergeben. Damit ist das Triple zum Einstieg in SMASS angelegt.

Der Austausch von Informationen hängt eng mit dem Observer-Pattern zusammen. In dem *MainView* kann der Benutzer z.B. eine Simulation anlegen. Hat der Benutzer diese Option ausgewählt, wird der Wunsch an den Controller mit der Methode *createSimulation(String)* weitergeleitet. Der Controller seinerseits ruft dann die Funktion *addSimulation(String)* aus der Modellklasse *SimulationKeeper* auf. Nachdem die Simulation angelegt ist, hat sich das Modell geändert. Dadurch ruft *SimulationKeeper* die *notifyObservers(Object)*-Methode aus der Basisklasse *Observable* auf. Innerhalb dieser

Methode wird dann die *update(Observable,Object)* der Benutzeroberfläche aufgerufen. Daraufhin aktualisiert sich die Ansicht *MainView* und stellt dem Benutzer die neu erstellte Simulation dar.

3.5.2. Fazit

Die eingeführte Benutzeroberfläche ermöglicht die Repräsentation der Daten (UR 4.0). Darüber hinaus gibt sie dem Benutzer die Möglichkeit, die Simulation zu steuern (UR 7.0) und dient damit als Benutzerinterface. Es wurden der grundsätzliche Ablauf und die verwendeten Konzepte für die Darstellung und die Benutzerinteraktion an Beispielen erläutert.

Mit der Einführung der „Frontend"-Schicht wird diese Funktionalität klar von den anderen Funktionalitäten getrennt (UR 16.0). Diese Strukturierung erhöht die Lesbarkeit des Quelltextes und gestaltet ihn dadurch wartbar (UR 12.0).

Die verwendeten Konzepte des Observer- und MVC-Patterns tragen zur hohen Modularität und Flexibilität innerhalb der „Frontend"-Schicht bei. Sie ermöglichen den Austausch der Benutzeroberfläche mit minimalem Aufwand. Zum Austausch muss lediglich ein neues Paket innerhalb von *view* angelegt werden, indem die Klassen z.B. von einem Web-Frontend liegen. In der Klasse *SMASS* muss eine Instanz von diesem Web-Frontend angelegt werden, dessen Klasse von *AView* ableiten muss. *AView* ist demnach die Schnittstelle für die „Frontend"-Schicht (UR 14.0). Für die Manipulation des Modells können die Klassen aus dem *controller*-Paket verwendet werden. Sie müssen für eine neue Benutzeroberfläche nicht modifiziert werden. Die Erläuterungen zu dem Web-Frontend zeigen, wie einfach SMASS erweitert werden kann (UR 13.0).

Mit den beiden eingeführten Patterns ist es in Zukunft auch möglich, mehrere Nutzerschnittstellen zur Verfügung zu stellen. Darüber hinaus könnten diese auch parallel aktualisiert werden, sofern sie die Schnittstelle *AView* umsetzen.

4. Bedienung

4.1. Installation

Bevor das SMASS-Framework mit seinen Services installiert werden kann, muss eine Java Laufzeitumgebung (JRE) installiert sein. Ist das noch nicht der Fall, kann eine aktuelle Laufzeitumgebung von der folgenden Internetseite heruntergeladen und anschließend installiert werden:

`http://www.java.com/en/download/manual.jsp`

Alternativ kann auch das Java Development Kit (JDK) von der CD (siehe Anhang D) installiert werden. Dieses beinhaltet neben der Laufzeitumgebung zusätzlich auch einige Entwicklungswerkzeuge, wie *jar* zum Erstellen von Java-Archiven oder den Java-Compiler *javac*. Diese werden benötigt, falls kleine Änderungen am Quelltext vorgenommen werden und der Quelltext unter der Verwendung der in 4.4 vorgestellten Dateien wieder neu kompiliert und archiviert werden soll.

Bei der Installation von der Laufzeitumgebung (JRE) oder des JDKs wird der Benutzer durch ein Installationsmenü geführt, bei dem nichts weiter beachtet werden muss. Der Benutzer sollte sich lediglich den Pfad merken, wo er die Software installiert.

Bei Windows-Systemen muss nach der Installation noch die Systemvariable für den Pfad in den Umgebungsvariablen gesetzt werden. Dies führt dazu, dass alle Java-Befehle von jedem Verzeichnis auf dem Computer aus benutzt werden können, ohne dass der absolute Pfad noch mit angegeben werden muss. Dies ist wiederum Voraussetzung für das Bearbeiten des Quelltextes und Starten von SMASS.

Die Systemvariable kann unter Windows XP durch den Aufruf des folgenden Menüs gesetzt werden:

Start ⇒ Systemsteuerung ⇒ System ⇒ Erweitert ⇒ Umgebungsvariablen

Unter Windows Vista sieht der Aufruf ähnlich aus:

Start ⇒ Systemsteuerung ⇒ System ⇒ Erweiterte Systemeinstellungen ⇒
Erweitert ⇒ Umgebungsvariablen

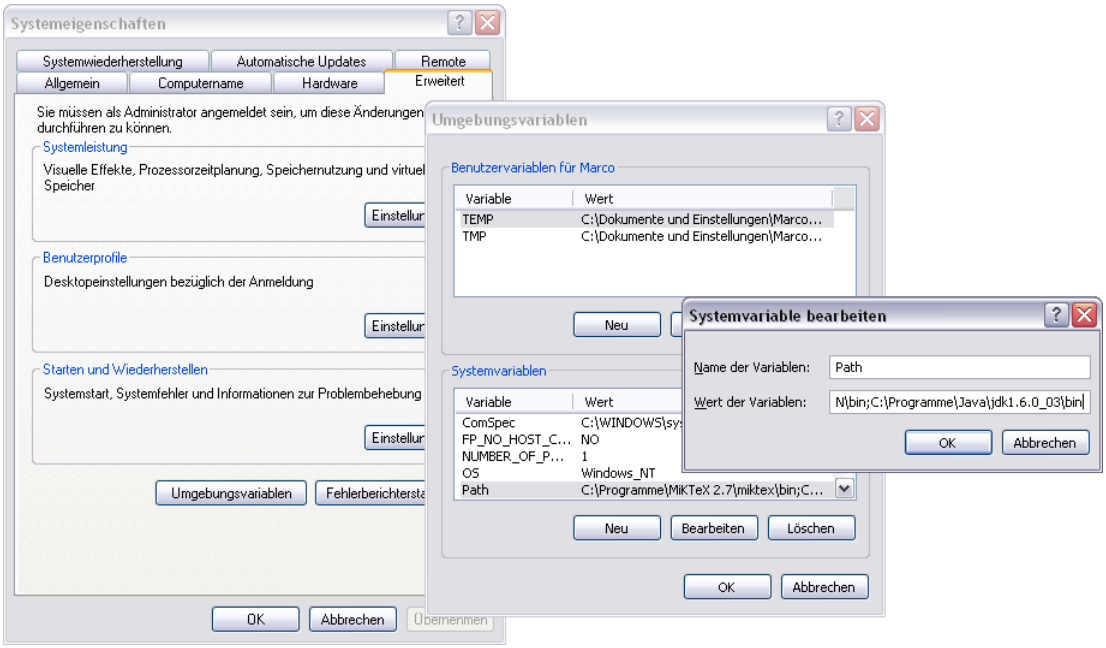


Abbildung 4.1.: Systemvariable in Windows XP setzen

Durch den Aufruf erscheint das Menü „Umgebungsvariablen“. Hier muss der Benutzer in dem Fenster „Systemvariablen“ das Feld „Path“ markieren und dann auf „Bearbeiten“ klicken. In dem sich jetzt öffnenden Dialog „Systemvariable bearbeiten“ kann nun der neue Pfad, der auf das *bin*-Verzeichnis in dem Java-Ordner verweist, mit einem Semikolon hinten angehangen werden. Für das JDK von der CD würde der Eintrag dem Folgenden ähneln:

$$\text{Wert der Variablen: } \ldots;C:\backslash Programme\backslash Java\backslash jdk1.6.0\_03\backslash bin$$

Nachdem die Java-Installation abgeschlossen ist, kann nun das SMASS-Framework installiert werden. Dazu müssen die Dateien aus dem Ordner *Executable\Framework* der beiligenden CD in ein beliebiges Verzeichnis kopiert werden. Zu den Dateien gehört das Archiv *Server.jar* mit den kompilierten Dateien, die Konfigurationsdatei *Configuration.xml*, das XML-Schema *smass_config_schema.xsd* und zwei Dateien zum Starten des Servers von SMASS. Für das Starten wurde eine Batch-Datei *StartServer.bat* für Windows-Systeme und ein Shell-Script *StartServer.sh* für Linux-Systeme angelegt. Je nachdem, welches Betriebssystem auf dem Rechner gerade läuft, muss natürlich nur eine der beiden Dateien kopiert werden. Unter Linux muss der Benutzer noch das Dateiattribut für die Shell-Skripte auf ausführbar setzen. Von dem Verzeichnis aus, indem sich die Shell-Skripte befinden, ist das Setzen der Attribute z.B. mit folgenden Befehl möglich:

$$find\ -name\ \text{“}*.sh\text{”}\ -exec\ chmod\ +x\ \{\}\ \backslash;$$

Wichtig für die spätere Ausführung ist nur, dass alle drei Dateien in ein und dasselbe Verzeichnis gelangen, da sonst die relativen Pfade in der Batch-Datei oder dem Shell-Skript nicht mehr stimmen und sie damit nicht mehr ausführbar sind. Die Pfade in der

Batch-Datei oder dem Shell-Skript können aber auch leicht mit einem Editor angepasst werden.

Analog dazu kann mit den beiden bereits erstellten Services verfahren werden. Deren kompilierte Dateien befinden sich ebenfalls in dem Ordner *Executable*. Für die Services wurden die Unterordner *ORB001* und *PWR000* angelegt. In diesen Ordnern befinden sich jeweils eine Archivdatei *<Service-Token>.jar* und die Dateien zum Starten des Services. Auch hier sind wieder eine Windows Batch-Datei *Start<Service-Token>.bat* und ein Linux Shell-Script *Start<Service-Token>.sh* erstellt worden. Weiterhin ist es wieder wichtig, das beide Dateien eines Services in ein und dasselbe Verzeichnis kopiert werden, damit die Pfadangaben in den Startdateien noch richtig sind. Ob die Services auf den gleichen Computer oder einen beliebigen anderen Computer installiert werden, ist dem Benutzer überlassen. Einzige Bedingung ist, dass die Computer über ein Netzwerk miteinander verbunden sind.

Mit dem Kopieren der Dateien ist die Installation von SMASS abgeschlossen.

4.2. Konfigurierung

Die Konfigurierung von einzelnen Simulationen von SMASS erfolgt mittels der in Kapitel 3.4.2 vorgestellten XML-Datei. Sie befindet sich im Ordner, wo der Benutzer die Dateien des Frameworks abgelegt hat. Die Konfigurationsdatei kann mit einem beliebigen Editor bearbeitet werden. Es wird aber empfohlen, einen Editor zu benutzen, der XML-Dateien validieren kann. Beispiele für solche Editoren sind, neben der Entwicklungsumgebung Eclipse, Altova XML Spy 2008, Exchanger XML und Liquid XML Studio 2008.

Um die Services beim Starten dem richtigen Satelliten zuordnen zu können, muss sich der Benutzer den Port aus der Konfigurationsdatei merken. Dieser Port ist standardmäßig auf 4444 eingestellt und muss bei Abweichungen oder der Verwendung von mehreren Satelliten in den Startdateien der Services (*Start<Service-Token>*.bat oder *Start<Service-Token>.sh*) geändert werden.

Zusätzlich können die Werte für die Simulationszeitsteuerung aus Tabelle 3.12 und die „CONFIG“-Nachricht an die Services aus Tabelle 3.16 über die Konsolenoberfläche geändert werden. Dies wird unter anderem im folgenden Unterkapitel erläutert.

4.3. Durchführen einer Simulation

Um SMASS zu starten, muss der Benutzer die Batch-Datei *StartServer.bat* unter Windows bzw. das Shell-Skript *StartServer.sh* unter Linux aufrufen. Daraufhin wird dem Benutzer ein Dialog angezeigt, welcher aus drei Bereichen besteht. Der erste Bereich ist die Anzeige der in der Konfigurationsdatei aufgeführten Simulationen. Diese sind durchnummeriert und mit einer Statusanzeige versehen, welche nach dem Start von SMASS für alle Simulationen auf „listed“ steht. Der zweite Bereich zeigt die Menüauswahl mit

den möglichen Optionen und der dritte Bereich ist das Eingabefeld. Das Hauptmenü sieht beispielsweise wie folgt aus:

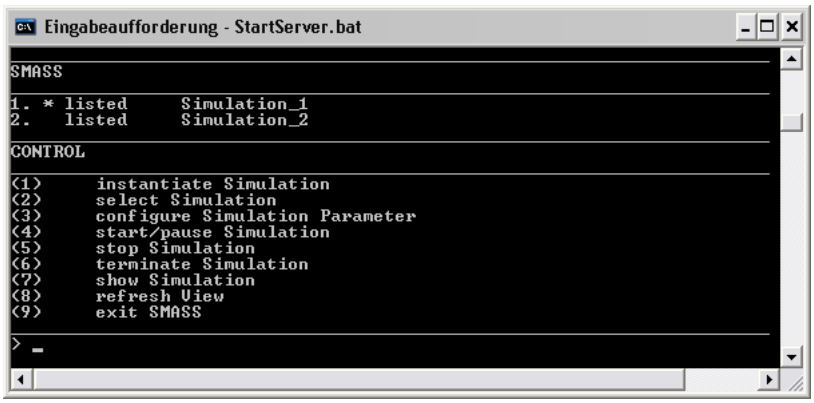


Abbildung 4.2.: Hauptmenü nach dem Start von SMASS

In dem Beispiel aus Abbildung 4.2 sind zwei mögliche Simulationen angezeigt, *Simulation_1* und *Simulation_2*. Standardmäßig ist immer die zuerst angezeigte Simulation ausgewählt, was an dem kleinen Sternchen vor der Simulation erkenntlich ist. In dem Beispiel soll nun aber die zweite Simulation benutzt werden, welche mit dem Menüpunkt „select Simulation“ ausgewählt werden kann. Dazu muss der Benutzer eine „2“ in das Eingabefeld am unteren Ende der Ansicht eingeben und mit Enter bestätigen. Anschließend wird er nach der Nummer der Simulation gefragt, die er auswählen möchte. Hat er diese ebenfalls eingegeben und mit Enter bestätigt, ist der Stern vor die zweite Simulation gewandert.

Als nächstes muss der Benutzer die ausgewählte Simulation instanziieren. Dies ist durch die Eingabe einer „1“ für den Menüpunkt „instantiate Simulation“ aus Abbildung 4.2 möglich. Hat der Benutzer die Eingabe mit Enter bestätigt, wechselt die Darstellung zum SIMULATION-Menü (vergleiche auch Tabelle 3.21).

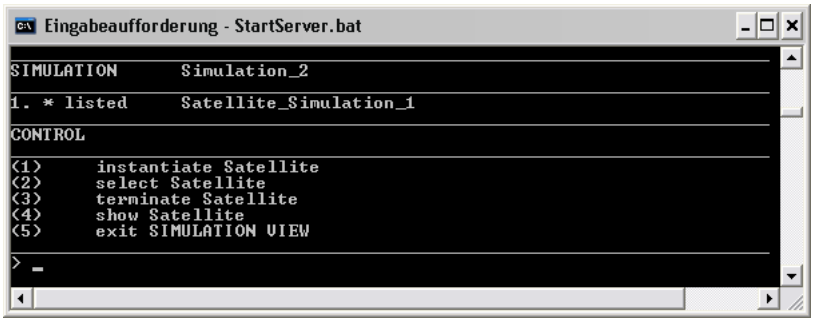


Abbildung 4.3.: Simulationsmenü

Wie in Abbildung 4.3 zu sehen ist, wurde für die Simulation nur ein Satellit eingetragen. Es könnten hier aber auch beliebig viele Satelliten auftauchen, abhängig von

den Daten in der Konfigurationsdatei. Es muss aber mindestens ein Satellit instanziiert werden, damit eine Simulation durchgeführt werden kann. Dazu gibt der Benutzer wieder eine „1“ für „instantiate Satellite“ ein und wechselt damit zum Menü SATELLITE SIMULATION.

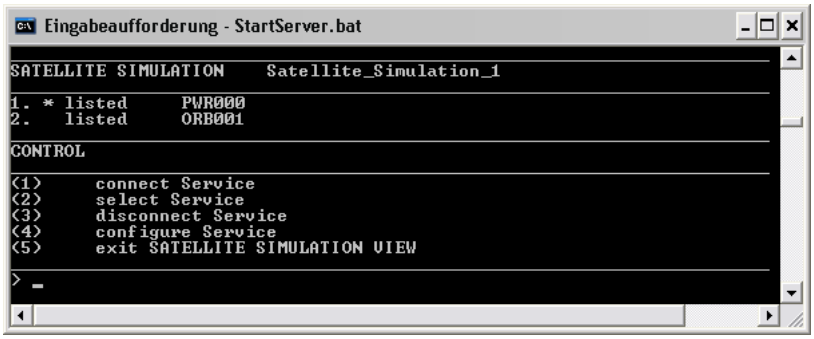


Abbildung 4.4.: Satellitensimulationsmenü

In dem Beispiel aus Abbildung 4.4 sind zwei Services aufgelistet, welche zu der Satellitensimulation „Satellite_Simulation_1“ gehören. Von diesen beiden Services muss nun mindestens einer beim Server angemeldet werden. Um einen ausgewählten Service anzumelden, muss der Benutzer zunächst die Option „connect Service“ auswählen. Daraufhin kommt eine Meldung, welche den Benutzer auffordert, den entsprechenden Service zu starten. In dem Beispiel handelt es sich um den Service PWR000. Diesen Service startet der Benutzer durch den Aufruf der Batch-Datei *StartPWR000.bat* für Windows oder des Shell-Scriptes *StartPWR000.sh* für Linux.

Ist ein anderer Port als 4444 in der Konfigurationsdatei angegeben oder befindet sich der Service nicht auf dem gleichen Computer wie das SMASS-Framework, so muss dies vorher in der Startdatei geändert werden. Dazu wurden die Variablen *myHost* und *myPort* in den Dateien *Start<Service-Token>.bat* bzw. *Start<Service-Token>.sh* angelegt.

Nachdem der Service PWR000 gestartet ist, ändert sich die Statusanzeige dieses Services von „listed“ in Abbildung 4.4 auf „connected“. Damit ist die Beispielsimulation erstellt und der Benutzer muss jetzt zum Starten der Simulation wieder zurück in das Hauptmenü. Dazu muss zunächst das Satellitensimulationsmenü aus Abbildung 4.4 durch den Menüpunkt fünf verlassen und in das Simulationsmenü aus Abbildung 4.3 zurückgekehrt werden. Auch diese Menü muss verlassen werden, was ebenfalls durch den fünften Menüpunkt möglich ist.

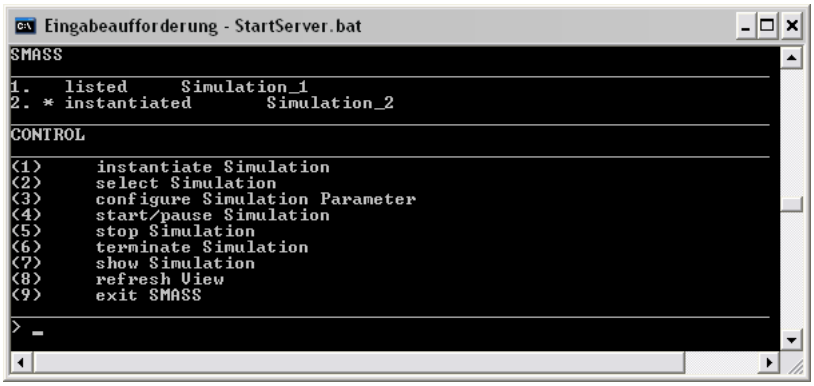


Abbildung 4.5.: Hauptmenü nachdem eine Simulation instanziiert wurde

Im Hauptmenü angelangt (siehe Abbildung 4.5), ist die Statusanzeige der Simulation auf „instantiated“ geändert worden. Durch die Option „start/pause Simulation“ kann der Benutzer nun die Simulation starten. Dadurch wechselt die Statusanzeige der Simulation von „instantiated“ auf „running“. Mit Hilfe der achten Option „refresh View“ des Hauptmenüs kann der Benutzer überprüfen, ob die Simulation bereits beendet ist, oder immer noch läuft. Ist sie beendet, wechselt die Statusanzeige wieder zu „instantiated“.

Die Ergebnisse der Beispielsimulation befinden sich in dem Ordner, wo sich auch die Konfigurationsdatei *Configuration.xml* befindet. Dort ist ein Ordner namens *Simulation_1* erstellt worden, der die in Kapitel 3.4.4 vorgestellte Struktur besitzt.

Nachdem eine Simulation beendet ist, könnte sie mit der vierten Option aus dem Hauptmenü wiederholt werden. Vorher ist es möglich, die Werte für die Simulationszeitsteuerung der Simulation und die CONFIG-Nachricht für die Services zu verändern. Ersteres kann aus dem Hauptmenü in Abbildung 4.5 mit der Option „configure Simulation Parameter“ gemacht werden. Daraufhin wird ein Menü angezeigt, indem die Parameter mit dem Namen, gefolgt von dem Wert und falls vorhanden der Einheit dargestellt werden (Abbildung 4.6).

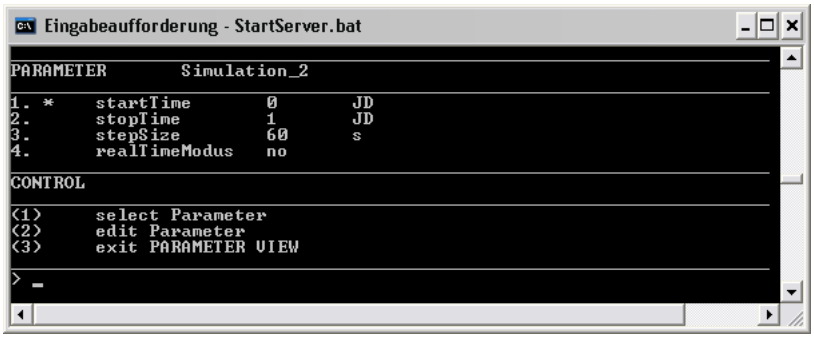


Abbildung 4.6.: Menü zur Editierung von Parameterwerten

In dem Parametermenü kann der Benutzer mit „select Parameter“ einen Parameter auswählen. Der Stern wandert dann vor den gewählten Parameter. Mit „edit Parameter“

ist es möglich den Wert des gewählten Parameters zu verändern. Die letzte Option beendet das Menü und es wird wieder das Hauptmenü angezeigt.

Analog dazu können auch die CONFIG-Werte eines Services bearbeitet werden. Dazu muss der Benutzer vom Hauptmenü aus Abbildung 4.5 mit der Option „show Simulation“ in das Simulationsmenü und von da aus mit „show Satellite“ in das Satellitensimulationsmenü gelangen. Wählt der Benutzer in dem Satellitensimulationsmenü (siehe Abbildung 4.4) die Option „configure Service“ erscheint wieder ein Menü wie in Abbildung 4.6, indem die Parameter des jeweiligen Services aufgelistet sind.

4.4. Quelltext ändern

Der Quelltext und die mit Javadoc erstellte Dokumentation des Frameworks und der beiden Services ORB001 und PWR000 ist in dem Verzeichnis *Code* auf der CD hinterlegt (siehe Anhang D). In diesem Verzeichnis befinden sich auch Batch-Dateien für Windows und Shell-Skripte für Linux, die eine neue Übersetzung des Quelltextes in Maschinensprache und anschließende Erstellung der **.jar*-Dateien ermöglichen. So können kleine Änderungen schnell und einfach mit einem einfachen Texteditor in den Quelltextdateien **.java* vorgenommen werden, ohne erst professionelle Entwicklungstools wie z.B. Eclipse zu installieren.

Werden die erstellten Kompilierungsdateien *Compile*.bat* für Windows bzw. *Compile*.sh* für Linux verwendet, muss die Verzeichnisstruktur innerhalb des *Framework*-, *ORB001*- und *PWR000*-Ordners wegen der Pfadangaben erhalten bleiben. Innerhalb der drei Ordner befindet sich jeweils ein *build*-Ordner, indem nach der neuen Übersetzung des Quelltextes die Archive **.jar* abgelegt werden. Die folgende Tabelle listet die Kompilierungsdateien auf und ordnet sie den von ihnen erstellten Archiven zu.

Ordner	****.bat* / *.sh*-Datei**	**erstelltes Archiv**	**Beschreibung**
Framework	*CompileServer.**	*Server.jar*	erstellt die Serverapplikation
	*CompileService.**	*Service.jar*	erstellt die Bibliothek, welche von den Services eingebunden und genutzt wird
ORB001	*CompileORB001.**	*ORB001.jar*	erstellt die Applikation zur Berechnung des Orbits
PWR000	*CompilePWR000.**	*PWR000.jar*	erstellt die Applikation zur Berechnung des Energiehaushaltes

Tabelle 4.1.: Kompilierungsdateien für SMASS

Für größere Änderungen oder Umbauten am System empfiehlt sich aber die Verwendung einer Entwicklungsumgebung. Für die Programmiersprache Java hat sich dabei Eclipse durchgesetzt (siehe Anhang A.2).

5. Einbinden eines neuen Services

Wenn ein neuer Service entwickelt wird, muss er in jedem Fall die beiden Klassen *<Service-Token>Application* und *<Service-Token>Implementation* beinhalten. Diese beiden Klassen setzen die in Kapitel 3.3.1 vorgestellten Schnittstellen zum Framework um. Darüber hinaus muss die Bibliothek mit der Kommunikation für die Services eingebunden werden, welche für Java *Service.jar* lautet. Damit die ersten Tests durchgeführt werden können, muss anschließend in der Konfigurationsdatei ein *service*-Element (siehe Kapitel 3.4.2) hinzugefügt werden.

Für die Erstellung eines neuen Services in Java gibt es auf der CD (siehe Anhang D) eine Vorlage in dem Ordner *NewJavaService*. Im Folgenden soll anhand des Services für die Energieversorgung (PWR000), welcher auf den Simple Power Simulator V2.1 von Prof. Dr. Kayal beruht und an das Framework angepasst wurde, das Einbinden eines neuen Services unter Verwendung der Vorlage Schritt für Schritt erklärt werden. Die Tabelle 5.1 listet diese Schritte auf und kann als Checkliste verwendet werden:

Schritt	**Beschreibung**
1	Kopieren der Vorlage und umbenennen des Ordners
2	Umbenennen der Klassen zur Schnittstellenimplementierung
3	Bearbeiten des Eintrittspunkts
4	Implementieren der Servicefunktionalität
5	Umbenennen und Bearbeiten der *Compile**-Dateien
6	Umbenennen und Bearbeiten der *Start**-Dateien
7	Eintragen des Services in die Konfigurationsdatei

Tabelle 5.1.: Schritte zur Erstellung eines neuen Services

5.1. Schritt 1 - Kopieren der Vorlage und Umbenennen des Ordners

Zunächst kopiert der Serviceentwickler die Vorlage in ein beliebiges Verzeichnis und benennt dieses eindeutig, z.B. mit dem Service-Token seines Services. Im Beispiel wurde der Ordner *NewJavaService* demnach in *PWR000* umbenannt. Die Ordnerstruktur in diesem Ordner ist in der Abbildung 5.1 dargestellt.

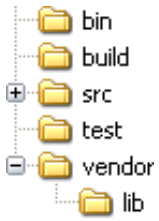


Abbildung 5.1.: Ordnerstruktur in *NewJavaService*

Durch die einheitliche Struktur für die Services ist es überhaupt erst möglich Vorlagen für das Übersetzen des Quelltextes und das Erstellen der **.jar*s (siehe auch Kapitel 4.4) anzulegen. Außerdem führt eine einheitliche Struktur dazu, dass die Wartung von Services in Zukunft sehr viel einfacher ist. Es ist klar strukturiert, wo welche Art von Dateien liegen. Tabelle 5.2 listet die Dateien aus den jeweiligen Ordnern auf.

Ordner	Inhalt	Beschreibung
bin	**.class*	Quelltextdateien in Maschinensprache
build	*<Service-Token>.jar*, *Start<Service-Token>.**	erstelltes ausführbares Servicearchiv und die zugehörigen Startdateien für Linux und Windows
src	**.java*, Java-Pakete	lesbare Quelltextdateien und weitere Java-Pakete
test	*Test.java, Java-Pakete	Testdateien und Pakete für Modultests
vendor\lib	*Service.jar*, **.jar*	benötigte Bibliotheken

Tabelle 5.2.: Inhalt der Ordnerstruktur in *NewJavaService*

5.2. Schritt 2 - Umbenennen der Klassen zur Schnittstellenimplementierung

Wie aus Tabelle 5.2 hervorgeht, befinden sich in dem Ordner *src* die Java-Pakete. In diesem Ordner existiert das Paket *services*, welches ein Unterpaket namens *NewService_implementation* besitzt. In dem Paket befinden sich die zwei Dateien mit den Vorlagen für die Klassen zur Implementierung der Schnittstelle. Diese müssen zusammen mit dem Paket *NewService_implementation* nun ebenfalls umbenannt werden. Dabei wird jeweils *NewService_* mit dem Service-Token ersetzt. Für den Service PWR000 lauten das Paket und die Dateien nach der Umbenenung *PWR000implementation*, *PWR000-Application.java* und *PWR000Implementation.java*.

5.3. Schritt 3 - Bearbeiten des Eintrittspunkts

PWR000Application.java stellt den Eintrittspunkt für die Serviceanwendung dar. In dieser Datei muss nur die Zeichenkette „<Service-Token>“ durch den aktuellen Service-

Token ersetzt werden. Für das Beispiel lautet dieser PWR000, sodass die Datei *PWR000-Application.java* folgendermaßen aussieht:

```
1 package services.PWR000implementation;
2
3 import services.serviceapplications.AServiceApplication;
4 import utilities.communication.IServiceMessageHandler;
5 import utilities.communication.handling.ServiceMessageHandler;
6
7 public final class PWR000Application extends AServiceApplication {
8
9     private PWR000Application() {
10        super();
11    }
12
13    public static void main(final String[] args) {
14        try {
15            PWR000Implementation implementation
16                    = new PWR000Implementation();
17            IServiceMessageHandler serviceMessageHandler
18                    = new ServiceMessageHandler(args[0], Integer
19                            .parseInt(args[1]), "PWR000",
20                                    implementation);
21            serviceMessageHandler.process();
22        } catch (ArrayIndexOutOfBoundsException ae) {
23            System.out.println("Argument for main method "
24                    + "of service is missing!");
25            System.out.println("Call should look like:");
26            System.out.println("java PWR000Application <host> "
27                    + "<port number>");
28        }
29    }
30 }
```

In diesem Programmausdruck, wie auch in den folgenden, wurden die Kommentare der Übersichtlichkeit halber weggelassen. Diese sind aber in der originalen Datei vorhanden und sollten ebenfalls angepasst werden. Für Java empfiehlt sich das Werkzeug Javadoc zu verwenden. Die nachfolgende Internetseite gibt einige Hinweise zur Gestaltung von Kommentaren mit Javadoc.

`http://java.sun.com/j2se/javadoc/writingdoccomments/`

5.4. Schritt 4 - Implementieren der Servicefunktionalität

In der Datei *PWR000Implementation.java* wird die Funktionalität des Services umgesetzt. Dazu implementiert die zugehörige Klasse das Interface *IService* (siehe Kapitel 3.3.1). In der Vorlage wird für den Service zur Energieversorgung zunächst wieder die Zeichenkette „<Service-Token>“ durch PWR000 ersetzt.

Anschließend werden die beiden Methoden *calculate* und *setConfiguration* mit Code gefüllt. Die Methode *setConfiguration* muss eine Variable vom Typ *String* zurückgeben. Wenn die Daten zur Konfiguration korrekt waren, muss der *String* „ok“ lauten. Dabei sind alle möglichen Kombinationen der Groß- und Kleinschreibung zulässig.

Die dritte Methode *getConfiguration* ist in der Vorlage bereits fertig gestellt und muss nicht weiter bearbeitet werden. Nachstehend ist die *Implementation*-Klasse für den Service zur Energieversorgung dargestellt.

```
1 package services.PWR000implementation;
2
3 import services.serviceapplications.IService;
4 import utilities.communication.message.DataObject;
5
6 public class PWR000Implementation implements IService {
7     // weitere Variablendeklarationen
8     private DataObject myConfiguration = null;
9
10    public PWR000Implementation() {
11    }
12
13    public final DataObject calculate(final DataObject parameters) {
14        // Quelltext zum Berechnen des aktuellen Ladestandes, welcher in result
15        // gespeichert wird
16        return new DataObject(result);
17    }
18
19    public final DataObject getConfiguration() {
20        return myConfiguration;
21    }
22
23    public final String setConfiguration(final DataObject configuration) {
24        // Quelltext zum Überprüfen der Konfigurationsdaten, Setzen der
25        // Konfiguration und Erstellen der Antwort
26        return response;
27    }
28 }
```

Bei der Implementierung sollten natürlich auch schon Tests vorgesehen werden. In Java gibt es dafür das Framework JUnit. Dieses Framework erleichtert die Erstellung und das Ausführen von automatischen Modultests. Die Testdateien und -pakete können in den Ordner *test* aus Abbildung 5.1 gespeichert werden.

5.5. Schritt 5 - Umbenennen und Bearbeiten der *Compile**-Dateien

Nachdem der Quelltext implementiert ist, kann dieser nun übersetzt und in ein ausführbares Archiv gepackt werden. Wird keine Entwicklungsumgebung benutzt, kann dies durch anpassen der *Compile_ NewService.bat* für Windows oder *Compile_ NewService.sh* für Linux erreicht werden. Diese beiden Dateien befinden sich in dem kopierten Ordner, der vormals *NewJavaService* hieß. In dem Beispiel handelt es sich um ein Windows-Betriebssystem, weshalb die Batch-Datei benutzt wird. Diese wird zunächst umbenannt in *CompilePWR000.bat*. Anschließend wird sie mit einem Texteditor bearbeitet. Der Variablen *myServiceToken* in Zeile 14 wird der Service-Token zugeordnet, sodass die Zeile in dem Beispiel wie folgt aus:

```
13 ...
14 set myServiceToken=PWR000
15 ...
```

Für den Fall, dass ein Linux-System verwendet wird, muss die Datei *Compile_NewService.sh* umbenannt werden. Auch in dieser Datei wurde eine Variable angelegt, welche mit dem Service-Token gefüllt werden muss. Diese Variable steht in Zeile 15 und ist im Anschluss dargestellt.

```
14 ...
15 myServiceToken=PWR000
16 ...
```

5.6. Schritt 6 - Umbenennen und Bearbeiten der *Start**-Dateien

Damit der erstellte Service auch ausgeführt werden kann, muss nun noch die Datei zum Starten des Services bearbeitet werden. Die Linux- und die Windows-Variante befinden sich in dem Ordner *build* aus Abbildung 5.1. Da in dem Beispiel für den Service zur Berechnung des Energiehaushalts ein Windows-Betriebssystem verwendet wird, muss die Datei *Start_NewService.bat* umbenannt werden. Auch sie erhält wieder den Service-Token als Teil des Namens. Nach der Umbenennung lautet die Datei im Beispiel *StartPWR000.bat*.

Analog zu der Datei zum Übersetzen und Archivieren aus dem vorherigen Schritt muss auch hier noch die Variable für den Service-Token gesetzt werden. Diese steht in der Zeile zehn von *StartPWR000.bat*.

Ebenso analog zu Kapitel 5.5 kann die Linux-Variante für die Startdatei bearbeitet werden. Nach dem Umbenennen muss auch hier wieder der Service-Token der Variablen zugewiesen werden. Diese Variable befindet sich in Zeile elf des Shell-Skriptes.

5.7. Schritt 7 - Eintragen des Services in die Konfigurationsdatei

Um den Service jetzt zu benutzen, muss ein *service*-Element einer Satellitensimulation in der Konfigurationsdatei hinzugefügt werden. Dabei wird der Service-Entwickler von der XML-Schema-Definition *smass_config_schema.xsd* unterstützt, sofern sein Editor die Validierung mit Hilfe eines Schemas unterstützt.

Bevor der Eintrag in die XML-Datei vorgenommen werden kann, muss zunächst der neue Service-Token als zulässiger Wert für das Attribut *serviceToken* in das Schema eingetragen werden. Die Angaben zu diesem Attribut befinden sich in den Zeilen 126 bis 133 der Datei *smass_configschema.xsd*. Nachfolgender Ausdruck zeigt die betreffende Stelle in der XSD-Datei nach dem Eintrag des neuen Service-Tokens.

```
124 ...
125     <xs:attribute name="serviceToken" use="required">
126         <xs:simpleType>
127             <xs:restriction base="xs:string">
128                 <xs:enumeration value="ORB001"/>
129                 <xs:enumeration value="PWR000"/>
130             </xs:restriction>
131         </xs:simpleType>
132     </xs:attribute>
133 ...
```

Das Service Element muss die drei vorgeschriebenen Attribute aus Tabelle 3.15 besitzen. Darüber hinaus hat der Service die vier Parameter *capacity_of_batteries*, *consumed_power*, *bus_voltage* und *solar_array_power*.

Der Service berechnet den Ladezustand der Batterien. Dieses Ergebnis wird in den *result*-Tag des Service-Elementes eingetragen.

Vom Service zur Orbitberechnung benötigt PWR000 die Information über Schatten oder Sonne. Dieser Parameter wird im *calculate*-Tag eingetragen.

Das vollständige *service*-Element für PWR000 sieht nach Berücksichtigung von Kapitel 3.4.2 wie folgt aus:

```
...
  ...
    ...
      <service name="Power" serviceToken="PWR000" simulationPosition="1">
        This service is a re-implementation of Prof. Dr. Kayals Simple Power
        Simulator V2.1.
        <parameter name="capacity_of_batteries" symbol="C" unit="Ah"
            value="4">
          This value defines the initial and maximum capacity of all
          batteries.
        </parameter>
        <parameter name="consumed_power" symbol="P" unit="W" value="4">
          This value defines the mean consumed power.
        </parameter>
        <parameter name="bus_voltage" symbol="U" unit="V" value="5">
          This value defines the bus voltage of the satellite.
        </parameter>
        <parameter name="solar_array_power" symbol="P" unit="W" value="8">
          This value defines the mean power of the solar arrays, if they
          are in the sun.
        </parameter>
        <result>
          <parameter name="capacity_of_batteries" position="1" symbol="C"
              unit="Ah">
            As result, this service calculates the charge status of the
            batteries.
          </parameter>
        </result>
        <calculate>
          <parameter name="eclipse" position="1" fromService="ORB001"
              initialValue="1" unit="">
            The power service needs the information, if the satellite is
            in the Earth shadow or not.
          </parameter>
        </calculate>
      </service>
    ...
  ...
...
```

Nachdem nun alle Vorbereitungen getroffen wurden, kann die erste Simulation mit dem neuen Service erfolgen. Die Vorgehensweise zum Durchführen einer Simulation wurde in Kapitel 4.3 erläutert. Sollte die Erwartung an die Ergebnisse nicht erfüllt werden und die Ursache in dem Quelltext liegen, so sind die Schritte aus Kapitel 4.4 zu befolgen.

6. Tests

6.1. Funktionstest mit ORB001

Der erste Test ist als reiner Funktionstest gedacht. In diesem Test wird eine Simulation mit einem Satelliten angelegt. Für diesen Satelliten wird ausschließlich der an das Framework angepasste Orbitservice angemeldet. Die Ergebnisse werden mit den Originaldaten des Bahnmodells von Giese ([Gie05]) verglichen.

Für den Test wurde ein nahezu kreisförmiger Orbit verwendet, der sich auf einer geostationären Bahn bewegt. Dieser Orbit wird für den Zeitraum von drei Tagen von 00:00 Uhr des 01.01.2004 bis 00:00 Uhr des 04.01.2004 simuliert. Als Bodenstation wurde wie in den Originaldaten das ILR der TU-Berlin angegeben. Nachstehende Daten wurden für die Simulation in die Konfigurationsdatei eingetragen:

	Name	**Wert**	**Einheit**
ORB001	große Halbachse	42164,2	km
	Exzentrizität	0	-
	Inklination	0	°
	Länge des aufsteigenden Knotens	112,946	°
	Perigäumsabstand	0	°
	Wahre Anomalie	0	°
	Epoche	2453005,5	JD
	Radius der Bodenstation	6364,803	km
	Geografische Länge der Bodenstation	13,326	°
	Geografische Breite der Bodenstation	52,517	°
Simulationszeit	Startzeit	2453005,5	JD
	Stoppzeit	2453008,5	JD
	Schrittweite	240	s
	Realzeitmodus	nein	-

Tabelle 6.1.: Eingangsdaten für den Funktionstest mit ORB001

Bei der Anpassung des Orbitservices wurde auf die Berechnung der Daten verzichtet, welche nur für die grafische Ausgabe in dem Original verwendet wurden. Das Diagramm in Abbildung 6.1 zeigt nun den relativen Fehler, bezogen auf die Originaldaten, einer Auswahl von Ergebnisparametern.

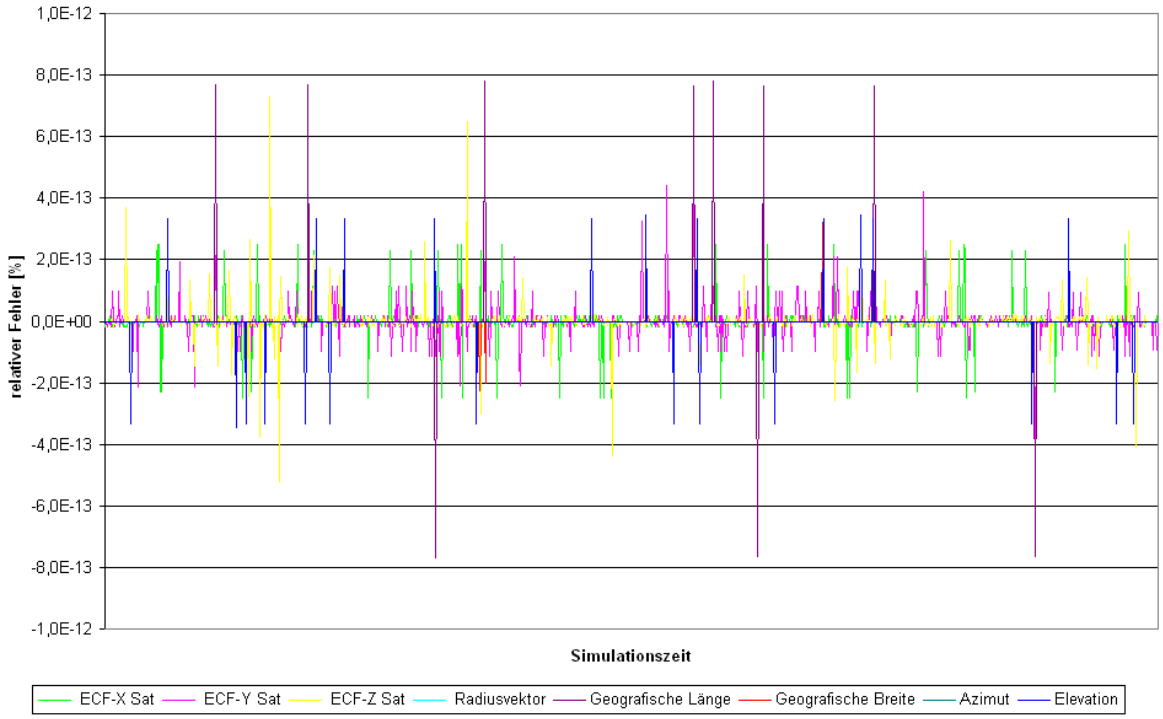


Abbildung 6.1.: Relativer Fehler zwischen Daten des originalen und des angepassten Services

In dem Diagramm zeigt sich, dass SMASS grundsätzlich funktioniert. Die Berechnungen des angepassten Services weichen um maximal $8 \cdot 10^{-13}\%$ ab. Umgerechnet auf den verwendeten Orbit würde das bei einer Größe von $42000km$ in einer der drei ECF Richtungen gerade einmal $4 \cdot 10^{-4}mm$ ausmachen.

Die Ursache der Differenz konnte nicht exakt ermittelt werden. Es wird vermutet, dass sie bereits im Bereich der Rechengenauigkeit in Java oder Excel, mit dem dieses Diagramm erstellt wurde, liegt. Eine weitere Ursache könnte sein, dass diese Daten im neuen Framework in SI-Einheiten umgerechnet werden, bevor sie in den Dateien gespeichert werden.

Darüber hinaus kommt durch die Simulationszeitsteuerung zwangsläufig ein Fehler in die Berechnung. Die Schrittweite wird in Sekunden angegeben, während die Start- und Stoppzeit in Tagen angegeben wird. Bei der Umrechnung von Sekunden auf Tagen muss durch $86400s/d$ geteilt werden. Der Bruch $1/86400$ ist aber ein periodischer Bruch, welcher nur mit begrenzter Genauigkeit dargestellt werden kann. Die Simulationszeit für jeden Schritt, welche sich aus der Addition der Schrittweite auf die vorherige Simulationszeit ergibt, wird aber zur Berechnung der Orbitdaten verwendet.

Durch diesen ersten Test können auch die folgenden Funktionen und Bestandteile von SMASS nachgewiesen werden, ohne die der Service nicht die Ergebnisse hätte berechnen können:

- Schema für XML-Datei
- Einlesen der Konfigurationsdatei
- Anmelden eines Services
- Versenden und Empfangen von Nachrichten
- Steuerung der Simulationszeit

6.2. Test des Datenaustauschs zwischen ORB001 und PWR000

Der zweite Test untersucht den Austausch von Daten zwischen den Services. Dazu wurde eine neue Simulation angelegt, welche einen Satelliten mit dem ORB001- und dem PWR000-Service beinhaltet. Wie bereits erwähnt, benötigt der Service PWR000 vom Orbitservice ORB001 die Information, ob der Satellit sich im Erdschatten befindet. Es wurde diesmal ein Orbit im LEO verwendet, der eine nahezu sonnensynchrone Bahn besitzt. Die Länge des aufsteigenden Knotens wurde so gewählt, dass im Simulationszeitraum vom 01.01.2004 bis 04.01.2004 in etwa 63% der Bahn in der Sonne liegen. Alle Eingangsdaten sind in der folgenden Tabelle dargestellt:

	Name	**Wert**	**Einheit**
ORB001	große Halbachse	7000	km
	Exzentrizität	0	-
	Inklination	98	°
	Länge des aufsteigenden Knotens	112,946	°
	Perigäumsabstand	0	°
	Wahre Anomalie	0	°
	Epoche	2453005,5	JD
	Radius der Bodenstation	6364,803	km
	Geografische Länge der Bodenstation	13,326	°
	Geografische Breite der Bodenstation	52,517	°
PWR000	Anfangs- und Maximalkapazität der Batterien	4	Ah
	Verbrauchte Energie	4	W
	Busspannung	5	V
	Generierte Leistung aus den Solarzellen	8	W
Simulationszeit	Startzeit	2453005,5	JD
	Stoppzeit	2453008,5	JD
	Schrittweite	60	s
	Realzeitmodus	nein	-

Tabelle 6.2.: Eingangsdaten für den Test zum Datenaustausch

Mit den Daten aus Tabelle 6.2 und der Festlegung der Simulationsreihenfolge derart, dass zunächst ORB001 und im Anschluss PWR000 berechnet wird, ergibt sich das Diagramm aus Abbildung 6.2 für die ersten zwölf Stunden des Simulationzeitraums.

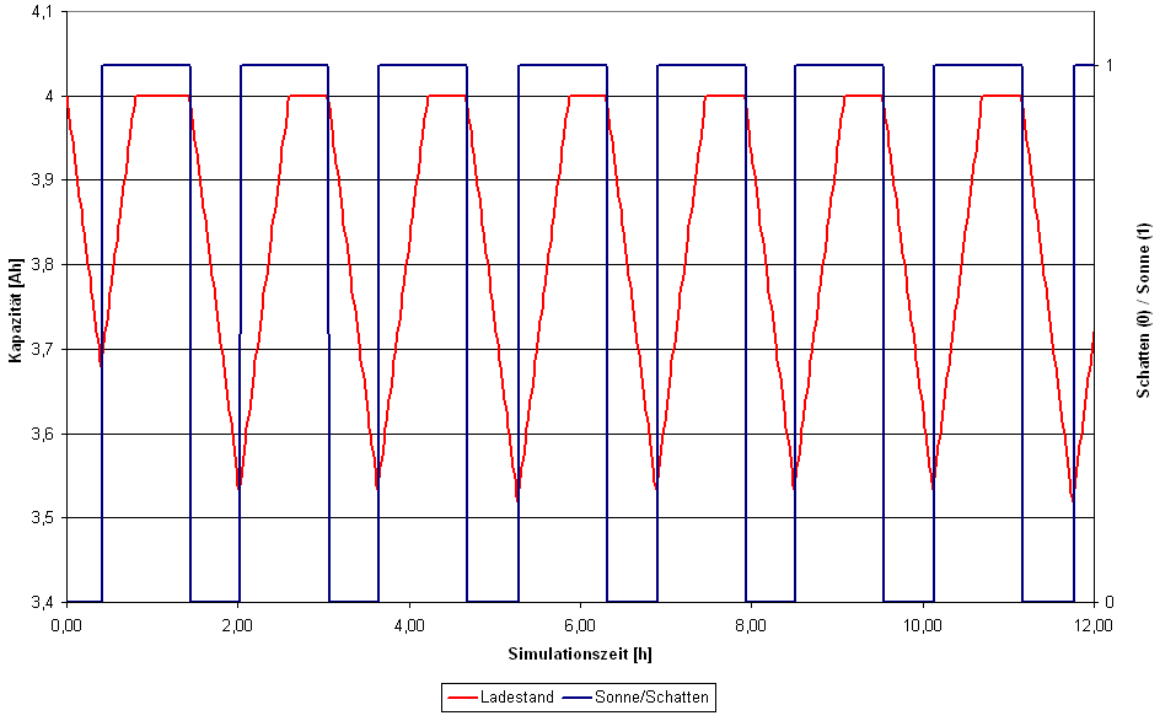


Abbildung 6.2.: Datenaustausch zwischen ORB001 und PWR000

An dem Diagramm kann nachvollzogen werden, dass im Moment des Schatteneintritts des Satelliten der Ladestand der Batterien sinkt. Sobald aber der Satellit wieder in die Sonne gelangt, laden sich die Batterien wieder bis zu ihren Maximum auf. Es ist somit gewährleistet, dass die Berechnung des Schattens aus dem Orbitservice korrekt an den Service zur Berechnung des Energiehaushaltes weitergeleitet wird.

Für den Fall, dass die Simulationsreihenfolge der Services umgekehrt wäre, würde der Ladestand der Batterien immer einen Simulationsschritt hinterherhängen. Das würde bei der gegebenen Schrittweite von $60s$ bedeuten, dass PWR000 immer erst eine Minute nach dem Ereignis die Information über Sonne oder Schatten bekommen würde. Dieses Beispiel zeigt noch mal deutlich den Einfluss der Simulationsreihenfolge auf die Aktualität der Daten, was bereits in Kapitel 3.1.4 angesprochen wurde.

6.3. Test der „schwachen Echtzeit“

In diesem Test geht es um die Überprüfung der Option zur „schwachen Echtzeit“. Dazu wurden die Zeitstempel für den Aufruf der Methode *calculate* aus den Services ausgegeben und mit den erwarteten verglichen. Zusätzlich wurden noch die Zeitstempel aus

der *Simulation*-Klasse ausgegeben, damit die Laufzeit bis zum ersten Service gemessen werden kann. Für diesen Test wurde eine Simulation mit zwei Satelliten angelegt. Bei der ersten Satellitensimulation werden sowohl der Orbit- als auch der Energieservice angemeldet. Bei der zweiten Satellitensimulation wurde nur der Orbitservice angelegt. Die Eingangswerte für die Simulation änderten sich im Vergleich zu denen aus dem vorherigen Testfall (Tabelle 6.2) nur im Bereich der Zeitsteuerung.

	Name	**Wert**	**Einheit**
ORB001 (Satellit 1 und 2)	große Halbachse	7000	km
	Exzentrizität	0	-
	Inklination	98	°
	Länge des aufsteigenden Knotens	112,946	°
	Perigäumsabstand	0	°
	Wahre Anomalie	0	°
	Epoche	2453005,5	JD
	Radius der Bodenstation	6364,803	km
	Geografische Länge der Bodenstation	13,326	°
	Geografische Breite der Bodenstation	52,517	°
PWR000 (Satellit 1)	Anfangs- und Maximalkapazität der Batterien	4	Ah
	Verbrauchte Energie	4	W
	Busspannung	5	V
	Generierte Leistung aus den Solarzellen	8	W
Simulationszeit	Startzeit	2453005	JD
	Stoppzeit	2453005,02	JD
	Schrittweite	3	s
	Realzeitmodus	ja	-

Tabelle 6.3.: Eingangsdaten für den Test zur „schwachen Echtzeit“

Für die Ausgabe der Zeitstempel in den Services wurde die Klasse *Date* aus dem Paket *java.util* des JDKs verwendet. In dieser Klasse ist die Methode *getTime()* implementiert, welche die Millisekunden seit 00:00 Uhr GMT am 01.01.1970 ausgibt. In den Messwerten zur „schwachen Echtzeit“ wurden Abweichungen im Millisekundenbereich zu den erwarteten Werten festgestellt (siehe Abbildung 6.3).

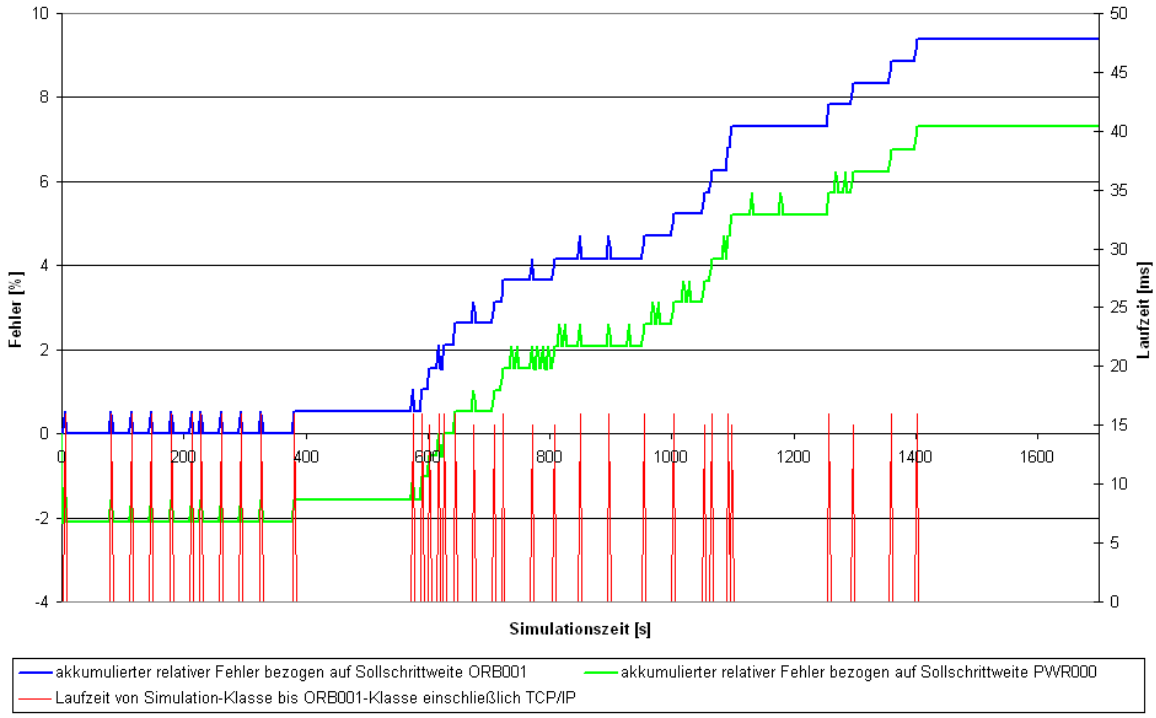


Abbildung 6.3.: Test der „schwachen Echtzeit“ - Satellit 1

In dem Diagramm sind die akkumulierten relativen Fehler der Schrittweite in Prozent des Orbit- und Energieservices für die ersten 1700s dargestellt. Eine Abweichung von 100% würde einem Wert von 3s Abweichung entsprechen. Dazu ist mit der roten Kurve die Laufzeit ausgehend vom Aufruf zur Berechnung innerhalb des ersten Satelliten in der *Simulation*-Klasse bis zum Aufruf der *calculate*-Methode in der *ORB001*-Klasse gezeigt.

Es fällt zunächst einmal auf, das die akkumulierten Fehler für beide Services zwar grundsätzlich die gleiche Tendenz haben, die Kurve für den PWR000 allerdings einen negativen Offset hat. Dies ist mit der Initialisierungsphase und dem ersten Berechnungsschritt zu erklären. Dieser verursacht größere Verzögerungen zwischen der Berechnung in ORB001 und PWR000 als die nachfolgenden Simulationsschritte. Dadurch wird die *calculate*-Methode des Energieservices (PWR000) beim zweiten Mal bereits vor dem Ablauf von der Schrittweite von drei Sekunden aufgerufen. Als Ursache hierfür wurde die Positionierung der Echtzeitsteuerung in der *Simulation*-Klasse identifiziert. Dadurch ist es nur möglich, eine Mindestzeit für den ersten Service eines Satelliten zu gewähren. Da die Berechnung innerhalb dieses Services aber nicht immer exakt die gleiche Zeit in Anspruch nimmt, kann es für den zweiten Service auch zu negativen Abweichungen von der Sollschrittweite kommen. Diese liegt in dem Beispiel aus Abbildung 6.3 aber nur bei ca. 2%, was bei einer Schrittweite von 3s einer Größe von 60ms entspricht.

Die Ausschläge und Treppen in den Kurven zu den relativen Fehlern lassen sich mit der TCP/IP-Verbindung erklären. Dafür ist die Laufzeit für den Orbitservice in dieses Diagramm mit aufgenommen worden. Immer dann, wenn die Laufzeit mehr als eine hal-

be Millisekunde beträgt und damit messbar ist, gibt es einen Ausschlag bzw. eine Stufe in der Kurve zum akkumulierten Schrittweitenfehler. Dies bestätigt, dass die Nachrichtenlaufzeit für eine TCP/IP-Verbindung nicht vorhersagbar ist ([Nat04] S.33f) und sich deshalb nicht für Echtzeit eignet. Darüber hinaus ist die Laufzeit auch plattformabhängig, was in weiteren Tests nachgewiesen wurde. In Abbildung 6.3 handelt es sich um Ungenauigkeiten von ca. $15ms$, während auf anderen Plattformen Fehler von bis zu $50ms$ festgestellt werden konnten.

Anhand des Diagramms kann außerdem vermutet werden, dass sich der akkumulierte Fehler auf einen Wert einpendelt, da die Sprünge zum Ende hin weniger werden. Dies könnte mit der Prozessverwaltung des Computer zusammenhängen. Diese versucht den Ablauf permanent zu optimieren, bis eine Art eingeschwungener Zustand erreicht ist. Ähnliches wird im Bereich der TCP/IP-Kommunikation angewendet. Um diese Vermutung weiter zu untermauern, müssten allerdings noch mehr Testreihen mit mehr Stützwerten aufgenommen werden. Dies ist aber sehr zeitaufwendig, da die Simulationszeit bei aktivierter Option zur „schwachen Echtzeit“ auch der realen Zeit entspricht.

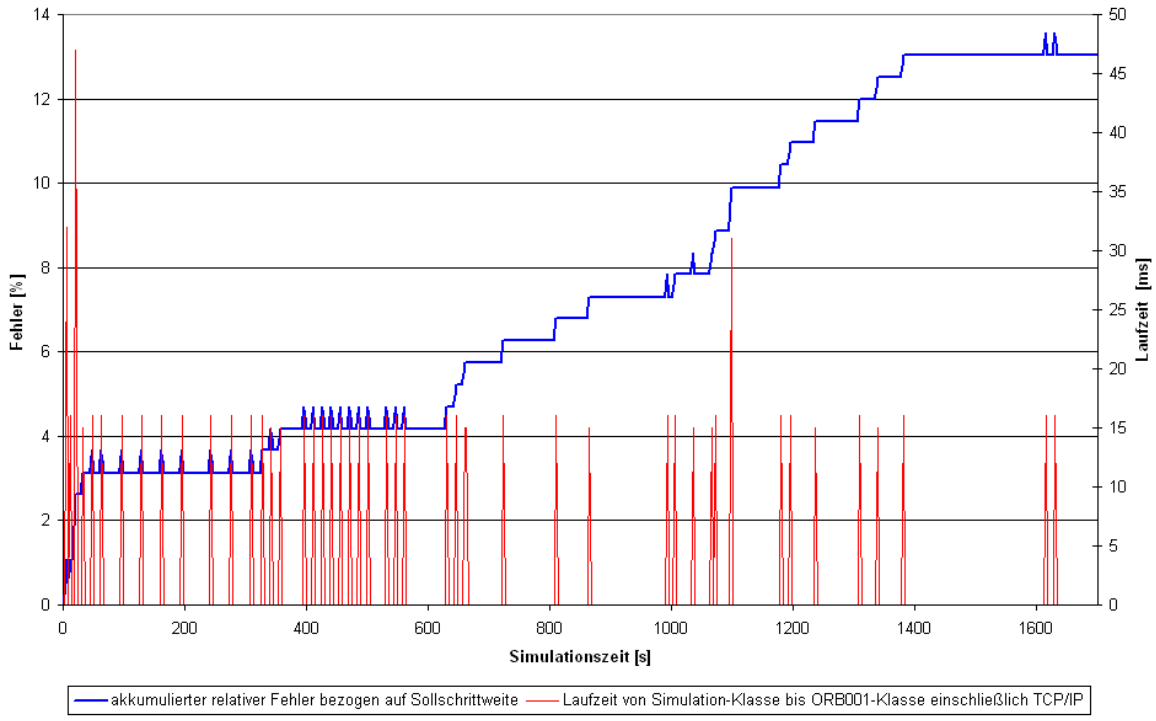


Abbildung 6.4.: Test der „schwachen Echtzeit“ - Satellit 2

Für den zweiten Satelliten konnte grundsätzlich ein ähnliches Verhalten festgestellt werden. Da es sich in der Abbildung 6.4 um den ersten und einzigen Service handelt, ist der relative Fehler positiv. Darüber hinaus sind wieder die Sprünge aufgrund der TCP/IP-Verbindung zu erkennen. An dieser Stelle sollte aber auch erwähnt werden, dass wirkliche Echtzeit auch nur mit Echtzeitbetriebssystemen erreicht werden kann.

Das Ziel der Einführung der „schwachen Echtzeit“ ist es, Mindestlaufzeiten zu gewähren. Diese können von Simulationsschritt zu Simulationsschritt auf eine Sekunde genau gewährleistet werden, wobei im Millisekundenbereich Abweichungen zu erwarten sind. Die akkumulierte Abweichung wird mit der Anzahl der Simulationsschritte steigen, wie die Abildung 6.4 zeigt. In dem gezeigten Test liegt die absolute Abweichung nach einer knappen halben Stunde für den ersten Satelliten bei ca. $0,3s$ und für den zweiten Satelliten bei ca. $0,4s$.

Zusätzlich zur „schwachen Echtzeit“ konnte mit dem Test die Funktionalität von mehreren Satelliten in einer Simulation bestätigt werden.

6.4. Mehrere parallele Simulationen

Mit SMASS ist es möglich, mehrere Simulationen parallel laufen zu lassen. Um diese Funktion zu testen, wurden zwei Simulation in der Konfigurationsdatei angelegt. Beide Simulationen mit den zugehörigen Services liefen auf einem Computer. Die erste Simulation bestand aus einem Satelliten, während die zweite Simulation zwei Satelliten beinhaltete. Zu beachten war dabei, dass drei verschiedene Ports benutzt werden. Einer für die erste Simulation und zwei für die beiden Satelliten der zweiten Simulation.

In der ersten Simulation war die „schwache Echtzeit“ aktiviert. Damit soll unter anderem untersucht werden, wie der Einfluss von parallel betriebenen Simulationen auf die Qualität der „schwachen Echtzeit“ aussieht. Die Daten für diese Simulation sind in folgender Tabelle dargestellt:

	Name	**Wert**	**Einheit**
ORB001	große Halbachse	7000	km
	Exzentrizität	0	-
	Inklination	98	°
	Länge des aufsteigenden Knotens	112,946	°
	Perigäumsabstand	0	°
	Wahre Anomalie	0	°
	Epoche	2453005,5	JD
	Radius der Bodenstation	6364,803	km
	Geografische Länge der Bodenstation	13,326	°
	Geografische Breite der Bodenstation	52,517	°
Simulationszeit	Startzeit	2453005,5	JD
	Stoppzeit	2453005,75	JD
	Schrittweite	5	s
	Realzeitmodus	ja	-

Tabelle 6.4.: Eingangsdaten für Simulation 1

Die zweite Simulation beinhaltet zwei Satelliten im LEO. Für den Orbit und die Simulationszeit wurde das Beispiel aus dem ersten Test der Diplomarbeit zum Server ([Pfe07])

verwendet. Als Simulationszeitraum wurde dort die Spanne vom 01.01.2008 12:00 Uhr bis 02.01.2008 00:00 Uhr gewählt. Für den Service zum Energiehaushalt wurde zunächst ein Satellit mit einer durchschnittlichen Leistung der Solarzellen von $8W$ verwendet. Der zweite Satellit hingegen generiert durchschnittlich $6W$ an Leistung. So können die beiden Satellitenkonfigurationen direkt miteinander verglichen werden. Tabelle 6.5 listet die Daten für die Konfigurationsdatei der zweiten Simulation auf.

	Name	**Wert**	**Einheit**
ORB001 (Satellit 1 und 2)	große Halbachse	7000	km
	Exzentrizität	0	-
	Inklination	60	°
	Länge des aufsteigenden Knotens	0	°
	Perigäumsabstand	0	°
	Wahre Anomalie	0	°
	Epoche	2454467	JD
	Radius der Bodenstation	6364,803	km
	Geografische Länge der Bodenstation	13,326	°
	Geografische Breite der Bodenstation	52,517	°
PWR000 (Satellit 1)	Anfangs- und Maximalkapazität der Batterien	4	Ah
	Verbrauchte Energie	4	W
	Busspannung	5	V
	Generierte Leistung aus den Solarzellen	8	W
PWR000 (Satellit 2)	Anfangs- und Maximalkapazität der Batterien	4	Ah
	Verbrauchte Energie	4	W
	Busspannung	5	V
	Generierte Leistung aus den Solarzellen	6	W
Simulationszeit	Startzeit	2454480,5	JD
	Stoppzeit	2454481	JD
	Schrittweite	2	s
	Realzeitmodus	nein	-

Tabelle 6.5.: Eingangsdaten für Simulation 2

Die folgende Abbildung zeigt den relativen Fehler der Schrittweite bezogen auf den Sollwert von $5s$ der ersten Simulation.

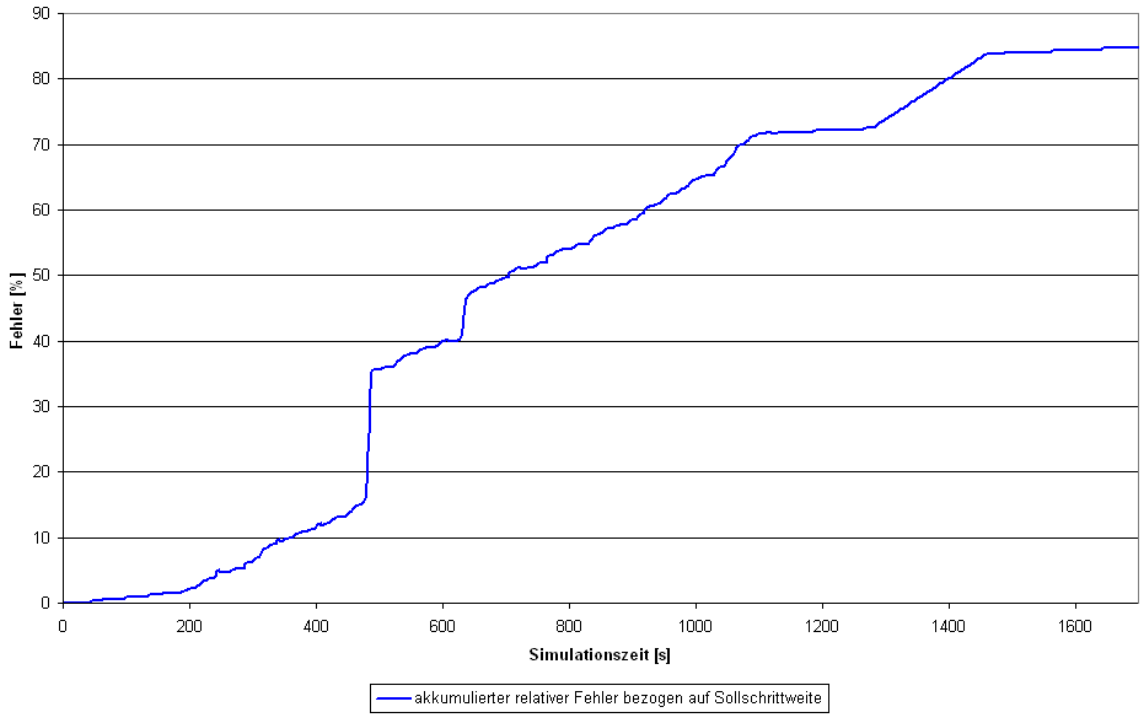


Abbildung 6.5.: relativer Fehler der Schrittweite in Simulation 1

Die erste Simulation mit aktivierter „schwacher Echtzeit“ wurde zuerst gestartet. Nach ca. $200s$ wurde die zweite Simulation gestartet. In der zweiten Simulation war die Option zur „schwacher Echtzeit“ nicht aktiviert, wodurch der Computer stark ausgelastet war. Auch die Netzwerkverbindung war durch die zweite Simulation stärker belastet, als im voherigen Test aus Kapitel 6.3. Deshalb ist der Anstieg der Fehlerrate deutlich steiler. Grundsätzlich zeigt sich aber ein ähnliches Verhalten wie in Abbildung 6.4.

Der starke Anstieg bei ca. $480s$ kann nur durch interne Prozesse verursacht worden sein. Zu diesem Zeitpunkt wurde wahrscheinlich ein Prozess mit hoher Priorität gestartet.

Anhand dieses Tests zeigt sich der Einfluss von anderen Prozessen des Computers auf die Qualität der Schrittweite. Umso stärker der Computer, auf dem der Server läuft, und die Netzwerkverbindung ausgelastet sind, umso größer wird der Fehler in der Schrittweite. Da auch alle Services auf dem einem Computer liefen, konnte demnach kein besseres Ergebnis erwartet werden. Die Abweichung nach einer halben Stunde liegt hier bereits bei $4,3s$. Eine Möglichkeit, die Genauigkeit der Schrittweiten zu erhöhen, ist das Auslagern der Services auf andere Computer.

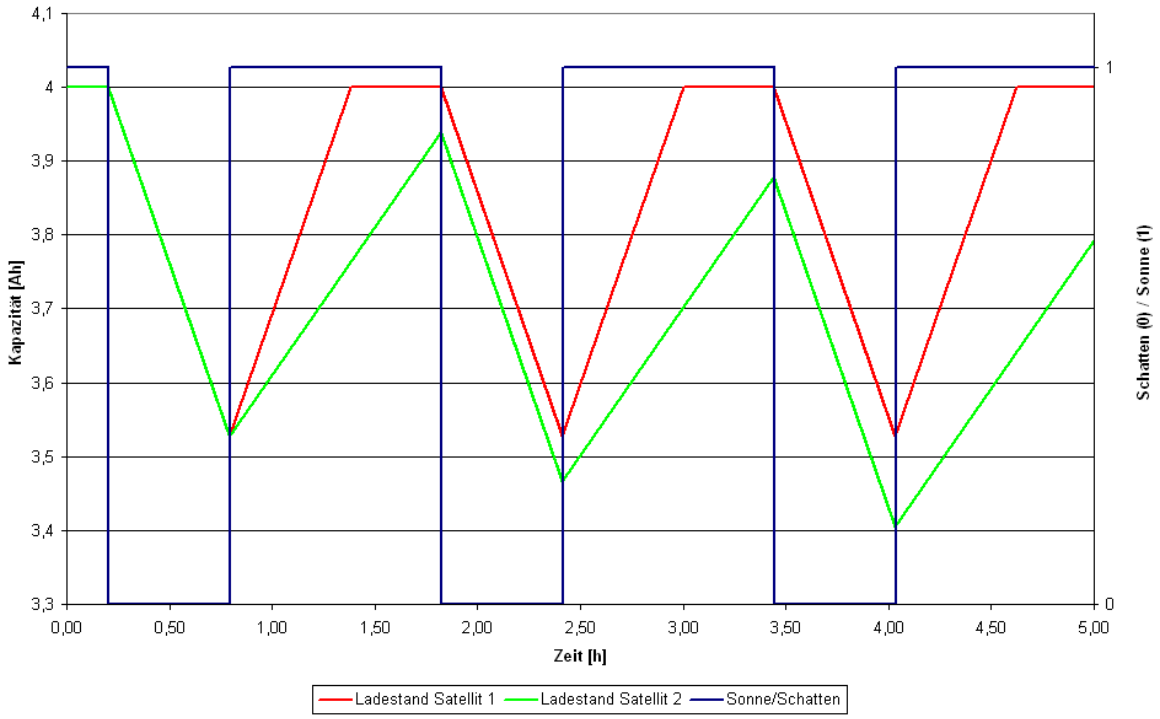


Abbildung 6.6.: Vergleich zweier Solarzellenkonzepte

In dem Diagramm aus Abbildung 6.6 sind die Ladestände der beiden verschiedenen Satelliten der zweiten Simulation dargestellt. Es ist deutlich zu erkennen, dass der zweite Satellit mit der generierten Leistung von $6W$ aus den Solarzellen es nicht schafft, die Akkus in der Sonnenphase wieder vollständig aufzuladen. Somit kann die Konfiguration des zweiten Satelliten im nächsten Iterationsschritt des Entwurfs angepasst oder verworfen werden.

Mit der Option, mehrere Satelliten in einer Simulation unterzubringen, können sofort mehrere Satellitenkonfigurationen mit einem Mal simuliert und anschließend verglichen werden. Es muss dafür nur eine Simulation durchgeführt werden, was den Entwurf von Satelliten deutlich beschleunigt.

Durch den Test in diesem Kapitel wurde nachgewiesen, dass mehrere Simulationen parallel betrieben werden können. Darüber hinaus wurde belegt, dass auch die Funktion, mehrere Satelliten in eine Simulation zu integrieren, erwartungsgemäß arbeitet. Außerdem wurde eine Abhängigkeit der Option zur „schwachen Echtzeit“ von der Auslastung des Computers herausgestellt. Es ist demzufolge empfehlenswert, bei einer Simulation mit aktivierter Echtzeitoption die Services und den Server auch physisch auf verschiedene Computer zu verteilen und möglichst keine weiteren Simulationen oder Prozesse auf dem Server-Computer auszuführen. Ausnahme wäre die Tatsache, dass es sich bei dem Server-Computer um eine echtzeitfähig Plattform handelt.

6.5. Betriebssystemunabhängigkeit und verteiltes Simulieren

In dem letzten Test wurde die Lauffähigkeit von SMASS auf verschiedenen Betriebssystemen getestet. Dabei wurden sukzessiv die Services auf andere Computer ausgelagert. Außerdem wurden auch alle möglichen Kombination von Betriebssystemen innerhalb eines Testfalls aus Tabelle 6.6 für den Server und die Services ausprobiert.

Testfall	**Computeranzahl**	**Betriebssysteme**
1	1	Windows XP Prof. V.2002 SP3
2	1	Windows XP Prof. V2002 SP2
3	3	Windows XP Prof. V.2002 SP3, Windows Vista Home Premium SP1, Windows XP Home Edition V.2002 SP2
4	2	Windows XP Prof. V2002 SP2, Suse Linux Enterprise V.10 als VMware-Image
5	3	Windows XP Prof. V2002 SP2, Suse Linux Enterprise V.10 als VMware-Image, Solaris 10 (SunOS 5.10)

Tabelle 6.6.: Tests zur Betriebssystemunabhängigkeit und verteilten Simulation

Neben dem weitverbreiteten Windows XP Home Betriebssystem wurden auch andere Varianten von Windows getestet. Durch die in Kapitel 4 vorgestellten Dateien für Linux- bzw. Unix-Systeme im Allgemeinen bestand auch die Möglichkeit ein Linux- und ein Solaris-Betriebssystem in die Tests mit einzubeziehen.

Die Besonderheit bei Testfall vier aus Tabelle 6.6 bestand darin, dass es sich bei dem Linux-System um ein VMware-Image handelte. Auf dem Computer lief als Hauptbetriebssystem Windows XP Professional. Eine der drei Komponenten von SMASS lief auf dem virtuellen Linux, eine weitere Komponente unter dem Hauptbetriebssystem des Computers und die letzte Komponente auf einem anderen Computer.

Der Computer mit dem Solaris-Betriebsystem im Testfall fünf wurde über eine SSH-Verbindung in die Tests mit einbezogen. Er befand sich nicht in Reichweite.

Alle Tests konnten erfolgreich durchgeführt werden. Einzige Ausnahme bestand im fünften Testfall. Für den Solarisrechner waren leider die Ports gesperrt und das Benutzerkonto besaß keine Administratorrechte. Es konnten somit keine Ports explizit freigegeben werden. Es wurde aber der Server installiert und gestartet. Alle Schritte zur Durchführung einer Simulation aus Kapitel 4.3 bis zum Anmelden der Services konnten durchgeführt werden, sodass davon auszugehen ist, dass SMASS auch auf einem Solaris-Betriebssystem läuft.

7. Zusammenfassung und Ergebnisse

Die vorliegende Diplomarbeit „Serviceorientiertes Framework für ein Satellitenmissionssimulationssystem“ wird in diesem Kapitel zusammengefasst. Dazu gehört auch die Herausstellung der Ergebnisse und das Aufzeigen des Wegs von der Problemstellung über das entwickelte Framework bis hin zu den Tests.

Nach der Einleitung wurden in dem zweiten Kapitel die grundsätzliche Aufgabe von SMASS erläutert und daraus Anforderungen abgeleitet (Tabelle 2.1). Anschließend wurde das existierende System analysiert und die Schwachstellen aufgezeigt. Diese liegen vor allem in der nicht vorhandenen Modularität und in den fehlenden Schnittstellen. Daraus resultiert die schlechte Wart- und Erweiterbarkeit. Als weitere Schwachstelle wurde die mehrfache Implementierung der Kommunikation auf Seiten der Berechnungsmodule identifiziert. Aus diesen Problemen ergaben sich die Ziele für diese Arbeit (Kapitel 2.3). Es sollte ein neues Framework für SMASS entwickelt werden. Dabei lag der Schwerpunkt auf der sauberen Trennung der Funktionalitäten und einer damit verbundenen modularen Struktur.

In Kapitel drei wurde das Konzept der Schichten- und Paketstruktur vorgestellt und deren Vor- und Nachteile stichpunktartig aufgelistet. Als nächstes folgte eine Übersicht über die definierten Schichten „Simulation“, „Utilities“, „Services“, „Data Handling“ und „Frontend“, welche in den sich anschließenden Unterkapiteln erläutert wurden.

Die „Simulation“-Schicht beinhaltet das Modell der Simulationen, welches aus der Laufzeitrepräsentation und der Geschäftslogik (business logic) besteht. Zunächst wurde auf die Struktur dieser Schicht eingegangen, gefolgt von der Erklärung des Verhaltens einer Simulation. Auf den Aspekt der „schwachen Echtzeit“ und des Datenaustausches zwischen Services wurde noch gesondert eingegangen. Mit der Option zur „schwachen Echtzeit“, welche eine Mindestlaufzeit zwischen den Simulationsschritten gewährleistet, ist die Anforderung UR 11.0 nach der möglichen Einbindung von Hardware-Modulen an Stelle von Services umgesetzt worden. Durch die Möglichkeit des Datenaustauschs zwischen Services können Disziplinen einander beeinflussen (UR 3.0) und ein Satellit kann als ganzes simuliert werden (UR 2.0). Es müssen keine statischen Annahmen mehr getroffen werden, da die berechneten Werte aus den anderen Disziplinen direkt verwendet werden.

Die zweite Schicht, „Utilities“, beinhaltet allgemeine Hilfsklassen. Hier wurden neben dem Paket für mathematische Funktionen und dem Paket für die Bearbeitung einer XML-Datei auch das Paket für die Kommunikation untergebracht. Die drei identifizier-

ten Funktionalitäten im Bereich der Kommunikation, welche der Aufbau der Netzwerkverbindung, das Nachrichtenformat und die Verarbeitung von Nachrichten sind, wurden erläutert. Das Nachrichtenformat wurde im Vergleich zum Ausgangszustand durch den Wegfall der Prüfsumme leicht verändert, da sie kein Mehrwert an Sicherheit ergab. Außerdem konnte die Anzahl der zulässigen Nachrichtentypen auf vier Varianten reduziert werden. Durch die Einführung dieser Schicht wird die Kommunikation zwischen Server und Client standardisiert (UR 15.0). Über die Netzwerkverbindung können mehrere Computer miteinander kommunizieren, was Voraussetzung für das verteilte Rechnen aus UR 8.0 ist. Die Wahl des weit verbreiteten TCP/IP-Standards für die Netzwerkkommunikation beruhte auf der angestrebten Hard- und Software-Unabhängigkeit (UR 9.0/10.0).

In der dritten Schicht wurden die Services untergebracht. An dieser Stelle wurde der Begriff Service eingeführt, da diese von der Kommunikation losgelöst sind und jetzt eine Dienstleistung in Form der Berechnung eines Simulationsschritts für eine Disziplin anbieten. Dafür wurde die in Kapitel 3.3.1 anhand des Orbitservices erläuterte Schnittstelle eingeführt. Der Orbitservices beruht auf dem Modul von Giese ([Gie05]) und wurde an das Framework angepasst. Durch die Einführung der Services ist die Entwicklung für weitere Disziplinen deutlich vereinfacht worden (UR 13.0), da nur die zwei Schnittstellen aus Kapitel 3.3.1 implementiert werden müssen. Die Kommunikation ist bereits durch das Framework gewährleistet und wird über eine Bibliothek eingebunden. Für Services, welche Java als Programmiersprache verwenden, ist die Bibliothek (*Service.jar*) bereits erstellt worden. Darüber hinaus blieb die gute Skalierbarkeit von SMASS durch das mögliche verteilte Rechnen trotz der Einführung der Services erhalten.

Für das Laden und Speichern der Konfigurationsdaten und der Ergebnisse wurde die Schicht „Utilities“ eingeführt. Die erstellte Konfigurationsdatei auf XML-Basis erhöht die Lesbarkeit im Vergleich zu der vorher benutzten Textdatei und steigert so die Benutzerfreundlichkeit (UR 17.0). Sie spiegelt die zentrale Stelle für die Konfigurationsdaten UR 6.0 wieder. Mit dem Laden und Speichern der Ergebnisse wurde UR 5.0 umgesetzt und die Infrastruktur für den Datenaustausch zwischen den Services geschaffen. Diese beziehen ihre Daten aus der Persistenz für die Ergebnisse. Durch die Trennung zwischen Konfiguration und Ergebnisse sowie der Anlegung von Schnittstellen (UR 14.0) ist es außerdem möglich, die Schicht komplett auszutauschen oder unterschiedliche Persistenzen für beide Bereiche zu verwenden. Dies ist insofern interessant, da es nicht Teil der Aufgabe der Diplomarbeit war, eine ausgereifte Datenverarbeitung zu entwickeln. So kann das für die Vorführung implementierte Dateisystem durch höherwertige Systeme ersetzt werden.

Mit der fünften Schicht wurde die Benutzeroberfläche umgesetzt. Diese beinhaltet die Darstellung der Simulationen (UR 4.0) und ermöglicht die Benutzerinteraktion mit SMASS (UR 7.0). Bei der Beschreibung der Struktur dieser Schicht wurde auf die beiden verwendeten Patterns Observer und MVC eingegangen. Zu der Struktur gehört auch hier wieder eine Schnittstelle, wie es in UR 14.0 gefordert wurde. Durch diese Schnittstelle kann die Schicht komplett ausgetauscht werden. Obwohl es auch nicht Teil der Diplomarbeit war, wurde ein einfaches Konsolen-Frontend für die Vorführung entwickelt. Dieses

kann durch die Schnittstellendefinition mit einer grafischen Oberfläche ersetzt werden.

Die Erfüllung der Anforderungen an die einfache Wart- und Erweiterbarkeit können nicht mit ja oder nein beantwortet werden, da dies keine quantitativen Größen sind. Aber mit Hilfe der Struktur des Frameworks und der Trennung der Funktionalitäten (UR 16.0) wurde versucht, diese Anforderungen umzusetzen. Um insbesondere die Lesbarkeit des Quelltextes zu gewährleisten, wurde das Plug-in Checkstyle für Eclipse (siehe Anhang A.2) verwendet. Dieses überprüft den Quelltext und warnt den Benutzer, sollten die Konventionen zum Java-Quelltext nicht eingehalten werden.

An das dritte Kapitel „Aufbau und Funktion“ schloss sich das Kapitel „Bedienung“ an. In diesem Kapitel wurde zunächst die Installation von SMASS erläutert. Dann folgten einige Bemerkungen zur Konfiguration einer Simulation. Das nächste Unterkapitel 4.3 beschrieb detailliert, wie eine Simulation vom Anlegen bis zum Beenden durchgeführt werden kann und ist als eine Art Benutzeranleitung zu verstehen. Im letzten Unterkapitel wurden die nötigen Maßnahmen erläutert, um kleine Änderungen am mitgelieferten Quelltext durchzuführen und die dafür bereitgestellten Batch-Dateien für Windows- bzw. Shell-Skripte für Linux-Systeme zu benutzen.

Das fünfte Kapitel erläuterte Schritt für Schritt anhand des angepassten Energieservices von Prof. Dr. Kayal, wie ein neuer Service in das SMASS-Framework integriert werden kann. Dazu kann die Vorlage auf der CD im Ordner *NewJavaService* benutzt werden. Es wurde die Implementierung der Schnittstellen gezeigt und auf die Stellen hingewiesen, wo die eigentliche Servicefunktionalität integriert werden muss. Abgeschlossen wurde das Kapitel mit den Erläuterungen zur Änderung des XML-Schemas und dem Eintrag in die Konfigurationsdatei.

Im sechsten Kapitel wurden Tests beschrieben, welche die Funktion des Frameworks nachgewiesen haben. Der erste Test galt dabei als Basistest, bei dem auch die korrekte Arbeitsweise des angepassten Orbitservices herausgestellt wurde.

Dem schloss sich ein Test zum Datenaustausch des Orbitservices mit dem Service zur Berechnung des Energiehaushalts PWR000 an. Dieser Test verlief ebenfalls erfolgreich. Es konnte nachgewiesen werden, dass der PWR000-Service zum richtigen Zeitpunkt die Information über Schatten oder Sonne erhält.

Als drittes wurde die Option der „schwachen Echtzeit“ überprüft. Die hierbei festgestellten Abweichungen im Millisekundenbereich festgestellt sind vornehmlich auf die TCP/IP-Verbindung zurückzuführen. Die Abweichungen betrugen im Test bei einer Schrittweite von $5s$ nur eben diese $5s$ pro halbe Stunde Simulationszeit. Diese Größe ist allerdings davon abhängig, wie die Auslastung des Computers und der Netzwerkverbindung ist. Darüber hinaus ist der Fehler auch von der Plattform an sich abhängig.

Im vierten Test wurde die Funktionalität, mehrere Simulationen parallel zu betreiben, nachgewiesen. Für eine der Simulationen wurde die Option zur „schwachen Echtzeit“ aktiviert. Dadurch konnte erneut der Einfluss der Auslastung des Computers, auf dem der Server läuft, bestätigt werden.

Der letzte durchgeführte Test untersuchte die Lauffähigkeit von SMASS auf verschiedenen Betriebssystemen und das verteilte Simulieren. Die beiden Services wurden auf andere Rechner ausgelagert. Es konnte so neben der Lauffähigkeit von SMASS unter Windows XP Home, Windows XP Professional, Windows Vista, Suse Linux Enterprise und Solaris 10 auch das verteilte Rechnen bewiesen werden.

Zusammenfassend ist zu sagen, dass SMASS mit dieser Arbeit erheblich weiterentwickelt wurde. Es sind alle Anforderungen aus Kapitel 2 erfüllt worden. SMASS ist nun sehr modular aufgebaut und kann an beliebigen Stellen erweitert werden, ohne es von Grund auf neu zu entwickeln. Das ist sicherlich einer der größten Vorteile des Frameworks. Darüber hinaus wurde die Kommunikation standardisiert und aus den Services ausgelagert, deren Entwicklung dadurch stark vereinfacht wurde. Die Funktion des Frameworks wurde anhand der Implementierung in Java nachgewiesen. Dafür wurden zusätzlich eine Datenverarbeitung auf Dateibasis und eine Konsolenoberfläche implementiert. Es können so erste Simulationen mit den beiden Services zur Orbitberechnung und Energieversorgung durchgeführt werden. Die Ergebnisse der Simulationen können ausgewertet und so die Entwicklung von Satellitenmissionen voran getrieben werden.

8. Ausblick

An dieser Stelle sollen noch einige Ideen und Vorschläge erörtert werden, welche die Weiterentwicklung von SMASS zum Ziel haben.

Es wurde im vorherigen Kapitel bereits erwähnt, dass die Oberfläche von SMASS sehr einfach ist und auf eine Standardkonsole aufbaut. An dieser Stelle sollte in Zukunft als nächstes angesetzt werden. Es könnte z.B. eine grafische Oberfläche entwickelt, welche dann auch mehr Informationen zu den Simulationen bereitstellen würde. Als weitere Idee wäre ein Web-Frontend denkbar. Dadurch wäre es möglich, SMASS über das Internet zu steuern. Mit der jetzigen Oberfläche ist es aber auch schon möglich, SMASS fernzusteuern. Dazu muss eine SSH-Verbindung zum Server-Computer eingerichtet werden, ähnlich wie es bei dem Test in Kapitel 6.5 mit dem Solaris-Computer gemacht wurde.

Weiteres Optimierungspotential steckt in der Steuerung der „schwachen Echtzeit“. Die negativen Abweichungen bei mehreren angemeldeten Services könnte durch die Verlagerung der Steuerung von der *Simulation*- in die *SatelliteSimulation*-Klasse minimiert werden. Dort müssten dann die Zeitstempel der einzelnen Services gespeichert und miteinander verglichen werden. Außerdem könnte über eine Art „regeneratives“ Warten nachgedacht werden, um den akkumulierten Fehler abzubauen. Wünschenswert wäre auch eine Warnmeldung, falls der akkumulierte Fehler der Schrittweite einen gewissen Grenzwert überschreitet.

Im Bereich der Datenverarbeitung ist es denkbar, die Speicherung der Ergebnisse von den Textdateien auf eine Datenbank umzustellen. Damit würden sich noch weitere Möglichkeiten für die spätere Verarbeitung und auch für die Visualisierung ergeben. Im Bereich der Konfiguration ist mit der XML- und XSD-Datei schon eine gute Lösung gefunden worden. Das Schema zu der Konfigurationsdatei kann aber auch noch erweitert werden. So sollte z.B. überprüft werden, dass Namen gemäß den Angaben in Kapitel 3.4.2 eindeutig sind. Ebenso sollten die Attribute des *service*-Elementes hinsichtlich ihrer Eindeutigkeit innerhalb eines *satelliteSimulation*-Elementes kontrolliert werden. Darüber hinaus könnten auch noch Parametergrenzen für die konkreten Services eingeführt werden, wie dies bereits in der Arbeit von Pfeiff ([Pfe07]) vorgesehen war. Diese würden dem Benutzer später die Editierung der Werte für neue Satellitenkonfigurationen erleichtern.

Um mögliche Fehler oder Unstimmigkeiten während laufender Simulationen analysieren zu können, wäre es denkbar, die verschickten Nachrichten in eine Datei zu schreiben. Diese könnte dann im Nachhinein noch mal überprüft werden, um die Ursache für die Fehler und Unstimmigkeiten herauszufinden.

Für die weitere Bearbeitung des Quelltextes sollte auch ein Mechanismus zum Loggen

eingeführt werden. Für Java bietet sich hier das Framework log4j an. Damit könnten Testausgaben eine Priorität zugewiesen und zentral an- bzw. ausgeschaltet werden.

Aufgrund von Zeitproblemen während der Diplomarbeit wurden die Modultests für das entwickelte Framework vernachlässigt. An dieser Stelle sollte auch noch nachgebessert werden und für möglichst alle Klassen solche Tests aufgestellt werden. Diese reduzieren die Komplexität des Testens eines Systems, da sich auf ein bestimmtes Modul (Unit) konzentriert wird. Wie in Kapitel 5.4 bereits erwähnt, eignet sich JUnit besonders für Java.

In der Schicht *Simulation* wurde während der Verfassung der Dokumentation noch ein weiterer Optimierungsbedarf hinsichtlich der Struktur entdeckt. Die Funktionalitäten können noch weiter getrennt werden in Verwaltungs- und Steuerungsklassen. Die *Simulation*-Klasse könnte z.B. noch in eine Klasse für die Verwaltung der Satelliten und in eine Klasse für die Steuerung der Simulation aufgeteilt werden. Analoges gilt für die *SatelliteSimulation*-Klasse.

In Kapitel 3.1.1 wurde erwähnt, dass Satellitenschwärme in einer Simulation simuliert werden können. Als Erweiterung wäre nun auch denkbar, ähnlich wie bei den Services, diese ebenfalls Daten untereinander austauschen zu lassen. Dafür müsste aber auch eine Simulationsreihenfolge für die Satellitensimulationen eingeführt werden, damit auch hier die Aktualität der Daten gewährleistet werden kann.

Als letzte Anmerkung sollen noch die Fehlermeldungen genannt werden. Diese sollten in der nächsten Entwicklungsstufe zentralisiert und globalisiert werden. Dazu könnte, wie in Kapitel 3.2 bereits vorgeschlagen, ein Paket in der *Utilities*-Schicht definiert werden. Alle existierenden Fehlermeldungen sind im Quelltext mit einem Kommentar versehen, der wie folgt lautet:

```
...
// TODO error messages should be defined somewhere central
...
```

Dadurch können sämtliche Fehlermeldungen im Quelltext leicht gefunden werden. Bei Benutzung von Eclipse, tauchen diese auch alle in der Ansicht „Tasks“ auf.

Das entwickelte Framework bildet für diese Ideen und Vorschläge eine gute Grundlage, auf die in der weiteren Entwicklung aufgebaut werden kann. Alle oben gemachten Vorschläge sind einfach umzusetzen, da die Funktionalitäten klar getrennt sind und dadurch ihre Abhängigkeiten untereinander minimiert wurden. Die vorgeschlagenen Erweiterungen bezüglich der Oberfläche und Datenverarbeitung könnten z.B. in nachfolgenden Diplom- oder Studienarbeiten umgesetzt werden, wodurch die Benutzerfreundlichkeit von SMASS weiter steigen würde.

Literaturverzeichnis

[Ber07a] Berlin, Marco: *Implementierung und Test von Algorithmen zur Lagebestimmung des Picosatelliten BeeSat.* Institut für Luft- und Raumfahrttechnik, TU Berlin, Mai 2007.

[Ber07b] Berlin, Marco: *SMASS - Satellite Mission Analysis and Simulation System - Benutzerhandbuch.* Institut für Luft- und Raumfahrttechnik, TU Berlin und SISTEC, DLR, November 2007.

[BK04] Briess, Klaus und Hakan Kayal: *A Mission Analysis and Simulation System for cost effective Earth Observation Missions with Micro-, Nano- and Pico-Satellites*, Oktober 2004. IAC-04-IAA.4.11.3.05.

[Bog99] Boger, Marko: *Java in verteilten Systemen - Nebenläufigkeit, Verteilung, Persistenz.* dpunkt-Verlag, 1 Auflage, 1999.

[CK06] Cellier, François E. und Ernesto Kofman: *Continous System Simulation.* Springer, 1 Auflage, 2006.

[ES04] Eilebrecht, Karl und Gernot Starke: *Patterns kompakt - Entwurfsmuster für effektive Software-Entwicklung.* Spektrum Akademischer Verlag GmbH, 2004.

[Fal06] Falss, Danny: *JavaServer Faces Vorstellung des Standard-Framework von Sun*, 2006. Verfügbar unter: `http://www.fh-wedel.de/~si/seminare/ws06/Ausarbeitung/10.JavaServerFaces/pdf/jsf.pdf` [06. August, 2008].

[FFSB05] Freeman, Eric, Elisabeth Freeman, Kathy Sierra und Bert Bates: *Entwurfsmuster von Kopf bis Fuß.* O'Reilly, 1 Auflage, 2005.

[Fle06] Fleischer, Simon: *Framework zur sicheren, verteilten Kommunikation ohne PKI-Unterstützung.* Diplomarbeit, Technische Universität Darmstadt, September 2006. Verfügbar unter: `http://www.project-midmay.de/docs/Fleischer06_Diplom.pdf` [23. April, 2008].

[GHJV94] Gamma, Erich, Richard Helm, Ralph Johnson und John Vlissides: *Design Patterns.* Addison-Wesley Publishing Company, 2 Auflage, 1994.

[Gie05] Giese, Torsten: *Aufbau eines Bahnmodells für das Satellitenentwurfszentrum der TU-Berlin.* Institut für Luft- und Raumfahrttechnik, TU Berlin, Mai 2005.

[HL88] Hallmann, Willi und Wilfried Ley: *Handbuch der Raumfahrttechnik.* Carl Hanser Verlag, 1988.

[JH06] Jordan, Christa und Friedhelm Hübner: *Fremdsprachliche Begriffe verstehen und richtig anwenden.* Verlag Das Beste GmbH - Verlagshaus Stuttgart, 2006.

[KY02] Kredel, Heinz und Akitoshi Yoshida: *Thread- und Netzwerkprogrammierung mit Java - Praktikum für die parallele Programmierung.* dpunkt-Verlag, 2 Auflage, 2002.

[Mül04a] Müller, Christian: *Open Source Entwicklung eines Persistenz-Frameworks.* Diplomarbeit, Technische Universität Kaiserslautern, September 2004. Verfügbar unter: `http://agrausch.informatik.uni-kl.de/lehre/arbeiten/SS-04/store/DA-Mueller.pdf` [23. April, 2008].

[Mül04b] Müller, Udo: *Java - Das Lehrbuch.* mitp-Verlag, 1 Auflage, 2004.

[Nat04] Natterer, Simon: *Anwendbarkeit von Java in verteilten Realzeit-Systemen.* Diplomarbeit, Universität Ulm, 2004. Verfügbar unter: `http://www.natterer-legau.de/files/diplomarbeit.pdf` [04. August, 2008].

[Pfe07] Pfeiff, Mark: *Entwurf und Implementierung des Steuerungsservers für das Satellitenentwurfszentrum der TU Berlin.* Institut für Luft- und Raumfahrttechnik, TU Berlin, Oktober 2007.

[Rau06] Rausch, Andreas: *Vorlesung Softwarearchitektur verteilter Systeme - 1. Einführung und Überblick.* Verfügbar unter: `http://agrausch.informatik.uni-kl.de/lehre/SS-06/SAVS/vorlesung/SAVES-1-Einleitung-Und-Ueberblick.pdf` [29. April, 2008], 2006.

[RNB88] Renner, Udo, Joachim Nauck und Nikolaos Balteas: *Satellitentechnik.* Springer, 1988.

[RSSW03] Ratz, Dietmar, Jens Scheffler, Detlef Seese und Jan Wiesenberger: *Grundkurs Programmieren in Java - Band 2 - Programmierung kommerzieller Systeme.* Carl Hanser Verlag, 1 Auflage, 2003.

[SSEHW+06] Scholze-Stubenrecht, Werner, Birgit Eickhoff, Angelika Haller-Wolf, Evelyn Knörr, Anja Konopka, Ursula Kraif, Franziska Münzberg, Ralf Osterwinter, Carsten Pellengahr, Karin Rautmann, Christine Tauchmann, Olaf Thyen und

Marion Trunk-Nussbaumer: *Duden - Die deutsche Rechtschreibung - Band 1.* Bibliographisches Institut und F.A. Brockhaus AG, 24 Auflage, 2006.

[Vog02] Vogt, Ingo: *Implementierung von Entwurfsmustern*, 2002. Verfügbar unter: `http://www.ordix.de/ORDIXNews/1_2002/java_3.html` [06. August, 2008].

[WL91] Wertz, James R. und Wiley J. Larson: *Space Mission Analysis and Design.* Kluwer Academic Publishers, 1991.

A. Verwendete Software

A.1. Enterprise Architect

Enterprise Architect (EA) ist ein UML-Modellierungswerkzeug. Mit diesem Programm wurden sämtliche UML-Diagramme dieser Arbeit erstellt. Um sich das komplette Modell von SMASS auf der CD im Ordner *Code\Framework\design* anzusehen, muss der Enterprise Architect installiert werden. Unter folgendem Link kann eine 30 Tage Testversion heruntergeladen werden:

`http://www.sparxsystems.de/default.asp?nav=3x14`

Die einzige bekannte Einschränkung besteht darin, dass bei der Exportierung von Diagrammen ein Hinweis auf die Testversion hinzugefügt wird. Es wurde mit dem EA auch eine Dokumentation des Modells im *RTF*-Format erstellt, welche sich ebenfalls in dem oben genannten Ordner auf der CD befindet. In dieser Dokumentation befinden sich aber nur die Diagramme. Kommentare und Querverbindungen zwischen den Diagrammen konnten nicht exportiert werden.

A.2. Eclipse

Für die Umsetzung des mit dem EA entwickelten Frameworks in Java und das Schreiben der Diplomarbeit wurde die Entwicklungsumgebung Eclipse benutzt. Eine aktuelle Version kann unter folgendem Link heruntergeladen werden:

`http://www.eclipse.org/downloads/`

Eclipse zeichnet sich durch seine Erweiterungsmöglichkeiten aus. Es wurden unter anderem das Checkstyle-, Subclipse- und Texlipse-Plug-in installiert.

Checkstyle überprüft den Java-Quelltext bezüglich standardisierter Richtlinien. Zu diesen Richtlinien gehört das Setzen von Sichtbarkeitsmodifizierern vor Klassen, Attributen und Methoden, aber auch Formatierungsrichtlinien (Code Convention). Durch die Verwendung wird der Quelltext übersichtlicher und kann leichter nachvollzogen werden. Weitere Information finden sich auf den beiden Internetseiten

`http://eclipse-cs.sourceforge.net/`
`http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html`

Subclipse ist ein Plug-in für die Verwendung einer Versionskontrolle innerhalb von Eclipse. Damit kann der Quelltext verwaltet und versioniert werden. Es ist auch möglich, dass mehrere Entwickler parallel an ein und demselben Quelltext arbeiten. Weitere Information findet sich auf der folgenden Internetseite:

`http://subclipse.tigris.org/`

Texlipse ermöglicht die Verwendung von LaTeX in Eclipse. Somit steht der Editor von Eclipse für das Schreiben der Dokumente zur Verfügung. Vorher muss allerdings eine LaTeX-Distribution installiert werden. Für Windows bietet sich MiKTeX an. Weitere Informationen zu Texlipse sind auf der folgenden Internetseite zu finden:

`http://texlipse.sourceforge.net/`

B. Vorgehensweise

B.1. Modellierung

Die Modellierung war in diesem Fall nötig geworden, um sich erst einmal einen klaren Überblick zu verschaffen und ein Systemverständnis zu gewinnen. Bei der konkreten Anwendung der Modellierung auf das vorliegende Problem wurde auf das Prinzip der Objektorientierung zurückgegriffen, da nur so die gewünschte hohe Modularität gewährleistet werden kann.

B.1.1. Verwendete Modellierungssprache

Für eine objektorientierte Modellierung gibt es mehrere Modellierungssprachen. Für diese Aufgabe wurde die UML verwendet.

B.1.2. Verwendetes Modellierungswerkzeug

Als Modellierungswerkzeug wurde der Enterprise Architect verwendet, welcher auch im Anhang A.1 beschrieben ist.

B.1.3. Verwendete UML-Elemente

Dieser Abschnitt soll kurz die verwendeten Elemente aus der UML erläutern.

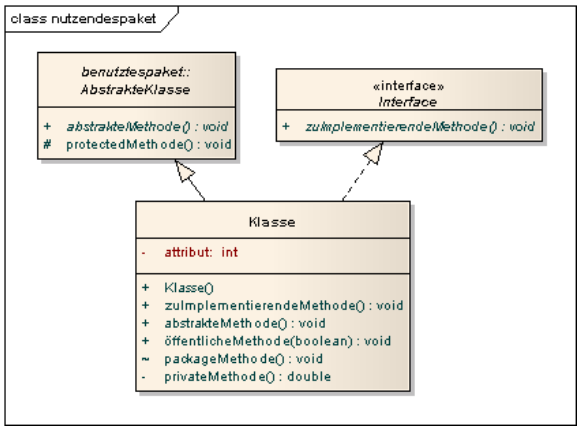


Abbildung B.1.: UML Klassen und Interfaces

In Abbildung B.1 sind drei Elemente dargestellt. Das sind die „normalen“ Klassen, die abstrakten Klassen und die Interfaces. Zwischen diesen Strukturelementen können nun verschiedene Verbindungen modelliert werden. Die gestrichelte Verbindung in Abbildung B.1 zwischen der Klasse und dem Interface stellt die Implementierung des Interfaces *Interface* durch die Klasse *Klasse* dar. Dafür müssen alle Methodenvorlagen des Interfaces realisiert werden. In dem Beispiel ist dies die Methode *zuImplementierendeMethode()*.

Darüber hinaus kann eine Klasse aber auch mittels Vererbung weitere Funktionalität erhalten. Dies ist mit dem durchgezogenen Pfeil in Abbildung B.1 dargestellt. Dabei leitet *Klasse* von der abstrakten Klasse *AbstrakteKlasse* ab. Damit *Klasse* nun nicht selber zu einer abstrakten Klasse wird, muss sie alle abstrakten Methoden implementieren. Außerdem kann sie alle Methoden benutzen, die in der abstrakten Klasse bereits definiert wurden. In diesem Beispiel ist dies die Methode *protectedMethode()*. Diese kann aufgrund der Sichtbarkeit, welche auf *protected* gesetzt ist, nur von erbenden Klassen benutzt werden. Darüber hinaus gibt es noch private Methoden, welche nur von der Klasse benutzt werden können, die sie besitzt, und Methoden die mit dem Sichtbarkeitsmodifizierer *package* versehen. Diese können von allen Klassen im Paket benutzt werden. Die Methode *öffentlicheMethode* kann hingegen von allen Klassen benutzt werden, da sie mit dem Modifizierer *public* versehen wurde.

Bei der Darstellung der abstrakten Klasse ist außerdem noch die Paketzugehörigkeit angegeben (*benutztespaket*). Pakete werden in einem Paketdiagramm dargestellt (siehe Abbildung B.2). In den Paketen kann man die zugehörigen Klassen, Interfaces und Unterpakete erkennen. Auch Pakete können miteinander in Beziehung stehen. Die in Abbildung B.2 gezeigte „use“-Beziehung drückt aus, dass „nutzendespaket“ eine Klasse von „benutztespaket“ verwendet.

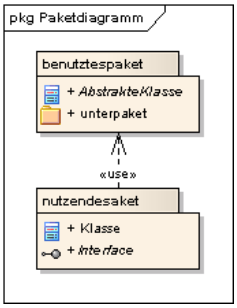


Abbildung B.2.: UML Pakete

Neben den Klassendiagrammen, wie in Abbildung B.1, werden in dieser Arbeit auch noch Zustands- und Aktivitätsdiagramme verwendet. Ein Beispiel für ein solches Zustandsdiagramm ist in Abbildung B.3 gezeigt.

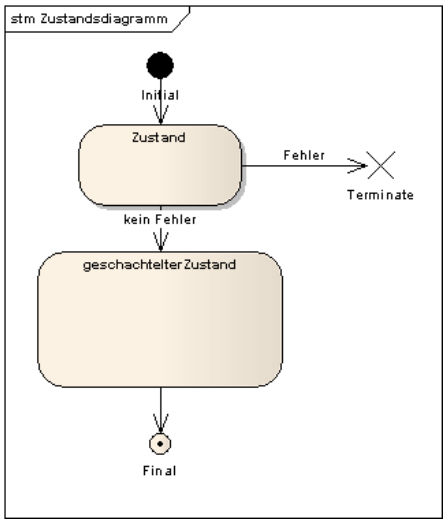


Abbildung B.3.: UML Zustandsdiagramm

Es sind zwei Arten von Zuständen dargestellt, in denen sich ein Objekt, also eine Instanz einer Klasse, befinden kann. Zu einem ist das der einfache Zustand und zum anderen ein geschachtelter Zustand. Einen geschachtelten Zustand zeichnet aus, dass er in sich weitere Unterzustände oder auch weitere geschachtelte Unterzustände vereint. Des weiteren hat jedes Zustandsdiagramm mindestens einen Eingangspunkt (in Abbildung B.3 *Initial*) und mindestens einen Austrittspunkt (in Abbildung B.3 *Terminate* und *Final*). Von einem Zustand zu einem anderen Zustand gelangt man über eine sogenannte Transition, was nichts weiter als ein Übergang ist. Den Übergängen können noch Namen gegeben werden, welche deutlich machen, wann dieser Übergang stattfindet. In dem Beispiel sind zwei Ausgangspunkte dargestellt, um zu zeigen, das es nicht immer einen finalen Zustand gibt. Es kann auch vorkommen, dass ein Objekt nach dem letzten Übergang zerstört wird (*Terminate*).

Zum Schluss soll noch das Aktivitätsdiagramm vorgestellt werden. Dieses ist grundsätzlich ähnlich aufgebaut, wie das Zustandsdiagramm (siehe Abbildung B.4).

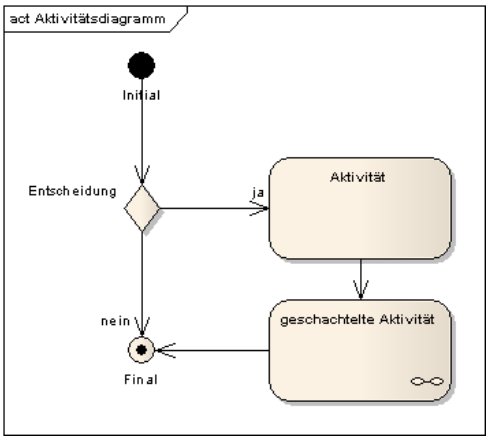


Abbildung B.4.: UML Aktivitätsdiagramm

Auch das Aktivitätsdiagramm besteht aus einem Eintrittspunkt (*Initial*), einem Austrittspunkt (*Final*) und einfachen bzw. geschachtelten Aktivitäten. Als neues Element wird in diesem Diagrammtyp die rautenförmige Entscheidung eingeführt. Diese symbolisiert eine Überprüfung einer Bedingung, wie sie im Quelltext z.B. mit einer *if*-Abfrage realisiert werden könnte.

B.2. Umsetzung

Die Umsetzung des Softwaremodells erfolgte mittels der IDE Eclipse. Diese Entwicklungsumgebung ist als Open Source verfügbar und bestens geeignet für Java. Weitere Erläuterungen zu Eclipse sind im Anhang A.2 zu finden.

Für die Umsetzung wurde zunächst das erstellte Modell mit dem Modellierungswerkzeug in Quellcode umgewandelt. Diesen Vorgang nennt man auch Quelltexterzeugung oder Code Engineering. Es werden also die UML Diagramme in Quelltextfragmente übersetzt. Dann wurden diese Fragmente ausgefüllt und dabei das Systemverständnis vertieft. Wenn neue Aspekte hinzukamen, also neue Klassen erzeugt werden sollten, wurde versucht, dies zunächst im Modell zu machen, da leider das Reverse Engineering, also die Erstellung eines Modells aus dem Quelltext, mit dem Enterprise Architect nicht richtig funktioniert hat. Deshalb musste das Modell immer per Hand auf den aktuellen Stand gebracht werden.

C. Patterns

In diesem Teil des Anhangs sollen die beiden verwendeten Patterns Observer und MVC erläutert werden. Für weiterführende Informationen wird auf die einschlägige Literatur verwiesen (z.B. [FFSB05] und [GHJV94]).

C.1. Observer-Pattern

„Dieses Verhaltensmuster ermöglicht es, dass die Änderung des Zustands eines Objekts dazu führt, andere Objekte zu benachrichtigen und automatisch zu aktualisieren.

Die zentralen Objekte in diesem Muster sind das Subjekt und die Beobachter. Ein Subjekt (Observable) kann mehrere abhängige Beobachter (Observer) besitzen, die sich für den Zustand des Subjekts interessieren. Ändern sich die Daten des Subjekts, werden die angemeldeten Beobachter darüber informiert.“[Vog02]

C.1.1. Struktur des Observer-Patterns

„Das Subjekt kennt seine Beobachter und bietet Schnittstellen zum Hinzufügen und Entfernen von Beobachtern. Die Aktualisierungsschnittstelle des Beobachters bildet die Methode update(). Ein konkretes Subjekt speichert den für einen konkreten Beobachter relevanten Zustand und benachrichtigt über notify() die angemeldeten Beobachter, wenn sich sein Zustand ändert. Dazu ruft das konkrete Subjekt auf allen registrierten Beobachtern deren update()-Methode auf. In dieser Aktualisierungsmethode holt sich der konkrete Beobachter den Zustand des konkreten Subjekts.

Somit sind die Daten des konkreten Subjekts (subjectState) und die der angemeldeten konkreten Beobachter (observerState) synchronisiert (siehe Abbildung C.1).“[Vog02]

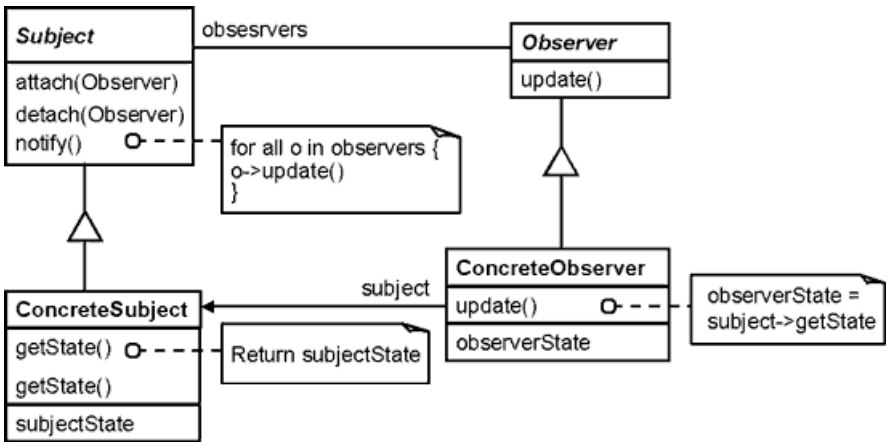


Abbildung C.1.: Struktur des Beobachtermusters [Vog02]

C.1.2. Das Observer-Pattern in Java

„Wegen der häufigen Verwendung, insbesondere beim Programmieren von grafischen Benutzungsschnittstellen, unterstützt die Java Standard API den Entwickler bei der Umsetzung des Beobachtermusters. Das Subjekt ist ein Exemplar der Bibliotheksklasse java.util.Observable. Ein Beobachter implementiert das Interface Observer.[...]“[Vog02]

C.2. MVC-Pattern

„Das MVC-Konzept unterstützt die Aufteilung der Schichten [...]. Es wird eine Entkopplung der Elemente erreicht. Dabei ist das Modell für die Datenhaltung zuständig. Die Präsentation der Daten erfolgt durch die View und der Controller vermittelt zwischen diesen beiden Komponenten.

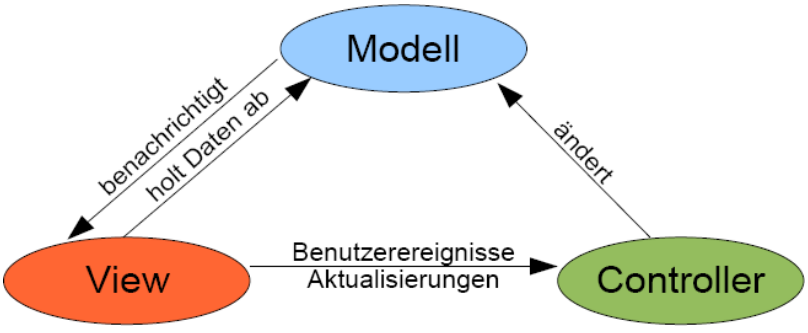


Abbildung C.2.: Struktur des MVC

Das MVC-Konzept arbeitet dabei ereignisorientiert. Eine View registriert sich im Modell und wird dann bei Änderungen informiert, so dass sich die View die neuen Daten

abholen kann. Sollen durch eine View Daten geändert werden, so wir ein entsprechendes Ereignis im Controller ausgelöst und der Controller aktualisiert die Daten im Modell, das dann wiederum die View benachrichtigt.

Durch die strikte Trennung soll erreicht werden, dass sich die jeweiligen Elemente nur noch mit der ihnen zugedachten Aufgabe befassen und so nicht nur die Wartbarkeit der Anwendung erhöht wird, sondern auch Wiederverwendbarkeit erreicht wird.

[...] Lediglich die Schnittstelle zum Controller muss mit dem Applikationsentwickler abgestimmt werden. Daraus ergibt sich, dass die View-Komponente leicht austauschbar wird. Dadurch können unter identischen, logischen Abläufen und auf gleichen Daten andere Sichten erzeugt werden[...].

Die Aufgabe des Controllers im Rahmen des MVC-Konzeptes besteht darin die Ablaufsteuerung zu übernehmen. Des weiteren kümmert er sich um Datenaktualisierungen im Modell nach Eintreffen eines entsprechenden Ereignisses. Er wird wie schon erwähnt von Applikationsentwicklern gepflegt.

Bei der Wiederverwendbarkeit ist vor allem die Verwendung von eigens erstellten Komponenten zu sehen, denn sind diese einmal erstellt, so können sie in jeder Anwendung verwendet werden. Auch Modelle sind wiederverwendbar, aber Views und Controller meistens nur gemeinsam aus der vorher schon erwähnten engen Verbindung.

Nachteile der strikten Trennung sind ein erhöhter Aufwand bei der Erstellung und der Ablauf ist nicht immer sofort zu erkennen, wodurch das Verständnis erschwert wird.

Ausgehend von der Schichtentrennung lassen sich das Frontend bzw. Backend im MVC-Konzept eindeutig der View bzw. dem Modell zuordnen und die Ablaufsteuerung sowie Aktualisierung der Daten übernimmt der Controller. Die Geschäftslogik der Anwendung kann entweder im Modell oder im Controller untergebracht werden. Bei umfangreichen Projekten wird diese sogar oft komplett ausgelagert[...].“[Fal06]S.4f

D. Inhalt der CD

Inhalt	**Verzeichnis**
Quelltext mit Javadoc Dokumentation	*Code*
ausführbare Programme	*Executable*
JDK	*JDK1.6.0_03*
**.tex*-Dateien der Diplomarbeit	*Latex*
Vorlage für neuen Java-Service	*NewJavaService*
Originalversionen der beiden Services	*OriginalServices*
Testdateien für Kapitel 6	*Tests*
Diplomarbeit als **.pdf*	Wurzelverzeichnis