der Bundeswehr
Universität München

# Visualisierung von Fusionsmodellen

Bachelor Thesis von
Stephan Tzschoppe
1006374

UniBwM – IB 16/2009

Aufgabenstellung:
Prof. Dr. Stefan Pickl

Betreuung:
Dipl.-Inf. Marco Schuler

Institut für Theoretische Informatik,
Mathematik und Operations Research
Fakultät für Informatik
Universität der Bundeswehr München

Neubiberg
18.12.2009

# Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken und Zitate sind als solche kenntlich gemacht.

Es wurden keine anderen, als die in der Arbeit angegebenen Quellen und Hilfsmittel benutzt. Die Arbeit wurde weder einer anderen Prüfungsbehörde vorgelegt, noch veröffentlicht.

Neubiberg, 18. Dezember 2009 _____

Unterschrift

# Inhaltsverzeichnis

# 1 Einleitung

## 1.1 Motivation

Wir leben in einer Zeit stetig voranschreitender Technologisierung. Dies äußert sich für jeden sichtbar auf vielerlei Art und Weise. Massenspeicher mit vor Jahren noch unvorstellbaren Kapazitäten, stetig wachsende Prozessorleistung und massiver Fortschritt in der Datenübertragung sind hier nur als Beispiele zu nennen. Solche Entwicklungen sind es, die das Sammeln, Verarbeiten und Speichern von riesigen Datenmengen erst ermöglichen. Diesen Fortschritt gilt es zu nutzen und auf mögliche Anwendungsfelder auszuweiten.

Betrachtet man ein Schlachtfeld, sei es im Rahmen einer Übung, einer kriegerischen Auseinandersetzung oder eines Konflikts, so werden auch hier Informationen gesammelt und ausgewertet. Man stelle sich folgende Situation vor: Drei eigene Panzer bewegen sich durch das Gelände. Plötzlich klären sie zwei feindliche Fahrzeuge auf. Jeder einzelne eigene Panzer setzt eine Meldung ab und beschreibt, was er sieht. Dies führt zu sechs Meldungen. Der S2[1]-Offizier muss nun aus diesen Meldungen ein Lagebild erstellen. Dabei gilt es aus der vorhandenen (teilweise redundanten) Information die zwei statt sechs feindliche Fahrzeuge zu erkennen.

Für das geschilderte Beispiel scheint es nicht notwendig, diese Informationsauswertung zu automatisieren. In der Realität hingegen [2] erfordert es viel Zeit, diese Arbeit zu erledigen. Und genau dies ist ein großes Problem, denn je älter die Lageinformation ist, umso weniger aussagekräftig ist sie. Entscheidungen, die darauf basierend getroffen werden, können daraufhin falsch oder unverhältnismäßig sein. Um diesen Missstand zu

[1] Der S2-Offizier ist verantwortlich für die Militärische Sicherheit, Militärisches Nachrichtenwesen mit Aufklärung und Zielfindung, elektronisch Kampfführung und eben die Wehrlage

[2] Ein Panzerbataillon der Bundeswehr umfasst zum Beispiel ungefähr 40 Kampfpanzer

beseitigen, gilt es, den S2-Offizier bei seiner Arbeit technisch zu unterstützen.

## 1.2 Ziel der Arbeit

Eine technische Unterstützung kann in unterschiedlichen Abstufungen geschehen. So können die separaten Meldungen zu einer großen Meldung zusammengefasst werden. Dies ist aber nicht sonderlich hilfreich, wenn zum Beispiel aus vielen einzelnen Datensätzen einfach eine zusammenhängende Datensammlung erstellt wird. Denn wenn diese in einem textuellen Format vorliegt, ist sie von einem Menschen nur schwer zu verstehen. Weiterhin können die auftretenden Redundanzen nicht einfach erfasst, geschweige denn überhaupt genutzt werden.

Aus diesem Grund möchte ich mich in dieser Arbeit mit Möglichkeiten auseinandersetzen, eine menschenlesbare Sicht auf multisensorische Daten zu erstellen. Zum Einen sollen Visualisierungsmöglichkeiten aufgezeigt werden. Weiterhin soll bei unterschiedlichen Ansätzen, eine Bewertung dieser vorgenommen werden.

Ergebnis dieser Betrachtungen wird ein Framework zur Darstellung multisensorischer Daten. Dieses verwende ich dann prototypisch dazu, die eingangs erwähnte Problemstellung des S2-Offiziers zu bearbeiten und ihm eine, für seine Lageerstellung hilfreiche, Sicht auf die eingehenden Meldungen zu geben.

Die Erweiterbarkeit des Frameworks soll durch dessen Verwendung zur Darstellung von NDP [ABK] Daten gezeigt werden.

## 1.3 Aufbau der Arbeit

# 2 Theoretische Grundlagen

## 2.1 Multisensor-Daten

### 2.1.1 Eigenschaften von Multisensor-Daten

### 2.1.2 Fusion als notwendige Voraussetzung zur Datenverarbeitung

[Var97] [HL97] [Hal92]

## 2.2 Visualisierungsmöglichkeiten

# 3 Verwendete Technologien

In diesem Kapitel werden die wichtigsten Technologien, die im Laufe der Arbeit verwendet werden, vorgestellt und erläutert.

## 3.1 Java

## 3.2 XML

## 3.3 jMonkeyEngine

## 3.4 Google Code

# 4 Entwurf und Umsetzung einer Visualisierungsumgebung

In diesem Kapitel werden der Entwurf eines Visualisierungsframeworks und eine prototypische Implementierung auf dessen Basis beschrieben. Dazu werden die einzelnen Komponenten des Frameworks erklärt und ihr Zusammenspiel erläutert. Basierend auf dem Wissen über die Bestandteile können die Möglichkeiten der Erweiterbarkeit und der Individualisierbarkeit skizziert werde. Die Grundlage hierfür stellt zum einen eine prototypische Implementierung einer Visualisierung einer militärischen Lage und des weiteren der Versuch, Daten des NDP [ABK] dreidimensional darzustellen.

Abschließend werden interessante Aspekte der Implementierung aufgezeigt, die neben den umgesetzten Implementierungsentscheidungen auch andere Wege aufzeigen sollen.

Dem Leser, der sich vorrangig für den entstandenen Prototyp interessiert, dem sei der Abschnitt 4.2 empfohlen. Die Details des Frameworks sollten für das Verständnis des Funktionsumfangs und die Bedienung keine Rolle spielen, können aber bei Bedarf nachgeschlagen werden.

## 4.1 Entwurf eines Visualisierungsframeworks

Das Ergebnis dieser Arbeit soll nicht nur eine potentielle Visualisierungs-Umgebung sein, sonder ein Framework, das zum einen die Möglichkeit bietet, mit geringem Aufwand Daten anzeigen zu können und auf der anderen Seite aber umfangreiche Erweiterungsmöglichkeiten zulässt. Wie das Framework konkret entworfen und umgesetzt wurde, soll im Folgenden dargestellt werden.

### 4.1.1 Eingabedaten und Datenmodell

Grundlage für die weitere Arbeit sollen Eingabedaten sein, die bestimmte Voraussetzungen erfüllen. Diese gestellten Bedingungen werden im Folgenden kurz beschrieben. Darauf basierend wird das Datenmodell erläutert, in das die Quelldaten überführt werden sollen.

#### Eingabedaten

Um Daten dreidimensional visualisieren zu können, müssen diese bestimmte Voraussetzungen erfüllen. Diese sollen hier kurz aufgezeigt werden.

**Identifizierbarkeit** Unabhängig von der Art der Visualisierung ist es unabdingbar, dass jedes einzelne Datum identifizierbar ist. Aus diesem Grund sollte in den Quelldaten bereits eine eindeutige Benennung vorliegen. Sollte das nicht der Fall sein, muss dieser Misstand beim Importieren der Daten spätestens behoben werden. Probleme, die sich ergeben können, wenn diese Bedingung nicht beachtet wird, sind zum einen, dass keine aussagekräftigen Berechnungen auf den importierten Daten durchgeführt werden können. Auch ist eine Markierung eines gezielten Datensatzes unmöglich.

**Lokalisierbarkeit** Aus dem Ziel, die Eingabedaten im dreidimensionalen, kartesischen Raum darzustellen ergibt sich eine ganz logische Voraussetzung: Es sollte mindestens für jede der drei Dimensionen eine Eigenschaft der Daten existieren, die eine Positionierbarkeit möglich macht. Zwar ist es genauso möglich, die Daten auf einer Linie anzuordnen und somit nur eine Positionierungseigenschaft vorauszusetzen oder sich analog auf den zweidimensionalen Raum zu beschränken. Der daraus resultierende Informationsverlust muss aber in Kauf genommen werden.

Die Voraussetzungen an den Typ der für die Lokalisierung herangezogenen Eigenschaften sind nicht sehr streng. Zwar ist es zielführend, wenn es sich hier um kontinuierliche oder zumindest diskret ganzzahlige Größen handelt. Ist dies nicht der Fall, so müssen diese lediglich zur Berechnung der Anzeigekoordinaten mit einer geeigneten Abbildungsvorschrift umgerechnet werden.

| *mittelbar geeignet* | *unmittelbar geeignet* |
|---|---|
| IP [ABK] Adressen | Entfernungsangaben |
| MAC [ABK] Adressen | Ortsangaben (Länge/Breite) |
| Zeichenketten allgemein | Zeitangaben |

Tabelle 4.1: Beispiele für unmittelbar und mittelbar geeignete Größen

**Weitere Eigenschaften** Die Darstellung von Daten im dreidimensionalen Raum, lediglich basierend auf der Position und dem Namen, bietet noch keine hohe Anschaulichkeit. Um die in [Verweis, Visualisierungsmöglichkeiten] dargestellten Visualisierungen auch verwenden zu können, braucht es weitere Eigenschaften, die in optische Information umgewandelt werden kann.

Die Art der Daten die für solche zusätzliche Veranschaulichung verwendet kann, muss nicht genau spezifiziert werden. Trotzdem gibt es Typen, die sich für bestimmte Visualisierungsmöglichkeiten besser eignen. Als einfaches Bespiel seien hier Aufzählungsdatentypen genannt, die sich gerade zu anbieten, in Objekte in unterschiedlichen Farben darzustellen. Dabei können auch unterschiedliche Enumerationen gleichzeitig visualisiert werden, indem pro Eigenschaft eine Farbe verwendet wird, die je nach Wert der Eigenschaft in unterschiedlichen Helligkeiten dargestellt wird. Der Kreativität sind hier keine Grenzen gesetzt.

### Datenmodell

Das im Rahmen dieser Arbeit entwickelte Datenmodell setzte den Grundstock für das später entworfene Framework und hat zum Ziel, durch generische Gestaltung in der Lage zu sein, unterschiedlichste Quelldaten ohne Anpassung speichern zu können.

Umgesetzt wurden diese Anforderungen durch die Trennung in eine Datensammlung und deren Daten auf der einen, und Eigenschaften auf der anderen Seite.

**Datensammlung mit Daten** Wie in 4.1 dargestellt, besteht das wesentliche Datenmodell aus zwei Klassen.

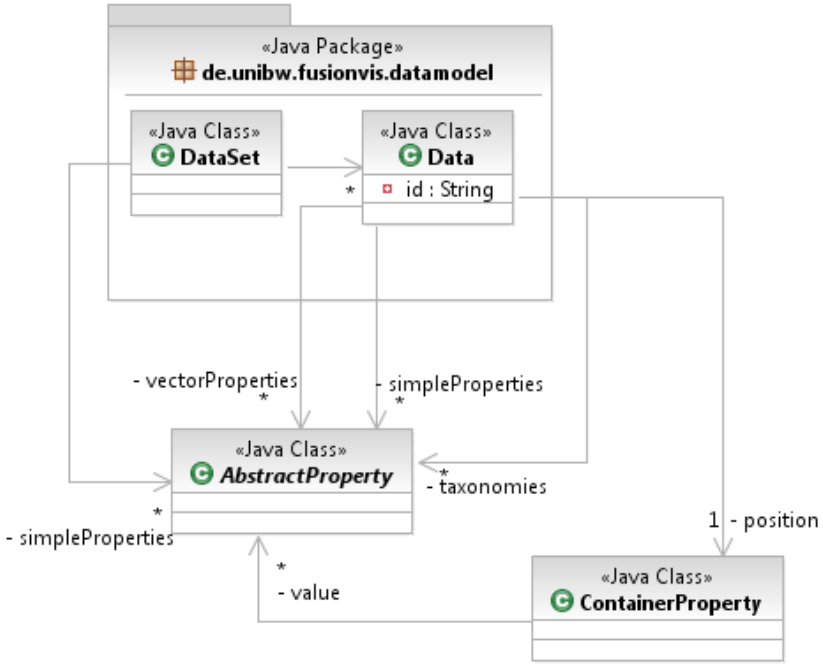


Abbildung 4.1: Übersicht des Datenmodells

**DataSet** Die Klasse DataSet kapselt die einzelnen Datensätze in Form einer Liste. Diese Klasse ist die zentrale Datenstruktur, die für alle weiteren Prozesse die notwendigen Informationen bereithält. Sie besteht im Wesentlichen aus zwei Bestandteilen. Das erste ist die angesprochene Liste der gespeicherten Daten. Weiterhin kann sie Eigenschaften erfassen, die nicht einem bestimmten Datum zu Eigen sind, sondern global für die gesamte Datensammlung gelten. Das benutzen dieser Eigenschaften ist aber fakultativ um eine hohe Flexibilität zu gewährleisten. Die Typisierung der Eigenschaften wird weiter unten dargestellt.

**Data** Die Klasse Data Kapselt ein einzelnes Datum. Wie bereits in 4.1.1 dargelegt, sind die notwendigen Bestandteile eines Datensatzes ein Bezeichner, der innerhalb einer Datensammlung eindeutig sein muss, sowie eine Positionsangabe. Diese ist mithilfe einer zusammengesetzten Eigenschaft festgehalten. Das Eigenschaftssystem wird weiter unten noch detailliert beschrieben.

Zusätzlich zu diesen beiden obligatorischen Angaben erfasst die Data-Klasse weiterhin drei Listen von Eigenschaften.

- Einfache Eigenschaften

- Zusammengesetzte Eigenschaften
- Taxonomien

Die einfachen Eigenschaften sind in der Lage textuelle und Numerische Merkmale eines Datensatzes zu erfassen. Sie sind Grundlage die spätere Visualisierung. Die Zusammengesetzten Eigenschaften ermöglichen das Ablegen von vektoriellen Größen, wie zum Beispiel einer Blickrichtung, Geschwindigkeiten, oder Beschleunigungen usw. Auch ist es möglich mit ihrer Hilfe baumartig strukturierte Eigenschaften zu erfassen.

Die Taxonomien umfassen eine Liste möglicher Klassifikationen, die einem Datensatz zu Eigen sein können. Auch ihre Angabe ist nicht zwingend erforderlich.

Der Aufbau einer Data-Klasse kann in 4.2 nachvollzogen werden.

**Eigenschaften** Die Eigenschaften der Datensammlung, wie auch die der Daten an sich, sollten so generisch aufgebaut sein, dass sie für möglichst viele unterschiedliche Anwendungsgebiete ohne Änderung und Anpassung übernommen werden können.

Beim Entwurf fiel auf, dass die auftretenden Eigenschaften nach zwei Kriterien zerfallen.

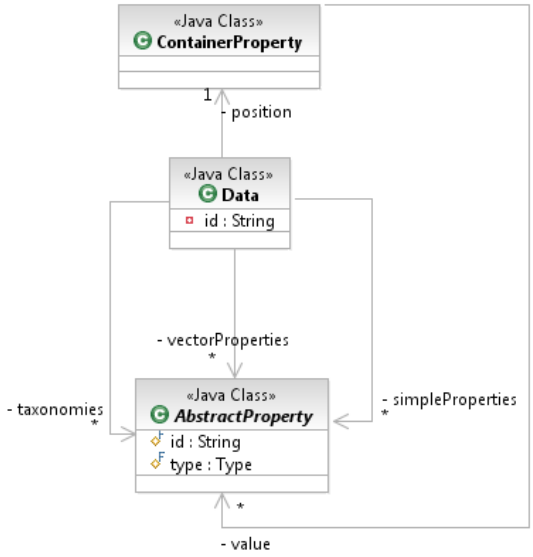


Abbildung 4.2: Aufbau der Klasse Data

Auf der einen Seite kann nach Dimension der Eigenschaften unterschieden werden. So kann zum Beispiel gespeichert werden, ob die Einheit in einem Schlachtfeldsimulator feindlich, freundlich oder neutral ist. Weiterhin ist es wäre es möglich, eine Gewichtsangabe zu speichern. In beiden fällen handelt es sich um eine einfache, weil eindimensionale, Eigenschaft.

Im Gegensatz gibt es zusammengesetzte Eigenschaften wie vektorielle Größen. In diesen Bereich fallen auch Gliederungsinformationen, oder Unterstellungsverhältnisse. Eine zusammengesetzte Eigenschaft besteht damit entweder aus einfachen, oder wiederum aus zusammengesetzten Eigenschaften. Diese Unterscheidung führte im Entwurf zur Auswahl des Composite-Patterns (nach [GHJV02]) für die Modellierung des Sachverhalts.

Das zweite Kriterium, in das die Informationen der Daten zerfallen, ist der Typ dieser. Um einen Kompromiss zwischen einem kompakten Datenmodell und einem breiten Spektrum unterstützter Typen zu gewährleisten, fiel die Entscheidung auf folgende Datentypen:

- Boolean
- Char
- Date [ANM]
- Float
- Integer
- String

Um die grundlegenden Bedürfnisse zu stillen, hätte auch eine Auswahl eines Fließkommadatentyps, mit dem sich auch ganze Zahlen darstellen lassen, und ein Zeichenkettendatentyp, mit dem alle anderen Informationen gespeichert werden können, ausgereicht. Eine noch kleinere Teilmenge, die nur den String-Datentyp umfasst, wäre unter Ausnutzung von programmiersprachenspezifischen Typumwandlungen auch denkbar gewesen. Diese beiden Möglichkeiten wurden aber mit Hinblick auf die komfortablere Handhabung verworfen.


Abbildung 4.3: Eigenschaftssystem im Datenmodell

Der resultierende Entwurf ist in 4.3 dargestellt. Er besteht aus der abstrakten Eigenschaft, den konkreten Implementierungen nach Datentypen und der Zusammengesetzten Eigenschaft.

**AbstractProperty** Die abstrakte Klasse dient als Muster für die konkreten Implementierungen. Sie wird über einen Bezeichner eindeutig identifiziert. Es bietet sich hier an, menschenlesbare Namen zu verwenden, die zugleich beschreibenden Charakter haben. Zusätzlich hat jede Eigenschaft einen Typ. Dieser ist bereits in der abstrakten Klasse implementiert, um zur Laufzeit ohne Kenntnis der genauen Implementierung den jeweiligen Typ der Eigenschaft abzufragen.

Der Satz an abstrakten Methoden bildet die notwendige Schnittstelle um den Wert einer Eigenschaft zu lesen und zu schreiben. Hier stellt sich die Frage, warum eine abstrakte Klasse und nicht etwa ein Interface gewählt wurde. Die Antwort ergibt sich aus der Möglichkeit die Typinformationen und den Bezeichner in der allgemeinen Eigenschaft vorhalten zu können. Das entlastet den Programmierer bei der Umsetzung der konkreten Implementierungen, da er sich darum nicht mehr kümmern muss.

Aus Sicht des Composite-Patterns stellt AbstractProperty die Component-Klasse

dar.

**Typ-Property** Die Implementierungen der abstrakten Klasse tragen in der Umsetzung die Verantwortung, einen zum Typ passenden Wert zu speichern. Dieser wird in der Ausgestaltung der Manipulationsmethoden gesetzt und gelesen. Wo notwendig sind Typumwandlungen durchzuführen, bei unsinnigen Operationen (zum Beispiel das Umwandeln eines bool'schen in ein Datum) sollten Exceptions geworfen werden.

Eine Typ-Property ist die im Composite-Pattern als Leaf bezeichnete Klasse.

**ContainerProperty** Die zusammengesetzte Eigenschaft verwaltet ihre zugehörigen Eigenschaften in einer Liste. Sie kann aus Typ-Property-Klassen bestehen, oder wiederum aus zusammengesetzten Eigenschaften. Aus diesem Grund enthält die Liste, die den Wert der ContainerProperty darstellt, nur AbstractProperty-Klassen. Darum stellt diese Klasse den Composite dar.

**Erweiterbarkeit** Das Datenmodell ist an der Stelle der Eigenschaften erweiterbar. Um einen neuen Typ einzuführen müssen vier Schritte durchgeführt werden:

In der Enumeration der Typen ist der neue einzuführen. Weiterhin muss in der abstrakten Klasse die Zugriffs- und Manipulationsmethoden abstrakt definiert werden. Daraus folgt, dass bestehende Typen diese Methoden implementieren müssen. Macht eine Typumwandlung Sinn, dann kann diese umgesetzt werden. Meist können die Methoden jedoch auf das Werfen einer Exception reduziert werden.

Abschließend muss nun eine neue Typ-Property Klasse erstellt und implementiert werden, die von ihrem abstrakten Vorbild erbt. Durch diese vier Schritte hat man somit das Datenmodell um einen Typ erweitert.

### 4.1.2 Übersicht über das Frontend des Frameworks

Das erstellte Framework besteht im Wesentlichen aus zwei Teilen. Auf der einen Seite stehen die für den Benutzer sichtbaren Klassen, an denen nicht notwendigerweise eine Anpassung vorgenommen werden muss, um ein lauffähiges Programm zu erstellen. Zu diesen Klassen gehört auf oberster Ebene die FusionVisForm. Diese ist die dem Benut-

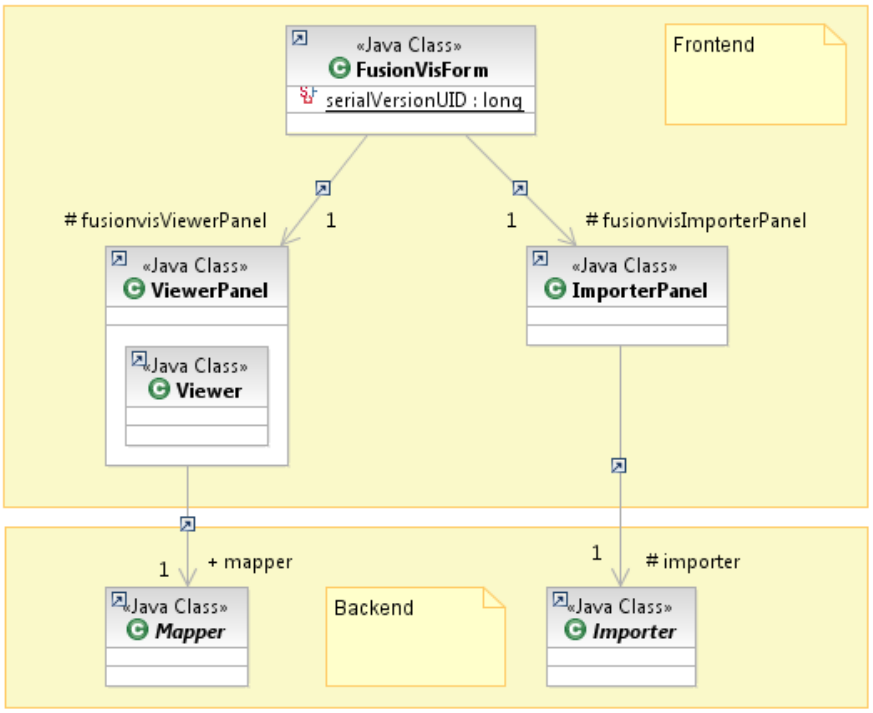


Abbildung 4.4: Übersicht des Frameworks (Frontend/Backend)

zer angezeigte Programmoberfläche. Sie besteht aus einem Menü mit implementierter Funktion zum Auswählen von XML-Dateien und zum Schließen des Programms.

Die Form enthält weiterhin zwei Panels, die es dem Benutzer ermöglichen, eine textuelle und eine visuelle Darstellung der Daten zu sehen.

**ImporterPanel** Für die textuelle Sicht ist das ImporterPanel vorgesehen. Es bietet die Möglichkeit, die Daten zu nach bestimmten Eigenschaften zu filtern und die Daten mit ihren Eigenschaften in Textform anzuzeigen. Das Model, auf dem gearbeitet wird, ist das in 4.1.1 beschriebene Datenmodell. Somit ist das ImporterPanel auch völlig unabhängig von dem genauen Inhalt der Daten, solange diese in das beschriebene Model überführt wurden. Dafür ist der Importer zuständig, der im nächsten Abschnitt beschrieben wird.

**ViewerPanel** Das zweite Panel ist das ViewerPanel. Es ist verantwortlich für die dreidimensionale Anzeige der Daten. Dazu ist es notwendig, das vorhandene Datenmodell in die Struktur eines Szenenbaums zu überführen. Der Szenenbaum wird in [REF Kapitel 4 jme] beschrieben. Dieser Schritt wird vom Mapper bewerkstelligt,

der im nächsten Abschnitt erklärt wird.

Der erstellte Szenenbaum kann nun von einem Viewer angezeigt werden. Dieser beinhaltet eine OpenGL-Anzeigefläche, die die dreidimensionale Darstellung übernimmt und es dem Benutzer erlaubt, sich in den visualisierten Daten zu bewegen, Datensätze mittels Mousepickings (siehe 4.3.2) auszuwählen und diese somit im ImporterPanel zu inspizieren. Auch der umgekehrte Weg ist möglich, also das Auswählen eines Datums in der textuellen Ansicht, was eine Hervorhebung seiner Visualisierung im ViewerPanel zur Folge hat.

Zusammenfassen kann man zum Fronend des Frameworks sagen, dass die von den konkreten XML-Daten unabhängigen Funktionen bereits implementiert sind. Der Anwender muss lediglich festlegen, was schließlich wie dargestellt werden soll. Das Was ist der Prozess des Überführens von XML in das Datenmodell des Frameworks und wird im Importer festgeschrieben. Das Wie ist die Frage nach der Art und Weise, wie die Daten dreidimensional dargestellt werden sollen und wird im Mapper beantwortet.

### 4.1.3 Übersicht über das Backend des Frameworks

Das Backend besteht im Wesentlichen aus den zwei oben erwähnten Komponenten, Importer und Mapper. Im Folgenden sollen die Funktion der beiden erläutert und notwendige Schritte der Spezialisierung beschrieben werden.

Die Spezialisierung bezieht sich auf das Implementieren abstrakter Methoden, die in den beiden als abstrakt gekennzeichneten Klassen die eigentliche Funktion enthalten sollen.

#### Importer

Der Importer ist, wie bereits im letzten Abschnitt angemerkt, dafür verantwortlich, die Daten aus einer XML-Datei zu lesen. An die Struktur muss eine Voraussetzung gemacht werden, um den Import automatisiert durchführen zu können: Die Datensätze müssen als Elemente des Wurzelknotens in der XML-Dateien stehen. Ein Beispiel für eine XML-Datei, die diese Anforderung erfüllt ist in Listing 4.1.

Ist die Voraussetzung erfüllt, liest der Importer die XML-Datei aus und extrahiert die

DOM-Knoten der Datenelemente. Die Wahl des DOM-Parsers für diese Aufgabe wird noch in [REF] genauer erläutert.

```
<Situation>
  <Units>
   <Unit>
      <Name>FriendlyTank1</Name>
      <Location>
        <Lat>52.796629714678467</Lat>
        <Lon>9.89990561649954</Lon>
        <LastModified>2009-02-20T10:25:36+01:00</LastModified>
      </Location>
    </Unit>
    <Unit>
      <Name>FriendlyTank2</Name>
      <Location>
        <Lat>52.794038961891268</Lat>
        <Lon>9.9011727699025922</Lon>
        <LastModified>2009-02-20T10:25:36+01:00</LastModified>
      </Location>
    </Unit>
  </Units>
</Situation>
```

Listing 4.1: Beispiel für eine wohlgeformte XML-Datei

Der Importer hält bereits notwendige Datenstrukturen vor. Zum einen sind dies zwei Membervariablen (*id*, *position*). Diese dienen dem Speichern der XML-Elementnamen, die den Wert für Position und Bezeichner eines Datums liefern. Ähnlich gibt es drei Listen, die die Elementnamen enthalten, die als einfache, zusammengesetzte Eigenschaften oder als Taxonomie extrahiert werden.

**Spezialisierung** Zur Spezialisierung eines Importers für ein bestimmtes Datenformat müssen zwei Schritte durchlaufen werden. Als erstes sollten die in das Datenmodell als Eigenschaft von Daten zu übernehmenden XML-Elemente auf die oben angesprochenen Membervariablen und Listen verteilt werden.

Als nächstes muss nun die Methode *extractDataFormNode()* implementiert werden. Ihr

wird ein Knoten des DOM-Dokumentbaums übergeben. Dieser zeigt auf ein Element, dass als Datum in das Datenmodell übernommen werden soll. Die Aufgabe der Funktion ist es nun nach den Vorstellungen des Anwenders ein Data-Objekt zu erzeugen, das die Eigenschaften enthält, die in den vorher spezifizierten Listen stehen.

**Hilfsfunktionen** In der Input-Klasse sind bereits zwei Hilfsfunktionen implementiert, die dem Anwender die Arbeit beim erzeugen der Data-Objekte vereinfachen sollen. Die Methode *parseDate()* dient dem Auslesen eines Datums aus einem String. Für die genauere Dokumentation sei hier auf das JavaDoc verwiesen.

Die Methode *extractSimpleProperty(Node, Type)* erzeugt eine einfache Eigenschaft (siehe 4.1.1) eines angegebenen Typs aus einem DOM-Knoten. Um beispielsweise in Zeile 4 von Listing 4.2 eine Eigenschaft *Name* als String zu extrahieren, muss die Hilfsfunktion lediglich mit der Node aufgerufen werden, die auf dieses Element zeigt und dem gewünschten Typen (hier: *Type.TString*).

```
protected Data extractDataFromNode(Node unitNode) throws Exception {
  Data result = null;
  NodeList list = unitNode.getChildNodes();
  // Erster Durchlauf zum Finden des Bezeichners
  for (int i = 0; i < list.getLength(); i++) {
    if (list.item(i).getNodeType() != Node.ELEMENT_NODE)
      continue; // wenn kein Element, dann skip
    Node element = list.item(i);
    if (element.getNodeName().equals(id))
      result = new Data(element.getTextContent());
  }
  // Zweiter Durchlauf für das setzen der restlichen Eigenschaften
  for (int i = 0; i < list.getLength(); i++) {
    if (list.item(i).getNodeType() != Node.ELEMENT_NODE)
      continue; // wenn kein Element, dann skip
    Node element = list.item(i);
    // Position setzen
    if (element.getNodeName().equals(position))
      result
          .setPosition(extractContainerProperty(element,
              "Position"));
    else if (simplePropertyList.contains(element.getNodeName()))
```

```
        if (element.getNodeName().equals("IsPlatform"))
24        result.addAbstractProperty(extractSimpleProperty(element,
              Type.TBool));
26      else
          result.addAbstractProperty(extractSimpleProperty(element,
28            Type.TString));
      else if (vectorPropertyList.contains(element.getNodeName()))
30      result.addContainerProperty(extractContainerProperty(element,
            element.getNodeName()));
32    else if (taxonomyList.contains(element.getNodeName()))
        result.addTaxonomie((extractSimpleProperty(element)));
34    else ; // skip
    }
36  return result;
  }
```

Listing 4.2: Beispielhafte Implementierung der *extractDataFromNode()*-Methode

### Mapper

Bereits im vorletzten Abschnitt wurde angesprochen, dass der Mapper dafür Sorge trägt, dass das Datenmodell in eine dreidimensionale Darstellung überführt wird. Da nicht für jedes Problem auch dieselbe Visualisierung zielführend ist, besteht hier die Möglichkeit zur Individualisierung.

Bei der individuellen Ausgestaltung sind zwei unterschiedliche Aspekte zu unterscheiden: Auf der einen Seite ist die Frage, wie die Daten an sich dargestellt werden, zum Beispiel in Bezug auf Form und Farbe und das sonstige Erscheinungsbild. Auf der anderen Seite muss spezifiziert werden, in welcher Art und Weise die Dimensionen der Daten dargestellt werden.

Resultat des Mappingvorgangs soll ein Szenenbaum sein. Er ist die für die Grafikengine lesbare Form des Datenmodells. Dieser Szenenbaum muss bei jeder Veränderung des Datenmodells aktualisiert werden. Als solche zählt zum Beispiel das Filtern von Daten nach bestimmten Eigenschaften, oder das Erstellen einer Sicht auf das Datenmodell.

**Spezialisierung** Wie oben bereits beschrieben hat die Spezialisierung im Mapper zwei Aspekte. Auf der einen Seite kann die Gestalt eines einzelnen Datums spezifiziert werden und auf der anderen Seite die Position dieses Datums im dreidimensionalen Raum. Eine vollständige Trennung ist in der derzeitigen Version nicht gelungen. Inwiefern sich das auswirkt, wird im Folgenden beschrieben.

Position und Projektion Eine grundlegende Entscheidung bei der Visualisierung von Daten ist, wie aus den Rohdaten eines Datums die Koordinaten im dreidimensionalen Raum gewonnen werden. Bereits in Abschnitt 4.1.1 wurde beschrieben, dass solche Rohdaten existieren müssen. Ebenfalls ist darauf verwiesen worden, dass ungeeignete Rohdaten (weder in Gleitkomma-, noch in ganzzahliger Darstellung) an dieser Stelle durch einen geeigneten Homomorphismus umzuwandeln sind.

Der erste Schritt zur Individualisierung des Mappers ist die Bestimmung der Ausmaße der Projektionsfläche. Dieser begriff meint die Ebene, die trivialisiert durch Länge und Breite aufgespannt wird. Ihre Festlegung erfolgt durch die Implementierung der *getSize()*-Methode. Hier gibt es zwei Varianten der Umsetzung. Es kann eine feste, von den Daten unabhängige Projektionsfläche gewählt werden. Eine maßstabsgetreue Abbildung entfällt somit. Vorteil dieser Art ist, dass bereits zur Implementierungszeit abschätzbar ist, wie groß die maximalen Entfernungen ausfallen. Somit können Entscheidungen, wie die Positionierung der Kamera [REF] sinnvoll getroffen werden.
Eine andere Möglichkeit ist eine von den Eingabedaten abhängige Projektionsfläche. Hier entfällt das abschätzbare Ausmaß der Darstellung zum Wohle der maßstabsgetreuen Abbildung.
Eine der wichtigsten Verwendungen dieser Methode liegt im späteren Erzeugen des Gittermusters, welches eine bessere Einschätzung der Dimensionen ermöglicht.

Der zweite Schritt zur Individualisierung ist die Festlegung der Skalierung der Position auf den einzelnen Dimensionen. Die Notwendigkeit hierfür kann sich aus ungünstig gestalteten Rohdaten ergeben. Um zum Beispiel Zeiten in Ortsangaben umzuwandeln, kann man auf die interne Darstellung eines Zeitpunktes in die Zahl der Millisekunden seit dem 01.01.1970 zurückgreifen. Hieraus ergeben sich jedoch schon für kurze Zeitspannen riesige Zahlenwerte, die zu ebensogroßen Ausmaßen der Darstellung führen würden. Mit der Implementierung der Methode *getDimensionFactors()* kann diesem Umstand Rechnung getragen werden. Ist eine Skalierung nicht notwendig, so kann die triviale Skalierung mit dem Faktor 1 erfolgen.

Der letzte Schritt zur Positionierung ist die Festlegung eines Datums auf eine genaue Koordinate in der dreidimensionalen Darstellung. Implementiert werden muss dies gleichzeitig mit der Festlegung der Gestalt eines Datensatzes. Wie dies zu erfolgen hat, ist in der allgemeinen Beschreibung zur Positionierung in Abschnitt [REF jme, Positionierung] beschrieben. Der Ort dieser Implementierung ist die im Folgenden beschriebene Methode.

Gestalt Die Gestalt eines Datums wird von der Methode *extractNodeFromData()* bestimmt. Sie hat die Aufgabe einen Szenenknoten (siehe Abschnitt [REF]) für jeden Datensatz zu erstellen. Bestandteil muss auf jeden Fall ein Grafikobjekt, wie in Abschnitt [REF, Visualisierung eines Datums] beschrieben, sein. Im letzen Abschnitt wurde bereits angesprochen, dass ebenfalls die Festlegung der Koordinate des Grafikobjekts an dieser Stelle zu erfolgen hat.
In dieser Methode kann der Benutzer neben Form, Größe, Position und sonstigen Eigenschaften, wie zum Beispiel Renderingoptionen (siehe Abschnitt [REF jme, Alphablending]) auch die Texturierung von Objekten festlegen. Es ist jedoch nicht immer zweckmäßig, für jedes Objekt eine individuelle Textur zuzuweisen. Aus diesem Grund gibt es eine weitere Methode, die es gilt zu implementieren.

Die Methode *texture()* hat zur Aufgabe die Grafikobjekte eines übergebenen Szeneknotens zu texturieren. Dies kann anhand von bestimmten Eigenschaften geschehen. Enumerationen eignen sich hierbei sehr gut. Wie in Abschnitt [REF Visualisierung eine Datums] beschrieben, sollte man sich an dieser Stelle auf eine noch vom Kurzzeitgedächtnis erfassbare Menge an Farben beschränken. Ist es jedoch nicht notwendig, so kann die Implementierung dieser Methode leer verbleiben.

**Hilfsfunktionen** Hilfsfunktionen der Art, wie sie aus dem Importer bekannt sind, gibt es im Mapper nicht. An dieser Stelle soll jedoch die Methode *getDataRoot()* erwähnt werden, die den Zugriff auf den vom Mapper erstellten Szenegraphen ermöglicht. Sie ist in zwei Ausprägungen vorhanden: Einerseits ohne Angabe von Argumenten um an die Visualisierung des Datenmodells zu gelangen, das bei der Erstellung des Mappers übergeben wurde.
Andererseits kann auch ein Datenmodell, welches zum Beispiel durch Filtern entstanden ist, übergeben werden. Es wird automatisch zum neuen Modell des Mappers und daraus wird der neue Szenengraph erzeugt.

### 4.1.4 Zusammenspiel der Komponenten

#### Visualisierungsprozess

#### Importer und Mapper

#### Viewer und Importer

#### Viewer und Mapper

## 4.2 Visualisierung einer militärischen Lage

## 4.3 Kernaspekte der Implementierung

### 4.3.1 Mapper

#### Streckung der Eingabedaten auf den Projektionsbereich

#### Entfernungsberechnung mithilfe der Haversine Formel

#### Informationsverlust durch Projektion eines Kugelabschnitts auf eine Ebene

### 4.3.2 Viewer

#### Mousepicking

#### Bewegungskegel

# 5 Fazit und Ausblick

## 5.1 Bewertung

## 5.2 Weiterführende Arbeit

## 5.3 Fazit

# Literaturverzeichnis

[GHJV02] Gamma, Erich, Richard Helm, Ralph Johnson und John Vlissides: *Design patterns: abstraction and reuse of object-oriented design*, Seiten 701–717. Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[Hal92] Hall, David L.: *Mathematical Techniques in Multisensor Data Fusion*. Artech House, Inc., Norwood, MA, USA, 1992.

[HL97] Hall, D. L. und J. Llinas: *An introduction to multisensor data fusion*. Proceedings of the IEEE, 85(1):6–23, 1997.

[Var97] Varshney, P. K.: *Multisensor data fusion*. Electronics & Communication Engineering Journal, 9(6):245–253, 1997.

# Abbildungsverzeichnis

# Listings