TECHNISCHE UNIVERSITÄT MÜNCHEN

Technische Universität München
Fakultät für Informatik
Lehrstuhl für Software und Systems Engineering
Prof. Dr. Dr. h.c. Manfred Broy

Systementwicklungsprojekt

Entwicklung einer Komponente für die Auswertung von Positionsdaten

Dimitri Alexeev

Aufgabensteller:	Prof. Dr. Dr. h.c. Manfred Broy
Betreuer:	Dipl.-Inf. Sebastian Winter
Abgabedatum:	15. November 2005

1. Einführung

In einem modernen Fahrzeug verarbeitet der Fahrer eine gewaltige Menge von Informationen. Zahlreiche Anzeige- und Bedienelemente, die eigentlich den Fahrkomfort und die Sicherheit erhöhen sollen, können unter Umständen kontraproduktiv werden. Der Fahrer kann sehr schnell „vom Lenken abgelenkt“ werden und sich sowie andere in Gefahr bringen. Da die Anzahl verschiedener Kontrolleinheiten in den letzten Jahren stark gewachsen ist, rückt nun der ergonomische Aspekt in den Vordergrund.

1.1. Das TUMMIC-Projekt

Um Fahrkomfort und Sicherheit in Fahrzeugen der nächsten Generation zu erhöhen werden in einem Fahrsimulator an der TU München im Rahmen des TUMMIC-Projektes unterschiedlichsten Bedienkonzepte entworfen und getestet. TUMMIC steht für „Thoroughly User-Oriented Man-Machine Interface in Cars“. Das Projekt entstand in enger Kooperation mehrerer Institute der TU München (Lehrstuhl für Ergonomie, Fachbereich für Augmented Reality, Lehrstuhl für Mensch-Maschine-Kommunikation und Lehrstuhl für Software- und System-Engineering), sowie IPSK der LMU und des Lehrstuhls für Allgemeine und Angewandte Psychologie der Universität Regensburg.

Das Projekt beschäftigt sich in erster Linie mit der Verbesserung der Mensch-Maschine-Interaktion. Das Ziel ist ein integriertes multimodales Bedienkonzept für Fahrerassistenz- und -informationssysteme im Auto. Im Rahmen des Projekts wurde am Lehrstuhl für Ergonomie (LfE) ein Anwendungskomplex unter Verwendung von Augmented Reality Werkzeugen gebaut. Mit diesem Fahrsimulator kann man unterschiedlichsten Bedienkonzepte testen und vergleichen.

1.2. Aufgabenstellung und Ziele der Arbeit

Die Aufgabe dieses Systementwicklungsprojektes ist die Entwicklung einer Komponente für den Fahrsimulator. Während der Fahrer in dem Fahrsimulator auf einer virtuellen Strecke fährt, soll die Komponente, abhängig von der Position des Fahrzeugs, verschiedene Ereignisse auslösen. Anschließend werden andere Komponenten des Systems die ausgelösten Ereignisse empfangen, verarbeiten und grafisch bzw. akustisch darstellen.

Als Beispiel kann man die Positionserkennung von StVO-Zeichen nennen. Abhängig von aktueller Geschwindigkeit und Position kann der Fahrer gegebenenfalls vor einem

riskanten Manöver gewarnt werden. Es sind auch andere positionsbezogene Ereignisse oder *points of interest* (POIs) denkbar.

Das Eintreffen in den Geltungsbereich eines POI soll also in Echtzeit erkannt, interpretiert und weitergeleitet werden. Eine weitere Anforderung an die Komponente ist eine benutzerfreundliche Konfigurationsschnittstelle. Diese soll dem Benutzer des Fahrsimulators die Bearbeitung von POI-Daten erleichtern.
Die Entwicklung besteht aus folgenden Schritten:

1. Spezifikation der Komponente.
2. Auswahl eines geeigneten Algorithmus.
3. Realisierung der Komponente in Java.
4. Entwicklung einer graphischen Nutzerschnittstelle zur Konfiguration.
5. Integration in den Fahrsimulator.

In dem Zielsystem existierte bereits eine Komponente für die Positionsanalyse. Sie beinhaltete allerdings keine Benutzerschnittstelle. Die Entstellungen erfolgten mithilfe einer Konfigurationsdatei. Die neue Version soll die Konfiguration der Komponente erleichtern und Anpassungen während zur Laufzeit ermöglichen. Der eingesetzte Positionsanalyse-Algorithmus soll optimiert werden.

1.3. Aufbau der Arbeit

Die Ausarbeitung beginnt mit einem Überblick über verwendete Technologien im Kapitel Grundlagen. Dort werden Besonderheiten der Interaktion zwischen mehreren Programmen in einem Anwendungskomplex erläutert. Außerdem sollen einige Aspekte von Java SWING – einer API zum Programmieren von grafischen Benutzeroberflächen betrachtet. Im nächsten Kapitel wird die eigentliche Aufgabenstellung analysiert. Als Ergebnis dieser Analyse wird ein geeigneter Lösungsansatz gewählt. Im Kapitel Implementierung werden einzelne Schritte der Realisierung von Applikation beschrieben. Im Vordergrund stehen dabei die technischen Details. Zum Schluss wird eine Zusammenfassung der durchgeführten Arbeit präsentiert sowie ein paar Erweiterungsmöglichkeiten angedeutet.

2. Grundlagen

Der Schwerpunkt des SEP liegt in der Weiterentwicklung der Komponente zur Erzeugung von positionsbezogenen Ereignissen. Die Komponente Kontextserver verwaltet die globalen Informationen, die für situationsbezogene Entscheidungen relevant sind. Der Kontextserver ist in den Fahrsimulator integriert und kommuniziert mit anderen Subsystemen mittels DWARF Architektur. Die Abbildung 1 zeigt die Einordnung der Komponente in das Gesamtsystem. Der Kontextserver bildet eine Schnittstelle zwischen gelieferten Positionsdaten und dem dazugehörigen semantischen Datenmodel.

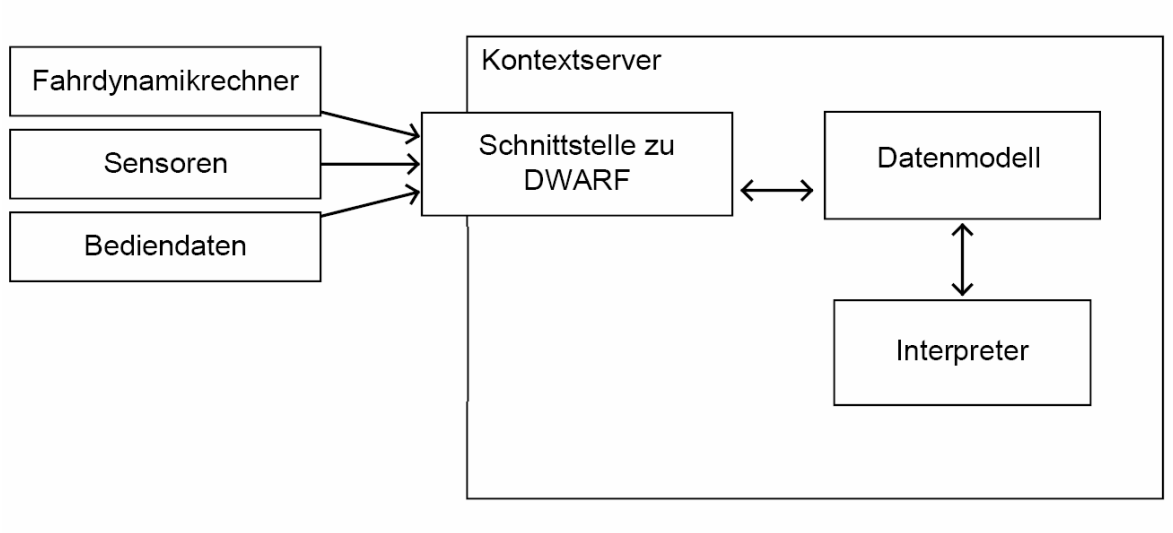

Abbildung 1: Einbettung in das Gesamtsystem

Nachfolgend werden die grundlegenden Technologien und Systeme präsentiert die bei der Umsetzung des Fahrsimulators und insbesondere der Kontextserver-Komponente eingesetzt wurden.

2.1. DWARF

Am Lehrstuhl für Computer Aided Medical Procedures wird im Rahmen des Projekts DWARF [2] eine Softwarearchitektur entwickelt, die das Erstellen von Augmented-Reality-(AR)-Systemen auf verteilten Computersystemen durch vorgefertigte Komponenten vereinfachen soll. Das Konzept von DWARF (Distributed Wearable Augmented Reality Framework) ist die Aufteilung des Systems in mehrere kooperierende Dienste, die auf verschiedenen Systemen parallel laufen. Die Dienste werden durch eine intelligente Middleware miteinander verschaltet und erledigen automatisch grundlegende AR- Aufgaben wie Benutzerpositionsbestimmung oder dreidimensionale Anzeige.

Dank Einsatzes der CORBA-Technologie ist eine Kommunikation zwischen Diensten möglich, die in verschiedenen Programmiersprachen geschrieben sind, etwa Java und C++. Jeder Dienst in DWARF beinhaltet einen Object Request Broker, der Teil einer CORBA-Implementierung ist. Grundsätzlich lässt sich das DWARF-Framework in die folgenden drei Elemente aufteilen:

1. *Conceptual architecture* - ist eine allgemeine Beschreibung von AR-Systemen.
2. *Main services* - liefern die Grundfunktionalität des AR- Systems.
3. *Middleware* - ist zuständig für die Kommunikation zwischen den einzelnen Diensten.

Einer der *main services*, der für das SEP besonders wissenswert ist, ist das Tracking-Subsystem. Wie die Abbildung 2 zeigt, besteht das Tracking-Subsystem aus verschiedenen Tracking-Einheiten (GPS-Tracker, optische Tracker, Trägheitstrascker, usw.) und einem Tracking-Manager. Die einzelnen Tracking-Einheiten liefern Positions- und Orientierungsdaten und geben diese an den Tracking-Manager weiter. Der Tracking-Manager wiederum kann die Positionsdaten mehrerer Tracker kombinieren und/oder filtern und anschließend weitergeben, mit Angaben zu Genauigkeit, Verzögerung, Updaterate usw.

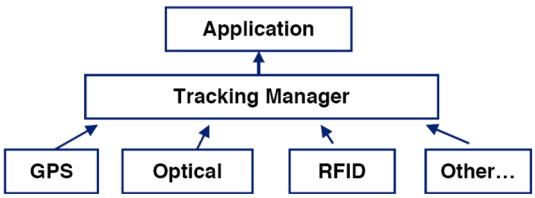

Abbildung 2: Aufbau von Tracking- Subsystem

2.2. CORBA

CORBA steht für Common Object Request Broker Architecture. CORBA ist ein Standard zur Entwicklung verteilter objektorientierter Anwendungen. Es wurde von dem herstellerunabhängigen Konsortium OMG (Object Management Group) [1] eingeführt. Ziel ist es, eine Integrationsplattform zur Realisierung verteilter Objekte zu schaffen, die unabhängig von Rechnerarchitektur, Programmiersprachen und Betriebssystem ist. Das Paradigma der verteilten Programmierung hat einige Vorteile gegenüber klassischen Programmierparadigmen. Verteilte Komponenten ermöglichen:

- echtes paralleles Rechnen
- verteilte Datenhaltung
- verschiedene Hersteller können zusammenarbeiten ohne den jeweiligen Source-Code zu kennen.

Wie auf der Abbildung 3 zu sehen, bildet der *object request broker* (ORB) die Basiskomponente für die Kommunikation in verteilten Anwendungen. Er dient dazu, Client/Server-Beziehungen zwischen Objekten aufzubauen. Ein Client ist dabei ein Objekt, das eine Operation eines anderen Objektes aufrufen möchte. In der Regel werden die Implementierungen mehrerer Objekte zu einem Programm zusammengefasst, das dann als Server fungiert [4].

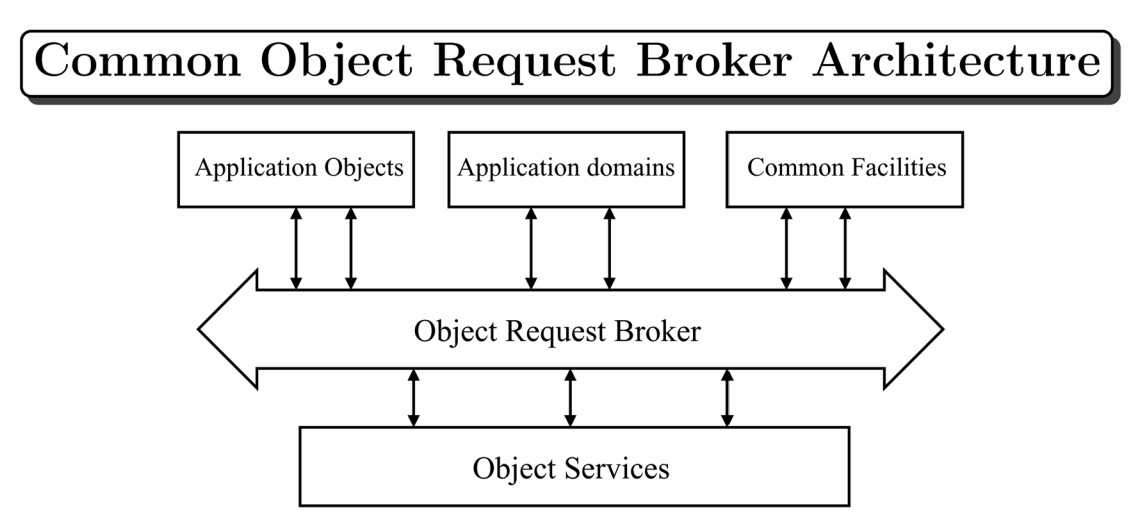

Abbildung 3: CORBA Komponenten

Um die Entwickler bei der Arbeit zu unterstützen, hat die OMG eine Reihe von systemnahen Diensten standardisiert, die die Funktionalität des ORB erweitern, die *object services*. Mit seiner Fähigkeit, verteilte Objekte zu lokalisieren, ist der *naming*-Service der wohl fundamentalste CORBA-Dienst. Er wurde als einer der ersten Dienste von der OMG spezifiziert und ist inzwischen von nahezu allen ORB-Herstellern implementiert.

Der *event*-Service (Ereignisdienst) hat die Aufgabe, die asynchrone Kommunikation zwischen anonymen Objekten zu ermöglichen. Er tritt dabei als eine Art Vermittlungsstelle zur Entkopplung der Aufrufer und der Aufgerufenen auf. Objekte, die an einem bestimmten Ereignis interessiert sind, registrieren sich hierfür beim Ereignisdienst. Dieser benachrichtigt bei Auftreten dieses Ereignisses alle angemeldeten Objekte. Es gibt auch andere Object Services, diese beschäftigen sich beispielsweise mit der persistenten Speicherung von Objekten, mit der transaktionsorientierten Bearbeitung entfernter Aufrufe oder mit Sicherheitsmechanismen.

Ein bedeutender Aspekt bei CORBA ist auch die Programmiersprachen-Transparenz der Objektimplementierungen. Hierzu stellt die OMG eine Schnittstellenbeschreibungssprache bereit, die IDL (*Interface Definition Language*). Mit ihr werden die verteilten Objekte an

ihren äußerlich sichtbaren Eigenschaften in einer standardisierten und implementationsunabhängigen Weise beschrieben. Um ein Objekt zu implementieren, kann dann auf verschiedene Programmiersprachen zurückgegriffen werden, z.B. C++, Java, aber auch nicht objektorientierte Sprachen wie C. Für diese Sprachen definiert die OMG *language mappings*, die angeben, wie die in IDL angegebenen Interfaces und Datentypen in Konstrukte der Programmiersprache umgesetzt werden.

2.3. Java SWING

Swing ist eine frei verfügbare Klassenbibliothek von SUN Microsystems. Es stellt für Java einen kompletten Satz an Oberflächenelementen bereit, wie z.B. Toolbars, Tooltips, Internal Frames und Progress Bars. Das *look&feel* der Benutzeroberfläche lässt sich dynamisch verändern.

Die Architektur von SWING basiert auf einer modifizierten Version des bekannten Model-View-Controller-Patterns. Im klassischen Model-View-Controller-Design besteht jede Komponente aus drei Teilen: dem Model, dem View und dem Controller. Im Model-Teil einer MVC-basierten Komponente befinden sich die Werte bzw. Eigenschaften der Komponente, welche ihre Semantik definieren. Im Controller-Teil der Komponente werden Benutzer-Aktionen ausgewertet und an das Modell weitergeleitet. Der View-Teil ist schließlich für die visuelle Darstellung der Komponente (z.B. auf dem Bildschirm) verantwortlich. Die Vorteile einer MVC-basierten Architektur liegen klar auf der Hand, durch die Trennung der einzelnen Teile kann z.B. das Aussehen einer Komponente verändert werden, ohne Ihr Verhalten zu modifizieren. Aus der strikten Trennung von View und Controller ergibt sich allerdings auch ein Problem: Die Kommunikation zwischen dem View- und dem Controller-Teil einer Komponente kann sehr schnell äußerst komplex und unüberschaubar werden. Aus diesem Grund modifizierten die Swing-Entwickler die klassische MVC-Architektur und fassten den View- und den Contoller- Teil zu einem View-Controller zusammen [3].

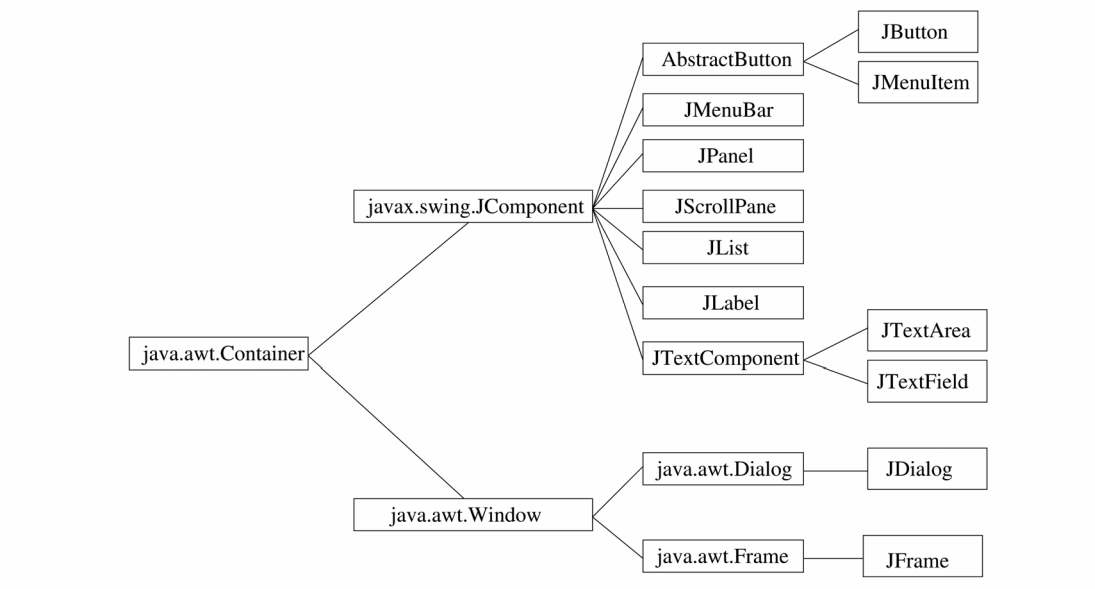

Abbildung 4: Swing- Komponenten

Im Gegensatz zu plattformspezifischen grafischen Bibliotheken benutzen Swing-Komponenten nur noch in sehr eingeschränkter Weise plattformspezifische GUI-Ressourcen. Wie die Abbildung 3 zeigt, werden alle GUI-Elemente abgesehen von Top-Level-Fenstern und Dialogen von Swing selbst erzeugt. Diese Vorgehensweise bietet einige Vorteile:

- Die plattformspezifischen Besonderheiten fallen weg, und der Code zur Implementierung der Dialogelemente vereinfacht sich deutlich.
- Die Unterschiede in der Bedienung entfallen auch. Der Anwender findet auf allen Betriebssystemen dasselbe Aussehen und dieselbe Bedienung vor.
- Swing ist nicht mehr darauf angewiesen, Features auf der Basis des kleinsten gemeinsamen Nenners aller unterstützten Plattformen zu realisieren. Komplexe Dialogelemente wie Bäume, Tabellen oder Registerkarten können plattformunabhängig realisiert werden und stehen nach der Implementierung ohne Portierungsaufwand auf allen Systemen zur Verfügung. Tatsächlich stellt Swing eine sehr viel umfassendere und anspruchsvollere Menge an Dialogelementen als das AWT zur Verfügung.

Der Grundstein für Komponenten, die von Java selbst erzeugt und dargestellt werden können, wurde im JDK 1.1 mit der Einführung der Lightweight Components („leichtgewichtige“ Komponenten) gelegt. Dabei wird die `paint`-Methode eines Component-Objects nicht mehr an die betriebssystemspezifische Klasse weitergeleitet, sondern in den Komponentenklassen überlagert und mit Hilfe grafischer Primitivoperationen selbst implementiert.

3. Anforderungsanalyse und Design

Zusammengefasst in drei Sätzen könnte die Aufgabe des SEP wie folgt aussehen:

Einer im Fahrsimulator dargestellten virtuellen Umgebung liegt ein dreidimensionales Modell zugrunde. Im Bezug auf dieses Modell besitzt das Fahrzeug bestimmte Koordinaten, deren Veränderung in Echtzeit ausgewertet werden soll. Die Ergebnisse der Auswertung von Positionsdaten sollen in einem einheitlichen Format für andere bereits vorhandenen sowie zukünftigen Komponenten des Fahrsimulators bereitgestellt werden.

Das nächste Kapitel beschäftigt sich mit der Spezifikation von Anforderungen sowie Design von Anwendungsklassen. Dabei werden einzelne Bausteine der Aufgabenstellung diskutiert und jeweils ein passender Lösungsansatz gewählt.

3.1. Anwendungsfälle

Im Rahmen des SEP war eine bereits im System agierende Komponente zu erweitern. Die bestehende Komponente erledigte im Hintergrund alle mit Positionsanalyse verbundenen Berechnungen. Wie auf der Abbildung 5 zu sehen, wird die Steuerung des Fahrsimulators im DWARF Framework interpretiert und verarbeitet. Unter anderem werden dabei Veränderungen von Koordinaten verfolgt. Die aktuelle Position wird von der zu erweiternden Komponente `ContextServer` analysiert. Anschießend verarbeiten andere Komponenten im DWARF-System eventuell auftretende POI-Ereignisse und stellen diese visuell bzw. akustisch dem Fahrer des Fahrsimulators dar.

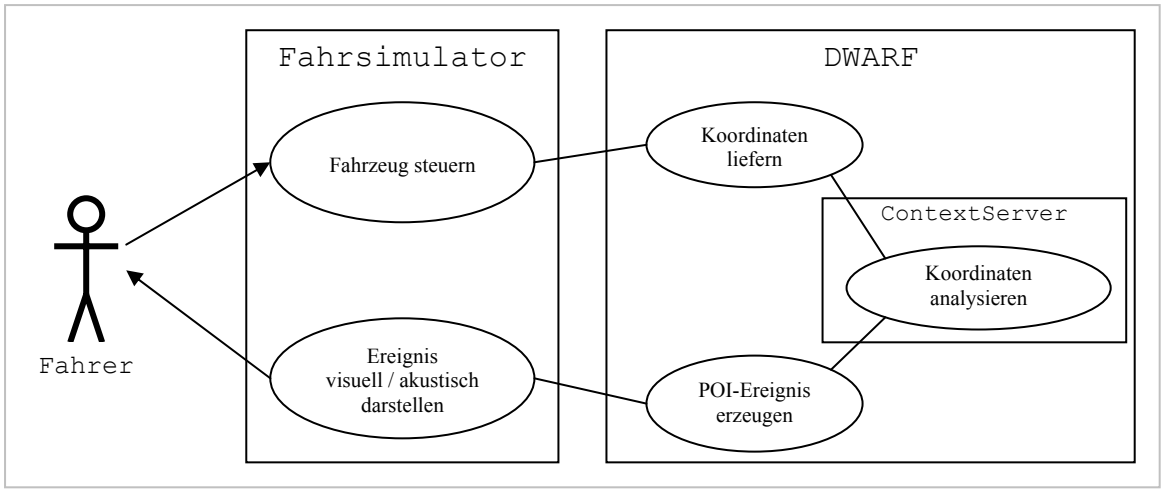

Abbildung 5: Use-Case-Diagramm „Auslösung von Ereignissen durch Steuerung des Fahrzeuges“

Im Zuge der Erweiterung der existierenden Komponente war ein neuer Anwendungsfall zu realisieren. In Anbetracht der Tatsache, dass der `ContextServer` die Positionsanalyse auf Basis einer statischen Menge von POI-Definitionen durchführte, war eine dynamische Veränderung des POI-Kontextes nicht möglich. Deswegen war eine der funktionalen Anforderungen, die Bearbeitung des Datenbestandes in Echtzeit zu ermöglichen. Die Abbildung 6 zeigt, wie ein neuer Akteur durch eine Interaktion mit dem `ContextServer` die Menge von POIs verwaltet und erweiterte Statusinformationen während der Positionsanalyse beobachtet.

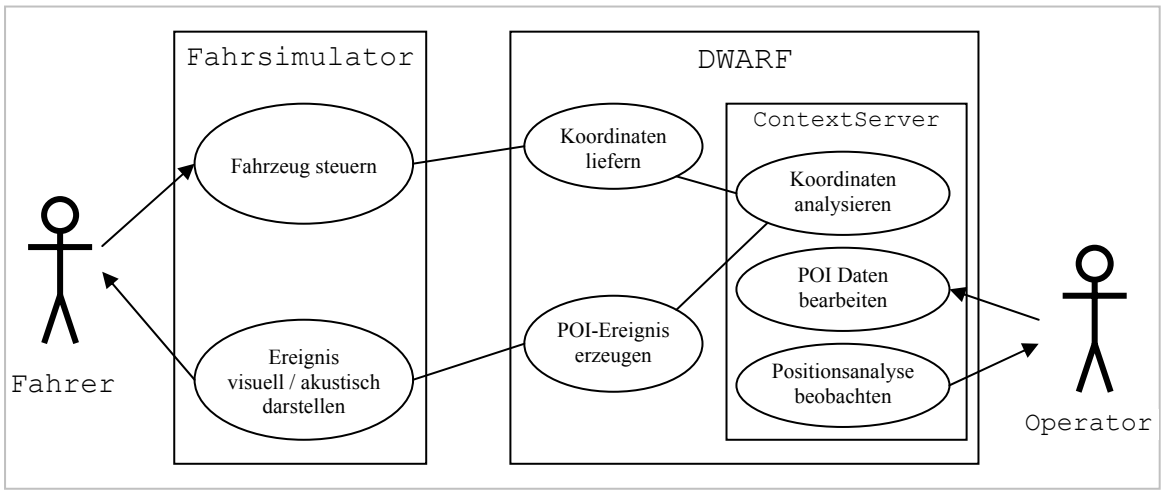

Abbildung 5: Use-Case-Diagramm „Direkte Interaktion mit ContextServer“

Neben genannten funktionalen Anwendugsfällen wurden einige wichtige nicht-funktionalen Anforderungen formuliert. Zum einen sollte der Algorithmus der Positionsanalyse optimiert werden. Die dazugehörigen Datenstrukturen sollten ebenfalls verbessert werden. Es wurde außerdem ein großer Wert auf Benutzerfreundlichkeit der Bedienoberfläche gelegt. Die Bearbeitung von POIs sollte möglichst einfach gestaltet werden.

3.2. Kommunikation mit dem Gesamtsystem

Um Positionsdaten des Fahrzeuges auswerten zu können, sollen diese über ein speziell eingerichtetes Ereignis-Strom von der anderen Komponente bezogen werden. Dafür wird die im Rahmen von DWARF-TUMICII-Projektes implementierte CORBA Schnittstelle verwendet, die mit einer Häufigkeit von 20 Ereignissen pro Sekunde die Fahrzeugkoordinaten liefert. Außerdem soll eine dynamisch veränderbare Menge von POIs zur Verfügung stehen. Ist der Abstand zwischen einem POI (z.B. Verkehrszeichen) und dem Auto kleiner oder gleich einer bestimmten Nahbereich- Konstante, so wird ein Ereignis ausgelöst und anschließend an andere Komponenten des Fahrzeugsimulators gesendet. Jeder POI besitzt eigene 3D-Koordinaten. In der Tat wird jedoch auf die dritte Komponente (Höhe über Meeresspiegel) verzichtet; darauf wird später noch einmal eingegangen.

Das Sequenzdiagramm auf der Abbildung 7 zeigt, wie das Gesamtsystem mithilfe dynamisch erzeugter Nachrichten mit der zu implementierenden Komponente kommunizieren kann. Über bereits implementierte Schnittstellen gelingen positionsbezogene Informationen in unsere Komponente `contextServer`. Dort wird die Fahrzeugposition analysiert und je nach Ergebnis eine POI Nachricht erzeugt und zurück an das Gesamtsystem gesendet.

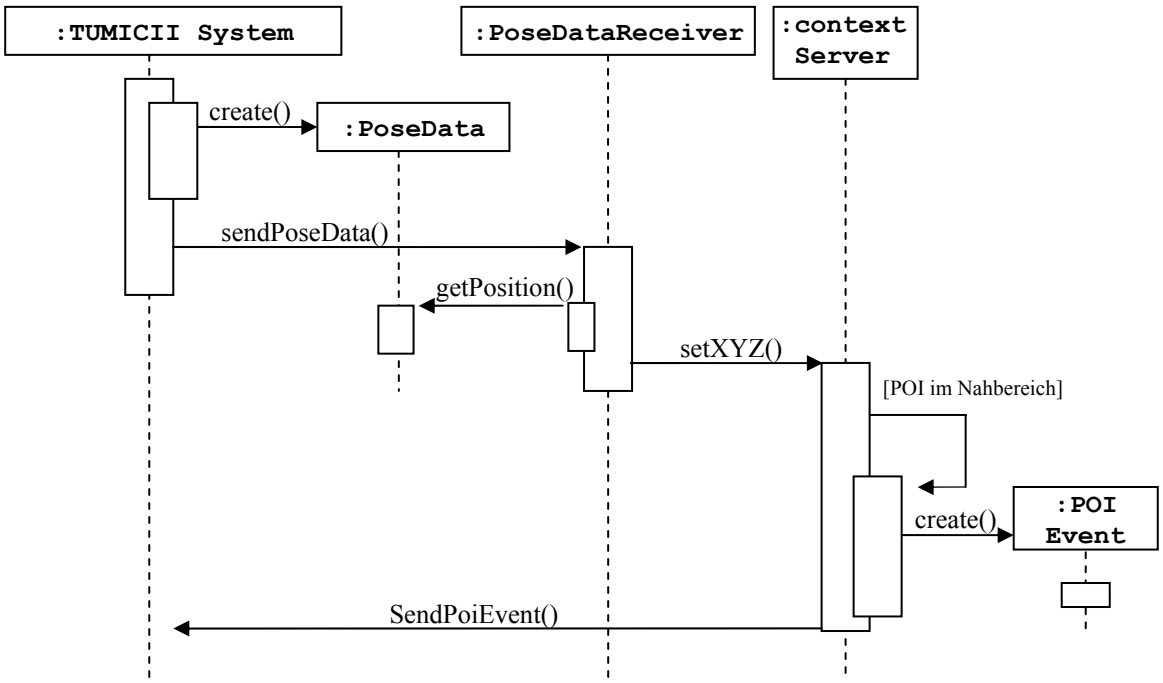

Abbildung 7: Sequenzdiagramm Nachrichtenaustausch

Jeder POI soll mit einem vordefinierten Ereignis verknüpft werden. Die Menge von Ereignissen stellt dabei eine Palette von allen möglichen ortsgebundenen Verkehrseigenschaften dar. Da die Menge von POIs nicht injektiv auf die Menge von Ereignissen abgebildet ist, führt jede Veränderung von Ereigniseigenschaften zur Änderung von allen damit verknüpften POIs. Jedes Ereignis bildet eine Datenstruktur, die über CORBA an andere Komponenten des Simulators kommuniziert werden kann.

3.3. Problemanalyse des Algorithmus

Die alte Version der Komponente realisierte eine simple Routine, die immer dann ausgeführt wurde, wenn das System Positionsdaten des Fahrzeuges geliefert hat. Der Algorithmus berechnete den Abstand zwischen dem Fahrzeug und POIs und erzeugte bei Unterschreitung einer vorgegebenen Nahbereich-Konstante ein POI-Ereignis. Leider wurde das POI-Ereignis nicht nur einmal erzeugt, sondern mehrmals, solange das Auto den Nahbereich des entsprechenden POI nicht verlassen hat. Außerdem wurde das Gesamtsystem mit der Bearbeitung von den Ereignissen oft überlastet, weil die Koordinatenquelle mit 20 Positionsnachrichten pro Sekunde eine Lawine von POI-Meldungen verursachte. Man hat versucht, die Überlastung des gesamten Frameworks zu vermeiden, indem man die Anzahl von Positionsnachrichten auf 5 Meldungen pro Sekunde reduzierte. Die Herabsetzung der Positionsdaten-Abtastrate hat aber dazu geführt, dass das Auto sehr oft die POIs „übersprungen“ hat, ohne ein POI-Ereignis auszulösen.

Unter Berücksichtigung der oben genannten Problempunkte wurde eine Liste von Anforderungen erstellt, die in der modifizierten Version des Algorithmus erfüllt werden müssen:

1. Bei jeder Änderung Koordinaten des virtuellen Fahrzeuges soll der Abstand zwischen dem Auto und den POIs überprüft werden.
2. Falls der Abstand zu einem POI einen bestimmten Wert unterschreitet, soll das mit dem POI verknüpfte Ereignis ausgelöst werden.
3. Solange sich das POI im Nahbereich des Autos befindet, dürfen keine weiteren Ereignisse von diesem POI ausgelöst werden.
4. Ein neues Ereignis kann durch den POI erst beim *Wiedereintreffen* in den Nahbereich ausgelöst werden.
5. Die Abtastrate von Fahrzeugkoordinaten darf nicht den Wert von 20 Hz unterschreiten.

Um das (Wieder)-Eintreffen des virtuellen Fahrzeuges richtig zu behandeln, soll eine Liste von POI's geführt werden, die zuvor im Nahbereich vom Auto gewesen sind. Die Abbildung 8 zeigt, wie abhängig von dem Inhalt der Liste sowie POI's im Nahbereich eine oder mehrere neue „Kollisionen" erkannt werden können.

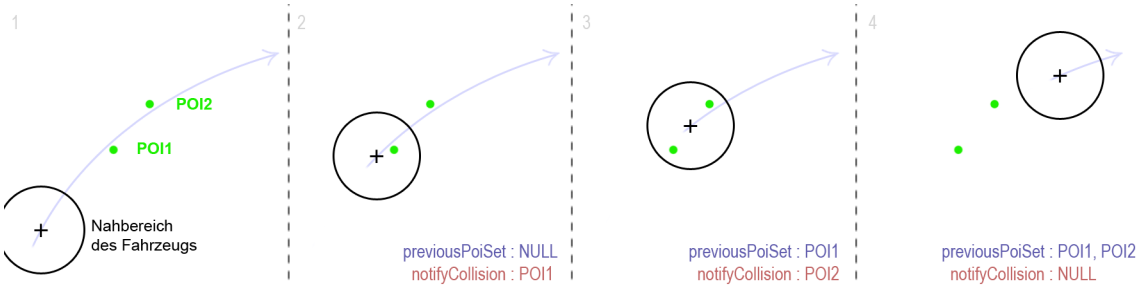

Abbildung 8: Betreten und Verlassen des Nahbereiches eines oder mehreren POIs

Der Inhalt der Liste von zuletzt im Nahbereich registrierten POI's, kurz *previousPoiSet,* wird immer wieder mit dem aktuellen Zustand vergleichen. Befindet sich im Nahbereich ein POI, der gleichzeitig keine Eintragung im *previousPoiSet* hat, so wird das als Eintreffen in den Nahbereich des Fahrzeuges, kurz *collision*, interpretiert.

Der modifizierte Algorithmus ist nicht perfekt: Es können rein hypothetisch Situationen vorkommen, in denen ein sehr schnell fahrendes Auto ein POI vorbeifährt, ohne ein Ereignis auszulösen. Anhand von Testfahrten am Ende der Implementierungsphase sollte deswegen geprüft werden, ob die gewählte Positionserfassungsrate sowie die Nahbereichskonstante eine reibungslose *collision*-Erfassung tatsächlich ermöglichen.

3.4. Design der Datenstruktur

Um die Flexibilität der Komponente zu erhöhen soll sowohl POI-Menge als auch Ereignismenge austauschbar sein. Damit kann für jeden Test ein geeignetes Datenumfeld vorbereitet werden. In diesem Zusammenhang sind zwei Datenmengen von maßgeblicher Bedeutung: POI- Koordinaten und POI-Ereignis-Definitionen. Die POI-Koordinaten bestimmen die Positionen, an welchen die orttsbezogene Ereignisse ausgelöst werden sollen. Diese Daten sind jedoch von der semantischen Bedeutung des Ereignisses getrennt. Wie die Abbildung 9 zeigt, können POI-Ereignis-Definitionen separat verwaltet und über einen eindeutigen Primärschlüssel dynamisch referenziert werden.

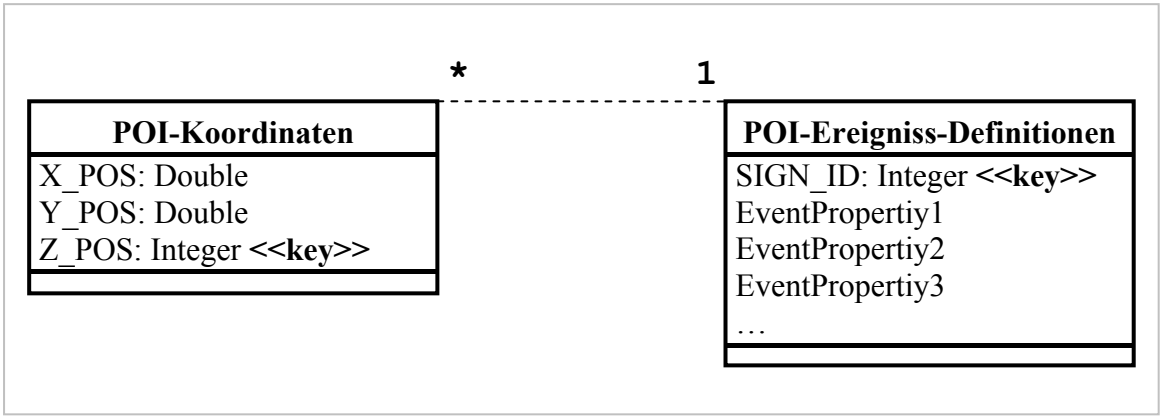

Abbildung 9: 1:n Assoziation von POI-Koordinaten und Ereignis-Definitionen

Solche strukturelle Trennung ermöglicht es beispielsweise, für eine bestimmte POI-Koordinaten-Menge verschiedene Ereignissmengen zu hinterlegen und je nach Bedarf auszutauschen. Bei verschiedenen Experimenten können dadurch unterschiedliche Ereignissmuster definiert werden. Andersrum kann eine einmal definierte Ereignissmenge wiederverwendet werden, indem mehrere POI-Koordinaten-Mengen auf einem und demselben Eriegnis-Datenfeld arbeiten.

3.5. Analyse des Laufzeitverhaltens

Das Laufzeitverhalten des Algorithmus in der Abhängigkeit von der Anzahl n der Elemente in der gegebenen POI-Menge kann anhand von der auf der Abbildung 10 dargestellten schematischen Realisierung berechnen.

```
public void Algorithmus() {

     ArrayList poiModel = // hier wird eine Liste von POIs referenziert
     double x = // hier wird die aktuelle x-Koordinate referenziert
     double z = // hier wird die aktuelle z-Koordinate referenziert

     double dx;
     double dz;

     LinkedList newPoiSet = new LinkedList();

     for (int i = 0; i < poiModel.size(); i++)  {
         // (1)
         PoiElement pe = poiModel.get(i);

         dx = pe.xPos - x;
         dz = pe.zPos - z;
         if ((dx*dx +dz*dz) < collisionRadiusQuadrat)  {
              // (2)
              newPoiSet.add(i);
              // (3)
              if (!previousPoiSet.contains(i)) notifyCollision(i);
         }
     }
```

Abbildung 10: Berechnung des Abstandes von POIs

Bei der Berechnung der Abstände zwischen dem Fahrzeug und den POI-Elementen werden alle n Elementen in einer Schleife bearbeitet. Es ergibt sich eine lineare Komplexität. Für den Inhalt der Schleife gilt außerdem:

1. Die Zugriffszeit (1) beim Lesen eines Elementes aus dem Datenfeld `poiModel` vom Typ `ArrayList` ist konstant und hängt nicht von der Anzahl der Elemente ab.
2. Die Zeit beim Anhängen (2) eines neuen Elementes an die verknüpfte Liste `newPoiSet` vom Typ `LinkedList` ist konstant und hängt nicht von der Anzahl der Elemente ab.
3. Die Suche nach einem Element (3) in der verknüpften Liste `previousPoiSet` hat lineare Komplexität. Man vermeidet jedoch, in dem Nahbereich eines POI einen anderen POI zu platzieren, geschweige denn gleich mehrere POIs sehr nah zu einander zu setzen. Unter Berücksichtigung dieser Besonderheit ist die Komplexität der Suche nahezu konstant.

Der Algorithmus hat somit die Komplexität O(n). Der Speicherverbrauch ist ebenfalls linear: Die POIs werden in einem ein-dimensionalen Datenfeld gelagert.

3.6. Aufbau der Applikation

Das Grundgerüst der Komponente ist vor allem geprägt durch die Art und Weise, wie die Anwendung in der Zukunft mit dem Benutzer interagieren soll. Eine der wichtigsten Anforderungen an die Komponente ist eine benutzerfreundliche Konfigurations-Schnittstelle. Diese soll die Bearbeitung von POI Daten effizienter machen. Die Anforderungen and die GUI wurden in enger Zusammenarbeit mit den anderen Entwicklern des Fahrsimulators immer wieder optimiert. Zwei Eigenschaften der Benutzerschnittstelle beistimmen die Aufbau der Gesamten Applikation: Das MVC-Prinzip sowie die *optionale* Möglichkeit, die Anwendung ohne GUI zu starten.

Wie bereits erwähnt wurde für die Entwicklung der grafischen Oberfläche die JAVA-Swing-Bibliothek gewählt. Diese ist nach dem Prinzip der klaren Trennung von *model* und *view-controller* aufgebaut. Dem Prinzip wurde bei dem Design der grafischen Benutzerschnittstelle befolgt, nicht zuletzt um die nachfolgende Implementierung von backend/frontend Ausführungsmodi zu vereinfachen. Die Applikation soll am Ende in zwei verschiedenen Modi ausgeführt werden: Entweder im Backend, in Form eines Dienstes; oder im Frontend, mit einem grafischen Benutzerinterface. In Begriffen der MVC Paradigma entspricht eine im Hintergrund gestartete Komponente einem Modell, dessen Verhalten weder sichtbar (*view*), kontrollierbar (*controller*) ist.

Die getrennte Ausführung vom *model* und *view-controller* erzwingt uns, ganz besondere Mechanismen für die Kommunikation zwischen einzelnen Bestandteilen der Applikation einzusetzen. Tatsächlich stellen sich direkte Methodenaufrufe zum Teil als unmöglich heraus, weil es zu der Übersetzungszeit nicht bekannt ist, ob die Komponente vollständig im Fordergrund oder nur teilweise – im Hintergrund – ausgeführt werden soll. Deswegen wird nach dem Vorbild des DWARF Netzwerkes ein Nachrichtenmodell eingeführt. Die Abbildung 11 zeigt, wie eine von Hauptkomponenten des grafischen Benutzerinterfaces `GraphicPanel` einen Nachrichtenstrom zum dem Positionsmodell `AutoProperties` aufbaut. Sie benutzt dabei das Observer-Observable Muster.

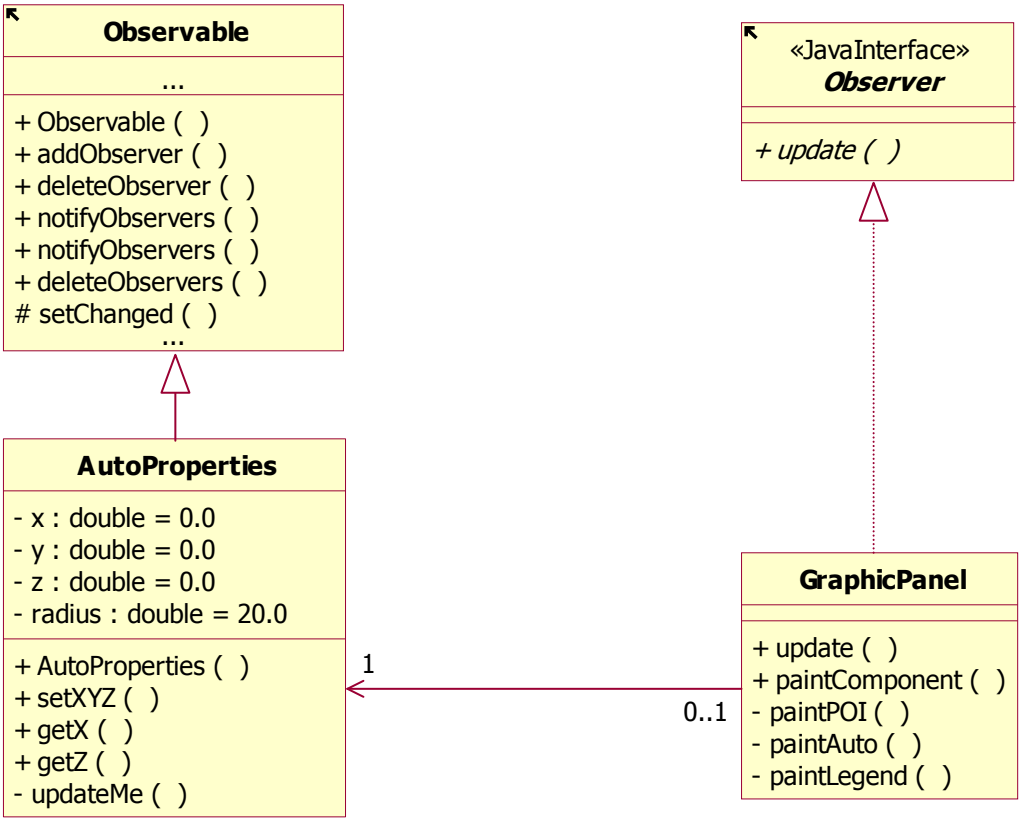

Abbildung 11: Observer-Observable Pattern

Dieses Pattern kommt immer dann zum Einsatz, wenn es mehrere Nachrichtenquellen sowie Nachrichtenempfänger gibt oder die Topologie der Verbindungen dynamisch veränderbar ist. Die Beobachter-Klassen müssen nicht zu der Kompilierungszeit bekannt sein und können dynamisch während der Laufzeit eine Verbindung zu den Objekten der Beobachtung aufbauen.

Neben `GraphicPanel` benutzt auch die zweite GUI-Komponente `EditScreen` dasselbe Kommunikationsmuster. Allerdings wird hier nicht nur die Fahrzeugposition überwacht sondern auch das Auftreten von POI-Ereignissen. Das entsprechende Model soll in der Klasse `PoiCollisionNotifyer` bereitgestellt werden, die ebenfalls Observable-Eigenschaften vererbt.

Auf der Abbildung 12 wurden alle Klassen zusammengefasst, die mithilfe von Observer-Observable Muster miteinander kommunizieren. Diese Klassen sind essentiell für die gesamte Anwendung und bilden das Gerüst der Applikation. Dank Nachrichtenkommunikation bestehen neben gezeichneten Assoziationen keinerlei weitere Klassen-Beziehungen. Einzige Ausnahme sind die beiden GUI Klassen; Es besteht eine Kompositionsbeziehung zwischen `EditScreen` und `GraphicPanel`.

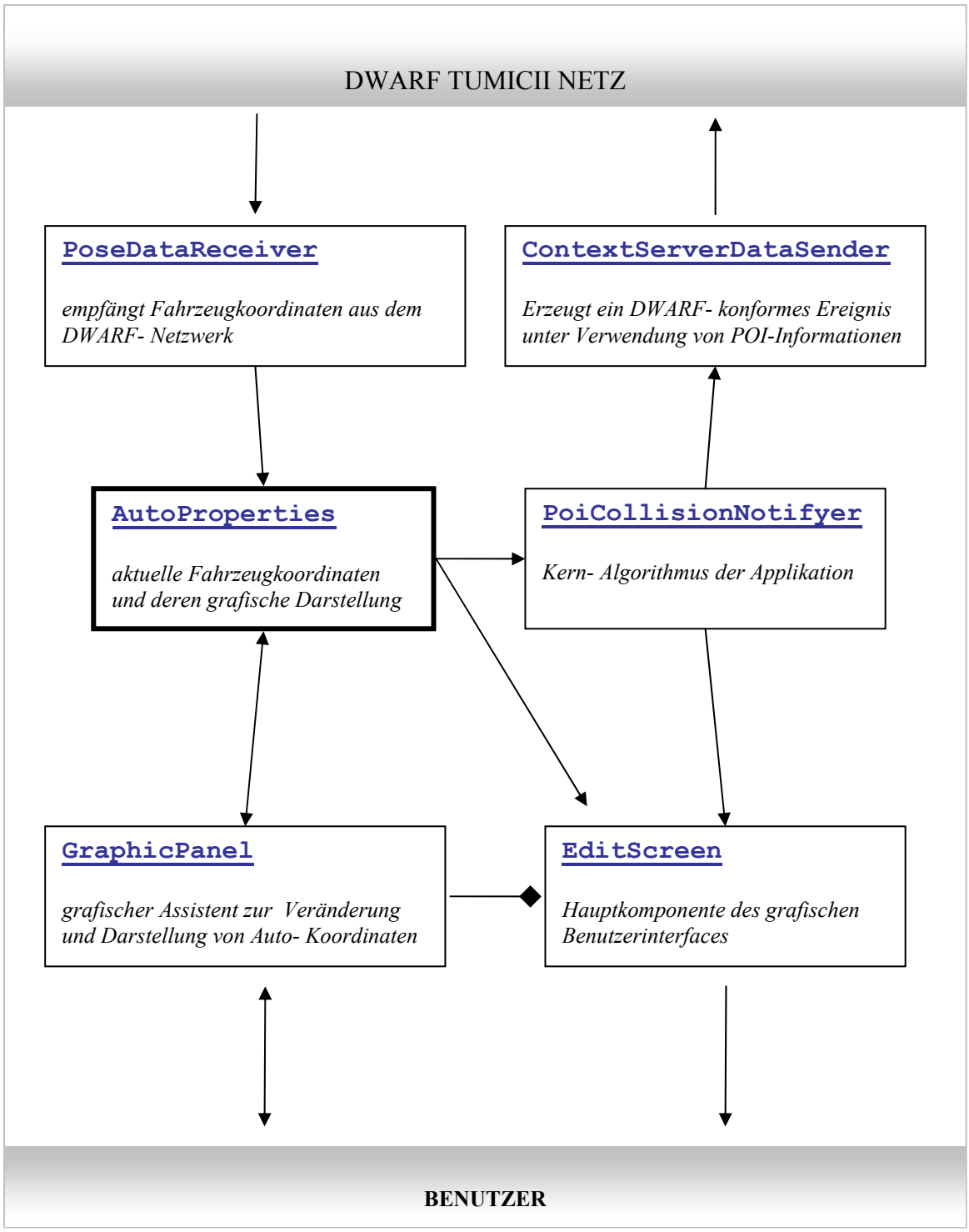

Abbildung 12: Objekte, deren Veränderung von anderen Instanzen beobachtet wird

Diese Klassen haben bei der Beschreibung der Implementierung im nächsten Kapitel eine besondere Beachtung verdient. Die restlichen Klassen werden immer dann kommentiert, wenn dies für das Verständnis der Projektergebnisse relevant ist.

4. Implementierung

Basierend auf der Problemstellung während der Spezifikationsphase wurden konkrete Realisierungsschritte erarbeitet. Die Aufteilung der Arbeitsschritte erfolgte vor allem unter Berücksichtigung der drei wichtigsten Teilaufgaben:

1. Algorithmus zur Erkennung und Behandlung von POI- relevanten Koordinaten;
2. Design der grafischen Benutzerschnittstelle;
3. Integration in das Gesamtprojekt.

Im ersten Schritt wurde das Konzept der Positionsanalyse realisiert. Es wurden alle dafür notwendigen Datenstrukturen angelegt sowie Ereignisschnittstellen entworfen. Im zweiten Teil der Realisierung wurde eine grafische Schnittstelle zur Steuerung des Positionsalgorithmus sowie zur Bearbeitung von POI Datenbanken entwickelt. Im letzten Schritt wurde die *standalone*-Applikation in das Gesamtsystem des Fahrsimulators integriert. Anschließend standen Tests auf Stabilität und Korrektheit der Komponente im Programm.

Nachfolgend werden die technischen Aspekte der Implementierung erläutert, die vor allem für die Weiterentwicklung der Komponente hilfreich sein können. Aus mehreren Gründen wurde für die Entwicklung der Komponente ausschließlich Java 1.4 verwendet. Zum einen sind CORBA-Schnittstellen für Datenaustausch innerhalb des DWARF Netzwerkes bereits in Java implementiert, zum anderen ist die Gestaltung von grafischen Elementen des Benutzerinterfaces mithilfe SWING 2 Klassenbibliothek besonders einfach.

4.1. Implementierung des Positionsanalyse-Algorithmus

Den Kernalgorithmus der Komponente, wie er im Punkt 3.3 beschrieben wurde, realisieren in der Interaktion mit einander die zwei Klassen:

- Klasse `AutoProperties` implementiert das Zustandsmodel des Fahrzeuges
- Klasse `PoiCollisionNotifyer` implementiert die Analyse der Veränderungen von `AutoProperties`

Klasse AutoProperties

Beschreibung:

Enthält stets die aktuelle Position des Fahrzeuges. Sie vererbt außerdem Eigenschaften und Methoden von `java.util.Observable` und benachrichtigt alle Beobachter, sobald das Positionsmodell geändert wurde.

Fields:

```
+ acceptedChanger : java.lang.Class
```

Gibt an, welche von den beiden Positionsquellen momentan die Änderungen am Positionsmodel-Zustand vornehmen darf: `PoseDataReciever` oder `GraficPanel`

Methods:

```
+ getX() : double
```

Gibt die aktuelle X-Koordinate des Fahrzeuges an.

```
+ getZ() : double
```

Gibt die aktuelle X-Koordinate des Fahrzeuges an.

```
+ getGeneralPath() : java.awt.geom.GeneralPath
```

Berechnet eine grafische Darstellung des Fahrzeuges im 2D Koordinatensystem.

```
+ setXYZ(double x, double y, double z, java.lang.Class changer)
```

Aktualisiert die Position des Fahrzeuges. Der Aufruf muss mit dem Klassennamen des aufrufenden Objektes authentifiziert werden.

Klasse PoiCollisionNotifyer

Beschreibung:

Eine Instanz dieser Klasse ist sofort nach dem Start der Komponente verfügbar und kommuniziert mit anderen Teilen der Anwendung über interne Ereignisse. Sie beobachtet jede Änderung der Fahrzeugkoordinaten. Um das Wiedreintreffen richtig zu behandeln, wird im Datenfeld `previousPoiSe`t eine Liste von POIs geführt, die zuvor im Nahbereich vom Auto gewesen sind.

Fields:

```
- previousPoiSet : LinkedList
```

Ermöglicht es, neu eintreffende POIs von den bereits behandelten zu unterscheiden und eventuell mehrere Ereignisse von der gleichen POI zu vermeiden.

Methods:

```
+ update(Observable o, Object arg)
```

Die Methode ist ein Teil des Beobachter-Musters. Jede Änderung der Fahrzeugkoordinaten stoßt die Berechnung der Kollisionsbedingungen in dieser Methode an.

Klasse PositionAnalyserComponents

Beschreibung:

Die Klasse selbst enthält keine Methoden. Sie aggregiert aber die bereits beschriebenen Modelle und fasst somit die einzelnen Teile des Komponentenmodels zusammen. Die Klasse wird sofort nach dem Start der Applikation instanziiert und erzeugt automatisch den gesamten Kontext der Komponente.

Fields:

```
+ autoProps : AutoProperties
```

eine aggregierte Instanz von Positionsmodel

```
+ contextServerDataSender : ContextServerDataSender
```

Kommunikationsscnittstelle mit DWARF

```
+ poseDataReceiver : PoseDataReceiver
```

Kommunikationsscnittstelle mit DWARF

```
+ defaultPdefFile : java.io.File
```

Vorgegebene Ereignis-Definitionen-Datenquelle

```
+ defaultPoiFile : java.io.File
```

Vorgegebene POI-Datenquelle

```
+ poiConnector : PoiConnector
```

Schnittstelle zur Veränderung von POI-Datenquelle

```
+ pdefConnector : PdefConnector
```

Schnittstelle zur Veränderung von Ereignis-Definitionen-Datenquelle

```
+ editScreen : EditScreen
```

Grafische Benutzerschnittstelle

```
+ gui : boolean
```

Ausführungsmodus der Komponente

```
+ poiCollisionNotifier : PoiCollisionNotifier
```

eine aggregierte Instanz von dem Ereignis-Berechnungs-Model

Zwei Klassen aus dem Package `de.bmw.tummicii.contextServer` bieten Schnittstellen für Erzeugung, Darstellung und Bearbeitung von POI- sowie Ereignis-Tabellen an: Es handelt sich um `PoiConnector` und `PdefConnector`. Beide Klassen beinhalten Methoden zur Verwaltung von Tabellen im Dateisystem. Klasse `PoiConnector` verwaltet außerdem eine grafische Darstellung der POI-Menge im Koordinatensystem. Beide Klassen erzeugen unter anderem Objekte vom Typ `javax.swing.table.AbstractTableModel`. Diese Objekte ermöglichen tabellarische Darstellung und Bearbeitung von Daten. Die Einzelheiten der Implementierung von Schnittstellen zur `javax.swing.JTable` sind den Klassen `PoiTableModel` und `PdefTableModel` zu entnehmen.

Wie im Punkt 3.2 beschrieben, besitzt jede Ereignis-Definition ein Primärschlüssel und kann dadurch in der POI-Koordinaten-Tabelle referenziert werden. Diese Verknüpfungsart ist vor allem in Datenbank- Strukturen vertreten, jedoch wäre eine Datenbank mit nur zwei Tabellen keine effiziente Lösung gewesen. Stattdessen werden POI- Koordinaten und Ereignis-Definitionen in zwei ASCII- Dateien verwaltet. Beide Tabellen sind TSV-formatiert (Tab Separated Values).

POI- Koordinaten-Tabelle hat folgendes Format:

```
X_POS [tab] Z_POS [tab] SIGN_ID [cr]
```

Die beiden Werteangaben X_POS und Z_POS stehen für die Koordinaten des Punktes in horizontaler Ebene. Man erlaubte sich im Rahmen des Fahrsimulators anzunehmen, dass das Auto immer auf dem Boden bleibt. Daher ist die Höhe des Punktes über Meeresspiegel in unserem Spezialfall irrelevant. Das Wert SIGN_ID gibt den Primärschlüssel zu der Ereignis-Definition aus der zweiten Tabelle.

Die Ereignis- Definitionen- Tabelle hat deutlich mehr Felder:

```
sign_id, gefahrstelle, kurveRechts, kurveLinks, doppelkurveRechts,
doppelkurveLinks, schneeOderEisglaette, schleudergefahr, kinder,
baustelle, wildwechsel, gegenverkehr, ueberholverbot,
richtgeschwindigkeit, zulaessigeHoechstgeschwindigkeit,
laengeGefahrstrecke, entfernungGefahrstrecke, text.
```

Die meisten Feldernamen sind selbsterklärend und beschreiben qualitative sowie quantitative Eigenschaften des ortsgebundenen Ereignisses. Das Feld „text“ dient zur Beschreibung von akustischen Sprachausgabe.

4.2. Grafische Benutzerschnittstelle

Die grafische Benutzerschnittstelle soll die Bearbeitung von POIs vereinfachen und deren Bezug auf die virtuelle Umgebung visualisieren. Es wurde entschieden, die Steuerung der Komponente in einem Fenster zusammenzufassen. Die Container-Klasse für das Fenster samt alle Bedienelementen ist die Klasse `EditScreen`. Eine detaillierte Beschreibung der einzelnen Schritte bei der GUI-Entwicklung wäre an dieser Stelle weniger sinnvoll: Bei den grafischen Elementen sagt ein Screenshot öfters mehr aus als ein Klassendiagramm. Deswegen beschränken wir uns hier auf Beschreibung der implementierten Funktionen.

Die obere Hälfte der graphischen Oberfläche fasst Bedienelemente zur Bearbeitung von POI- Koordinaten- sowie Ereignis- Definitionen- Tabellen. Die üblichen Normen der tabellarischen Datenbearbeitung sind eingehalten. Man kann zum Beispiel nach beliebigen Spalten sortieren, dafür soll die Kopfzeile der zu sortierenden Spalte angeklickt werden. Die Editierung von einzelnen Zellen ist mit einem Doppelklick möglich. Dabei werden die Benutzerangaben auf Plausibilität geprüft. Falsche Angaben, beispielsweise Buchstaben in einem numerischen Feld, werden sofort abgefangen und rot markiert, noch bevor das mit der Tabelle verbundene Datenmodel geändert wurde.

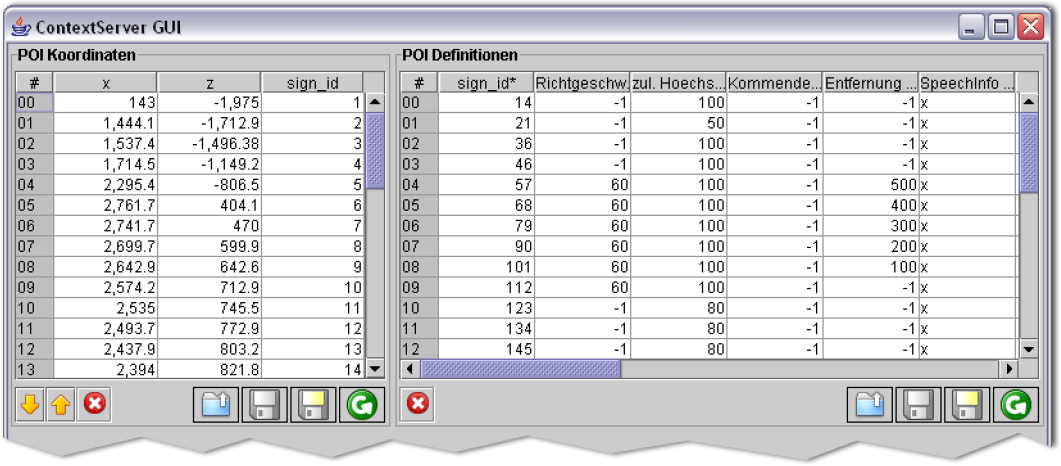

Abbildung 13: Graphische Benutzerschnittstelle, obere Hälfte

Nachfolgend werden die einzelnen Bedienelemente beschreiben. Eine kurze Funktionserklärung bekommt man übrigens auch während der Arbeit: Wenn Sie Ihre Maus ein paar Sekunden lang über ein Werkzeug oder eine Einstellung halten, erscheint ein kleines *tooltip*-Fenster mit Anweisungen und Informationen über den Bedienelement.

Folgende Bedienelemente stehen zur Verfügung:

Ausgewählten Datensatz niedriger/ höher positionieren

Verändert die Datensatz-Reihenfolge in der Datenbank. Nicht zu verwechseln mit der automatischen Spaltensortierung. Die Funktion wurde eingeführt, um Reihenfolge mit dem Vorkommen von POIs auf der virtuellen Strecke zu synchronisieren.

Datensatz löschen

Ausgewählte Zeile wird aus der Tabelle entfernt. Die Änderung ist sofort wirksam. Wird die Tabelle danach gespeichert, bleibt die Änderung permanent wirksam.

Konfigurationsdatei öffnen

Eine Tabelle aus dem Dateisystem laden

Konfigurationsdatei speichern

Die geöffnete Tabelle speichern

Konfigurationsdatei speichern unter

Die geöffnete Datei mit unter neuem Namen speichern

Konfigurationsdatei neu laden

Die bereits geöffnete Datei wird aus dem Dateisystem neu geladen.
Eventuelle Änderungen gehen dabei verworfen.

Die untere Hälfte der graphischen Benutzerschnittstelle liefert in Echtzeit ausführliche Informationen zum aktuellen Stand der Fahrsimulation im Bezug auf Position des Fahrzeuges. Im Fenster [POI Map] befinden sich eine Visualisierung der POIs sowie der Fahrzeugposition. Außerdem kann man die Komponente vom Gesamtsystem ganz oder teilweise isolieren indem man ankommenden und/oder ausgehenden Nachrichtenstrom deaktiviert.

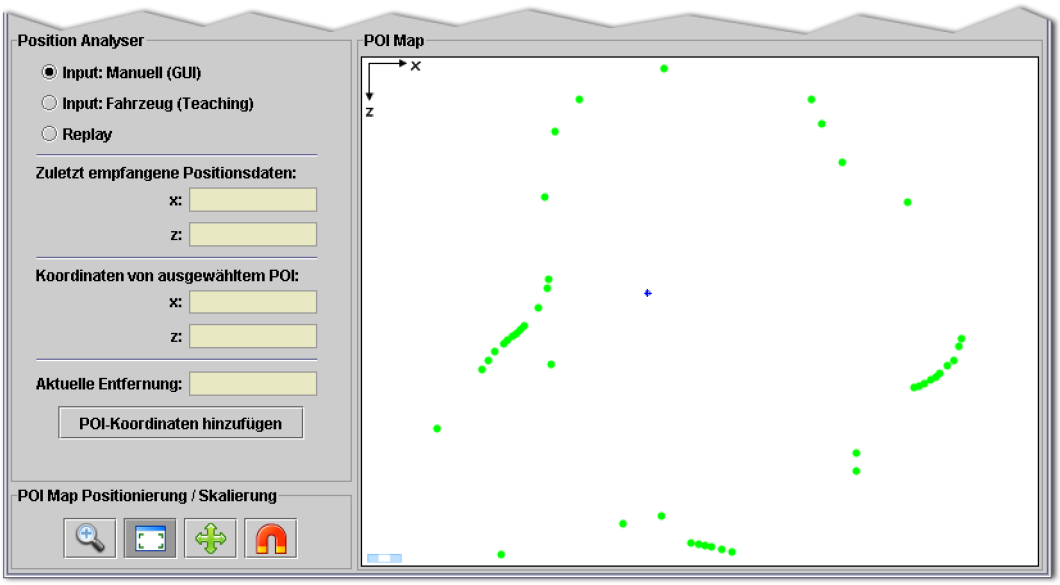

Abbildung 14: Graphische Benutzerschnittstelle, untere Hälfte

Für Testzwecke kann man eingehende Positionsnachrichten mit dem Mauszeiger simulieren. Mit einem Click&Drag durch rechte Maustaste verschiebt man den Fahrzeug in dem [POI Map] Fenster. Diese Funktion wird allerdings erst dann möglich, wenn man den ankommenden Strom von Positionsnachrichten deaktiviert. Dies geschieht durch die Auswahl „Input: Manuell“ im Bereich [Position Analyser].

Folgende Bedienelemente stehen zur Verfügung:

Input : Manuell	**Manuelle Positionierung des Fahrzeugs** Benutzen Sie die rechte Maustaste, um das Fahrzeug auf dem POI Map zu positionieren
Input : Fahrzeug	**Automatische Positionierung des Fahrzeugs** Fahrzeug-Positionsdaten werden über CORBA geliefert. (s. PoseService)
Replay	**contextServer zum verschicken von POI-Events wird aktiviert** Fahrzeug-Positionsdaten werden über CORBA geliefert, ausgewertet und zugehörige POI- Events werden erzeugt.
POI-Koordinaten Hinzufügen	**Aktuelle Fahrzeugposition in POI Datenbank kopieren** Aktuelle Fahrzeug-Positionsdaten werden in die letzte Zeile der geöffneten POI Datenbank kopiert.
	**Zoom** Bestimmen Sie mit der linken Mouse Taste den neuen Viewport
	**Zoom to Content** Aktueller Viewport wird an den aktuellen Inhalt automatisch angepasst
	**Scroll** Verschieben Sie mit der linken Mouse Taste den aktuellen Viewport
	**Scroll to car** Der Viewport wird automatisch zu der aktuellen Fahrzeugposition verschoben.

4.3. Integration in das Gesamtsystem

Die Eingliederung der Komponente in den Fahrsimulator fängt beim Auswählen des richtigen Speicherortes an. Zu dem Zeitpunkt des Starts von SEP existierte bereits eine Komponente, die die Fahrzeugposition analysierte. Die Komponente `contextServer` befand sich im eigenen Java-Package: `de.bmw.tummicii.contextServer`. Um Integrations-Aufwand zu minimieren wurde entschieden, die alte Komponente einfach durch die neue zu ersetzen.
Das notwendige Refactoring-Aufwand ist dabei ebenfalls gering, denn das Format von eingehenden und ausgehenden Nachrichten nicht geändert wurde. Zu beachten ist jedoch, dass die neu hinzugekommenen Java-Quellen dem Compiler explizit bekannt gemacht werden müssen. Eine Gesamtliste der zu kompilierenden Quellen wird in einer speziellen Variable `sourcefiles` in der Konfigurationsdatei `Makefile.am` geführt. Diese Konfigurationsdatei befindet sich in einem der DWARF-Quellenverzeichnisse, unter [DWARF-HOME]/src/services/tumicii/ContextServer/ .

Die statische main-Methode, die den Einstiegspunkt in die Anwendung darstellt, befindet sich, wie bei der alten Komponente, in der Klasse `ContextServerService`. Diese Klasse realisiert die DWARF-Schnittstelle `de.tum.in.DWARF.TemplateService`. Die beiden Methoden `createAbilityObject` und `createNeedObject` signalisieren Bereitschaft der Komponente, Nachrichten nach außen zu versenden bzw. von außen zu empfangen. Die Einzelheiten der Schnittstellen- Konfiguration wurden aus dem Quelltext ausgelagert, sodass evtl. Änderungen der Nachrichten-Eigenschaften kein erneutes Kompilieren nach sich zieht. Die Eigenschaften des DWARF-Dienstes `ContextServerService` sind in der Konfigurationsdatei `ContextServer.xml` zusammengefasst. Diese Konfigurationsdatei befindet sich in einem der DWARF-Applikationenverzeichnisse, unter [DWARF-HOME]/applications/tumicii/ **.** Die zusätzlichen Attribute `configFilePath` und `configFilePath2` verweisen auf POI- bzw. Definitionen-Tabelle, die standardmäßig bei dem Start der Komponente verwendet werden sollen. Man kann den Dienst aber auch unter Verwendung von anderen Tabellen starten, ohne `ContextServer.xml` zu ändern: Dafür verwendet man optionale Aufrufparameter poifile=[Pfad] bzw. pdeffile=[Pfad].

Zum Schluss noch ein wichtiger Hinweis: Standardmäßig wird die Komponente als DWARF- Dienst im Hintergrund gestartet. Um zusätzlich noch die grafische Schnittstelle zu laden, setzt man den optionalen Aufrufparameter `gui` auf „true“. Der Befehl könnte zum Beispiel so aussehen:

```
java –jar ContextServer.jar gui=true
```

5. Zusammenfassung und Ausblick

Im Rahmen dieses SEPs wurde eine Komponente entwickelt, die die Steuerung des Fahrsimulators vereinfacht und die positionsbezogene Daten dank grafischer Darstellung veranschaulicht. Außerdem wurde der Algorithmus der Positionsanalyse optimiert.
In der Integrationsphase hat sich das modulare Konzept des DWARF-Netzwerkes noch mal bestätigt: Die auf Applikations- ja sogar auf physikalischer Ebene getrennten Komponenten arbeiten zusammen, lassen sich jedoch zu jedem Zeitpunkt austauschen und erweitern. Die Eingliederung der im Rahmen des SEPs entwickelten Anwendung in das Gesamtsystem verlief reibungslos und mit denkbar geringem Aufwand.

Mit den zur Verfügung stehenden Informationen wie Position von Auto sowie Positionen von POIs bleibt das Aufgabenfeld sehr begrenzt. Zusätzliche Daten, beispielsweise Koordinaten des Straßenpfades, würden ein bedeutendes Erweiterungspotenzial schaffen:

- Die Visualisierung der Fahrzeugposition könnte noch anschaulicher dargestellt werden
- Die Berechnung von Entfernungen könnte verbessert werden, denn bis jetzt wurde mangels Informationen zum Straßenpfad stets durch eine Fluglinie approximiert.
- Der Berechnungsaufwand könnte gesenkt werden, denn ein Straßenpfad kann auf ein ein-dimensionales Datenfeld abgebildet werden; bis jetzt wurde Positionsanalyse in einem drei-dimensionalen Raum durchgeführt.
- Statt Punkte (*points of interest*) könnte man im Bezug auf Straßenpfad ganze Bereiche AOS (*area of interest*) definieren, welche sich sogar überschneiden dürften.

Der Bereich Positionsanalyse erstreckt sich aber viel weiter, über solche Aufgabenfelder wie Routeplanung und Tank- und andere Assistenten, die zur Steigerung des Fahrtkomforts und der Sicherheit in modernen Fahrzeugen beitragen.

Literaturverzeichnis

[1] Object Management Group (OMG): Common Object Request Broker Architecture.
Part 2: CORBA Overview. Revision 3.0.3. Mar. 2004.
11 Nov. 2005 <http://www.omg.org/docs/formal/04-03-12.pdf >

[2] Project Distributed Wearable Augmented Reality Framework.
Ed. Marcus Toennis. Revision r1.6. Nov. 2005
Technische Universität München. Lehrstuhl für Informatikanwendungen in der Medizin & Augmented Reality.
11 Nov. 2005 <http://ar.in.tum.de/Chair/ProjectDwarf>

[3] Marc Loy, Robert Eckstein, Dave Wood, James Elliott & Brian Cole: Java Swing.
O'Reilly, 2002

[4] Andreas Sayegh: CORBA - Standard, Spezifikationen, Entwicklung.
O'Reilly, 1999