Helmut Schmidt Universität
Universität der Bundeswehr Hamburg

Henry Winkel

Entwurf einer Klassenhierarchie für militärische Simulationen in DIS Developing a class hierarchy for military simulations in DIS

Studienarbeit

Fakultät für Elektrotechnik

Studiengang: Elektrotechnik und Informationstechnik Matr.-Nr. 874650 / ET2014
Übernahme: 30. Mai 2018
Betreuer: Univ.-Prof. Dr. phil. nat. habil. Bernd Klauer
Weiterer Prüfer: Univ.-Prof. Dr.-Ing. habil. Udo Zölzer

Erklärung

Hiermit versichere ich, dass ich diese Arbeit selbständig verfasst, keine anderen als die im Quellen- und Literaturverzeichnis genannten Quellen und Hilfsmittel, insbesondere keine dort nicht genannten Internet-Quellen benutzt, alle aus Quellen und Literatur wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe und dass die auf einem elektronischen Speichermedium abgegebene Fassung der Arbeit der gedruckten entspricht.

Hamburg,

.....	.....
(Datum)	(Unterschrift)

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Abkürzungsverzeichnis

DIS	Distributed Interactive Simulation
IEEE	Institute of Electrical and Electronics Engineers
PDU	Protocol Data Units
ESPDU	Entity State PDU
OOP	objektorientierte Programmierung
UML	Unified Modeling Language
IDE	Integrated Development Environment
HLA	High-Level Architecture
VLS	Vertical Launching System

Inhaltsverzeichnis

1 Einleitung

In den letzten Jahrzehnten spielten Simulatoren in zivilen und militärischen Bereichen eine immer größere Rolle. Durch sie konnte die Ausbildung von Piloten aber auch von normalen Fahrzeugbesatzungen und Soldaten verbessert werden. Durch den Einsatz von Simulationen können Szenarien in sicherer Umgebung geübt werden. Neben der Ausbildung von Personal können in Simulationen auch Fahrzeuge oder Geräte getestet werden, ohne diese konstruieren zu müssen. Um die Ausbildung oder das Simulieren von Fahrzeugen noch effizienter zu gestalten, kam die Forderung auf, einzelne Simulationen miteinander zu vernetzten und interagieren zu lassen. Das Vernetzen von Simulation integriert zum Beispiel Flugsimulatoren und Panzersimulatoren in ein Szenario und lässt diese zusammen interagieren. Eine Möglichkeit um Simulationen miteinander zu vernetzen, ist Distributed Interactive Simulation (DIS).
In dieser Studienarbeit werden erste Erfahrungen mit DIS gemacht, und es wird eine Einschätzung getroffen, ob eine open soucre Library die nötige Funktionalität liefert, um eine Simulation zu erstellen. Dabei wird zu Beginn der Arbeit ein Überblick über den DIS Standard gegeben. Außerdem werden die wichtigsten PDU des Institute of Electrical and Electronics Engineers (IEEE) Standard erklärt. Anschließend wird eine Beispielsimulation beschrieben, die im Rahmen der Erprobung der Library erstellt wurde. Hierbei wird besonders auf die Erstellung einer Entity State PDU und auf die gelösten Probleme eingegangen. Abschließend wird eine Einschätzung getroffen, ob die verwendete Library die nötigen Funktionen enthält oder ob es nötig ist weitere, Libarys einzubinden oder eigene Funktionen zu erstellen.

2 Grundlagen

Im folgenden Kapitel werden die Grundlagen beschrieben und erklärt, die im Rahmen dieser Masterarbeit verwendet wurden. Zuerst wird die Programmiersprache C++ erklärt und anschließend wird besonders auf die objektorientierte Programmierung eingegangen, die hier als Grundlage dient. Dabei wird zunächst die Grundidee beschrieben. Anschließend wird das Konzept der Klassen und das der Polymorphie dargestellt. Abschließend werden die Grenzen der objektorientierten Programmierung beschrieben. Weiter wird die Entity State PDU (ESPDU) des DIS Protokoll beschrieben. Ein Überblick über die wichtigsten Inhalte des DIS-Protokoll wird im Grundlagenteil der Studienarbeit „Beurteilung der Open-DIS C++ Library zur Simulation von militärischen Szenarien“ der Professur für Technische Informatik an der Helmut Schmidt Univesität der Bundeswehr Hamburg gegeben.

2.1 C++

C++ ist eine objektorientierte Programmiersprache, die eine Weiterentwicklung der Sprache C ist. Jedoch ist C nicht komplett in C++ enthalten. Bei der Entwicklung von C++ ist man trotzdem dicht an der Syntax von C geblieben. C++ wurde in den 1980er Jahren von Bjarne Stroustrup entwickelt. Für die erste Version, die „C with Classes“ hieß, versuchte er durch das Nutzen von Teilen der Programmiersprachen Simula und BCPL die Sprache C zu erweitern. Das Konzept der objektorientierten Programmiersprache wurde aus der Sprache Simula entnommen, die als die erste Programmiersprachen das Klassenkonzept beinhaltet. BLCP, welche ein Vorgänger von C ist, lieferte den prozeduralen Anteil. 1983 wurde „C with Classes“ zu C++ umbenannt. Das erste Release von C++ 1.0 wurde 1985 veröffentlicht. 1998 wurde C++ als Standard „ISO/IEC 14882“ festgelegt. Im laufe der Zeit wurde C++ immer weiter verbessert und bekam neue Funktionen. Die aktuellste Version von C++ ist C++17, welches 2017 veröffentlicht wurde. C++ bringt nicht nur eine Reihe von Vereinfachungen und Ergänzungen mit sich, sondern auch diverse Verschärfungen und Änderungen. C++ bringt als großen Vorteil die

Möglichkeit objektorientiert zu Programmieren. Was genau objektorientierte Programmierung ist, wird in den nachfolgenden Kapitel erklärt. Ein weiterer großer Vorteil von C++ sind Templates. Durch Templates werden unter anderem Container bereitgestellt, die eine dynamische Speicherverwaltung ermöglichen. Container können Listen, Vektoren oder Warteschlagen dynamischer Größe sein, deren Inhalt Elemente einer beliebigen Datenstrukturen sein kann. Neben den Containern, die Klassen-Templates sind, gib es noch Funktions-Templates und Variablen-Templates, welche hier nicht weiter erklärt werden, da sie in dieser Masterarbeit keine Anwendung finden.
Eine weitere Neuerung in C++ ist die Möglichkeit, Speicher zu reservieren. In C++ kann man durch den Befehl „new“ Speicher eines beliebigen Typs reservieren. Dieser Befehl ersetzt das „malloc“ aus C. Das Gegenstück dazu ist das „delete“. Mit diesem Befehl kann man den reservierten Speicher wieder freigeben.
In C++ gibt es neben den Pointer auch die Möglichkeit, Referenzen zu benutzen. Der Unterschied zwischen einer Referenz und einem Pointer ist, dass eine Referenz prinzipiell nur eine neue Bezeichnung für eine bereits vorhandene Variable ist. Neben den genannten Änderungen gibt es noch viele weitere, die jedoch im Rahmen dieser Masterarbeit keine oder nur am Rande benutzt wurden.
[5] [2] [4] [1]

2.2 Objektorientiertes Programmieren

2.2.1 Grundidee

Die Grundidee hinter der objektorientierte Programmierung (OOP) ist das Abbilden von Objekten der realen Welt. Diese Objekte kommunizieren durch Nachrichten miteinander, welche sie unterschiedlich interpretiert werden können. Im Zuge von immer größer werdenden Softwareprojekte hat sich herausgestellt, dass es zum Beispiel mit der Programmiersprache C sehr schwer geworden ist, den Überblick über den Programmcode und den Programmfluss zu behalten. Objektorientierte Programmiersprachen helfen den Überblick zu behalten und sie bringen Techniken mit sich, Programmteile wieder zu verwenden. Das ermöglicht das schnellere und fehlerfreier Entwickeln von Programmen. Ein Nachteil von Programmiersprachen, die ausschließlich objektorientiert sprechen, ist, dass Funktionen nicht alleine existieren können. Dies führt dazu, dass kleine Programme unnötig kompliziert werden. Ein Sprache, die sowohl objektorientiert als auch prozedural verwendet werden kann, ist C++. Wie bereits erwähnt, wird diese Sprache im Rahmen

dieser Masterarbeit verwendet. Dabei werden unter anderem Techniken wie die Vererbung und die Polymorphie verwendet. Diese Techniken werden in den folgenden Kapiteln beschrieben. Als Modelierungssprache wurde Unified Modeling Language (UML) und als Integrated Development Environment (IDE) wurde Eclipse mit der Papyruserweiterung verwendet. Alle folgenden Beispiele werden anhand dieser Umgebung beschrieben. [1] [3]

2.2.2 Klassen und Objekte

Wie oben genannt, ist die Grundidee hinter der OOP das Abbilden der realen Welt, um Probleme zu lösen. Ein zentraler Gegenstand hierbei ist die Klasse. Eine Klasse enthält, ähnlich wie das „struct“ in C, Daten, die Attribute genannt werden. Der Unterschied zu einem klassischen „struct“ ist jedoch, dass eine Klasse Funktionen beziehungsweise Methoden enthält, die auf diese Daten zugreifen können. Des Weiteren können Klassen auch von anderen abgeleitet werden, was Vererbung genannt wird. Die Vererbung wird im nachfolgenden Unterabschnitt erläutert.
Um einen unerlaubten Zugriff auf Methoden und Attribute einer Klasse zu verhindern, kann innerhalb der Klasse festgelegt werden, wer auf Inhalte der Klasse zugreifen kann. Dies gliedert sich in drei Gruppen. Funktionen und Methoden können „public“, „private“ oder „protected“ sein. Soll der Inhalt für jeden zugänglich sein, dann ist er „public“. Ist ein Methode oder Attribut „private“, dann darf nur von innerhalb der Klasse auf ihn zugegriffen werden. Bei „protected“ Inhalten können alle von der Klasse abgeleiteten Klassen auf den Inhalt zugreifen.

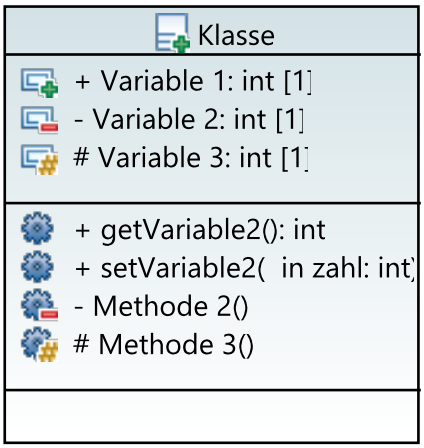

Bild 2.1: eine Klasse

In Bild 2.1 ist der exemplarische Entwurf einer Klasse zu sehen. Diese enthält drei Attribute, Variable 1 bis 3, außerdem beinhaltet sie vier Methoden. Ob ein Attribut oder Methode „public“, „private“ oder „protected“ ist, sieht man an dem Symbol, welches vor dem Namen steht. Das Kästchen mit dem Pluszeichen steht für ein „public“ Attribut, das Kästchen mit dem roten Balken steht für ein „private“ Attribut und das Kästchen mit dem gelben Raute Symbol steht für ein „protected“ Attribut. Analog gilt das auch für die Methoden, wobei das Pluszeichen bei den „private“ Methoden entfällt.
Um das unabsichtliche Überschreiben einer Variable zu verhindern, werden in der objektorientierte Programmierung Variablen zum Größten Teil als „private“ deklariert. Wie oben beschrieben, kann die Variable nur innerhalb der eigenen Klasse verändert werden. Um nun die Variable aktiv zu verändern und auszulesen, werden die sogenannten „get“ und „set“ Methoden verwendet. Dies sind Methoden, deren Aufgabe es ist, den Zugang zu den Attributen der Klasse zu ermöglichen. Des Weiteren kann mit Hilfe dieser, die korrekte Eingabe überprüft werden. Wie im Bild 2.1 zusehen, wurde nur für die Variable zwei eine Methode zum Beschreiben, „setVariable2()“ und zum Auslesen, „getVariable2()“, erstellt.

```
1  #include <iostream>
2
3  namespace Beispiel{
4  class Klasse:{
5  public:
6          int Variable1;
7          int getVariable2();
8          void setVariable2(int zahl);
9  protected:
10         int Variable;
11         void Methode3();
12 private:
13         int Variable2;
14         void Methode2();
15 };
16 }
```

Listing 2.1: Klasse.h

In Listing 2.1 ist die Header Datei der Beispielklasse aus dem Bild 2.1 dargestellt. Hier finden sich alle Methoden und Attribute wieder. Dabei wird eine Unterteilung nach „public“, „private“ und „protected“ durchgeführt. In der Header Datei werden nur die Attribute und die Methoden bekannt gegeben. Die Implementation der Methoden findet nicht hier, sondern in einer anderen Datei, der Source Datei, statt.

```
1  #include "Klasse1.h"
2
3  namespace Beispiel {
4  int Klasse1::getVariable2() {
5          return Variable2
6  }
7   void Klasse1::setVariable2(int zahl) {
8          Variable2 = zahl;
9   }
10 void Klasse1::Methode2() {
11 ...
12 }
13
14 void Klasse1::Methode3() {
15 ...
16 }
17
18 void Klasse1::Mehtode4() {
19 ...
20 }
21 }
```

Listing 2.2: Klasse.cpp

Im Listing 2.2 ist die zum Listing 2.1 zugehörige Source Datei dargestellt. Wie man sieht, sind hier nur die Implementationen der Mehtoden aufgeführt. Neben den Methoden werden hier auch die statischen Variablen initialisiert. Eine Sortierung, wie in der Header Datei, wird nicht vorgenommen, da die Zugriffsrechte dort schon festgelegt wurden.

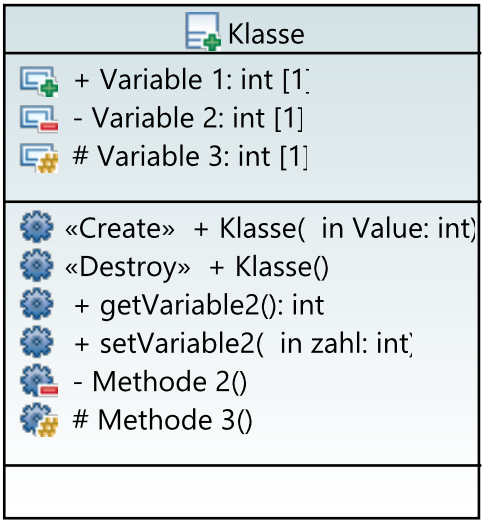

Bild 2.2: eine Klasse mit Konstruktor und Destruktor

Die Klasse stellt jedoch nur den Bauplan für ein Objekt dar. Um die implementierten Methoden zu nutzen, muss man ein Objekt der Klasse erstellt. Dazu kann ein „Konstruktor“ genutzt werden. Dieser hat den Vorteil, dass die Attribute mit Defaultwerten initialisiert werden können. Der „Konstruktor“ kann als normale Methode angesehen werden, die jedoch so wie die Klasse benannt sein muss. Er kann deshalb auch Eingabeparameter besitzen und Attribute auf bestimmte Werte setzen. In der IDE wird als Kennzeichnung für den Konstruktor „Create“ vor die Methode gesetzt. Möchte man nun das erstellte Objekt wieder löschen, muss man dafür ein „Destruktor“ verwenden. Dieser kann ebenfalls als Methode gesehen werden, die den Speicher, der das Objekt belegt, frei gibt. Des Weiteren können im „Destruktor“ weitere Anweisungen, wie zum Beispiel das anpassen eines Zähler, enthalten sein.
Im Bild 2.2 ist zu sehen, dass der Klasse aus Bild 2.1 ein „Konstruktor“ und „Destruktor“ hinzugefügt wurden. Diese werden oft als die ersten beiden Methoden angegeben. Da die Variablen eines Objektes immer nur so lange existieren, solange das Objekt selbst existiert, kann es zum Beispiel keine Variable geben, die Teil der Klasse ist, welche alle erzeugten Objekte zählt. Um dieses Problem zu lösen, kann man auf statische Variablen zurückgreifen, die auch ohne das erstellte Objekt existieren. Statische Variablen werden unterstrichen. Sie können ebenfalls „private“, „public“ oder „protected“ sein und werden auch dementsprechend gleich gekennzeichnet. Neben Attributen können auch Methoden statisch sein. Dies ist ein Vorteil, wenn man nur eine bestimmte Funktion der Klasse benutzen möchte, ohne erst das Objekt zu erstellen und anschließend wieder löschen zu

müssen. In Listing 2.3 ist die Header Datei, welche den Konstruktor und den Destruktor enthält, dargestellt. Die Implementierung erfolgt wie bei einer normalen Methode in der Source Datei.

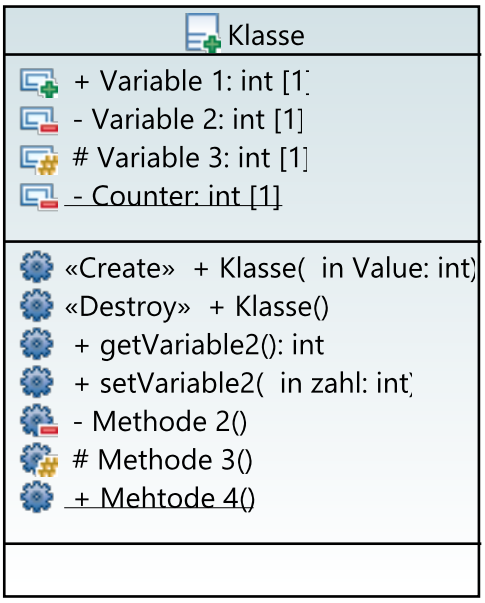

Bild 2.3: eine Klasse mit Statischer Variable

[1] [3] [2]

2.2.3 Vererbung

Ein Vorteil der OOP ist, dass Klassen von anderen Klassen Attribute und Methoden erben können. Das bedeutet, dass die erbende Klasse zugriff auf alle „public“ und „protected“ Methoden und Attribute hat. Die erbende Klasse wird Kindklasse genannt und die Klasse, von der geerbt wird, wird Elternklasse genannt. In Bild 2.4 sind zwei Klassen abgebildet, die Klasse 1 und die Klasse2. Die Klasse1 erbt von Klasse2 alle Attribute und Methoden. In diesem Beispiel erbt Klasse 1 das Attribut „Name“ und die Methoden „setName()“ und „getName()“ von Klasse2. Eine Elternklasse kann mehrere Kinder haben, was bedeutete, dass in Elternklassen Attribute und Methoden beinhaltet sein können, die jede ihrer Kindklassen benötigt. Das reduziert den Aufwand bei der Erstellung der Methoden, da sie nur einmal in den Elternklassen implementiert werden müssen.

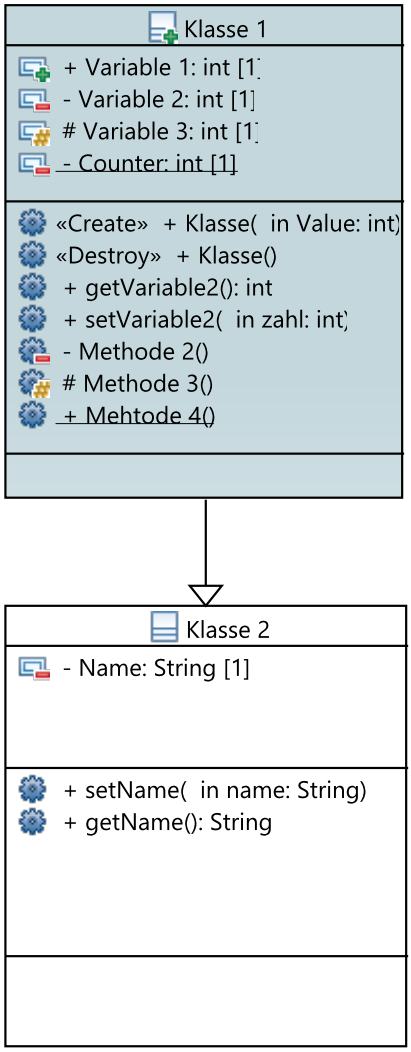

Bild 2.4: Vererbung

In Listing 2.3 ist die Header Datei der Klasse2 dargestellt. Das „public“ vor Klasse2 gibt an, wie von dieser Klasse geerbt wird. Die Methoden und Attribute der Elternklasse werden hier nicht angegeben, da diese in der Header Datei der Elternklasse deklariert und in der Source Datei implementiert werden.

```
1 #include <iostream>
2 #include "Klasse2.h"
3
4 namespace Beispiel{
5 class Klasse1: public Klasse2:{
6 public:
```

```
7           Klasse1(int Value);
8           ~Klasse1();
9           int Variable1;
10          int getVariable2();
11          void setVariable2(int zahl);
12  protected:
13          int Variable3;
14          void Methode3();
15  private:
16          int Variable2;
17          void Methode2();
18
19  };
20  }
```

Listing 2.3: Vererbung in c++

Die oben beschriebene Vererbung ist eine eine „public“ Vererbung. Neben dieser Art können Klassen auch „private“ oder „protected“ erben. Dies verändert die Zugriffsmöglichkeiten auf die Elemente, also die Methoden und Attribute der Elternklasse. Ist zum Beispiel eine Methode in der Elternklasse „public“ und wird diese durch eine „private“ Vererbung vererbt, so ist diese Methode in der Kindklasse „private“ und kann nur innerhalb dieser Klasse benutzt werden. In Tabelle 2.1 sind die Verschiedenen Methoden und deren Konsequenzen dargestellt.

Ableitung	**Privileg der Elternklasse**	**Privileg der Kindklasse**
private	public	public
	protected	protected
	private	kein Zugriff
private	public	private
	protected	private
	private	kein Zugriff
protected	public	protected
	protected	protected
	private	kein Zugriff

Tabelle 2.1: Vererbung von Klassen

In Listing 2.4 ist in Zeile 5 zusehen, wie man mithilfe eines Konstruktors, eine Objekt von Typ Klasse1 erstellt. In Zeile 6 wird eine Methode aus der eigenen Klasse benutzt. Dazu wird hinter den Namen des Objektes, hier ist der Name „Klasse1“, der Aufruf der Methode gesetzt. Der Aufruf einer Methode aus der Elternklasse erfolgt wie der Aufruf einer Methode der eigenen Klasse. Dies ist in Zeile 7 zusehen. Abschließend wird das Objekt durch Destruktor, in Zeile 8 zusehen, gelöscht.

```
1  #include <iostream>
2  #include "Klasse1.h"
3
4  int main(){
5  Beipiel::Klasse1 Klasse1(1);
6  Klasse1.setVariable2(2);
7  Klasse1.getName();
8  Klasse1.~Klasse1();
9
10 return 0;}
```

Listing 2.4: Benutzung der Klasse

[1] [3] [2]

2.2.4 Polymorphie

Neben der Vererbung ist die Polymorphie ein weiteres Grundkonzept der OOP. Polymorphie ermöglicht verwandte Objekte mithilfe gleichnamiger Funktionsaufrufe zu verarbeiten. Bevor jedoch die Polymorphie ausführlich erklärt werden kann, müssen zunächst einige Voraussetzungen erklärt werden. Zu Diesen gehört das Umwandeln von Klassen. Für das Umwandeln von Klassen beziehungsweise von Objekten, gibt es mehrere Möglichkeiten. Die Erste ist die direkte Zuweisung. Dabei muss man beachten, dass dies nur in eine Richtung funktioniert. Das heißt, dass nur Objekte einer Kindklasse in ein Objekt einer Elternklasse umgewandelt werden kann. Der Grund dafür ist, dass jedes Attribut der Ziel Klasse belegt werden muss und dabei gehen die eigenen Attribute und Methoden der Kindklasse verloren.
Eine weitere Methode ist das Umwandeln mithilfe eines Pointers. Dazu wird zunächst ein Pointer erstellt, der auf ein Objekt der Elternklasse zeigt. Diesen Zeiger kann die Speicheradresse eines Objektes der Kindklasse übergeben werden. Über diesen Zeiger kann nun auf Methoden zugegriffen werden, die in der Elternklasse enthalten sind. Auf

die Methoden der Kindklasse kann nicht zugegriffen werden. Die Umwandlung mittels Pointern ermöglicht, im Gegensatz zur Umwandlung durch Zuweisung, eine Konvertierung eines Objektes der Elternklasse in in Objekt der Kindklasse. Dabei muss jedoch zwingend ein „stativ_cast“ auf das Objekt angewendet werden, welches umgewandelt werden soll.
Das Umwandeln von Klassen ermöglicht es, Objekte unterschiedlicher Klassen gemeinsam zu verarbeiten. Um nach dem Umwandeln eines Objektes auch auf die dazugehörige Methode zugreifen zu können, muss die Elternklasse, in die das Objekt umgewandelt werden soll, virtuelle Funktionen genutzt werden.

```
1  #include <iostream>
2
3  namespace Beispiel{
4  class Klasse2 {
5  public:
6          void setName(std::string name);
7          std::string getName();
8          virtual void printName();
9  private:
10         std::string Name;
11 };
12 }
```

Listing 2.5: Klasse2.h

In Listing 2.5 ist die Header Datei der Elternklasse zu sehen. Diese verfügt über die virtuelle Methode „virtual void printName()“. Das „virtuel“ vor dem Funktionsaufruf sorgt dafür, dass diese Methode in allen Kindklassen implementiert werden muss. In der Kindklasse tauchte diese Mehtode als ganz normale Methode auf, dabei muss sie den gleichen Namen wie die virtuelle Methode aus der Elternklasse haben.
In Listing 2.6 wird gezeigt, wie man ein Objekt der Kindklasse mithilfe eines Pointers in ein Objekt der Eltern klasse umwandelt. Dabei wird in Zeile 5 zunächst ein Pointer erzeugt, der auf ein Objekt der Klasse2 zeigt. Diesem Pointer wird anschließend in Zeile 7 die Speicheradresse des Objektes der Klasse1 übergeben. Der Klasse1 wird nun mittels Funktionsaufruf der Name „hallo welt“ zugeordnet. Im Anschluss wird durch die Ausgabe überprüft, ob der Name dem Objekt zugeordnet wurde. Dies ist in Zeile 9 zu sehen. In Zeile 10 ist zu sehen, wie mit hilfe des Zuweisungsoperators „->“ die Fuknktion

„printName()“, der Elternfunktion, aufgerufen wird. Da es kein Objekt der Elternklasse gibt, könnte man jetzt meinen, dass dieser Aufruf einen Fehler erzeugen würde. Aufgrund aufgerufene virtuelle Methode, wird nun nicht die Mehtode der Elternklasse ausgeführt. Da das Objekt auf das der Pointer zeigt, ein Objekt der Kindklasse ist, wird nun entschieden, dass es die Implementation aus der Kindklasse verwendet. Diese Entscheidung erfolgt automatisch während der Laufzeit.
Wie erwartet, gibt das Programm zwei mal hintereinander „hallo welt“ aus. Das erste Mal kommt von dem „std::cout « Klasse1.getName() « std::endl;“ und das zweite Mal kommt entsprechend von dem „Klasse2->printName();“.

```
1  #include <iostream>
2
3  int main() {
4  tes::Klasse1 Klasse1(2);
5  tes::Klasse2 *Klasse2;
6
7  Klasse2 = &Klasse1;
8  Klasse1.setName("hallo␣welt");
9  std::cout << Klasse1.getName() << std::endl;
10 Klasse2->printName();
11 Klasse1.~Klasse1();
12
13 return 0;
14 }
```

Listing 2.6: Beispiel virtuelle Funktion

Virtuellen Methoden müssen auch eine Implementierung in der Elternklasse besitzen. Um dies zu umgehen, kann eine Methode „pure virtual“ sein. Dazu setzt man in der Header Datei die virtuelle Methode gleich 0. Dies hat zur Konsequenz, dass die gesamte Klasse abstrakt wird. Von abstrakten Klassen können keine Objekte erzeugt werden.
Neben den genannten Inhalten der Polymorphie gib es noch weitere, wie zum Beispiel die virtuellen Destruktoren oder die Möglichkeit sich den Objekttyp ausgeben zu lassen. Diese Inhalte wurden aber in dieser Masterarbeit nicht benutzt und werden daher nicht weiter betrachtet.
Die Polymorphie beschreibt also, wie man mit Objekten verwandter Klassen umgehen kann und wie diese auf einfache Weise verwaltet werden können ohne das der Anwender

zu jeder Zeit weiß, welche Methode er für welches Objekt benutzt muss. Diese Entscheidung wird ihm von Programm selbst abgenommen.
[1] [3] [2]

2.3 Distributed Interactive Simulation

Distributed Interactive Simulation (DIS) steht für einen Standard zur Steuerung und Überwachung von Simulationen, der 1993 im IEEE 1278 definiert wurde. Er wird für Simulationen im militärische und zivilen Umfeld genutzt und ermöglicht das Erstellen das Vernetzten von Simulationen und das Erstellen von simulierten Lagebildern. Dabei können die einzelnen Simulationen unterschiedliche Sprachen sprechen. Genauere Informationen über den Aufbau und die Inhalte von DIS können in der Studienarbeit „Beurteilung der Open-DIS c++ Library auf Funktionalität“ und im DIS Standard nachgelesen werden. Die verwendete Library ist wie in der aufgeführten Studienarbeit die „Open-DIS“ Library. Da im Rahmen dieser Masterarbeit ausschließlich mit der Entity State PDU (ESPDU) gearbeitet wurde, wird diese im kommenden Abschnitt vertieft erklärt.
Die ESPDU ist eines der Grundelemente von DIS. Mit ihrer Hilfe werden Entitäten dargestellt, welche für bestimmte Objekte wie Panzer, Schiffe oder Fahrzeuge stehen. Die ESPDU stellt eine Klasse dar, die folgende Attribute besitzt.

- PDU Header
- Entity ID
- Force ID
- Number of Variable Parameter Records(N)
- Entity Type
- Alternative Entity Type
- Entity Linear Velocity
- Entity Location
- Entity Orientation
- Entity Appearance
- Dead Reckoning Parameters

- Entity Marking
- Capabilities
- Variable Parameter record #1
- Variable Parameter record #N

Field size	Entity State PDU field	
96 bits	PDU Header	Protocol Version
		Exercise ID
		PDU Type
		Protocol Family
		Timestamp
		Length
		PDU Status
		Padding

Tabelle 2.2: PDU Header[6]

Der PDU Header beinhaltet die grundlegenden Informationen über die PDU und die Simulation. Dabei sollten die Werte für die „Protocol Version“ und der „Exercise ID“ bei allen Teilnehmern der Simulation gleich sein. Dies zeigt zum einen die verwendete DIS Version und die Simulationsnummer einer spezifischen Simulation.

Field size	Entity State PDU field	
48 bits	Entity ID	Site Number
		Application Number
		Entity Number
8 bits	Force ID	enumeration
8 bits	Number of Variable Parameter Records(N)	unsigned integer

Tabelle 2.3: Entity ID, Force ID und anzahl Parameter[6]

In Tabelle 2.3 ist angeben, wie sich der Inhalt der Entity ID zusammensetzt. Die Werte hängen dabei von einander ab. Die „Site Number“ gibt Auskunft darüber, an welchem geografischen Ort die Simulation stattfindet. Die „Application Number“ gibt an, welche Anwendung die ESPDU Simuliert hat. Die „Entity Number“ gibt an, ob es sich bei

der ESPDU um eine befreundetem, feindliche, neutrale oder um eine Einheit anderer Gesinnung handelt. Des Weiteren kann aus ihr entnommen werden, die wievielte Einheit es ist. Die „Entity Number“ hängt dabei von der „Site Number“ ab. Bei einer anderen „Site Number“ kann die sonst einmalige „Entity Number“ erneut vergeben werden. Die „ForceID“ gibt die eindeutige Nummer der Entity an und „Number of Variable Parameter Records(N)“ gibt Anzahl der Zusatzparameter an. Diese werden im weiteren Verlauf ausführlich erklärt.

Field size	Entity Sate PDU fields	
64 bits	Entity Type	Entity Kind
		Domain
		Country
		Category
		Subcategory
		Specific
		Extra

Tabelle 2.4: Entity Type[6]

Der Entity Type gibt Auskunft darüber, um was für eine spezielle Art es sich bei der ESPDU handelt und welcher Nation diese angehört. Der Inhalt des Entity Type wird durch die DIS Enumeration begrenzt. In der Enumeration werden militärische Fahrzeuge, wie Flugzeuge, Schiffe, Boote und Landfahrzeuge, verschiedener Nationen kodiert. Ein alternativer Entity Type kann im Feld „Alternative Entity Type“ angeben werden. Die „Entity Linear Velocity“, „Entity Location“ und „Entity Orientation“ sind dreidimensionale Vektoren. Der „Entity Linear Velocity“ Vektor gibt die vektorielle Geschwindigkeit, der „Entity Location“ Vektor die Position und der „Entity Orientation“ Vektor die Orientierung an.
Das „Entity Appearance“ Feld ist ein 32 Bit breiter Vektor, welcher Informationen über die Gesamterscheinung der ESPDU angibt. Beispielsweise kann in diesem Vektor kodiert sein, welche Lackierung die Entity hat, ob die Entity beschädigt ist oder ob sie sich selbstständig bewegen kann. Genauere Informationen sind in der DIS Enumeration beschrieben.
Die „Dead Reckoning Parameters“ geben an, welche Methode verwendet werden muss um die Position der Entity vorauszuberechnen. Dieses Feld dient zur Reduzierung der Netzwerkauslastung.

Das „Entity Marking“ Feld enthält Informationen über eindeutige Markierungen an der Entity. Dies können zum Beispiel Zahlen, Symbole oder Ländersymbole sein.
Das „Capabilities“ ist ein 32 Bit breiter Vektor, welcher angibt, welche Fähigkeiten eine Entity besitzt. Zu Diesen gehört Beispielweise die Fähigkeit, andere Entity's mit Munition oder Treibstoff zu versorgen. Ausführliche Informationen sind in der DIS Enumeration enthalten.

Field size	Entity State PDU fields	
128 bits	Variable Parameter record	Record Type
		Record-Specific fields

Tabelle 2.5: Variable Paramete record[6]

Im Variable Parameter record werden die Zusatzteile beschrieben, die an dem ESPDU verbaut sind. Dazu können Waffenanlagen und Sensoranlagen gehören. In Tabelle 2.6 ist ein Turm mit Kanone als Variable Parameter record in DIS dargestellt. Da sich der Turm drehen können und die Kanone sich anheben können soll, werden vier Parameter benötigt, die die Bewegung beschreiben. Dabei gibt es einen Parameter Record, der die aktuelle Lage beschreibt. In dem unten gezeigtem Beispiel gibt es einen Record für die horizontale Richtung und einen für die vertikale Richtung , also die Höhe, in der die Kanone zeigt. Die anderen beiden Records sind für die Änderungsraten, in der sich der dazugehörige Wert ändert. Diese Zusammensetzung der Parameter Records ist jedoch nicht fest vorgeschrieben und kann variiert werden. Diese Variation hat jedoch zur Folge, dass bestimmte Bewegungen nicht simuliert werden können. Zum Beispiel kann das Feld „Primary Gun Elevation Rate“ vernachlässigt werden, wenn sich zum einen die Höhe der Kanone sehr schnell ändert oder weil sich die Kanone überhaupt nicht bewegt und somit auch keine Änderungsrate existiert.

Record	Field name	Value	Discription
Turret Azimuth	Record Type	0	Articulated Part VP record
	Change Indicator	213	Increment by one for each change
	ID - Part Attached to	0	Tank chassis
	Parameter Type	4107	4096(primary turret) + 11 (azimuth)
	Parameter Value	-0.305	Angle in radians
Turret Auimuth Rate	Record Type	0	Articulated Part VP record
	Change Indicator	45	Increment by one for each change
	ID - Part Attached to	0	Tank chassis
	Parameter Type	4108	4096(primary turret) + 12 (azimuth rate)
	Parameter Value	-0.058	Rate in radians/s
Primary Gun Elevation	Record Type	0	Articulated Part VP record
	Change Indicator	187	Increment by one for each change
	ID - Part Attached to	1	Turret
	Parameter Type	4439	4416 (primary gun) + 13 (elevation)
	Parameter Value	0.267	Angle in radians
Primary Gun Elevation Rate	Record Type	0	Articulated Part VP record
	Change Indicator	34	Increment by one for each change
	ID - Part Attached to	1	Turret
	Parameter Type	4430	4416 (primary gun) + 14 (elevation rate)
	Parameter Value	0.384	Rate in radians/s

Tabelle 2.6: Variable Paramete record Beispiel[6]

3 Entwurf der Klassenhierarchie

3.1 Grundkonzept

Die Aufgabenstellung die im Rahmen dieser Masterarbeit gelöst werden soll, war das Erstellen einer Klassenhierarchie zur Erstellung von militärischen und zivilen Simulationsobjekten. Simulationsobjekte können hierbei Schiffe, Fahrzeuge, Flugzeuge oder Gebäude darstellen. Die Klassenhierarchie soll die Möglichkeit liefern Objekte zu erstellen und zu simulieren und diese anschließend in das DIS Standard oder ein Anderen, wie der High-Level Architecture (HLA) Standard, zu überführt.

3.2 Lösungsansätze

Während der Erstellung wurden drei verschiedene Lösungsansätze in Betracht gezogen. Diese Ansätze Unterschieden sich in der Abbildung der Wirklichkeit, in Bezug auf die Detailgenauigkeit, Einfachheit und der Möglichkeit, diese Modelle mit beliebigen Funktionen zu erweitern. Die Folgenden Lösungsansätze werden schematisch erklärt und es werden die dazugehörigen UML Diagramme gezeigt und erklärt. Es wird in den Diagrammen darauf verzichtet alle Methoden und Attribute anzugeben, da zwei der drei Ansätze nicht weiter verfolgt wurden und die Anzahl der Methoden und Attribute des dritten Ansatzes so groß ist, dass eine komplette Darstellung unübersichtlich ist.
Der erste Ansatz, der in Betracht gezogen wurde war der, sich nahe am DIS Standard zu bewegen und ausschließlich Klassen zu benutzen, die darin implementiert wurden. Es sollte eine Klasse erstellt werden, die die Attribute besitzt, die für Erstellung einer DIS PDU notwendig sind. Dabei sollten möglichst wenig eigene Methoden erstellt werden. Eine der wichtigsten eigenen Methoden wäre die Erstellung einer ESPDU aus den gespeicherten Daten. Parallel sollte es die Möglichkeit geben eine Konvertierung der Daten zu HLA zu implementieren. Die Implementierung dieses Ansatzes wäre schnell zu bewältigen, da nur wenige Klassen zu erstellen wären. Wie im Bild 3.1 zu sehen, hätten nur drei Klassen erstellt werden müssen. Eine der Klassen wäre nur für die Überführung der

ESPDU in HLA notwendig. Die anderen beiden wären nur für die Erstellung eines Objektes und der ESPDU zuständig. Die Klasse „Object“ speichert alle Daten und enthält alle Funktionen die notwendig sind. Sie erbt von der den Klassen „DISClass“ und „ConvertToHLA“ die Methoden für die Erstellung einer ESPDU und für deren Überführung in HLA.

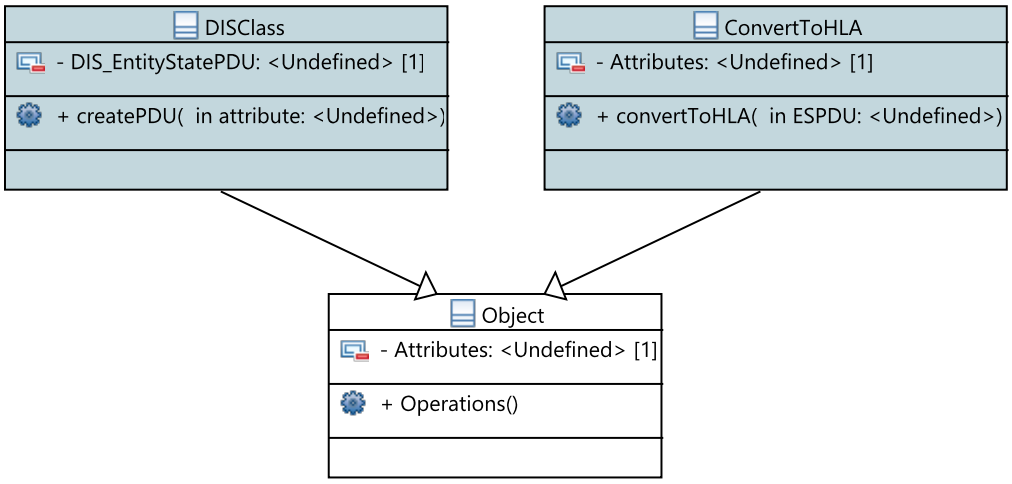

Bild 3.1: Ansatz 1

Der zweite Ansatz bestand darin, eine unabhängige Klassenhierarchie zu erstellen. Diese Hierarchie soll ihre eigenen Attribute und Methoden besitzen. Wie auch im ersten Ansatz, besitzt der zweite Ansatz eine Klasse „Object“. Zusätzlich existieren hier weitere Klassen, deren Elternklasse die Klasse „Object“ ist. Diese besitzt jetzt als Attribut eine ESPDU und zusätzlich noch Attribute die allgemein für ein Objekt benötigt werden. Die spezifischen Attribute sind in den Kindklassen enthalten.
Wie in Bild 3.2 zusehen, wird zunächst unterschieden, ob ein Objekt beweglich oder starr ein einem Ort ist. Ist ein Objekt mobil, wird unterschieden, ob es sich um ein ziviles oder militärisches Objekt handelt. Die Klassen „civil“ und „military“ stellen die unterste ebene dar und erben, wie im Bild 3.2 gezeigt, die Inhalte der Elternklassen. Die Entscheidung, ob ein Objekt beweglich oder starr ein einem Ort ist muss vorgenommen werden, da mobile Objekte sich fortbewegen können und demnach mindestens ein Attribut und die dazugehörige Interface- und Verarbeitungsmethoden mehr hat als ein festes Objekt. Eine dieser Verarbeitungsmethoden wäre die Berechnung der Positionsänderung in Abhängigkeit der Richtung, Geschwindigkeit und der Zeit. Die Entscheidung, ob ein Objekt zivil oder militärisch ist, wird benötigt, da sie sich grundlegend Unterscheiden. Ein mili-

tärisches Objekt, kann zum Beispiel aktiv an Kampfhandlungen teilnehmen wohingegen ein ziviles Objekt maximal passiv daran teilnehmen kann. Durch die Möglichkeit, aktiv an Kampfhandlungen teilzunehmen, kann das militärische Objekt Waffen besitzen, welche ihm die Teilnahme ermöglichen. Dies hat zur Folge, dass die dafür benötigten Attribute und Interface-Methoden in der Klasse implementiert werden. Des Weiteren werden Methoden benötigt, die das Schießen berechnen und simulieren.
Bei diesem Ansatz wird durch Methoden eine DIS Entity State PDU erstellt, welche als Attribut in der Klasse „Object“ gespeichert werden. Diese kann dann an dann an andere Teilnehmer der Simulation gesendet werden. Parallel kann dazu auch eine Methode implementiert werden, die die gespeicherten Daten in HLA oder in eine andere Simulationsumgebung konvertiert. DIS und HLA dienen hier nur als Schnittstelle zu anderen Simulationen.

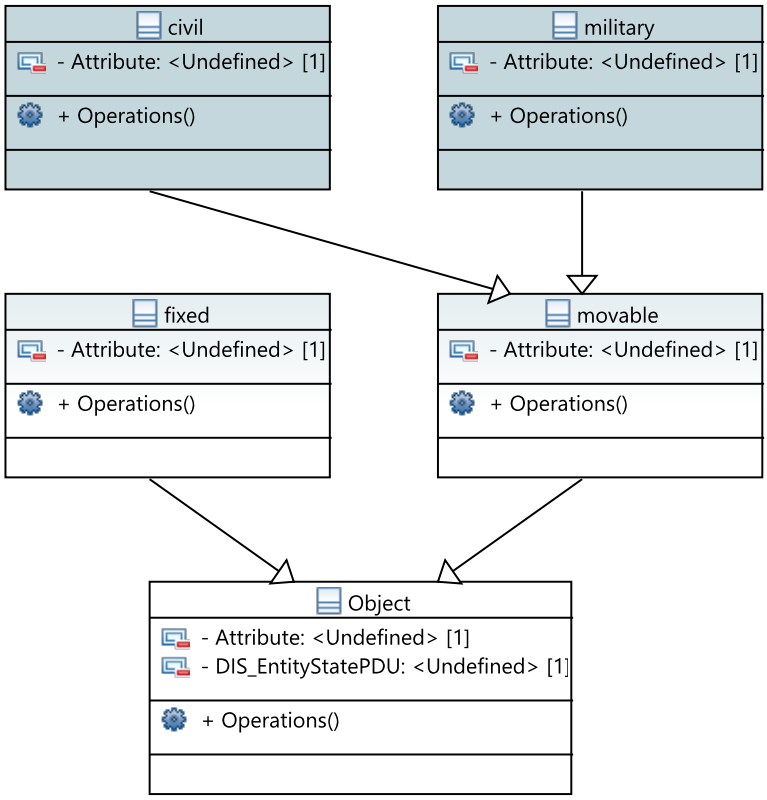

Bild 3.2: Ansatz 2

Der dritte Ansatz greift das Konzept des Zweiten auf, verfolgt jedoch eine andere Strategie. Im Bild 5.1, welches aufgrund der Größe im Anhang zu finden ist, wir das zum Ansatz gehörige UML Diagramm gezeigt. Auch hier ist die Elternklasse von der alle die

Klasse „Object“. In diesem Ansatz, wird eine grundlegend andere Unterscheidung vor genommen. Hier wird zunächst zwischen Objekten unterschieden, die sich selbständig und nicht selbständig fortbewegen können. Objekte, die sich von selbst bewegen können, besitzen eine Geschwindigkeit und eine Richtung in die das Objekt zeigt. Objekte, die sich nicht selbst bewegen können, können jedoch auch eine Geschwindigkeit besitzen beziehungsweise sie können sich auch fortbewegen. Der unterschied dabei ist jedoch das es bei diesen Bewegungen um entweder einmalige Positionsänderung handelt oder um derart kleine und daher vernachlässigbare Bewegungen. Ein Beispiel für ein Objekt, das ein Objekt der Klasse „nonDriven“, sind Bojen. Bojen bewegen sich im Idealfall nicht oder nur sehr wenig. Die Positionsänderung wird daher nur über das setzen einer neuen Position und nicht über einen Berechnung aus Geschwindigkeit, Zeit und Richtung vorgenommen. Von der Klasse „nonDriven“ sind die Klassen „movable“ und „fixed“ abgeleitet. Der Unterschied zwischen den Klassen beiden besteht darin, dass sich bei der Klasse „movable“ die Position ändern kann und es deshalb Methoden existieren müssen, die die Position anpassen können. Bei der Klasse „fixed“ wird vorausgesetzt, dass sich die Position nie ändern.
Die Klasse „driven“ ist eine Kindklasse von der Klasse „Object“ und stellt das Gegenstück zur Klasse „nonDriven“ dar. Sie ist die Elternklasse aller Klassen die Objekte darstellen, die sich selbstständig fortbewegen können. Anschließend wird zwischen Luftfahrzeugen, Landfahrzeugen, Seefahrzeugen und Unterseefahrzeugen unterschieden. Für jede Domäne existiert eine eigene Klasse. Diese Unterscheidung ist sinnvoll, denn dadurch kann für jede Domäne spezifische Methoden erstellt werden die zum Beispiel die Positionsangaben prüfen und dann gegebenenfalls Konsequenzen daraus ziehen kann. Des Weiteren kann dadurch das Objekt der Klasse genauer simuliert werden, es können beispielsweise Methoden implementiert werden, die die Flugbahn eins Flugzeugs oder die einer Rakete berechnen. Diese Methode darf jedoch nicht auf ein Objekt der Klasse „Surface“, „Subsurface“ und „Land“ angewendet werden. Die Unterscheidung zwischen militärischen und zivilen Objekten wird in jeder der Kinklassen der Klasse „driven“ getroffen. Dies hat zur Folge, dass jeweils zwei weitere Klassen hinzukommen, die jeweils das zivile und das militärische Objekt darstellen. Wie im Ansatz 2 beschrieben, ist das sinnvoll, da militärische Objekte andere Methoden und Attribute besitzen wie ein ziviles Objekt. Bei diesem Ansatz wird wie im Ansatz 2 die ESPDU als Attribut in der Klasse „Object“ gespeichert und dann gegebenenfalls an andere Teilnehmer verteilt.

Ein weiterer Wichtiger Teil dieser Ansätze ist das hinzufügen von Ausrüstung, da diese

ein Objekt erst einsatzfähig macht. Dafür wurde eine Klassenhierarchie entwickelt, die der Grundidee der Ansätze 2 und 3 entspricht. Diese Klassenhierarchie ist im Anhang im Bild 5.2 zu finden. Hier wird eine die Klasse „Equipment“ erzeugt, die die Ausrüstung widerspiegelt, die ein Objekt besitzt. Sie kann als Attribut in einer der Beliebigen Klasse eingefügt werden. Diese Klasse besitzt als Attribute zwei Container, in diesem Beispiel zwei Arrays, in denen jeweils alle vorhandenen Waffen und Sensoren gespeichert werden und einen String, indem der Name gespeichert wird. Dazu besitzt sie Methoden mithilfe denen auf die Waffen und Sensoren zugegriffen werden kann. Dabei werden Methoden der Polymorphie benutzt, welche im Abschnitt 2.2.4 beschrieben wurden. Des Weiteren wurde eine Klasse „Weapon“ erstellt, welche die Elternklasse für die Spezifischen Waffenklassen darstellt. Außerdem wurde eine Klasse „Munition“ erstellt, welche als Elternklasse für die Munitionsklassen darstellt. Die derzeitig verfügbaren Waffenklassen sind die Klassen „Cannon“ und „Vls“. Die Klasse „Cannon“ stellt eine Rohrwaffe dar, die als Attribut einen Array der Klasse „Bullet“ besitzt. Die Klasse „Vls“ stellt ein Vertical Launching System (VLS) , eine Senkrechtstartanlage für Flugkörper, dar. Dieses System besitzt als Munition Raketen, welche in mithilfe der Klasse „Rocket“ dargestellt werden. Die Raketen, werden wie in der Klasse „Cannon“, in einem Container gespeichert und verwaltet. Da sich die Munition der Waffenklassen grundsätzlich ähneln, sind die Klassen „Bullet“ und „Rocket“ von der Klasse „Munition“ abgeleitet. Neben den Waffenklassen existieren auch Sensorenklassen. Dazu gehören die Klassen „Camera“„Sonar“ und „Radar“. Diese Klassen werden wie Waffenklassen behandelt.

3.3 Vergleich der verschiedenen Lösungsansetzen

Im folgenden Abschnitt werden die im vorhergehenden Abschnitt erklärten Ansätze miteinander vergleichen. Es wird eine Bewertung der einzelnen Ansätze vorgenommen und abschließend wird der verwendete Ansatz genannt und es werden die Gründe für die Wahl dargestellt.
Der erste Ansatz, ist wie oben beschrieben, dem DIS sehr nahe. Vorteile dieses Ansatzes wäre, dass die Umsetzung sehr schnell wäre, und dass man sich nahe am DIS Standard bewegt, was die Teilnahme an DIS basierten Simulationen einfach gestalten würde. Nachteil dieses Ansatzsatzes ist, dass man von dem jeweiligen Standard abhängig ist. Des Weiteren kann man bei diesem Ansatz nur anhand des Inhalts der ESPDU unterscheiden um was für ein Objekt es sich handelt. Dies hat zur Folge, dass nicht gewährleistet werden kann, dass spezifische Objekte wie Schiffe, Flugzeuge oder Landfahrzeuge nur auf

die für sie geltenden Methoden zugreifen können und somit die Methoden selbst prüfen müssen um was für ein Objekt es sich handelt und dann entsprechende Berechnungen durchführt. Die Methoden, die für die speziellen Einsatzräume gelten, sind zum Beispiel Flugbahnberechnungen, Berechnungen die die Position überprüfen oder Methoden zur Routen Berechnung.
Der zweite Ansatz hat den Vorteil, dass er von jedem Simulationsstandard losgelöst ist. Er enthält zwar als Attribut eine ESPDU jedoch wird kann diese nur bei bedarf erstellt werden. Die Attribute, die in den Klassen enthalten sind, sind vom DIS Standard losgelöst. Bei der Erstellung einer ESPDU werden diese Attribute übersetzt und in der entsprechenden Form in der ESPDU gespeichert. Diese PDU wird ausschließlich zur Kommunikation mit anderen Teilnehmern verwendet. Ein Nachteil dieses Ansatzes ist das nicht unterschieden wird, um was es sich bei diesem Objekt handelt. Es wird zwar unterschieden, ob es sich um ein ziviles oder militärischen Objekt handelt jedoch werden beispielsweise militärische Schiffe, Flugzeuge und Landfahrzeuge gleich behandelt. Sie können, wie im Ansatz 1, auf die gleichen Methoden zugreifen, welche auch hier anhand Attributen entscheiden müssen, um was für ein Objekt es sich handelt.
Im dritten Ansatz wird konsequent zwischen zivilen und militärischen, Luft-, Land- und Seeeinheiten unterschieden. Hier können zwar die militärischen und zivilen Einheiten auf die gleiche Elternklasse zugreifen, die Methoden für den jeweiligen Einsatzraum enthält jedoch können sie nicht auf Methoden zugreifen die zu einem Einsatzraum gehören. Da Objekte nur noch auf die für sie gedachten Methoden zugreifen können, müssen diese Methoden nicht mehr prüfen, ob sie auf dieses Objekt angewendet werden dürfen. Die Implementierung dieser Methoden ist somit weniger komplex. Nachteile dieses Ansatzes sind, dass es für jedes Objekt des Einsatzraums jeweils eine Klasse für zivile und militärische Objekte gibt, was die Anzahl der Attribute und Methoden vergrößert. Des weiteren muss bei diesem Ansatz mehr darauf geachtet werden, dass Attribute und Methoden nicht doppelt implementiert sind, da dies den Quellcode unnötig unübersichtlich und komplex machen würde.
Vergleicht man die Ansetze mit einander, fällt auf, dass die Anzahl der Klassen mit jedem Ansatz gestiegen ist. Des weiteren muss man bei der Erstellung eines Objektes genau wissen, um was für eins es sich handeln soll. Bei ersten Ansatz kann man eine ESPDU mit den entsprechenden Werten erstellen ohne zu entscheiden, um was es sich bei dem Objekt handeln soll. Bei zweiten Ansatz, muss man zwischen festen und beweglichen Objekten und wenn es sich um ein bewegliches Objekt handeln soll, ob es ein militärischen oder ziviles Objekt handelt, entscheiden. Beim dritten Ansatz kann man

sich nur für ein Objekt entscheiden, welches auch als Klasse abgebildet ist. Benötigt man ein Objekt einer nicht vorhandenen Klasse, muss man den Ansatz um die benötigte Klasse erweitern. Diese Einschränkung bei der Auswahl ermöglicht es jedoch immer die korrekten Methoden auf die Objekte anzuwenden. Der erhöhte Implementierungsaufwand ermöglicht jedoch eine sichere und korrekte Benutzung. Abschließend stellt sich der dritte Ansatz als der beste heraus, da bei der Implementierung der Methoden nicht auf einen unzulässigen Gebrauch geprüft werden muss, da die Methode immer nur von dem Objekt benutzt werden kann, für den sie bestimmt ist. Für die Implementierung wurde daher dieser Ansatz genutzt, da zum Einen die Implementierung der Methoden die einfachere ist und weil sich dieser Ansatz, auf Grund seiner Struktur, potentiell am besten erweitern lässt.

3.4 Implementierung

Die Klassenhierarchie wurde mithilfe eines UML Diagramms entworfen, welches mit der Papyrus Erweiterung der Eclipse IDE erstellt wurde. Anhand dieses Diagramms wurde eine C++ Projekt generiert, das anschließend vervollständigt wurde. Dabei wurden weitere Methoden und Attribute hinzugefügt, die im UML Diagramm nicht aufgeführt sind. Bei der Implementierung traten einige Probleme auf. Die Größten werden im folgenden Abschnitt aufgezeigt und es wird die gefundene Lösung erklärt.
Die erste Aufgabe war es alle „get“ und „set“ Methoden zu implementieren. Im Listing 3.1 ist eine einfache Implementierung dieser Methoden zu sehen. Hier wird das Attribut „Type“, welches „private“ ist, gesetzt und abgerufen. Die unten gezeigte Implementierung ist Beispielhaft für die anderen Methoden.

```
1 void driven::setType(std::string /*in*/type) {
2 Type = type;
3 }
4 std::string driven::getType() {
5 return Type;
6 }
```

Listing 3.1: „get“ und „set“ Methode

Das nächste Problem war es die Konstruktoren zu erstellen. Hierbei galt es zunächst diese so einfach wie möglich zu halten. Jedoch wurde schnell sichtbar, dass die eingaben überprüft werden müssen. Der Grund dafür entstand aus der Forderung, dass ein Objekt

in das DIS Standard übersetzt werden soll. Da DIS über eine Enumeration verfügt, in der die Verschiedenen Arten einer Entity kodiert sind, muss das Programm erkennen, wie er das vorhandene Objekt übersetzen kann, sodass es zu einer der Objekte passt, die in der Enumeration kodiert sind. Um dies zu erfüllen wurde entschieden, dass es am besten wäre, diese Prüfung direkt im Konstruktor vorzunehmen und diese als String zu speichern, um die Leserlichkeit beizubehalten.

```
1  warShip::warShip(std::string /*in*/Name,
2  std::string /*in*/Type, std::string/*in*/ Country) {
3  object::setName(Name);
4  object::setCountry(Country);
5  object::setKind("Platform");
6  object::SetDomain("Surface");
7  object::setPosition(0,0,0);
8  object::incrementCounter();
9  driven::setType(Type);
10 equipment = NULL;
11 if (Type== "F124") {
12   driven::setCategory("Guided Missile Frigate (FFG)");
13   driven::setSubCategory("Sachsen Class (Type 124)");
14 } else if (Type == "F123") {
15   driven::setSubCategory("Brandenburg class (Type 123)");
16   driven::setCategory("Guided Missile Frigate (FFG)");
17 }else if (Type == "F122"){
18   driven::setSubCategory("Bremen class (Type 122)");
19   driven::setCategory("Guided Missile Frigate (FFG)");
20 } else{
21   std::string SubCategory , Category;
22   std::cout <<"Category as written in DIS Enum:";
23   getline(std::cin, Category);
24   std::cout <<"SubCategory as written in DIS Enum:"
25   << std::endl;
26   getline(std::cin, SubCategory);
27   driven::setSubCategory(SubCategory);
28   driven::setCategory(Category);
29 }}
```

Listing 3.2: Konstruktor „warShip“ Klasse

Im Listing 3.2 ist der Konstruktor der Klasse „warShip“ dargestellt. Hier ist zu sehen, dass für das Erstellen der Klasse nur die Attribute „Name“, „Type“ und „Country“ dem Konstruktor übergeben werden. Neben den genannten Attributen werden noch weitere benötigt, die das Objekt noch genauer spezifizieren. Die weiteren Attribute sind „Category“, „SubCategory“, „Kind“ und „Domain“. Diese weiteren Attribute sind jedoch nur für die korrekte Erstellung der ESPDU notwendig. Die Attribute „Category“ und „SubCategory“ sind DIS spezifisch und sollten möglichst dem Inhalt der Enumeration entsprechen, da sonst die Übersetzung nicht möglich ist. Die hier gezeigt Implementierung ist jedoch nur zu Testzwecken. Eine Elegantere Lösung wäre die Erstellung einer Datenbank beziehungsweise eines Containers, in dem die Möglichen Werte für die Attribute gespeichert sind und das Anhand des „Type“ Wertes entschieden wird, welche Werte die Attribute „Category“ und „SubCategory“ erhalten. Die Konstruktoren der Klassen „warPlane“„civilPlane“, „military“, „civil“ und „civilShip“ können auf die gleiche Weise implementiert werden, wobei das für die Klassen „military“ und „warPlane“ schon der Fall ist.

Das wohl größte Problem war die Übersetzung eines Objektes in eine ESPDU. Dafür wurde eine eigene Klasse erstellt, die nur dafür zuständig ist den „Entity Type“ zu erstellen. Diese Klasse enthält eine art Datenbank, die aus der DIS Enumeration entnommen wurde.

```
1  class DIS_enum {
2  public:
3  DIS_enum();
4  // ~DIS_enum();
5
6  DIS_EntityType_Variables getDISEntityType(std::string kind,
7     std::string domain, std::string country,
8     std::string category,  std::string subcategory,
9     std::string specific,  std::string extra);
10
11 private:
12 std::map<int,std::string> Kind;
13 std::map<int,std::string> DomainPlatform;
14 std::map<int,std::string> DomainMunition;
```

```
15 std::map<int,std::string> Country;
16 std::map<int,std::string> CategoryLand;
17 std::map<int,std::string> CategoryAir;
18 std::map<int,std::string> CategorySurface;
19
20 ...
21 ...
22 ...
23 };
```

Listing 3.3: Ausschnitt DIS_enum.h

Das Listing 3.3 ist ein Ausschnitt der Header Datei der „DIS_enum “ Klasse. In Zeile 6 ist die Funktion zusehen, die mithilfe der eingegebenen Strings einen Struct befüllt, mit dessen Hilfe das „Entity Type “ Feld befüllt wird. Bevor jedoch diese Funktion genutzt werden kann, muss zunächst der Konstruktor aufgerufen werden. Dieser dient zur Erstellung eines Objektes und zur Befüllung der Container, in diesem Fall der vorhandenen Maps. In Listing 3.4 ist ein Teil des Konstruktor abgebildet. Dieser Teil zeigt wie die Maps befüllt werden.

```
1  DIS_enum::DIS_enum(){
2  Kind[0]="Other";
3  Kind[1]="Platform";
4  Kind[2]="Munition";
5  Kind[3]="Life form";
6
7
8  DomainPlatform[0]="Other";
9  DomainPlatform[1]="Land";
10 DomainPlatform[2]="Air";
11 DomainPlatform[3]="Surface";
12 DomainPlatform[4]="Subsurface";
13 ...
14 ...
15 }
```

Listing 3.4: Ausschnitt DIS_enum.cpp

Die Maps, die derzeit erstellt wurden, können in der Header Datei gefunden werden. Diese Maps sind nur teilweise vollständig befüllt. Falls ein benötigter Eintrag nicht vorhanden ist, können die benötigten werte in der DIS Enumeration gefunden werden.
Die in Listing 3.3 zu sehende Funktion „DIS_EntityType_Variables getDISEntityType()“ hat als Rückgabe wert einen Struct, der Folgenden Integer Werte beinhaltet:

- Kind
- Domain
- Country
- Category
- SubCategory
- Specific
- Extra

Der Inhalt entspricht dem des „Entity Type “ Feldes. Als Input Parameter werden der Funktion die in Listing 3.3 zusehenden Strings übergeben. Da die Werte des „Entity Type “ von einander abhängen, ist die Struktur der Funktion die eines Baumes. In Listing 3.5 ist die erste Entscheidung zusehen. Es wird zunächst geprüft, um was es sich bei dem Objekt grundlegend handelt. Diese Information ist im Attribut „Kind “ gespeichert. In der derzeitigen Version der Funktion ist nur der „case 1“ implementiert.

```
1  for (std::map<int,std::string>::iterator it=Kind.begin();
2       it!=Kind.end(); ++it)
3  if (it->second == kind) {
4  help.Kind =  it->first;
5  }
6  switch (help.Kind) {
7  case 1:
8    for (std::map<int,std::string>::iterator
9         it=DomainPlatform.begin(); it!=DomainPlatform.end();
10        ++it)
11   if (it->second == domain) {
12   help.Domain =  it->first;
13   }
14   switch (help.Domain) { ... }
```

```
15 break;
16 case 2:
17 break;
18 ...
19 ...
20 ...
21 default : std::cout << "invalid␣Kind" << '\n';
22 std::cout << "Possible␣is:␣'Other'␣␣'Platform'␣␣'Munition'
23 'Life␣form'␣␣'Environmental'␣␣'Cultural␣Feature'␣␣'Supply'
24 'Radio'␣␣'Expendable'␣or␣'Sensor/Emitter'␣" << '\n';
25 }
```

Listing 3.5: Funktion „getDISEntityType()“ Teil 1

Trifft der Fall nun zu, so wird nun die Domäne Geprüft, also ib es sich um ein Land, Luft, See oder Untersee Objekt handelt. Diese Information ist im String „Domain “ gespeichert. In der vorliegenden Version, sind nur die Fälle implementiert, in denen die Domäne Land oder Surface ist. Die anderen Fälle müssen noch implementiert werden. Trift einer der Fälle zu, wird nun geprüft, welchem Land das Objekt angehört.

```
1  switch (help.Domain) {
2  case 0: // domain other
3  break;
4  case 1: // domain land
5    for (std::map<int,std::string>::iterator
6         it=Country.begin(); it!=Country.end(); ++it){
7    if (it->second == country) {
8    help.Country =  it->first;
9    } }
10   switch (help.Country) {...}
11 break;
12 case 2: // domain air
13 break;
14 case 3: // domain surface
15 break;
16 ...
17 default:
```

```
18 | . . .
```

Listing 3.6: Funktion „getDISEntityType()“ Teil 2

Hier kann zwischen den USA, Russland und Deutschland entschieden werden. Sollen weitere Nationen verfügbar sein, muss die entsprechende Map angepasst werden.

3.5 Fähigkeiten der Klassenhierarchie

Im folgenden Anschnitt wird beschrieben, was die fertige Klassenhierarchie für Möglichkeiten bietet. Die konkrete Implementierung wird für ausgewählte Methoden im Abschnitt 3.4 erklärt. Grundlegen kann jedoch gesagt werden, dass die Klassenhierarchie eine Grundlage liefert, die je nach Bedarf beliebig erweitert werden kann. Diese setzt sich aus dem Ansatz 3 und der Klassenhierarchie für die Ausrüstung zusammen. Mit der Klassenhierarchie ist es Möglich ein Objekt zu erstellen, das ein ziviles oder militärischen Schiff, ziviles oder militärisches Flugzeug oder ein ziviles oder militärischen Landfahrzeug sein kann. Diesem Objekt können folgende Informationen gegeben werden:

- Name
- Typ
- Kategorie
- Nation
- Fraktionsangehörigkeit
- Position
- Geschwindigkeit
- Lage
- Equipment

Neben den oben genannten Attributen besitzt ein Objekt noch weitere, die nicht vom User gesetzt werden müssen, da sich bei entsprechender Eingabe im Konstruktor automatisch gesetzt werden. Klassen, bei denen diese Attribute automatisch im Konstruktor gesetzt werden, sind die „warShip“ und die „military“ Klasse. Diese Möglichkeit wurde bei diesen Klassen experimentell Implementiert. Des weiteren wurden eine Methode implementiert, mithilfe derer sich die Bewegung eines Objektes berechnen lässt.

Nach der Erstellung eines Objektes der Grundklassen, können auch Ausrüstungsobjekte erstellt werden, die dann anschließend einem Objekt der Grundklasse als Attribut hinzugefügt wird. Als Grundlage für die Ausrüstung wurde das Model genutzt, welches im Abschnitt 3.2 beschrieben ist. Hierbei wurde sich auf zwei Waffen und auf drei Sensoren beschränkt. Weitere Klassen können, wie bei den Beispielklassen gezeigt, hinzugefügt werden.
Die Fähigkeiten

3.6 Offene Probleme

4 Ausblick und Fazit

5 Anhang

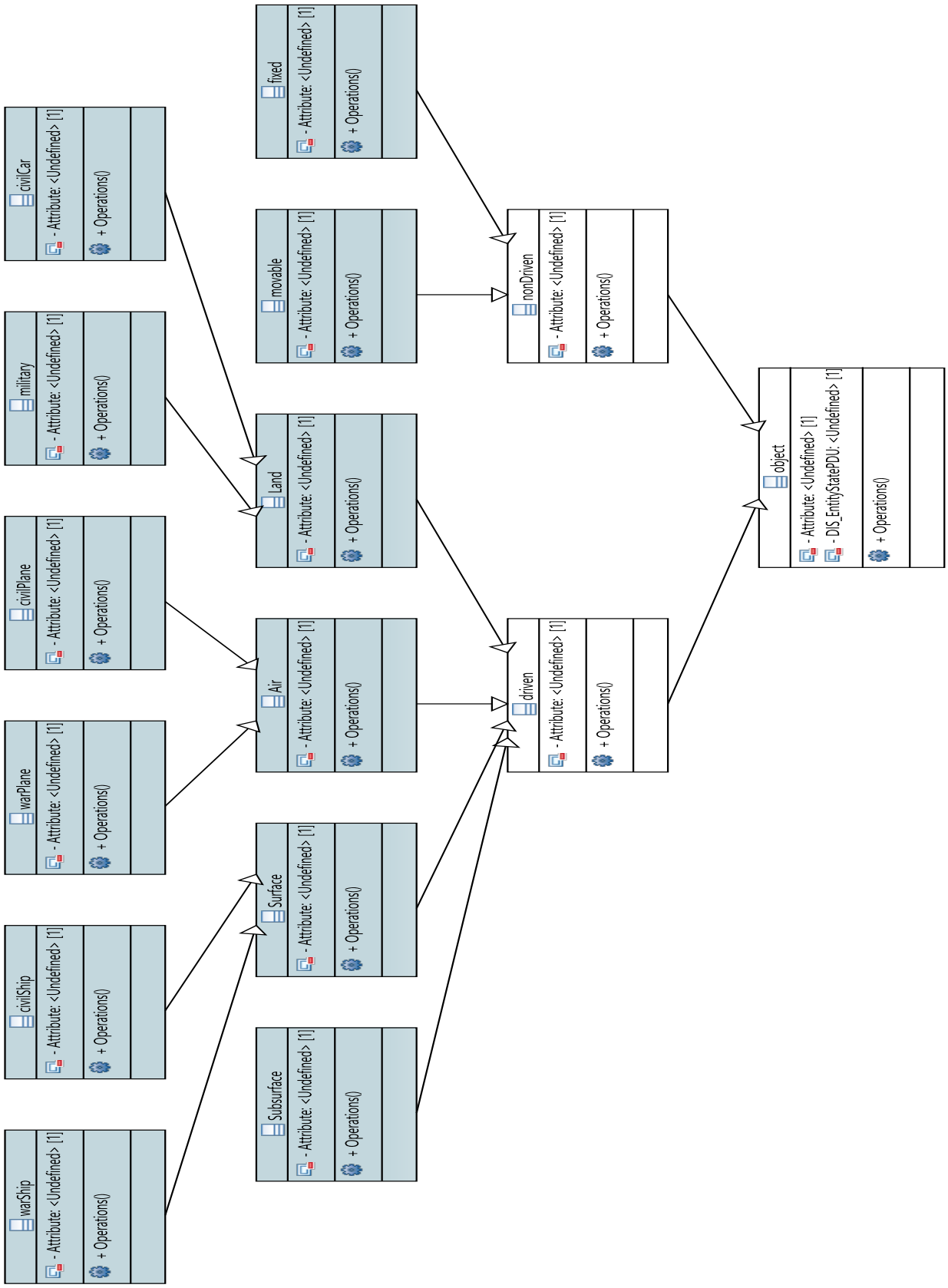

Bild 5.1: Ansatz 3

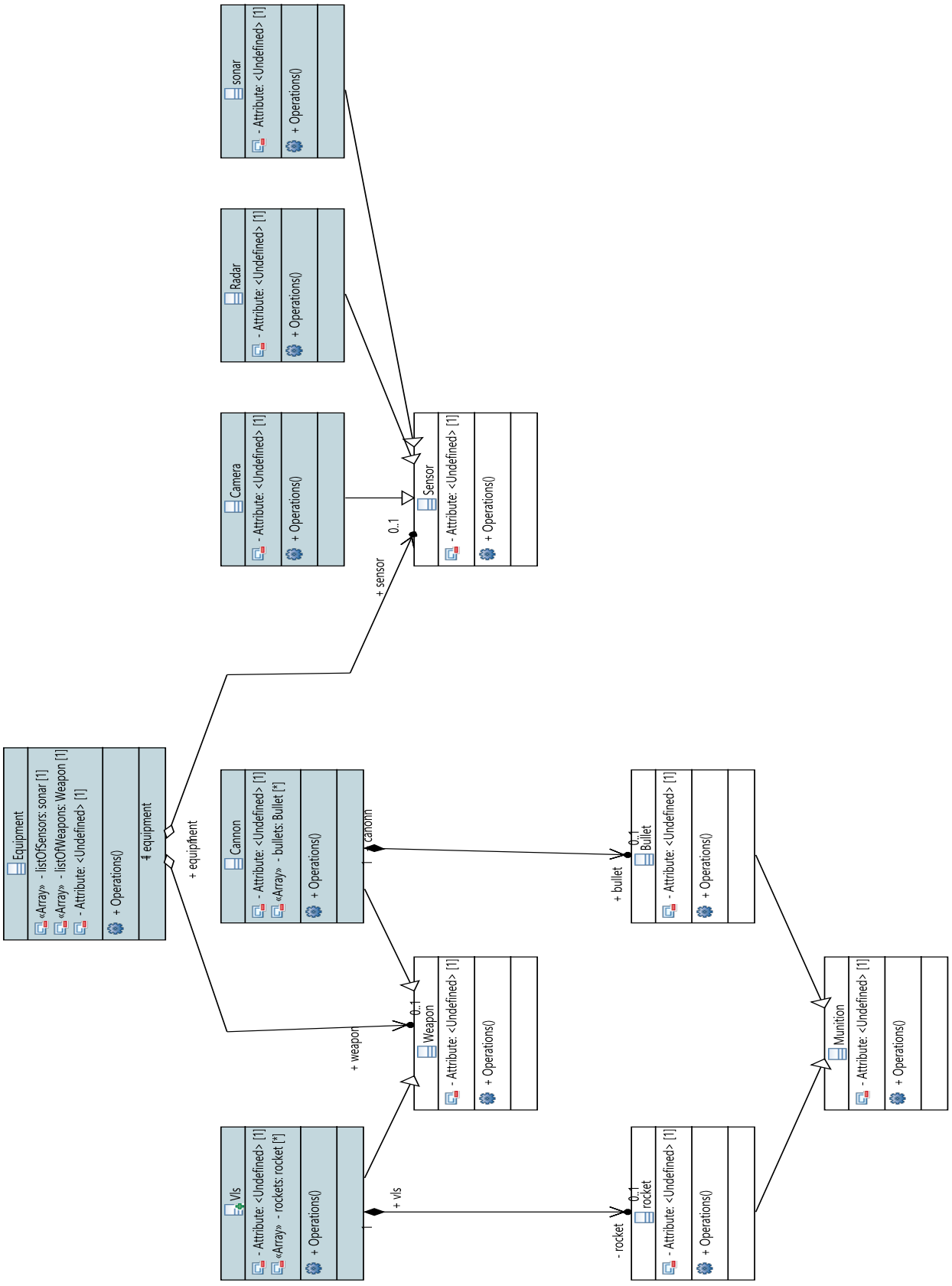

Bild 5.2: Ausrüstung

6 Literaturverzeichnis

[1] Helmut Erlenkötter. *C++ Objektorientiertes Programmieren von Anfang an.* 17. edition.

[2] Peter Krauß. Objektorientiertes programmieren in c++. `https://www.ldv.ei.tum.de/fileadmin/w00bfa/www/Vorlesungen/cpp/CPP-Skript.pdf`. [Online, Stand 09. August 2018].

[3] Prof. Dr. Alfred Irber. Objektorientierte programmierung (oop). `http://lsw.ee.hm.edu/~irber/oop/skriptum/skriptum_oop_cpp_ws12.pdf`. [Online, Stand 09. August 2018].

[4] Prof. Dr.-Ing. Wolfgang Schröder-Preikschat. Programmierung in c++. `https://www4.cs.fau.de/Lehre/SS05/V_VS/Uebung/folien/2-A6.pdf`. [Online, Stand 08. August 2018].

[5] Sebastian Meyer. C vs. c++ proseminar c - „grundlagen und konzepte“, 2013. `https://wr.informatik.uni-hamburg.de/_media/teaching/sommersemester_2013/cgk-13-meyer-c-vs-c_-ausarbeitung.pdf`. [Online, Stand 08. August 2018].

[6] SISO Standards Activity Committee of the IEEE Computer Society. Ieee std 1278.1-2012, ieee standard for distributed interactive simulation—application protocols.