CARL VON OSSIETZKY universität OLDENBURG

Anbindung einer Java-Implementierung an eine 3D-Animation

Individuelles Projekt

Andreea Taifas

Oldenburg, den 24. Januar 2005

C.v.O. Universität Oldenburg
Fakultät II - Departement für Informatik
Theoretische Informatik

Veranstalter:
Prof. Dr. E.-R. Olderog
Betreut durch:
Dipl. Inform. Michael Möller

Inhaltsverzeichnis

1 Einleitung **7**

1.1 ForMooS . . . 7

1.2 Aufgabenstellung . . . 8

1.3 Struktur dieses Dokuments . . . 8

2 Grundlagen **9**

2.1 Das holonische Fertigungssytem, HMS . . . 9

2.2 HMS-Implementierung . . . 10

2.3 3D-Animationssoftware . . . 11

2.3.1 Funktionsweise . . . 11

2.3.2 Darstellung . . . 13

3 Anforderungen **15**

3.1 Funktionale Anforderungen . . . 15

3.2 Weitere Anforderungen . . . 16

3.2.1 Programmiersprache . . . 16

3.2.2 Qualität . . . 16

4 Realisierungskonzept **19**

4.1 Entwurf einer abstrakten Schnittstelle . . . 19

4.2 Implementierung dieser Schnittstelle . . . 19

4.3 Integration der Animationsaufrufe in die HMS-Implementierung . . . 20

5 Entwurf der abstrakten Schnittstelle **21**

5.1 Verwendete Konzepte . . . 21

5.2 Entwurfsentscheidungen . . . 21

5.3 Beschreibung der Java-Interfaces . . . 22

6 Umsetzung der abstrakten Schnittstelle **27**

6.1 Allgemeine Grundlagen . . . 27

6.1.1 Java-Threads . . . 27

6.1.2 HMS-Implementierungsaspekte . . . 28

6.1.3 Kommunikation zwischen der HMS-Implementierung und der 3D-Animationssoftware . . . 32

6.2 Entwurf der Umsetzung . . . 32
6.2.1 Beschreibung der Klassen . . . 33
6.2.2 Der Kommunikationsablauf über die Socket-Schnittstelle . . . 37
6.3 Implementierung der Umsetzung . . . 42
6.3.1 Implementierungsdetails . . . 42
6.3.2 Koordinaten-Transformation: Die Klasse Mapping . . . 42
6.3.3 Integration der Animationsaufrufe in die HMS-Implementierung . 43
6.3.4 Dokumentation: JavaDoc . . . 44
6.4 Überblick über die gesamte Architektur . . . 45

7 Test der Schnittstelle 47
7.1 Bewertung des Ansatzes: Test . . . 47
7.1.1 Vorteile beim Testen . . . 47
7.1.2 Nachteile beim Testen . . . 48

8 Entwicklungsumgebung 51

9 Schlussbemerkungen 53
9.1 Fazit . . . 53
9.2 Ausblick . . . 54

Literaturverzeichnis 55

Abbildungsverzeichnis 57

Index 62

A Java API Dokumentation 63
A.1 Package hms.animation . . . 63
A.1.1 Interface Animation . . . 63
A.1.2 Interface Consumer . . . 65
A.1.3 Interface Hts . . . 66
A.1.4 Interface In . . . 68
A.1.5 Interface Machine . . . 69
A.1.6 Interface Producer . . . 71
A.1.7 Interface Workpiece . . . 72
A.2 Package hms.animation.i4d . . . 72
A.2.1 Class Agv . . . 73
A.2.2 Class CommunicationManager . . . 77
A.2.3 Class CommunicationManager.Instruction . . . 82
A.2.4 Class I4dAnimation . . . 83
A.2.5 Class InStore . . . 88
A.2.6 Class Machinetool . . . 91
A.2.7 Class OutStore . . . 94
A.2.8 Class WorkpieceSim . . . 97

A.3 Package hms.animation.test 98
A.3.1 Class Main 98
A.3.2 Class Main.HtsControl 99

1 Einleitung

Im Folgenden wird das Ziel des Projekts ForMooS [3] erläutert, welches die Grundlage für dieses Individuellen Projekt darstellt. Danach wird die Aufgabenstellung dieses individuellen Projektes vorgestellt. Abschießend wird die Struktur dieses Dokuments beschrieben.

1.1 ForMooS

Im Rahmen des Projektes ForMooS [3] wurde die systematische Entwicklung korrekter und zuverlässiger Software aus einer integrierten formalen Spezifikation untersucht. Zu diesem Zweck wurde die objekt-orientierte formale Spezifikation CSP-OZ [12] (eine Kombination der Prozeßalgebra Communicating Sequential Processes (CSP) mit der objektbasierte Spezifikationssprache Object-Z) in den Software-Entwicklungssprozess eingebettet, um die Vorteile der objekt-orientierten graphischen Modellierungssprache (UML) einerseits und der objekt-orientierten Implementierungssprache Java andererseits nutzen zu können. Von einem zuerst entwickelten UML-Modell wurde in diesem Prozess der Einbettung [14] ein UML-Profil für CSP-OZ entwickelt, durch das dem UML-Modell eine formale Semantik in Form einer CSP-OZ Spezifikation gegeben wird. Um die nötige Präzision der formalen Spezifikation in CSP-OZ weiter in der Implementierungssprache Java gewährleisten zu können, wurde danach eine weitere Brücke zwischen CSP-OZ und Java durch die Java Modeling Language JML (für die korrekten Zusicherungen in Java-Programmen) ergänzt um CSP_{jassda}-Spezifikationen geschlagen.

Da die Spezifikationssprache CSP-OZ besonders für den Bereich der reaktiven Systeme geeignet ist, wurde dieser Prozess für die Fallstudie „Produktionsautomatisierung“ aus dem Schwerpunktprogramm Software-Spezifikationen [4] der Deutschen Forschungsgemeinschaft (DFG) durchgängig durchgeführt.

Die so entstandene Java-Implementierung wurde auf Basis der CSP-OZ Spezifikation eines sogenannten holonischen Fertigungssystems (engl.: Holonic Manufacturing Systems, HMS) entwickelt, welches einen hohen Grad an Parallelität und Kommunikation miteinbezieht.

Holonische Fertigungssysteme sind Fertigungssysteme bei denen der Materialfluss in einem werkstattorientierten Produktionsumfeld mittels autonomer , flexibler und kooperativer Transportfahrzeuge realisiert werden. Die Koordination des Transportes und der Verarbeitung innerhalb der Fertigungsumgebung wird dezentral durch fünf Stationen und drei Transporter gesteuert.

1.2 Aufgabenstellung

Innerhalb dieses individuellen Projektes soll die vorhandene Java-Implementierung des HMS an die im DFG-Schwerpunktprogramm entwickelte Tcl/Tk 3D-Animationssoftware angeschlossen werden. Die Funktionalität dieser Java-Implementierung des HMS-Moduls soll nicht geändert werden.

Ziel des individuellen Projektes ist die Entwicklung einer Systematik für diese Anbindung, so dass eine Anpassung an andere 3D- oder 2D-Animationssysteme nur geringen Aufwand verursacht.

1.3 Struktur dieses Dokuments

In diesem Dokument folgt zunächst eine kurze Beschreibung der Grundlagen, auf denen dieses Projekt aufbaut. Danach werden die funktionalen Anforderungen vorgestellt, wie sie im Antrag zur Durchführung des Individuellen Projekts vereinbart wurden. Weiter wird ein konkretes Realisierungskonzept beschrieben. Im Anschluss daran werden der Entwurf der abstrakten Schnittstellle, die Umsetzung der abstrakten Schnittstelle sowie der Test der Schnittstelle beschrieben, die für die Realisierung der Aufgabenstellung für die Anbindung der vorhandene HMS-Implementierung an die Tcl/Tk 3D-Animation erstellt wurden. Abschließend folgen die Entwicklungsumgebung, ein Fazit und ein kurzer Ausblick.

2 Grundlagen

In diesem Kapitel sollen wichtige Grundlagen für dieses Individuelle Projekt erläutet werden. Zu diesem Zweck wird zunächst ein allgemeiner Überblick über die Komponenten des holonischen Fertigungssytems HMS gegeben. Danach wird die auf Basis der CSP-OZ Spezifikation entwickelte HMS-Implementierung vorgestellt. Abschließend wird auf die bestehende 3D-Animationssoftware eingegangen.

2.1 Das holonische Fertigungssytem, HMS

In einem holonischen Fertigungssytem (HMS) soll das Entgraten von Werkstücken (engl.: workpieces) in einem Produktionsumfeld mittels führerlosen Transportfahrzeugen realisiert werden. Hierbei werden die autonomen (d.h. ohne zentrale Steuerung) Transporter (engl.: automated guided vehicle, Agv) bei Bedarf von den Stationen: die Materiallager EIN (engl.: InStore) für die zu verarbeitenden Werkstücke und AUS (engl.: OutStore) für die entgrateten Werkstücke, sowie Werkzeugmaschinen (engl.: machine tools) gerufen, um Werkzeugteile zum Materiallager OutStore zu bringen oder sie zur weiteren Bearbeitung aus dem Lager InStore abzuholen. Da die Koordination des Transportes und der Verarbeitung innerhalb des Fertigungssystems dezentral durch fünf Stationen und drei Transporter gesteuert wird, wird von einem holonischen Fertigungssystem gesprochen.

Das Fertigungssystem ist in ein flexibles Produktionssystem eingebunden, das aus weiteren Fertigungssystemen besteht. In der Referenzfallstudie Produktionstechnik (Version 2.0) [7] wurden eine Grundstufe und eine Ausbaustufe des Fertigungssystems beschrieben.

In der Grundstufe des Fertigungssystems wird der Materialfluss von den drei autonomen Transportfahrzeugen realisiert, dessen Batteriekapazität so ausgelegt ist, dass während des Betriebes kein Aufladen der Batterie nötig ist. Für die fünf Stationen: drei Werkzeugmaschinen: einer Drei-Achsen-Fräsmaschine (engl.: mill), einer Fünf-Achsen-Fräsmaschine (engl.: drill) und einer Waschmaschine (engl.: wash) und die Materiallager: InStore und OutStore wird ein störungsfreier Betrieb [7] vorgesehen.

In der Ausbaustufe des Fertigungssystems ist von jedem Werkzeugmaschinetyp eine zweite Maschine vorgesehen, um somit einen alternativen Materialfluss zu ermöglichen. Zusätzlich können Störungen der Werkzeugmaschinen während der Verarbeitung der Werkstücke (z.B. Werkstückbruch) sowie der Kommunikation zwischen der Stationen und Transporter auftreten. Die Transporteranzahl in dieser Stufe ist auf sechs erhöht

und ein Aufladen der Batterie der Transporter muss mittels einer Batterieladestation (engl.: battery station) erfolgen. Auch ein Parkbereich (engl.: garage) für die nicht aktiven Transporter sowie ein Blockstreken-Management zur Verwaltung der Streken innerhalb des Fertigungssystems, um Kollisionen der Transporter zu vermeiden sind in der Ausbaustufe [7] enthalten. In der Ausbaustufe der Fallstudie sind drei Werkstückvarianten (engl.: eng, shaft und axle) vorgesehen. Von jeder Variante ist eine bestimmte vorgegebene Anzahl zu bearbeiten, bevor die Fertigung beendet wird.

2.2 HMS-Implementierung

Die Java-Implementierung des HMS wurde auf Basis der im Projekt ForMooS realisierten CSP-OZ Systemspezifikation des HMS entwickelt.

In dieser Implementierung meldet sich jeder Transporter an den Stationen an (*acquireOrder()*) und wartet auf Aufträge. Die Auswahl des Transporters erfolgt aufgrund des Angebots, das dieser schickt. Das Angebot hängt wiederum von der Distanz zur Station ab. Hierbei werden kürzere Entfernungen bevorzugt. Hat ein Transporter einen Auftrag (*newOrder()*) erhalten, transportiert es (*drive()*) ein Werkstück von bzw. zu der Station oder Ein- und Ausgangslager. Danach meldet es sich erneut an den Stationen und wartet auf Aufträge. Im entwickelten Beispiel wurde eine Konfiguration mit drei Transporter und drei Typen von Werkzeugmaschinen, sowie die Ein- und Ausgangslager gewählt. Der komplette Ablauf umfasst den Transport und die Bearbeitung von fünf Werkstücken. Die Reihenfolge des Materialflusses innerhalb des Fertigungssystems bezogen auf eine Werkstückvariante ist festgegeben. Sind alle bearbeiteten Werkstücke zum Ausgangslager (OutStore) gebracht worden, ist der Tages-Soll erreicht und die Produktion wird beendet.

Diese Implementierung des Fertigungsablaufs entspricht eher der Beschreibung der in der Referenzfallstudie [7] vorgestellte Grundstufe des Fertigungssystems, da hier keine Batterieladestation und kein Parkbereich (engl.: garage) vorhanden sind, wie in der Ausbaustufe vorgesehen.

Zur Zeit besteht keine Kommunikation zwischen dieser HMS-Implementierung und der 3D-Animationssoftware. Der Ablauf der HMS-Implementierung wird deshalb nur Textmeldungen auf der Konsole dargestellt.

2.3 3D-Animationssoftware

Im Rahmen des DFG-Schwerpunktprogramms wurde mittels Tcl/Tk eine 3D-Animationssoftware [8] entwickelt, die den Fertigungsablauf visualisiert, um die Entwicklung von HMS-Implementierungen und deren Testen zu unterstützen.

2.3.1 Funktionsweise

Für die visuelle Darstellung fungiert diese 3D-Animationssoftware als Server. Sie wartet im Hintergrund darauf, dass die HMS-Implementierung, im Folgenden allgemein auch als HMS-Steuerungssoftware bezeichnet, eine Verbindung zu ihr aufbaut. Die HMS-Steuerungssoftware spielt die Rolle des Clients und kommuniziert mit der 3D-Animationssoftware über eine Socket-Schnittstelle, wie in Abbildung 2.1 zu sehen ist. Nachdem die Socket-Verbindung erfolgreich aufgebaut wurde, können Nachrichten von der HMS-Steuerungssoftware zu ihr gesendet werden, die als stringbasierte Befehle interpretiert werden.

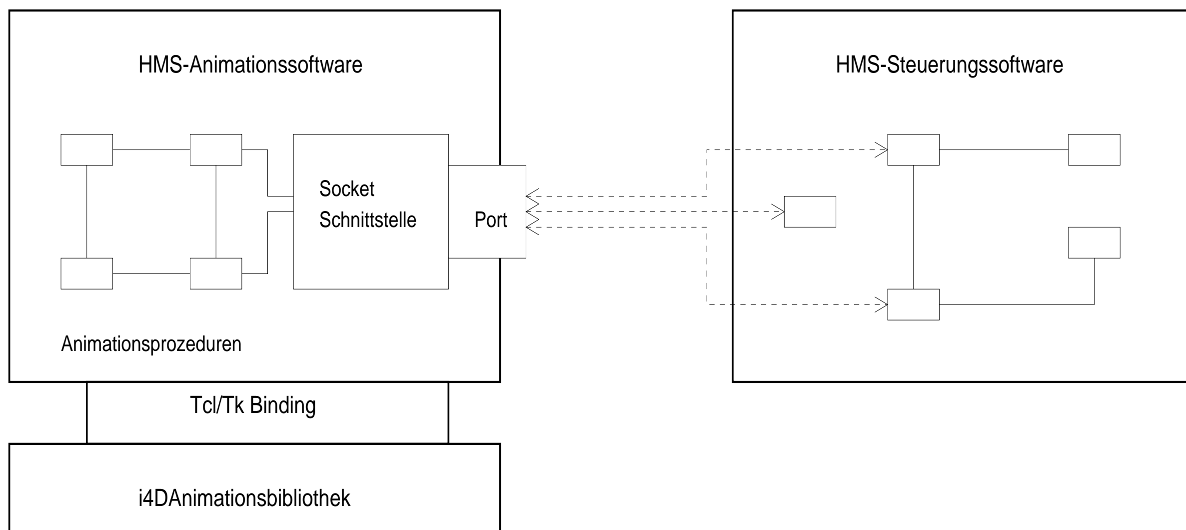

Abbildung 2.1: Socket-Schnittstelle der HMS-Animation

Die Kommunikation zwischen der 3D-Animationssoftware und der HMS-Steuerungssoftware ist in Abbildung 2.2 dargestellt. Auf jede eingehende Nachricht sendet die Animationssoftware eine Antwort zurück. Jede eingehende Nachricht hat eine eindeutige Nachrichten-Identifikation, die bei Antworten unverändert an den Absender zurückgeschickt wird, sodass für die HMS-Steuerungssoftware sichergestellt ist, dass die Nachricht angekommen ist. Die eingehenden Nachrichten sind in der linken Spalte zu sehen. Die Animationsantworte sind in der rechten Spalte eingetragen. Viele Antworten der Animationssoftware enthalten aber gleichzeitig auch eine Statusmeldung, die über die erfolgreiche oder misslungene Ausführung des Befehls Auskunft gibt, wie im Schaubild

zu sehen ist. Wenn bei einer solchen Antwort der Parameter *status* ungleich „ok“ ist, konnte der Befehl nicht oder nur teilweise ausgeführt werden und es wird eine genaue Fehlerursache ausgegeben. Im Bild 2.2 kann festgestellt werden, dass die visualisierten Nachrichten über die Socket-Schnittstelle erfolgreich durchgeführt werden konnten.

Die verwendeten String-Formate bei der Kommunikation über die Schnittstelle haben folgende Struktur:

Typ	Beschreibung
Eingehende Nachricht:	Format: commandname msgId args
Antwort:	Format: commandname msgId result

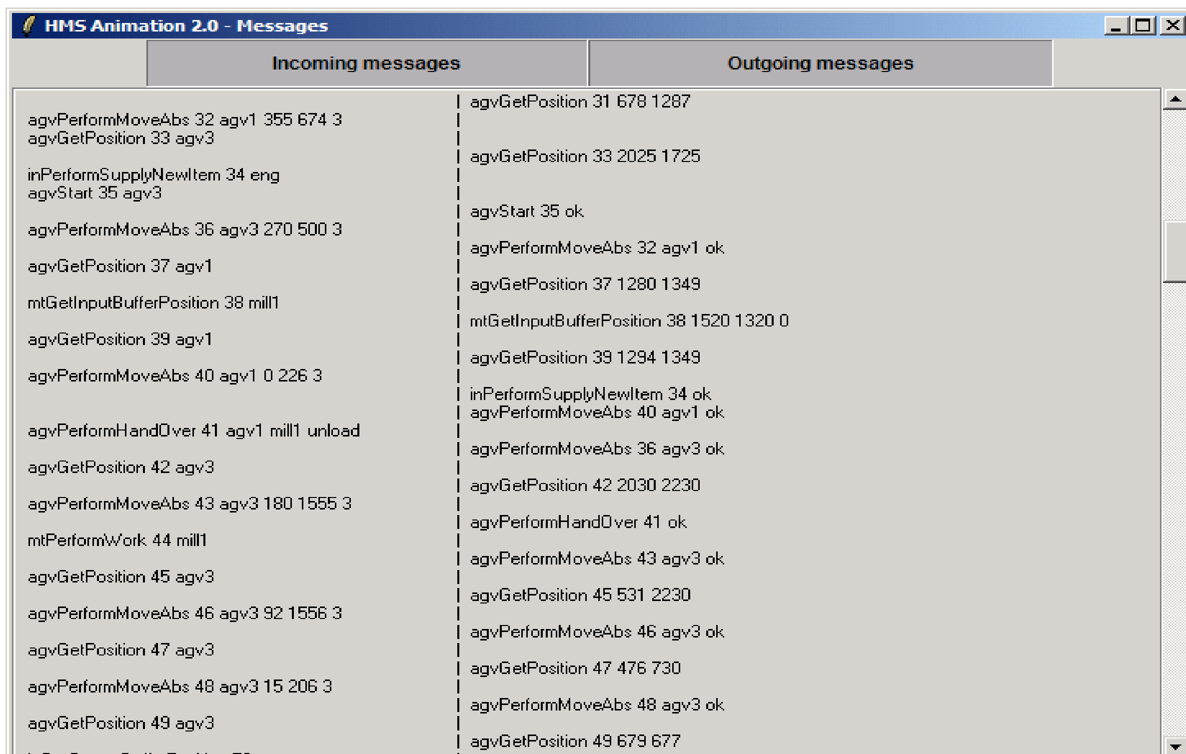

Abbildung 2.2: Die Kommunikation über die Schnittstelle

Bei vielen Animationsbefehlen hat man die Wahl zwischen zwei Modi:

- Start mit sofortiger Rückmeldung: Bei dieser Art von Animationsbefehlen wird der Animationsvorgang gestartet und ohne Verzögerung die entsprechende Antwortnachricht über den erfolgreichen oder misslungenen Start der Animation zurückgegeben. Diese Befehle haben ein „Start“ als Bestandteil ihres Namens, wie z.B. „agvStartMoveAbs“ oder „inStartSuppyNewItem“. Wie in der Schnittstellendefinition dieser 3D-Animationssoftware dokumentiert ist, bedeutet dies für eine HMS-Steuerung, dass der aktuelle Zustand des Animationsvorgangs mittels zusätzlicher Abfragen überwacht werden muss. Diese Abfragen beziehen sich auf die im Laufe des Vorgangs gerade angenommene Position des HTS, um zu erkennen, wann ein Animationsbefehl vollständig ausgeführt worden ist.

- Durchführung mit anschließender Rückmeldung: Bei dieser Art von Animationsbefehlen wird der Animationsvorgang gestartet und erst nach seiner erfolgten Ausführung sendet die Animationssoftware die Antwort an die HMS-Steuerungssoftware zurück. Befehle dieser Art haben ein „Perform“ als Bestandteil ihres Namens, wie z.B. „agvPerformMoveAbs“ oder „inPerformSupplyNewItem“.

2.3.2 Darstellung

Im Folgenden werden einige Aspekte erläutert, die mittels dieser 3D-Animationssoftware ermöglicht werden.

Die gesamte Fertigungsumgebung ist innerhalb eines rechteckigen Bereichs dargestellt - wie in Abbildung 2.3 zu sehen ist. Dieser Bereich ist durch die Koordinaten (0,0) und (4000,4000) bestimmt, wobei die Längeneinheit Zentimeter beträgt. Die dargestellte Fertigungsumgebung besteht aus fünf Stationen und drei Transporter. Die Stationen umfassen ein Eingangslager, drei Werkzeugmaschinen und ein Ausgangslager.

Da die Transporter ohne zentrale Steuerung innerhalb des Fertigungssystems fahren, hat jeder Transporter Aktoren und Sensoren, die mittels dieser 3D-Animationssoftware abgebildet sind. Auf der Seite der Aktorik können die Transporter somit Bewegungsabläufe und Werkstückübergaben simulieren. Auf der Seite der Sensorik erkennen die Transporter Kollisionen anhand der auf ihrer Frontseite angereicherten Sensoren. Zu den Sensoren zählen Berührungssensoren und ein Ultraschallmessgerät zur Abstandsmessung gegenüber anderen Objekten in ihrer näheren Umgebung.

Diese Ultraschallmessung wird in der 3D-Animation durch einen aufblitzenden Ausgangsstrahl visualisiert, wie im Bild dargestellt.

Innerhalb der Stationen gelangen die Werkstücke über Laufbänder und Senken zu den Puffern und Bearbeitungsplätzen. Die 3D-Animationssoftware unterstützt weiterhin auch die Visualisierung bei der Pufferung und bei der Bearbeitung von Werkstücken durch temporäre Farbänderungen oder animierte Teilobjekte.

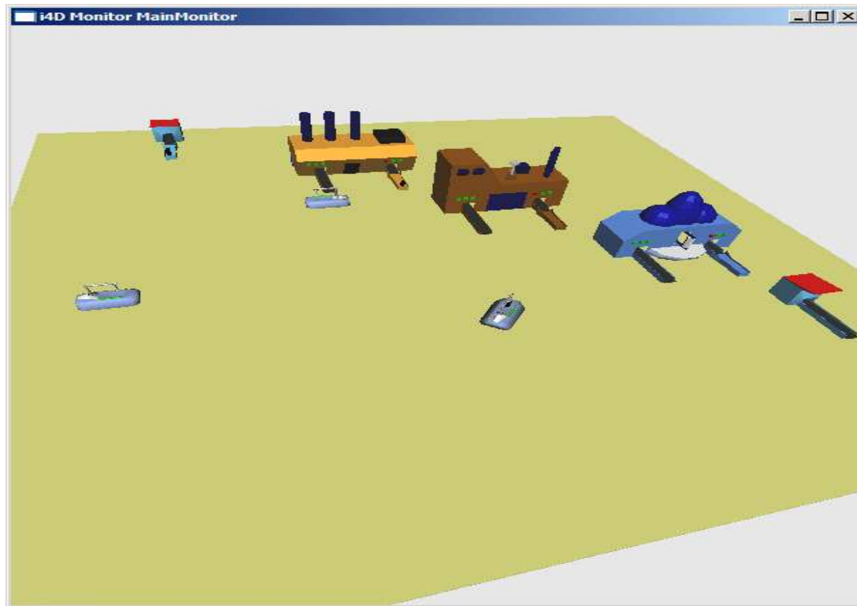

Abbildung 2.3: Das Fertigungssystem

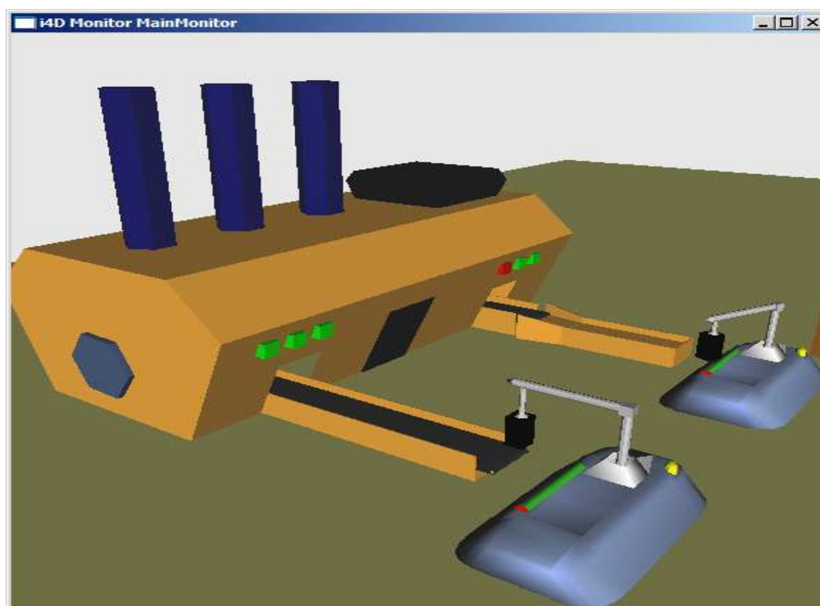

Abbildung 2.4: Das Beladen und Endladen von Werkstücken

Die Tcl/Tk 3D-Animationssoftware berücksichtigt allerdings keine Störungen während des Fertigungsablaufs, wie z.B. Werkzeugbruch oder Antriebsfehler der Transporter oder der Stationen. Mittels der Animationssoftware hat man lediglich die Möglichkeit, solche in der Fallstudie [4] beschriebene Störfälle visuell (durch rotes Blinken der Transporterlampe) mit einem Notfallbefehl in der 3D-Animation darzustellen.

3 Anforderungen

In diesem Kapitel sollen die funktionalen Anforderungen an die zu erstellende Schnittstelle zur 3D-Animation zuerst beschrieben werden. Im Anschluss daran werden weitere Anforderungen erläutert.

3.1 Funktionale Anforderungen

In der Vereinbarung über die Durchführung dieses Individuellen Projekts wurde die Aufgabenstellung für die Java-Anbindung der HMS-Implementierung an eine 3D-Animation festgelegt. Die wichtigsten Anforderungen an die zu erstellende Schnittstelle zur 3D-Animation werden hier nochmal genau beschrieben:

Animation der HMS-Implementierung: Die Hauptanforderung an die zu erstellende Schnittstelle ist die Animation der aus der CSP-OZ Spezifikation entstandenen HMS-Implementierung. Die Funktionalität der HMS-Implementierung wird (mit Ausnahme der Animationsaufrufe) nicht geändert.

Unabhängigkeit: Die Schnittstelle soll die Anforderungen verschiedener 3D- oder 2D-Animationssysteme berücksichtigen oder leicht an diese anzupassen sein. Die Schnittstelle soll also unabhängig von der zu verwendeten Animationssoftware zur Verfügung gestellt werden.

Kapselung spezifischer Eigenschaften: Aus dieser Unabhängigkeit ergibt sich die Notwendigkeit einer Kapselung aller spezifischen Eigenschaften der 3D-Animationssoftware.

Hoher Abstraktionsgrad: Klare, nachvollziehbare Funktionalität.

Erweiterbarkeit (optional): Die zu erstellende Schnittstelle könnte (optional) um Funktionalität, die nicht in der HMS-Implementierung benutzt wird, erweitert werden. Im Fall einer Erweiterung der Schnittstelle, muss auch die entsprechende Implementierung dieser Erweiterung für die Tcl/Tk 3D-Animation erfolgen. Diese Aspekte sollen schon beim Entwurf der Schnittstelle berücksichtigt werden.

Effizienz: Die Schnittstelle soll eine effiziente Zusammenarbeit der HMS-Implementierung und der Tcl/Tk 3D-Animationssoftware ermöglichen.

3.2 Weitere Anforderungen

In diesem Abschnitt werden weitere, nichtfunktionale Anforderungen an die Anbindung einer Java-Implementierung an eine 3D-Animation beschrieben.

3.2.1 Programmiersprache

Die anzubindende HMS-Implementierung ist in Java entwickelt. Die zu verwendenende 3D-Animation ist in Tcl implementiert und verwendet zur Darstellung Tk.

Tcl/Tk ist eine Kombination aus der Skriptsprache Tcl und dem Toolkit Tk, mit dem sich grafische Benutzungsoberflächen, sogenannte GUIs (Graphical User Interfaces), zusammenbauen lassen. Bei Tcl/Tk ist das Toolkit Tk eine Zusammenstellung von Funktionen zur Erstellung grafischer Fensterelemente.

Die Schnittstelle zwischen HMS-Implementierung und Tcl/Tk 3D-Animation, die in diesem Projekt erstellt wurde, sollte in Java implementiert werden.

3.2.2 Qualität

Die zu erstellende Schnittstelle soll den Anwendern einen komfortablen und einfachen Zugriff ermöglichen. Allgemeine Qualitätsanforderungen an die zu erstellende Schnittstelle, die erfüllt werden müssen, sind:

Funktionalität: Die Funktionalität der zu implementierenden Schnittstelle soll durch Testen überprüft werden, indem die Übereinstimmung zwischen funktionalen Anforderungen an diese und tatsächlich implementierter Schnittstelle untersucht wird. Dies ist vor allem dadurch bestimmt, dass die zu implementierenden Methoden der Schnittstelle von der HMS-Implementierung aufgerufen werden müssen, um derer Animation zu verwirklichen.

Zuverlässigkeit: Diese Anforderung lässt sich erreichen, indem die Schnittstelle vollständig, genau und konsistent implementiert wird.

Änderbarkeit und Wartung: Die Änderbarkeit der Schnittstelle ist vor allem durch ihrer Struktur bestimmt. Dies bedeutet, dass der zu ändernde Bereich eng begrenzt bleiben soll, also nicht über viele Klassen verteilt werden darf. Die Änderungsfreundlichkeit ist außerdem auch dadurch bestimmt, wie gut sie die Anwendungswelt modelliert und wie leicht sie sich erweitern lässt.

Übertragbarkeit: Die Schnittstelle sowie derer Implementierung werden mit der Programmiersprache Java realisiert, wodurch eine gewisse Plattformunabhängigkeit gewährleistet wird.

Dokumentation: Im Gegensatz zu der allgemeinen Vorgehensweise bei der Entwicklung von komplexen Softwaresystemen werden in diesem Individuellen Projekt hauptsächlich die resultierenden Ergebnisdokumente der Entwurfs- und der Implementierungsphase festgehalten. Der Quelltext der entwickelten Schnittstelle zwischen HMS-Implementierung und Tcl/Tk 3D-Animation wird durch JavaDoc dokumentiert. Mit JavaDoc bietet sich eine gute Möglichkeit, die in der Implementierung eingegebenen Kommentare direkt für die automatische Erstellung einer Programmdokumentation zu nutzen. Alle Klassen und Methoden werden hiermit näher erläutert.

4 Realisierungskonzept

Nachdem im vorherigen Kapitel die funktionalen Anforderungen an die zu erstellende Schnittstelle zur 3D-Animation dieses Individuellen Projekts erläutert wurden, soll im Folgenden der Realisierungsansatz der Schnittstelle beschrieben werden. Um die vorgestellten Anforderungen gewährleisten zu können, können folgende Aufgaben identifiziert werden:

4.1 Entwurf einer abstrakten Schnittstelle

Da die zu entwerfende Schnittstelle alle Funktionalitäten bereitstellen soll, die notwendig sind, um die HMS-Implementierung animieren zu können, wird diese Schnittstelle durch Java-Interfaces implementiert.

Java-Interfaces bieten die Möglichkeit, abstrakte Operationen zu spezifizieren, ohne die Implementierung dieser Methoden zur Verfügung zu stellen. Java-Interfaces definieren dabei, *was* getan werden sollte, aber nicht *wie* es zu tun ist. Dies bedeutet, dass sie durch eine höhere Abstraktion gekennzeichnet sind, um Implementierungsdetails zu verbergen. Dies hat zur Folge, dass die durch Java-Interfaces zur Verfügung gestellten Methoden, für verschiedene Animationssysteme danach implementiert werden können. Somit wird die im Abschnitt 3 vorausgesetzte Unabhängigkeit erfüllt.

Die Anforderung eine Erweiterung der Schnittstelle zu berücksichtigen, wird dadurch erfüllt, indem klare Schnittstellen geschaffen werden sollen. So können der Methodenumfang dieser Schnittstellen sowie neue Java-Interfaces leicht hinzugefügt werden. Klare Schnittstellen ermöglichen auch das schnelle Entwerfen und Implementieren von neuen Klassen.

4.2 Implementierung dieser Schnittstelle

Die entworfene Schnittstelle wird für die 3D Tcl/Tk-Animation implementiert. Daher müssen alle spezifische Eigenschaften in diesen Klassen gekapselt werden - wie in Kapitel 3 beschrieben. Diese Datenkapselung gemäß dem objekt-orientierten Entwurf macht dadurch auch ein Ändern der Funktionalität überschaubar.

Die Implementierung wird also die Verbindung zwischen der abstrakten Schnittstelle und der 3D Tcl/Tk-Animation sein, soll aber nicht von der HMS-Implementierung (hier als HMS-Modul bezeichnet) zugreifbar sein, wie in Abbildung 4.1 zu sehen ist.

Die Implementierung der Schnittstelle erfolgt mittels der Schnittstellendefinition zur 3D-Animationssoftware [8]. In diesem Dokument befindet sich die Beschreibung aller von der 3D-Animation unterstützenden Befehlen, zusammen mit den Wertebereichen der dafür benötigten Parameter. Das Dokument beinhaltet auch zwei UML-Zustandsdiagramme, welche die internen Zustände von Transportern und Werkzeugmaschinen visuallisieren. Somit spielt das Dokument eine wichtige Rolle bei der Implementierung der Schnittstelle.

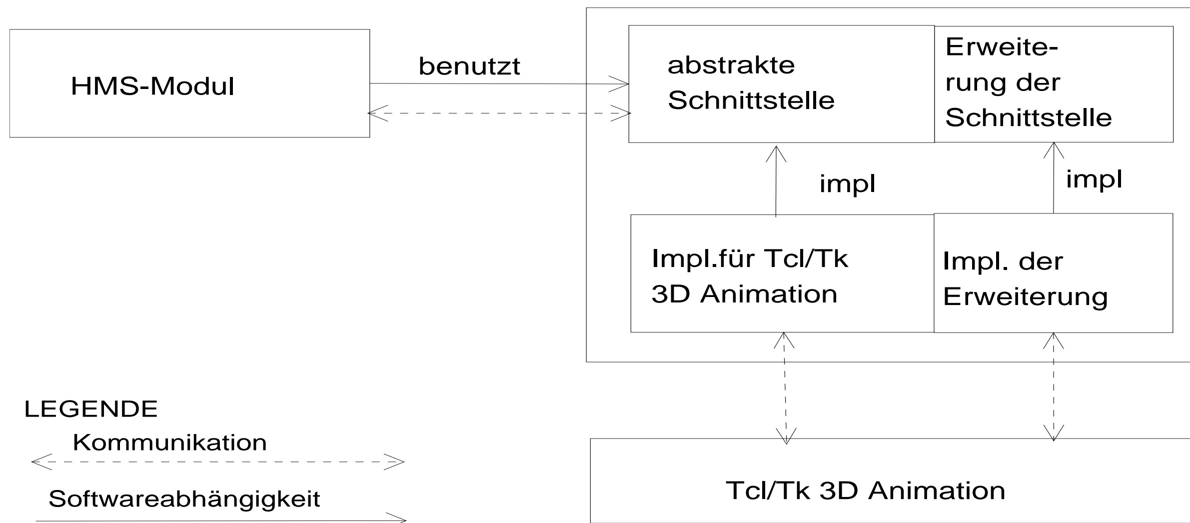

Abbildung 4.1: Realisierungsansatz der Schnittstelle

4.3 Integration der Animationsaufrufe in die HMS-Implementierung

Um die HMS-Implementierung animieren zu können, müssen nun die implementierten Methoden aufgerufen werden. Zu diesem Zweck sollen Methodenaufrufe geeignet in die HMS-Implementierung integriert werden. Die HMS-Implementierung greift dabei - wie in Abbildung 4.1 dargestellt - nur auf die abstrakte Schnittstelle zu, nicht auf die Klassen, die diese für die 3D Tcl/Tk-Animation implementieren. Die Funktionalität der HMS-Implementierung wird (mit Ausnahme der Animationsaufrufe) nicht geändert.

5 Entwurf der abstrakten Schnittstelle

In diesem Kapitel werden alle wichtigen Ergebnisse des Entwurfs der abstrakten Schnittstelle festgehalten. Der Entwurf dient als Grundlage für die abschließende Implementierung dieser Schnittstelle. Im Folgenden sollen einige wichtige Konzepte objekt-orientierter Programmiersprachen erläutert werden, die beim Entwurf der abstrakten Schnittstelle berücksichtigt wurden. Danach werden die Entscheidungen erläutert, die getroffen werden mussten, bevor mit dem Entwurf begonnen wurde. Die Java-Interfaces, visualisiert durch ein UML-Klassendiagramm, sowie deren Beschreibung zusammen mit ihren abstrakten Operationen folgen im Anschluss.

5.1 Verwendete Konzepte

Beim Entwurf dieser Schnittstelle wurden wichtige Prinzipien objekt-orientierter Programmierung angewandt. Diese sind:

Abstraktion: Verbergen von Implementierungsdetails, um eine weitgehende Unabhängigkeit der internen Programmierung gewährleisten zu können.

Polymorphie: Vererbung von Eigenschaften umgesetzt durch Ableiten von Interfaces.

Beim Entwurf dieser Schnittstelle war es entscheidend, ein zentrales Interface *Animation* - wie in Abbildung 5.1 zu sehen - zur Verfügung zu stellen. Dieses Interface stellt abstrakte Methoden bereit, die lediglich zur Erzeugung der Objekte innerhalb der Animationsumgebung verwendet werden können. Die entsprechenden Funktionalitäten des Animationsablaufs werden nicht miteinbezogen. Das Interface *Animation* ist also unabhängig von der verwendeten Animationssoftware. Somit lässt sich dieses Java-Interface für verschiedene Animationssysteme implementieren.

5.2 Entwurfsentscheidungen

Beim Entwurf der Schnittstelle wurden folgende Entscheidungen getroffen:

- Die Befehle, die zur Tcl/Tk 3D-Animationssoftware gesendet werden, werden in Methodenaufrufe umgesetzt, um die Kapselung der Daten gewährleisten zu können.

- Es wird bewusst darauf verzichtet, alle von der Tcl/Tk 3D-Animationssoftware unterstützten Befehle zu verwenden. Da die Hauptanforderung an die zu erstellende Schnittstelle die Animation der aus der CSP-OZ-Spezifikation entstandenen HMS-Implementierung ist, wären einige Nachrichten, die an die Tcl/Tk 3D-Animationssoftware geschickt werden können, wie z.B. „agvRemove“, „mtStopWorking“ oder „inRestart“, etc. überflüssig. Es ist daher sinnvoll, zuerst nur die notwendigen Methoden bereitzustellen. Dies ist mit keinen Nachteilen für den Animationsablauf der HMS-Implementierung verbunden. Eine Erweiterung der Schnittstelle könnte dann auch die zusätzlichen Methoden zur Verfügung stellen, sowie andere, die für die Ausbaustufe des Fertigungssystems zum Einsatz kommen.
- Wie in Abschnitt 2.3 erläutert wurde, hat man bei vielen Animationsbefehlen die Wahl zwischen zwei Modi: *Start mit sofortiger Rückmeldung* und *Durchführung mit anschließender Rückmeldung*. Beim Entwurf dieser Schnittstelle wurden Befehle verwendet, bei denen der Animationsvorgang gestartet wird und die Animationssoftware erst nach der erfolgten Ausführung eine Antwort an die HMS-Implementierung zurücksendet. Befehle dieser Art haben also ein „Perform“ als Namensteil wie z.B. „agvPerformMoveAbs“ oder „inPerformSupplyNewItem“.

5.3 Beschreibung der Java-Interfaces

Abbildung 5.1 zeigt die Java-Interfaces des Pakets *hms.animation*, die im Folgenden beschrieben werden.

Animation:

Dieses Interface kann von verschiedenen Animationsklassen implementiert werden. Es stellt verschiedene Methoden zum Erzeugen von Objekten in einer Animationsumgebung bereit. Das Interface legt mehrere Typen von Objekten fest, die zum Fertigungssystem gehören, damit alle Klassen, die das Interface implementieren, diese Typen von Objekten verwenden können. Diese Schnittstelle dient zur Ausführung der Darstellung diverser Objekten in einer (beliebigen) Animationsumgebung und gibt Auskunft über die erfolgreiche oder misslungene Ausführung dieser Darstellung. Das Interface enthält folgende Bestandteile:

- die Methode *public Hts createHts(Point postion, short direction)*, mit deren Hilfe ein Transporter als Objekt in der Animation visualisiert werden kann,
- die Methode *public Machine createMachine(Point postion, short direction)*, die verwendet wird, um verschiedene Werkzeugmaschinen zu erzeugen,
- die Methode *public In createInStore(Point postion, short direction)*, die das Materiallager InStore liefert und
- die Methode *public Consumer createOutStore(Point postion, short direction)*, die zur Erzeugung des Materiallagers OutStore dient.

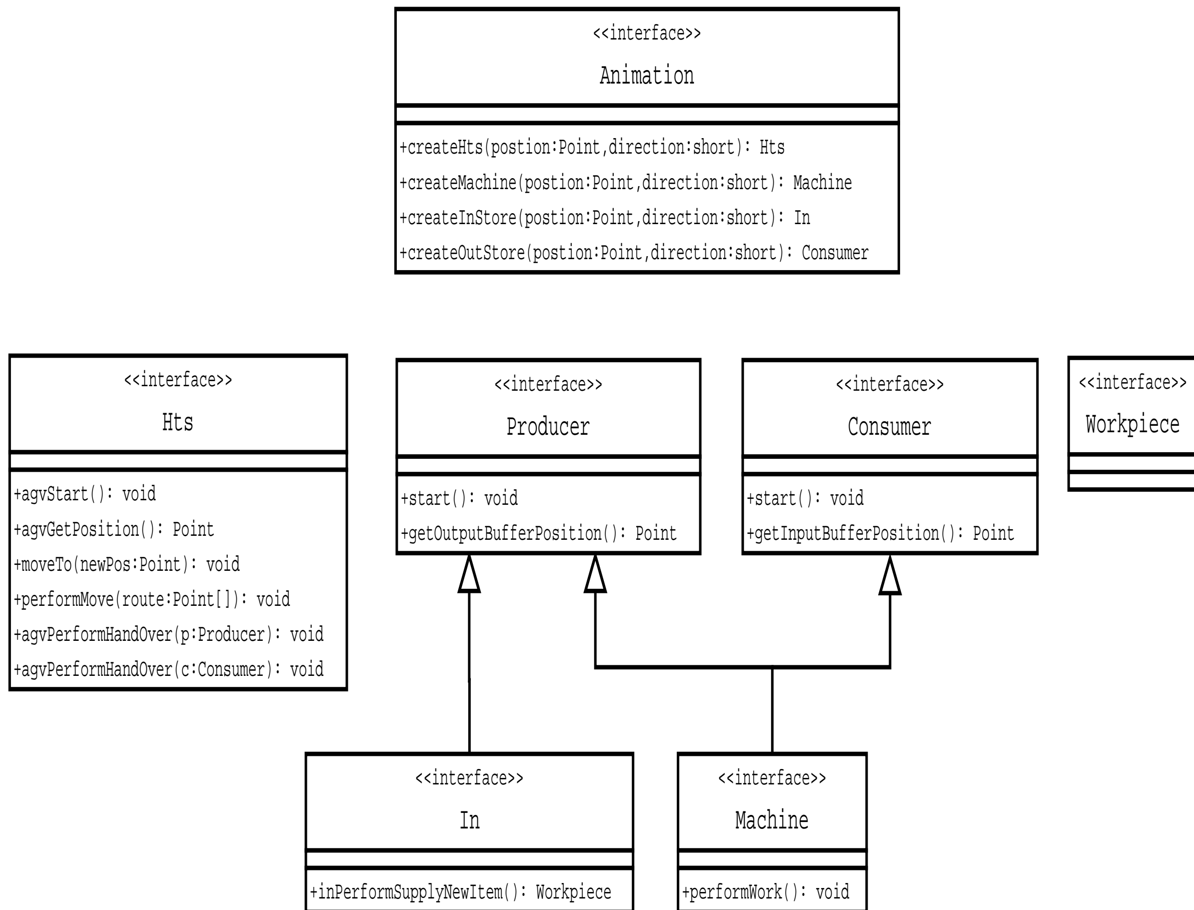

Abbildung 5.1: Überblick über die verwendeten Java-Interfaces

Producer:

Unter dem Begriff Producer sind alle Stationen in der Animationsumgebung zusammengefasst, die einen Ausgangspuffer haben, also das Materiallager InStore und die Werkzeugmaschinen (machine tools). Das Basis Interface *Producer* wurde bereitgestellt, um allgemeine Eigenschaften von Stationen mit einem Ausgangspuffer zu repräsentieren. Dieses Interface kann von verschiedenen Klassen implementiert werden. Eine implementierende Klasse liefert Methoden zum Starten eines Producers sowie zum Abfragen der Anlaufposition am Ausgangspuffer eines Producers.

Das Producer-Interface umfasst folgende Funktionen:

- die Methode *public void start()*, durch deren Aufruf ein Producer gestartet wird und
- die Methode *public Point getOutputBufferPosition()*, die benutzt wird, um die Anlaufposition am Ausgangspuffer eines Producers abzufragen.

Consumer:

Unter dem Begriff Consumer sind alle Stationen in der Animationsumgebung zusammengefasst, die einen Eingangspuffer haben, also das Materiallager OutStore und die Werkzeugmaschinen (machine tools). Eine implementierende Klasse dieses Interface liefert Methoden zum Starten eines Consumers sowie zum Abfragen der Anlaufposition am Eingangspuffer eines Consumers.

Das *Consumer*-Interface umfasst folgende Funktionen:

- die Methode *public void start()*, bei deren Aufruf ein Starten des Consumers erfolgt und
- die Methode *public Point getInputBufferPosition()*, die benutzt wird, um die Anlaufposition am Eingangspuffer eines Consumers abzufragen.

Machine:

Das Interface *Machine* wurde von den Interfaces *Producer* und *Consumer* abgeleitet. Es definiert die Eigenschaften von Werkzeugmaschinen. Diese haben einen Ausgangspuffer und einen Eingangspuffer. Aus diesem Grund vereinigt das Interface alle Methodendefinitionen der oben genannten Basis-Interfaces und stellt dazu folgende Funktion zur Vefügung:

- die Methode *public void performWork()*, die das Holen eines Werkstücks aus dem Eingangspuffer und dessen Bearbeitung veranlasst.

In:

Das abgeleitete Interface *In* erbt alle Methodendefinitionen des Basis-Interfaces *Producer* und erweitert dieses um die Methode:

- *public Workpiece inPerformSupplyNewItem()*, die beim Aufruf ein neues Werkstück im Eingangslager InStore erscheinen lässt.

Hts:

Dieses Interface stellt die Schnittstellendefinition für Transporter (Holonische Transportsysteme) bereit.

Das Interface *Hts* enthält folgende abstrakte Operationen:

- die Methode *public void agvStart()*, die beim Aufruf den Start eines Transporters veranlasst,
- die Methode *public Point agvGetPosition()*, mit deren Hilfe die aktuelle Position eines Transporters in der Animationsumgebung abgefragt werden kann,
- die Methode *public void moveTo(Point newPos)*, die als Parameter den neuen Wegpunkt bekommt, den ein Transporter anfahren soll,
- die Methode *public void performMove(Point[] route)*, die als Parameter die Wegpunkte einer Strecke bekommt, die ein Transporter anfahren soll,

- die Methode *public void agvPerformHandOver(Producer p)*, die das Entladen einer Station vom Typ Producer veranlasst und
- die Methode *public void agvPerformHandOver(Consumer c)*, die das Beladen einer Station vom Typ Consumer veranlasst.

Workpiece:

Das Interface *Workpiece* wurde erstellt, um Werkstücke als Objekte in der Animationsumgebung visualisieren zu können. Das Interface selbst stellt keine Methoden zur Verfügung, es wurde vielmehr aus Implementierungssgründen erstellt worden. Somit bietet es nur die Möglichkeit, Werkstücke zu initialisieren. Eine Klasse, die das Interface implementiert, kann hauptsächlich verschiedene Werkstückvarianten instanziieren.

6 Umsetzung der abstrakten Schnittstelle

Auf der Grundlage des Entwurfs - wie in Abschnitt 5.3 näher beschrieben - wird die Umsetzung der Schnittstelle für die Tcl/Tk 3D-Animation durchgeführt. Dieses Kapitel umfasst zuerst einige allgemeine Grundlagen, die eine entscheidende Rolle im Umsetzungsentwurf gespielt haben. Danach soll erläutert werden, wie die im vorigen Kapitel beschriebenen Java-Interfaces umgesetzt wurden. Zu diesem Zweck wurden zwei wesentliche Schritte durchgeführt: der Entwurf der Umsetzung der Schnittstelle und dessen Implementierung. Im Entwurf der Umsetzung werden die für die Tcl/Tk 3D-Animation entwickelten Klassen durch ein UML-Klassendiagramm dargestellt und näher erläutert. Nachfolgend wird auf die Implementierung eingegangen. Zum Schluss folgt ein Überblick über die gesamte Architektur.

6.1 Allgemeine Grundlagen

In diesem Abschnitt werden zuerst einige Grundlagen von Java-Threads erläutert, um die später beschriebenen Abläufe besser verstehen zu können. Danach wird auf wichtige HMS-Implementierungsaspekte eingegangen. Im Anschluss wird der Realisierungsansatz der Kommunikation zwischen der HMS-Implementierung und der 3D-Animationssoftware mittels Java-Threads vorgestellt.

6.1.1 Java-Threads

Java-Threads bieten eine Möglichkeit, eine nebenläufige Kommunikation über die Socket-Schnittstelle zu gestalten. Threads werden in Java durch die Klasse *Thread* und das Interface *Runnable* umgesetzt. In beiden Fällen wird der Thread-Body, also der parallel auszuführende Code, in Form der überschriebenen Methode *run()* zur Verfügung gestellt. Die Kommunikation mehrerer Threads erfolgt in Java auf Basis gemeinsamer Variablen. Führen mehrere Threads Änderungen auf den gemeinsamen Daten durch, müssen sie synchronisiert werden. Anderfalls können undefinierte Ergebnisse entstehen.

Zur Synchronisation paralleler Threads stellt Java das Monitorkonzept zur Verfügung. Ein Monitor verwaltet den Zugriff auf zu schützende Daten, indem er nur maximal einen Thread auf diesen Daten arbeiten lässt. Der Monitor unterstützt darüber hinaus die

Ablaufsynchronisation durch Methoden, mit denen Threads auf das Eintreten von bestimmten Zuständen warten können bzw. signalisieren können, dass sich der Zustand eines Objekts verändert hat. Die wartenden Threads werden vom Monitor in einer *Warteschlange* verwaltet. Monitore sind als Sprachkonzept in Java eingegangen. Dabei ist jedem Objekt, das geschützt werden muss, ein Monitor zugeordnet. Mit dem Schlüsselwort *synchronized*, wird das Objekt während der Ausführung der Methode für alle anderen Methoden, die ebenfalls als *synchronized* deklariert sind, gesperrt. Mit den Methoden *wait()*, *notify()* und *notifyAll()* der Javaklasse *Object* können Threads an einem Objekt warten und wieder aus der Warteschlange entfernt werden.

Man kann mit *synchronized* auch unter Angabe eines Objekts nur einen Anweisungsblock innerhalb einer Methode sperren. Immer nur ein Aufruf einer *synchronized*-Methode oder eines *synchronized*-Blocks erhält den *Lock* für das angegebene Objekt, um damit arbeiten zu können. Aufrufe anderer Threads werden solange blockiert. Ist der erste Aufruf abgearbeitet, erhält einer der blockierten Aufrufe in nichtdeterministischer Auswahl als nächster den *Lock* und kann abgearbeitet werden.

Durch die Blockierung beim Warten auf einen *Lock* ergibt sich die Gefahr eines *Deadlocks*. Zu einem *Deadlock* kann es kommen, wenn zwei Threads je ein Objekt gelockt haben und auf die Freigabe des jeweils anderen Locks warten. Deadlocks sind schwer zu finden, da sie von einer konkreten Aufrufreihenfolge verschiedener Threads abhängen, die an den Stellen, an denen die Threads eventuell nicht mehr weiterarbeiten können, nicht unbedingt nachvollziehbar ist. Daher sind *synchronized*-Aufrufe bei der nebenläufigen Programmierung mit Java sehr wichtig.

Darüber hinaus bietet Java auch Funktionen zur Verwaltung von Threads, auf die hier nicht weiter eingegangen wird, da sie für die Implementierung im Rahmen dieses Individuellen Projekts nicht benötigt werden.

6.1.2 HMS-Implementierungsaspekte

Die aus der im Projekt ForMooS realisierten CSP-OZ Systemspezifikation resultierende HMS-Implementierung besteht aus mehreren Klassen. Im Folgenden wird auf einige wichtige Klassen und Funktionen daraus eingegangen, um benötigte Aspekte für den Entwurf der Umsetzung festzuhalten. Die Klassenbeschreibungen in diesem Abschnitt geben nicht den vollständigen Inhalt der jeweiligen Klasse wieder. Eine vollständige Beschreibung der Klassen kann im Anhang dieses Dokuments, in der API-Beschreibung der entsprechenden Klassen nachgelesen werden. Danach werden relevante Implementierungsaspekte vorgestellt.

Die wichtigsten Klassen im Paket *hms.impl* sind in Abbildung 6.1 dargestellt. Zur besseren Übersichtlichkeit wird von allen Klassen in diesem Schaubild nur der Name angezeigt.

Die Klasse Machine: Die Klasse *Machine* ist von *Thread* abgeleitet. Ein Objekt dieser Klasse repräsentiert eine Station mit einem Namen und einer Position innerhalb

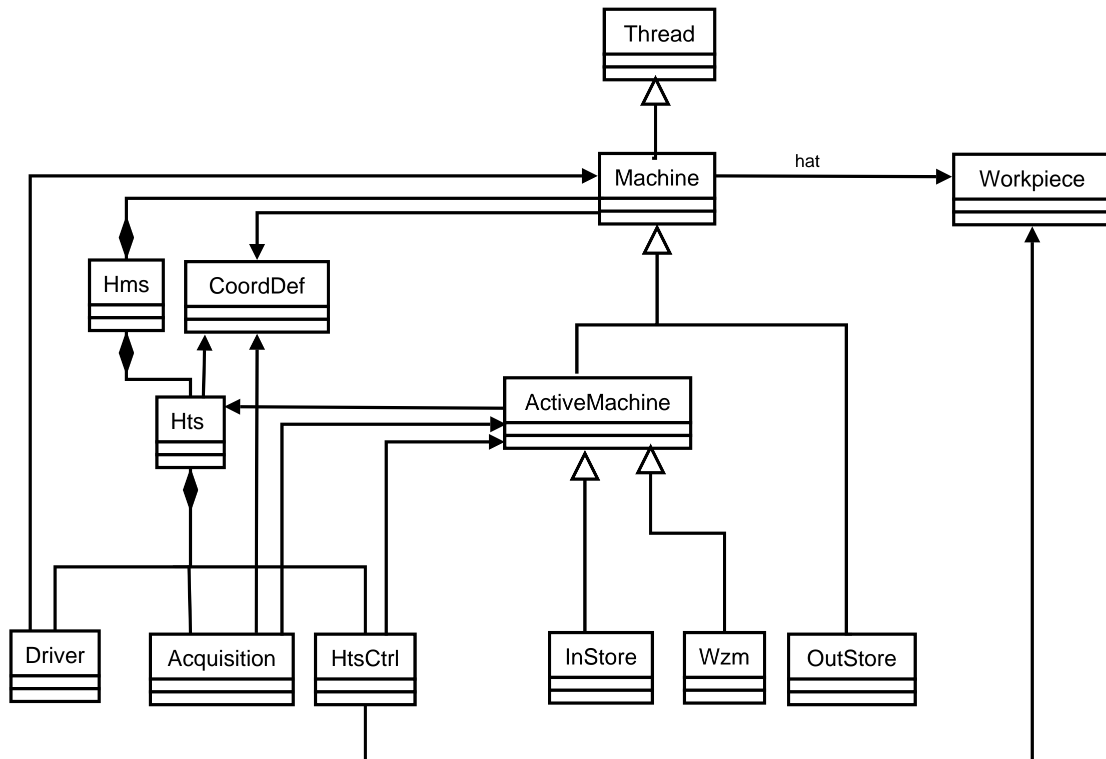

Abbildung 6.1: Das vereinfachte Klassendiagramm der HMS-Implementierung

des Fertigungssystems. Eine *Machine* ist also Teil des Fertigungssystems. Im Klassendiagramm 6.1 ist dies durch eine Komposition dargestellt. Die Position einer Machine kann mit Hilfe der Methode *CoordDef map()* abgefragt werden. Eine Station hat mehrere Werkstücke, die entweder zur Bearbeitung auf dem InputBuffer oder zur Abholung auf dem OutputBuffer bereit stehen. Deswegen stellt eine *Machine* auch Methoden zum Beladen (*WorkpieceSpec loadMachineHts(HtsSpec hts)*) und Entladen (*void loadHtsMachine(HtsSpec hts, WorkpieceSpec workpiece)*) von Werkstücken zur Verfügung. Die Kommunikation während des Ablaufs wird mittels der Methode *void log(String msg)* auf der Konsole dargestellt.

Die Klasse ActiveMachine: Die Klasse *ActiveMachine* ist von der Klasse *Machine* abgeleitet und um einige Methoden und Eigenschaften erweitert worden. Hierbei ist, wie der Name schon sagt, eine solche Machine aktiv. Dies bedeutet, dass sie mit den Transportern kommuniziert, indem sie durch die *doChoose()*-Methode eines davon auswählt und ihm (anhand seines Angebots - hier die Position zu ihr) einen Auftrag durch *loadMachineHts(MachineSpec ma, WorkpieceSpec workpiece))* erteilt. Dies geschieht nachdem sich alle Transporter bei ihr gemeldet haben. Als *ActiveMachine* werden InStore und die drei Werkzeugmaschinen bezeichnet. Das Beladen des Werkstücks auf dem Transporter wird vom *HtsCtrl* unterstützt.

Die Klasse Wzm: Die Klasse *Wzm* ist von der Klasse *ActiveMachine* abgeleitet und um

einige Methoden und Eigenschaften erweitert worden. Eine *Wzm* hat einen InputBuffer und einen OutputBuffer. Im Vergleich zu *InStore* muss eine *Wzm* zuerst abwarten, bis der Transporter das Werkstück entladen hat (*awaitLoadHtsMachine()*), danach muss sie das am Inputbuffer „wartende" Werkstück bearbeiten (*process()*). Nach der Bearbeitung des Werkstücks wählt sie durch den Aufruf *doChoose()* einen Transporter aus, der das am Outputbuffer „wartende" Werkstück entlädt. Dies erfolgt durch den Aufruf der *loadMachineHts(MachineSpec ma, WorkpieceSpec workpiece)*-Methode.

Die Klasse InStore: Die Klasse *InStore* ist, wie die Klasse *Wzm*, von der Klasse *ActiveMachine* abgeleitet, aber dadurch gekennzeichnet, dass sie nur einen OutputBuffer hat. Sie bearbeitet also keine Werkzeuge.

Die Klasse OutStore: Die Klasse *OutStore* ist von der Klasse *Machine* abgeleitet, also auch ein Thread. Dieses Materiallager wird nicht als aktiv während des Fertigungsablaufs betrachtet. *OutStore* hat genau wie die Werkzeugmaschinen einen InputBuffer, entnimmt also die fertig bearbeiteten Werkstücke. *loadMachineHts(HtsSpec hts)*

Die Klasse Hts: Die Klasse *Hts* ist auch von der *Thread*-Klasse abgeleitet. Ein Objekt dieser Klasse repräsentiert einen Transporter mit einem Namen und einer Position innerhalb des Fertigungssystems. Ein *Hts* ist wie eine *Machine* Teil des Fertigungssystems. Die Klasse *Hts* ist zusätzlich eine Aggregationsklasse, wie in Abbildung 6.1 angezeigt. Dabei handelt es sich um eine Komposition. Ihre Einzelteile: *HtsCtrl*, *Driver* und *Acquisition* sind als Threads realisiert, welche in der *run()*-Methode dieser Klasse gestartet werden.

Da die Teilobjekte aus denen sich ein *Hts* zusammensetzt mit diesem existentiell verknüpft sind, übernimmt ein *Hts* Operationen, die dann an die Einzelteile weitergeleitet (propagiert) werden. Jedem Teilobjekt werden bestimmte Aufgaben zugeordnet, die zur Realisierung des Materialflusses innerhalb des Fertigungssystems beitragen.

Die Klasse HtsCtrl: Der *HtsCtrl* kommuniziert mit der *ActiveMachine*, um mehrere Transporteraktivitäten steuern zu können. Zuerst veranlasst er durch die *acquireOrder()*-Methode das Melden des Transporters an die aktiven Maschinen. Danach muss der Transporter auf den durch die *ActiveMachine* zugeteilten Auftrag warten (*awaitNewOrder()*). Wenn ein Transporter einen Auftrag bekommt, gibt der *HtsCtrl* ihm an, zu welcher Maschine er fahren soll. Dies ist durch die *gotoMachine()*-Methode realisiert. Hierbei übernimmt der *Driver* die Steuerung der Bewegung. Am Ziel angekommen (*awaitArrived()*) steuert der *HtsCtrl* durch die *loadHtsMachine(master, master.loadHtsMachine(getTarget()))*-Methode, das Entladen des Werkstücks auf der Station. Weiterhin unterstützt der *HtsCtrl* das Beladen des Werkstücks auf dem Transporter.

Die Klasse Driver: Der *Driver*-Teil steuert durch die *drive()*-Methode die Bewegung des Transporters innerhalb des Fertigungssystem. Es wird solange gewartet, bis der *HtsCtrl* angibt, zu welcher Station gefahren wird. Der Driver kommuniziert mit allen Maschinen innerhalb des Fertigungssystems, um die aktuelle Position des Transporters zu ermitteln. Am Ziel angekommen wartet der Driver auf eine neue von HtsCtrl zugeteilte Fahrt.

Die Klasse Acquisition: Der *Acquisition*-Teil des Transporters kommuniziert mit den aktiven Maschinen beim Finden (*findOrder()*) von Aufträgen von diversen aktiven Maschinen. Dies erfolgt in Abhängigkeit der aktuellen Position des Transporters. Sie wird durch den Driver ermittelt.

Die Klasse Workpiece: Die Klasse *Workpiece* stellt ein Werkstück innerhalb des Fertigungssystems dar. Sie ist nicht als Thread realisiert. Es werden hauptsächlich verschiedene Werkstücke instanziiert.

Die Klasse Hms: Die Klasse *Hms* ist die Konfigurationsklasse des ganzen Fertigungssystems. Die HMS-Implementierung wird mittels der *main()*-Methode ausgeführt. Hierbei werden zuerst alle zu animierenden Objekten, die als Threads realisiert sind, instanziiert und innerhalb der Fertigungsumgebung positioniert. Durch die *run()*-Methode dieser Klasse werden danach die *start()*-Methoden dieser Threads aufgerufen. Die *run()*-Methode der jeweiligen Threads wird automatisch durch das Laufzeitsystem asynchron aufgerufen. Dies hat zur Folge, dass diese parallel zur main()-Methode ausgeführt werden. Die *run()*-Methode dieser Threads ist als Endlosschleife realisiert. Deswegen läuft die Ausführung der HMS-Implementierung solange, bis die Unterbrechung des Programms durch den Anwender erfolgt.

Es folgen einige Aspekte zum Umsetzungsentwurf:

- Die zu animierenden Objekte sind als Threads realisiert. Dies bedeutet, dass das Zuordnen der Antworten von der Tcl/Tk 3D-Animationssoftware zu den jeweiligen Threads ein wichtiger Aspekt beim Entwurf der Umsetzung ist.

  Dies erklärt sich wie folgt: Der Thread testet zuerst die while-Bedingung innerhalb eines kritischen Abschnitts. Falls sie nicht erfüllt ist oder ein anderer Thread gerade diesen kritischen Abschnitt ausführt, wird der aufrufende Thread blockiert. Ansonsten führt der aufrufende Thread (unter Ausschluss anderer Threads) den kritischen Abschnitt aus. Anschließend wird durch die *notify*-Methode ein blockierter Thread aufgeweckt. Um eine Sperre anzufordern oder freizugeben, wird ein *synchronized*-Konstrukt verwendet.

- Es ist erforderlich, dass diese Threads synchronisiert auf die Socket-Schnittstelle zugreifen. Zu diesem Zweck wird eine Klasse benötigt, die als Singleton zu entwickeln ist, um den Zugriff dieser Threads auf die Socket-Schnittstelle steuern zu können.

6.1.3 Kommunikation zwischen der HMS-Implementierung und der 3D-Animationssoftware

Die in diesem individuellen Projekt entwickelte Schnittstelle soll zuerst die HMS-Implementierung bei der Herstellung der Verbindung mit der Tcl/Tk 3D-Animationssoftware unterstützen. Wie in Abschnitt 2.3 erläutet wurde, spielt die HMS-Steuerungssoftware die Rolle des Clients und kommuniziert mit der 3D-Animationssoftware über eine Socket-Schnittstelle. Dieser Zusammenhang ist in Abbildung 2.1 dargestellt.

Daraus ergeben sich folgende Überlegungen zum Umsetzungsentwurf:

- Es muss zuerst eine Socket-Verbindung aufgebaut werden, damit die 3D-Objekte in der Animationsumgebung dargestellt werden können. Zu diesem Zweck muss eine Klasse entworfen werden, die diese Verbindung herstellt.
- Die HMS-Implementierung besteht aus mehreren, eigenständigen Threads, die jeweils als eigene Clients über die Socket-Verbindung mit der Animationssoftware verbunden werden. Die Socket-Verbindung wird also von allen Clients gemeinsam genutzt. Um dieser Anforderung zu genügen, müssen die entsprechenden Prozesse auf dem Socket synchronisiert werden.
- Über die Socket-Schnittstelle wird für jede eingehende Nachricht eine Antwort mit derselben Identifikation Id zurückgesendet. Deswegen werden lesende sowie schreibende Zugriffe auf dem Socket benötigt.
- Beim Datenaustausch zwischen der HMS-Implementierung und der 3D-Animation über diese Socket-Verbindung muss daher das Request-Reply-Konstrukt (der Sender wartet, bis der Empfänger die Nachricht empfangen und bearbeitet hat) verwendet werden. Da die String-basierten Befehle durch Methodenaufrufe umgesetzt werden müssen, bedeutet dies, dass ein Methodenaufruf erst beendet werden kann (Rücksprung), wenn der entsprechende Vorgang komplett animiert wurde.
- Die Tcl/Tk 3D-Animationssoftware verwendet Objektnamen, um die dargestellten Objekte während des Fertigungsablaufs anzusprechen. Da Java eine objektorientierte Programmiersprache ist, werden Objekte und Klassen verwendet, um die Anwendungswelt wiederzuspiegeln. Dies hat zur Folge, dass bei der Implementierung für jeden Objektnamen eine Objektinstanz geschaffen wird.

Um diese Überlegungen konkret umzusetzen, wurde zuerst ein grober Entwurf angefertigt, der danach schrittweise verfeinert wurde.

6.2 Entwurf der Umsetzung

Die oben beschriebenen Überlegungen wurden beim Entwurf der Klassen und deren Methoden berücksichtigt. Die in Bild 6.2 dargestellten Klassen entsprechen dem Umsetzungsentwurf der Schnittstelle und werden in diesem Abschnitt vorgestellt. Hierbei

werden nur die beiden Klassen *CommunicationManager* und *I4dAnimation* detailiert beschrieben. Die anderen Klassen implementieren die im Entwurf der abstrakten Schnittstelle spezifizierten Java-Interfaces und werden hier nur kurz beschrieben.

6.2.1 Beschreibung der Klassen

Die Klasse CommunicationManager

Die Klasse `CommunicationManager` ist von der *Thread*-Klasse abgeleitet, wie in Abbildung 6.2 dargestellt. Sie verwaltet die Kommunikation zwischen der HMS-Implementierung und der Tcl/Tk 3D-Animation und erfüllt folgende Anforderungen.

- Der Verbindungsaufbau erfolgt durch Öffnen eines Sockets zur Tcl/Tk 3D-Animation und Erzeugen eines *PrintWriters* und eines *Readers*.

  ```
  Socket socket = new Socket(String host, int port);
  PrintWriter channelOut  =  new PrintWriter(socket.getOutput-
                             Stream(), true);
  BufferedReader channelIn = new BufferedReader(
                             new InputStreamReader(socket.get-
     InputStream()));
  ```

  Sobald die Socket-Verbindung zur 3D-Animationssoftware erfolgreich hergestellt wurde, kann auf dem Socket gelesen und geschrieben werden und somit können Befehle bidirektional gesendet und empfangen werden.

- Alle Befehle der Kommunikation werden zusammen mit ihren IDs in einer *HashMap*-Liste festgehalten. In diese Liste werden die versendeten Befehle (zusammen mit deren zugehörigen IDs) eingefügt. Wenn Antworten von der Animationssoftware mit derselben ID empfangen werden, werden die Einträge aus der Liste gelöscht. Da die Socket-Verbindung von mehreren Clients gemeinsam genutzt wird, müssen diese Änderungen in der Liste synchronisiert werden, um gleichzeitige Änderungen verschiedener Threads zu verhindern. Dies wird erreicht, indem entsprechende *synchronized*-Blöcke in der *String[] sendAndReceive(String command, String[] arguments)*-Methode verwendet werden. Diese Methode dient dazu, die Synchronisation der Threads aus beiden Richtungen - von der HMS-Implementierung zur 3D-Animationssoftware einerseits und von der 3D-Animationsoftware zur HMS-Implementierung andererseits - zu erreichen.

- Auf die Antwort auf jede gesendete Nachricht muss gewartet werden, um einen Methodenaufruf der HMS-Implementierung zu beenden. Dies wird erreicht, indem die Methode *void handleAnswer(String answer)* verwendet wird. Auf diese Weise können die verschiedenen Threads miteinander kommunizieren. Dadurch

wird der zur Zeit laufenden Thread, der gerade gesendet hat, mit dem Aufruf der *wait()*-Methode des Instruction-Objekts in der *String[] sendAndReceive(String command, String[] arguments)*-Methode unterbrochen und als wartend markiert. Der wait-Zustand bleibt bestehen, bis der Antwort-Thread in der *void handleAnswer(String answer)*-Methode eine passende Nachricht empfängt und somit die *notify()*-Methode des Instruction-Objekts aufruft.

Folgende Tabelle zeigt den Zusammenhang zwischen der String-basierten Kommunikation und der objekt-orientierten Implementierung:

Tcl/Tk 3D-Animation	`CommunicationManager`
Befehle	Methode
String-Befehl: commandname msgId args	Methodenaufruf: sendAndReceive(String command, String[] arguments)
Antwort-String: commandname msgId result	Methoderückgabe: return result

Die Klasse `I4dAnimation`

Diese Klasse implementiert die Schnittstelle *Animation* - wie im Bild 6.2 dargestellt - und realisiert die HMS-Animation mittels der Tcl/Tk 3D-Animationssoftware. Bei der Erzeugung jedes Animationsobjekts in der Fertigungsumgebung wird die *connect()*-Methode dieser Klasse aufgerufen. Innerhalb der connect()-Methode wird ein *CommunicationManager*-Objekt erzeugt, das die Socket-Verbindung aufbaut. Alle Threads nutzen dieses *CommunicationManager* gemeinsam, um über die SocketSchnittstelle beliebige Nachrichten als Folge von Methodenaufrufen zur 3D-Animationssoftware zu senden, auf die von der 3D-Animationssoftware Antworten zurückgesendet werden.

Von dieser Klasse darf nur ein Objekt erzeugt werden, da auf diese Weise der Zugriff von Threads auf die Tcl/Tk 3D-Animation gesteuert werden kann. Also wurde sie als *Singleton* implementiert. Die Klasse *I4dAnimation* stellt damit eine globale Zugriffsmöglichkeit auf dieses Objekt zur Verfügung und instanziert es beim ersten Zugriff automatisch. Mit der Methode *getI4dAnimation()* kann auf diese einzige Instanz zugegriffen werden.

Mit dem Aufruf der *connect()*-Methode der Klasse *I4dAnimation* wird ein *CommunicationManager*-Objekt erzeugt und auf diese Weise kann die Verbindung zur 3D-Animationssoftware hergestellt werden. Die *run()*-Methode der *CommunicationManager*-Klasse wird parallel mit der *start()*-Methode ausgeführt.

Die Klasse *I4dAnimation* ist also eine wichtige Komponente zur Realisierung der im Abschnitt 3 geforderten Animation der HMS-Implementierung.

Die Klasse `Agv`

Die Klasse *Agv* implementiert die Schnittstelle *Hts*. Ein Objekt dieser Klasse repräsentiert einen Transporter in der Animationsumgebung. Die Klasse *Agv* kapselt die Agv-Operationen, die von der HMS-Implementierung abgefragt werden können. Im Einzelnen sind dies:

- die Methode *private Point[] route(Point from, Point to)*, die ein Array mit den Wegpunkten einer Strecke liefert, die ein Transporter zwischen zwei Stationen fahren soll,
- der Konstruktor *Agv(String agvId, CommunicationManager comm)*, der zum Erstellen eines Objektes der Klasse Agv dient,
- die Methode *protected String sendAndReceive(String command, String[] arguments)*, die einen String mit der Animationsantwort auf eine eingehende Nachricht zurückliefert,
- die Methode *public void agvStart()*, die beim Aufruf den Start eines Transporters veranlasst,
- die Methode *public Point agvGetPosition()*, mit deren Hilfe die aktuelle Position eines Transporters in der Animationsumgebung abgefragt werden kann,
- die Methode *public void moveTo(Point newPos)*, die als Parameter den neuen Wegpunkt bekommt, den ein Transporter anfahren soll,
- die Methode *public void performMove(Point[] route)*, die als Parameter die Wegpunkte einer Strecke bekommt, die ein Transporter fahren soll,
- die Methode *public void agvPerformHandOver(Producer p)*, die das Entladen einer Station vom Typ Producer veranlasst und
- die Methode *public void agvPerformHandOver(Consumer c)*, die das Beladen einer Station vom Typ Consumer veranlasst.

Jedes Agv hat eine eindeutige Identifikation, um innerhalb der Animationsumgebung angesprochen zu werden.

Die Klasse `InStore`

Die Klasse *InStore* implementiert das Interface *In*. Da das Interface *In* von *Producer* abgeleitet ist, erbt es alle Methodendefinitionen des Basis-Interfaces *Producer*. Daher implementiert die Klasse *InStore* alle Methoden des Interfaces *Producer* und stellt zusätzlich einen Konstruktor, die *inPerformNewItemType()*-Methode sowie die *String sendAndReceive(String command, String[] arguments)* zur Verfügung. Der Konstruktor dieser Klasse liefert nach dem Starten des Materiallagers ein neues Werkstück vom Typ „eng“.

Die Klasse *InStore* hat folgende Methoden:

- der Konstruktor *public InStore(CommunicationManager comm)*, der zum Erstellen eines Objektes dieser Klasse InStore dient,
- die Methode *public void start()*, die beim Aufruf den Start des Materiallagers InStore veranlasst,
- die Methode *protected String sendAndReceive(String command, String[] arguments)*, die einen String mit der Animationsantwort auf eine allgemeine eingehende Nachricht zurückliefert, falls der Befehl erfolgreich durchgeführt werden konnte,
- die Methode *public Workpiece inPerformSupplyNewItem()*, die beim Aufruf ein neues Werkstück im Eingangslager InStore erscheinen lässt und
- die Methode *public Point getOutputBufferPosition()*, die benutzt wird, um die Anlaufposition am Ausgangspuffer des InStore abzufragen.

Die Klasse `Machinetool`

Die Klasse *Machinetool* implementiert das Interface *Machine*. Sie stellt folgende Methoden zur Vefügung:

- der Konstruktor *Machinetool(String stationId, CommunicationManager comm)*, der eine Werkzeugmaschine initialisiert,
- die Methode *public void start()*, mit deren Hilfe eine Werkzeugmaschine gestartet werden kann,
- die Methode *public Point getOutputBufferPosition()*, die benutzt wird, um die Anlaufposition am Ausgangspuffer einer Werkzeugmaschine abzufragen,
- die Methode *public Point getInputBufferPosition()*, die benutzt wird, um die Anlaufposition am Eingangspuffer einer Werkzeugmaschine abzufragen
- die Methode *protected String sendAndReceive(String command, String[] arguments)*, die einen String mit der Animationsantwort auf eine eingehende Nachricht zurückliefert,
- die Methode *String getStationId()*, die zum Holen einer Werkzeugmaschine verwendet wird und
- die Methode *public void performWork()*, die das Holen eines Werkstücks aus dem Eingangspuffer und dessen Bearbeitung veranlasst.

Die Klasse `OutStore`

Die Klasse *OutStore* implementiert das Interface *Consumer*. Diese Klasse hat folgende Methoden:

- der Konstruktor *public OutStore(CommunicationManager comm)*,
- die Methode *public void start()*, die beim Aufruf den Start des Materiallagers OutStore veranlasst,
- die Methode *public Point getInputBufferPosition()*,die verwendet wird, um die Anlaufposition am Eingangspuffer des OutStore abzufragen und
- die Methode *protected String sendAndReceive(String command, String[] arguments)*, die einen String mit der Animationsantwort auf eine eingehende Nachricht zurückliefert.

Die Klasse `WorkpieceSim`

Die Klasse *WorkpieceSim* implementiert das Interface *Workpiece*. Hierbei können im Konstruktor hauptsächlich verschiedene Werkstückvarianten instanziiert werden. Es gibt drei Werkstückvarianten: „eng“, „shaft“ und „axle“. Jede dieser Werkstückvarianten ist dadurch gekennzeichnet, dass sie von bestimmten Werkzeugmaschinen in einer festen Reihenfolge bearbeitet wird. Da die Werkstückvariante „eng“ von allen Werkzeugmaschinen bearbeitet wird -zuerst von der Werkzeugmaschine „mill“, danach von der „drill“ und abschießend von der „wash“- ist es von Vorteil, diese Werkstückvariante zu wählen, um dadurch, eine längere Animation zu generieren.

Der Konstruktor dieser Klasse liefert nur Werkstücke vom Typ „eng“. Diese Klasse stellt auch die *String toString()*-Methode, die die String-Repräsentation eines Werkstücks liefert, sowie den Vergleich zweier String-Objekte *boolean equals(Object other)* zur Verfügung. Dies ist im UML-Klassendiagramm in Abbildung 6.2 zu sehen.

6.2.2 Der Kommunikationsablauf über die Socket-Schnittstelle

Das Sequenzdiagramm 6.3 zeigt den Kommunikationsablauf über die Socket-Schnittstelle, bei dem die Darstellung eines Transporters innerhalb der Animationsumgebung näher betrachtet wird.

Problematisch bei dieser Visualisierung erwies sich die Repräsentation von *synchronized*. Hierbei handelt es sich um ein Sprachkonstrukt von Java, das keinem Methodenaufruf entspricht. Ohne Repräsentation von *synchronized* würde das Sequenzdiagramm jedoch jegliche Aussagekraft verlieren. Da zu Beginn eines *synchronized*-Blocks ein *Lock* auf das Parameterobjekt angefordert wird, wurde im Sequenzdiagramm 6.3 das Schlüsselwort *synchronized* als expliziter synchroner Methodenaufruf auf dem Parameterobjekt

repräsentiert. Da der Aufruf synchron erfolgen muss und das aufrufende Objekt auf den Erhalt des Locks wartet, wurde dies entsprechend visualisiert. Am Ende des *synchronized*-Block wird der Lock durch einen synchronen Aufruf wieder zurückgegeben.

Die HMS-Implementierung beginnt ihre Ausführung durch den Aufruf der main()-Methode. Damit startet der main-Thread seine Ausführung - wie im Schaubild dargestellt. Der main-Thread ruft den Konstruktor zur Initialisierung eines Objektes der *Hts*-Klasse auf. Innerhalb des Konstruktors werden auch die Komponenten eines Hts: *HtsCtrl*, *Driver* und *Acquisition*, erzeugt. Zur besseren Übersichtlichkeit werden in Sequenzdiagramm diese Aufrufe nicht angezeigt. Folgend wird die *createHts()*-Methode der Klasse*I4dAnimation* aufgerufen. Das Objekt *I4dAnimation* sendet die Nachricht *connect()* an sich selbst. Innerhalb der connect()-Methode wird ein *CommunicationManager*-Objekt erzeugt, welches als Thread realisiert ist. Innerhalb dieses Konstruktors werden ein *Socket* mit Host und Port, sowie ein *PrintWriter* und ein *Reader* für den Socket erzeugt. Anschließend wird durch die *start()*-Methode des CommunicationManager die *run()*-Methode dieses Threads aufgerufen. Dadurch läuft der CommunicationManager-Thread parallel zum main-Thread. Weiterhin ruft der main-Thread die *sendAndReceive(String command, String[] arguments)*-Methode auf, um die „*agvCreate*“-Nachricht zur 3D-Animation zu senden. Da die beiden momentan laufenden Threads die HashMap und das Instruction-Objekt zur Kommunikation über die Socket-Schnittstelle gemeinsam benutzen, sind sie synchronisiert. Zur Synchronisation dieser Threads sowie weiterer, die im Laufe der main-Ausführung erzeugt werden, sind *synchronized*-Blöcke realisiert.

Innerhalb der sendAndReceive(String command, String[] arguments)-Methode wird ein *Instruction*-Objekt erzeugt, welches eine Nachricht bei der Kommunikation über die Socket-Schnittstelle darstellt. Weiterhin fordert der main-Thread den Lock auf das Parameterobjekt der HashMap-Liste an. Da der Lock verfügbar ist, wird er dem main-Thread zugeordnet und ist somit für andere Threads nicht mehr verfügbar. Dies ist im Sequenzdiagramm als Rückgabe vom Liste-Objekt angezeigt. Der main-Thread ruft innerhalb dieses *synchronized*-Blocks die *put(instruction.id, instruction)*-Methode der HashMap-Liste auf. Somit wird die Nachricht in die Liste gespeichert. Danach gibt der main-Thread den Lock frei. Folgend fordert der main-Thread den Lock auf das Parameterobjekt der Instruction an. Nach Erzeugen des Instruction-Objekts ist dessen *Lock* verfügbar. Daher wird er auch dem main-Thread zugeordnet. Innerhalb dieses kritischen Bereichs wird die *printl(instruction.toString())*-Methode des PrintWriters aufgerufen, um die „agvCreate“-Nachricht zur 3D-Animation zu senden. *wait()* wird von main-Thread danach aufgerufen, um dem CommunicationManager-Thread zu signalisieren, dass er auf die Animationsantwort wartet. Somit gibt der main-Thread diesen Lock frei. Dies bedeutet, dass falls andere Threads warten, einer von ihnen den Lock auf das Parameterobjekt der Instruction erhält.

Der CommunicationManager-Thread kann nun mit der Ausführung beginnen. Seine *run()*-Methode ruft zuerst die *handleAnswer(channelIn.readLine)*-Methode auf. Nachdem die Animationsantwort gelesen wurde, wird sie in seine Argumente aufgeteilt. Der *result[1]*-Argument repräsentiert die Nachricht-Id. Somit kann in der HashMap-Liste das Instruction-Objekt gesucht werden, das zu dieser Animationsantwort gehört. Um dies zu

realisieren, fordert der CommunicationManager-Thread den Lock des HashMap-Objekts an und er bekommt ihn. Die *get(result[1])*-Methode des HashMap-Objekts wird folgend aufgerufen. Sie liefert das Instruction-Objekt mit derselben Id zurück. Danach gibt der CommunicationManager-Thread den Lock des HashMap-Objekts frei. Weiter bekommt der CommunicationManager-Thread den Lock auf dem Instruction-Objekt, wenn eine passende Antwort in der Liste gefunden wird. Hier wird die Animationsantwort in einem *result*-Array festgehalten. Danach wird der main-Thread, der auf diese Animationsantwort wartet, durch *notify()* aufgeweckt. Anschließend gibt der CommunicationManager-Thread den Lock auf dem Instruction-Objekt frei. Innerhalb der run()-Methode werden zum Schluß alle Einträge in der Liste gelöscht. Diese sind aus Übersichtlichkeitsgründen nicht im Sequenzdiagramm dargestellt.

Der main-Thread, der auf die Animationsantwort gewartet hat, kann durch den Aufruf von notify() seine Ausführung weiter durchführen. Dies bezieht sich auf die sendAndReceive()-Methode. Folgend bekommt er den Lock auf dem HashMap-Objekt. Durch den Aufruf der *remove(instuction.id)*-Methode des HashMap-Objekts wird der Eintrag für diese Nachricht-Id aus der Liste gelöscht. Danach gibt der main-Thread den Lock auf diesem Objekt frei. Die sendAndReceive()-Methode liefert abschließend die Animationsantwort an dem Aufrufer zurück. Dies bedeutet, das ein Transporter mit einer eindeutigen Identifikation, einer Position und einer Ausrichtung in der Animationsumgebung erzeugt wurde. Abschließend wird *ichInderAnimation* im Hts-Konstruktor geliefert und somit dieser beendet.

Der Kommunikationsablauf bei der Erzeugung eines Transporters erfolgt hier - wie im Sequenzdiagramm dargestellt - durch die Kooperation dieser beiden Threads. Im Laufe der Ausführung der HMS-Implementierung nutzen mehrere Threads die Socket-Verbindung. Diese Kooperation verschiedener Threads beim Senden und Empfangen der Nachrichten über die Socket-Schnittstelle ist entscheidend für die Lauffähigkeit des gesamten Systems.

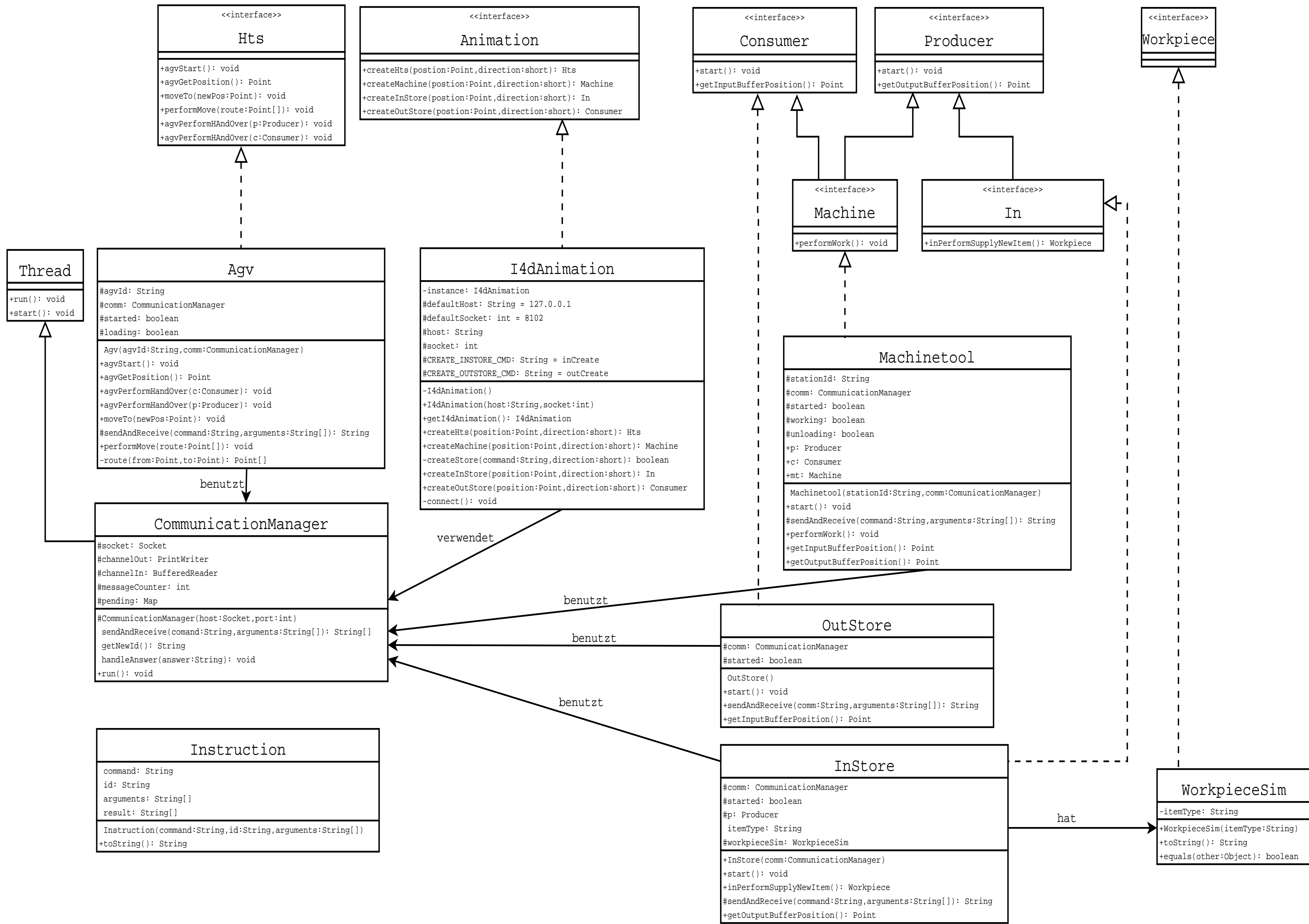

Abbildung 6.2: Das Klassendiagramm der implementierten Schnittstelle

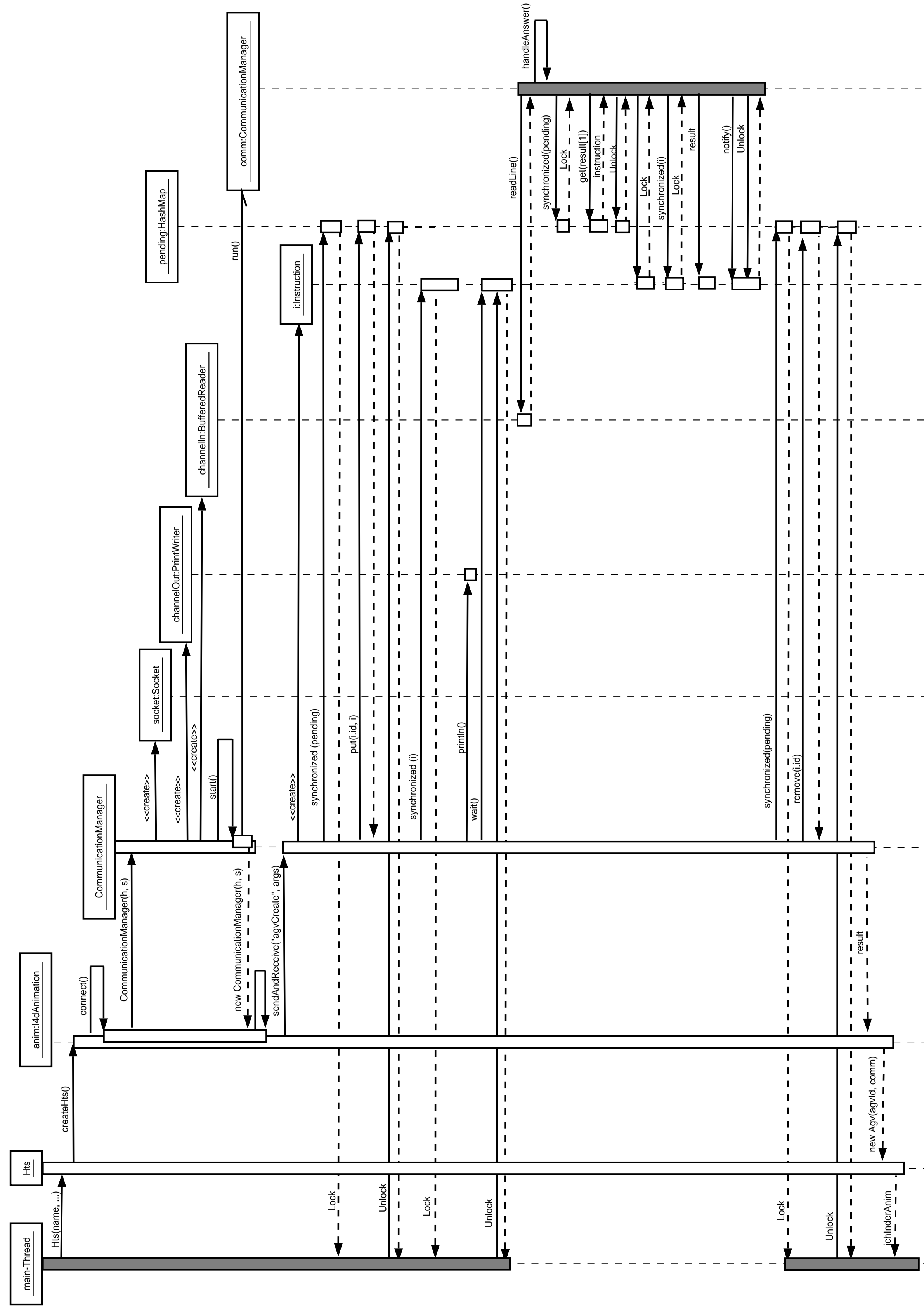

Abbildung 6.3: Der Kommunikationsablauf bei der Erzeugung eines Transporters

6.3 Implementierung der Umsetzung

Da im Entwurf der Umsetzung der abstrakten Schnittstelle alle Klassen näher beschrieben wurden, werden in diesem Abschnitt lediglich einige Details der Implementierung ergänzt. Auf eine ausführliche Beschreibung wird hier verzichtet. Im Anhang dieses Dokuments werden die entwickelten Klassen und Methoden ausführlich beschrieben.

In diesem Teil werden zunächst einige wichtige Aspekte beleuchtet, die bei der Implementierung bedeutend waren. Danach wird die Erweiterung der HMS-Implementierung um die Klasse *Mapping* erläutert. Des weiteren wird auf die Integration der Animationsaufrufe in die HMS-Implementierung eingegangen. Zum Schluss wird die gesamte Architektur des im Rahmen des Projekts ForMooS [3] entstandenen Softwareprodukts dargestellt.

6.3.1 Implementierungsdetails

Bei der Implementierung der Klassen musste zuerst ein Problem gelöst werden. Es ergab sich aus der Tatsache, dass das Dokument [8], das die von der Tcl/Tk 3D-Animationssoftware unterstützten Befehlen beschreibt, teilweise fehlerhaft und unvollständig ist. So ist hier zu erwähnen, dass die Nachrichtennamen einiger Befehle sowie deren Parameter unvollständig sind. Für dieses Problem wurde zwar eine Lösung gefunden, aber es war zeitaufwändig, in den Quelltext der Klasse AnimationControl zu schauen. Diese Klasse ist zusammen mit der Tcl/Tk 3D-Animationssoftware als kleine Simulation der Fertigungssystems verfügbar. Durch Überprüfung des Quelltextes dieser Klasse konnte das Problem gelöst werden.

Auch die im Dokument erklärte Funktionalität einiger Befehle führte zu Missverständnissen. Nur durch intensive Tests war sie nachvollziehbar.

6.3.2 Koordinaten-Transformation: Die Klasse Mapping

Die Klasse *Mapping* wurde aus mehreren Gründen implementiert. Zum Einen verwendet die HMS-Implementierung [13] zur Positionierung der Objekte innerhalb des Fertigungssystems Koordinaten, deren Längeneinheit nicht der der Tcl/Tk 3D-Animationssoftware entspricht. Um eine für die 3D-Animation passende Koordinaten-Transformation durchführen zu können, war es erforderlich, Methoden bereitzustellen, die die Visualisierung der Objekte innerhalb der Animationsumgebung ermöglichen. Aus diesem Grund sind folgende Methoden implementiert worden:

- die Methode *public static Point fromSpecToSim(CoordDef coord)*, die verwendet wird, um die Koordinaten eines Punkts vom Typ CoordDef mit einem Faktor zu multiplizieren. Die Methode liefert einen neuen Punkt, dessen Koordinaten dem durch die Tcl/Tk 3D-Animationssoftware festgelegten Animationsbereich entsprechen und

- die Methode *public static CoordDef fromSimToSpec(Point point)*, mit deren Hilfe im Unterschied zur ersten Methode *public static Point fromSpecToSim(CoordDef coord)* eine umgekehrte Umwandlung durchgeführt wird. Es wird also ein Punkt vom Typ CoordDef geliefert, dessen Koordinaten durch die Division durch einen Faktor entsprechend kleiner sind.

Zum Anderen war es erforderlich, eine Methode zur Verfügung zu stellen, die die Strecken innerhalb der Animationsumgebung festlegt, die die Transporter fahren müssen. In der HMS-Implementierung „weiss“ der Transporter, wo sich die Stationen befinden, aber nicht wie man dorthin kommt. Das war für die HMS-Implementierung irrelevant. Dieser Aspekt tritt jedoch in den Vordergrund, wenn der Fertigungsablauf simuliert werden soll. Daher wurde die Methode *public static Point[] moveTo(Point from, Point to)* implementiert. Sie liefert eine Route mit den Wegpunkten, die ein Transporter anfahren soll, um zur nächsten Station zu kommen. Es muss an dieser Stelle erwähnt werden, dass diese Methode geändert werden muss, wenn es eine andere Positionierung der Stationen gibt.

Die Methode *public static Point[] moveTo(CoordDef fromCoordDef, CoordDef toCoordDef)* wird verwendet, um die Koordinaten der Strecke passend für die 3D-Animation zu transformieren.

6.3.3 Integration der Animationsaufrufe in die HMS-Implementierung

In diesem Kapitel wird erläutert, wie die vorhandene HMS-Implementierung an die Tcl/Tk 3D-Animation angeschlossen wurde. Hierbei werden die notwendigen Änderungen an den Methoden der HMS-Implementierung beschrieben. Dabei werden alle geänderten Methoden in verschiedenen Klassen kurz besprochen.

Änderungen an der HMS-Implementierung

Die Anbindung der HMS-Implementierung an die Tcl/Tk 3D-Animation erfolgte über die in diesem individuellen Projekt entwickelte Schnittstelle. Zu diesem Zweck wurden während der Integration der Animationsaufrufe, mehrere Tests durchgeführt.

Die Erweiterung der HMS-Implementierung um Animationsaufrufe wurde schrittweise durchgeführt. Dies bezieht sich auf die Reihenfolge des Ablaufs innerhalb des Fertigungssystems. Im Folgenden wird die schrittweise durchgeführte Anbindung beschrieben:

- Zuerst mussten die Komponenten des Fertigungssystems innerhalb der Animationsumgebung dargestellt werden. Diese sind die drei Transporter, die den Materialfluss realisieren und die fünf Stationen. Die Stationen umfassen - wie im Abschnitt 2.1 beschrieben - ein Eingangslager InStore, drei Werkzeugmaschinen „mill“, „drill“

und „wash“ und ein Ausgangslager OutStore. Um diese Objekte innerhalb der Animationsumgebung darzustellen, ist jeder Konstruktor dieser zu animierenden Objekte um der Parameter *anim* erweitert worden. Dies hat zur Folge, dass beim ersten Aufruf eines dieser Konstruktoren ein Animation-Objekt erzeugt wird. Auf diese Art und Weise wird eine globale Zugriffsmöglichkeit auf dieses Objekt zur Verfügung gestellt. Dazu wurden die *create*-Methoden der Klasse *I4dAnimation* verwendet, um ein in der Animationsumgebung entsprechendes Objekt zurückzugeben.

- Nach der Initialisierung der Transporter und Stationen werden die Stationen mittels der *start()*-Methode gestartet. Dieser Aufruf erfolgt auch im Konstruktor der Stationen. Im Konstruktor des Materiallagers InStore wurde zusätzlich die *public Workpiece inPerformSupplyNewItem()*-Methode integriert, mit deren Hilfe ein neues Werkstück im Eingangslager InStore erscheint.
- Zu Beginn der Animation befinden sich die Transporter in ihrer Initialposition und warten auf Aufträge. Die initiale Position eines Transporters innerhalb der Animationsumgebung kann mittels der *fromSpecToSim(CoordDef coord)*-Methode der Klasse *Mapping* ermittelt werden. Wie im Abschnitt 2.2 erläutert wurde, erfolgt die Auswahl eines Transporters aufgrund des Angebots, das dieser schickt. Hat ein Transporter einen Auftrag erhalten, muss er zu der entsprechenden Station fahren, um ein Werkstück abholen. Dies wird durch den Aufruf von *performMove(Point[] route)* innerhalb der *drive()*-Methode der Klasse *Driver* erreicht. Das Abholen eines Werkstücks erfolgt mittels des Aufrufs von *agvPerformHandOver(Producer p)*. Sie wird innerhalb der *loadMachineHts(MachineSpec ma, WorkpieceSpec workpiece)*-Methode der Klasse *HtsCtrl* integriert.
- Das abgeholte Werkstück wird zu einer Werkzeugmaschine zur Bearbeitung transportiert. Dies wird auch durch den Aufruf von performMove(Point[] route) realisiert, wie oben beschrieben. Danach meldet sich der Transporter erneut an den Stationen und wartet auf Aufträge.
- Das Entladen des Werkstücks am Eingangspuffer der Werkzeugmaschinen und des Materiallagers OutStore erfolgt durch den Aufruf von *agvPerformHandOver(Consumer c)*. Diese Methode wird innerhalb der *loadHtsMachine(MachineSpec ma)*-Methode der Klasse *HtsCtrl* integriert.
- Die *performWork()*-Methode veranlasst das Holen eines Werkstücks aus dem Eingangspuffer einer Werkzeugmaschine und dessen Bearbeitung. Diese Methode wurde innerhalb der *process()*-Methode der Klasse *Wzm* integriert.

6.3.4 Dokumentation: JavaDoc

Um das entwickelte Java API jedem Leser dieses Dokuments zur Verfügung zu stellen, wurde ein TeXDoclet [9] verwendet. Dieses Programm ermöglicht es, aus der Doku-

mentation des Java-Quelltexts automatisch LaTeX-Code zu erstellen. Hierzu muss die Quelltext-Dokumentation den JavaDoc-Richtlinien genügen.

6.4 Überblick über die gesamte Architektur

Das entwickelte Softwareprodukt verteilt sich auf verschiedene Pakete, dargestellt in Abbildung 6.4.

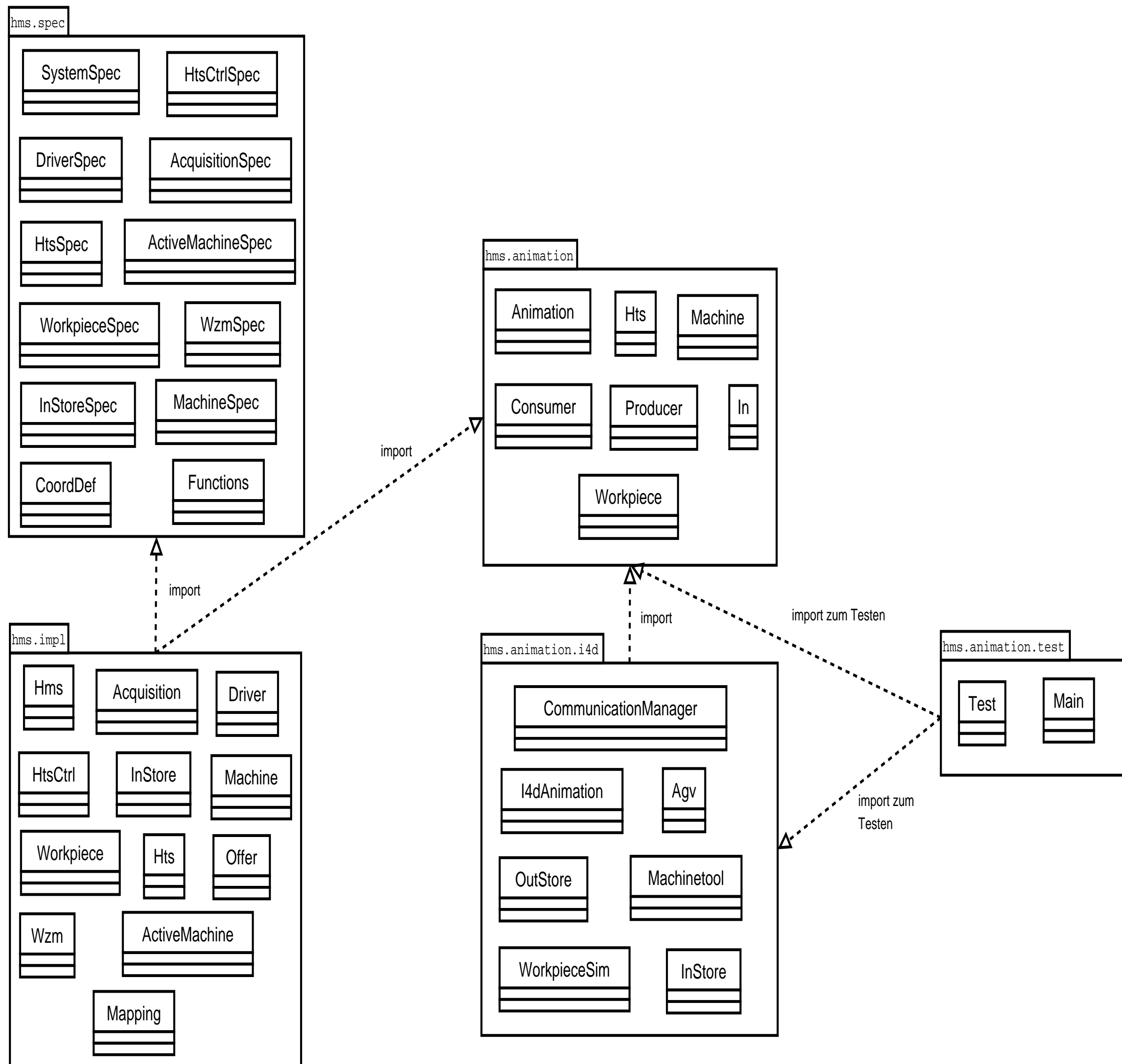

Abbildung 6.4: Überblick über die gesamte Architektur

Abbildung 6.4 zeigt das Paket-Diagramm dieses Softwareprodukts. Die Systemspezifikation des HMS *hms.spec* und seine Implementierung *hms.impl* waren bereits zu Anfang

dieses Individuellen Projekts vorhanden.

Die Pakete *hms.animation*, *hms.animation.i4d* und *hms.animation.test* sind durch im Rahmen dieses Individuelle Projekts entstanden.

Das Paket hms.spec: In diesem Paket befinden sich mehrere Java-Interfaces. Sie sind auf Basis der CSP-OZ-Spezifikationssprache entstanden und stellen alle nötigen Spezifikationsaspekte für das Paket *hms.impl* zur Verfügung.

CSP-OZ erlaubt es, Kommunikationsereignisse (im CSP-Teil) und Datenaspekte (im Object-Z-Teil) zu spezifizieren. Durch die Java Modeling Language JML ergänzt um CSP_{jassda}-Spezifikationen kommen weitere benötigte Spezifikationselemente hinzu. So konnten u.a. Klassen zur Strukturierung, Vererbungsoperatoren, Konzepte der abstrakten Datentypen und Zustandsinvarianten zur Spezifikation verwendet werden. In Abbildung 6.4 sind aus Gründen der Übersichtlichkeit nur die relevanten Klassen dargestellt.

Das Paket hms.impl: Ein UML-Klassendiagramm der wichtigsten Klassen dieses Pakets befindet sich in Abbildung 6.1. Die Klassen der HMS-Implementierung implementieren die in *hms.spec* spezifizierenten Klassen.

Das Paket hms.animation: Das Paket *hms.animation* wurde innerhalb dieses Individuellen Projekts neu erstellt. Die Java-Interfaces dieses Paketes sind in Abbildung 5.1 näher erläutert.

Das Paket hms.animation.i4d: Das Paket *hms.animation.i4d* wurde ebenfalls im Rahmen dieses Individuellen Projekts neu erstellt.

Das Paket hms.animation.test: In diesem Paket befinden sich die Testdaten. Die beiden Klassen *Test* und *Main* müssen einige Klassen aus dem Paket *hms.impl* und dem Paket *hms.spec* importieren. Dies wurde aus Übersichtlichkeitsgründen nicht im Paket-Diagramm dargestellt.

7 Test der Schnittstelle

In diesem Teil wird erläutert, wie die Funktionalität der Implementierung der Schnittstelle getestet wurde.

Während der Entwicklung wurde die Funktionalität der Implementierung der Schnittstelle getestet. Das Vorgehen beim Testen konzentrierte sich darauf, Abweichungen zwischen der Funktionalität dieser Schnittstelle und ihrer Anforderungen zu identifizieren. Zu diesem Zweck wurde die Funktionalität der HMS-Implementierung mittels der integrierten Animationaufrufe sowie der vorhandenen Tcl/Tk 3D-Animation getestet.

7.1 Bewertung des Ansatzes: Test

Die Funktionalität der einzelnen Teile der Anbindung der HMS-Implementierung an die Tcl/Tk 3D-Animatonssoftware wurde schon begleitend zum Entwurf und zur Implementierung getestet.

Der Realisierungsansatz erwies sich beim Testen als adäquat. Die im Kapitel 3 festgelegten Anforderungen wurden durch diesen Ansatz erfüllt.

Es konnte festgestellt werden, dass obwohl der implementierte Kontrollfluss dieser nebenläufigen Ablaufsteuerung während des Fertigungsablaufs explizit festgelegt wurde, dennoch in den Hintergrund trat. Hierbei obliegt es ausschließlich dem Betriebsystem, welches tatsächlich die Prozessorzeit zuteilt, in welcher Reihenfolge die einzelnen Anweisungen ausgeführt werden und nicht wie die vorhandene Implementierung es vorgibt.

Zu welchem Zeitpunkt zwischen den Threads umgeschaltet wird, hängt also von dem prozessorzeitzuteilenden Betriebssystem, der Hardware und möglicherweise auch von der Version des JDKs und von anderen Faktoren ab. Deswegen lässt sich während der Animation der HMS-Implementierung eine mangelnde Abstimmung bei der Koordination der Transporter feststellen. Dadurch treten Kollisionen zwischen den Transportern auf, die nicht zu vermeiden sind, ohne eine Überwachung der Strecken innerhalb der Fertigungsumgebung. Dies lässt sich nur durch den Einsatz eines Blockstrecken-Management-Systems realisieren, da es die Funktionalität der HMS-Implementierung benutzt.

7.1.1 Vorteile beim Testen

Die zur Verfügung gestellte Tcl/Tk 3D-Animation spielte beim Testen der Implementierung eine wichtige Rolle, da sie die Steuerung des Fertigungsablaufs visualisieren kann.

So konnten einige Fehler bei der Implementierung behoben werden:

- Dank des Ultraschallmessgerätes (zur Abstandsmessung gegenüber anderen Objekten) mit dem die Transporter aufgerüstet sind, konnten Kollisionen zwischen den Transportern und den Werkzeugmaschinen innerhalb der Animationsumgebung erkannt werden. Diese wurden dann entsprechend behoben.
- Fehler bei der Steuerung der Verarbeitung von Werkstücken durch die Werkzeugmaschinen ließen sich mittels der 3D-Animationssoftware erkennen, dank der unterstützenden Visuallisierung (durch temporäre Farbänderungen) der Bearbeitung. All diese Animationen während des Fertigungsablaufs erwiesen sich als vorteilhaft beim Testen.
- Bei der Entwicklung der von den Transportern anzufahrenden Route erwies sich die Tcl/Tk 3D-Animation auch als vorteilhaft.

7.1.2 Nachteile beim Testen

Es konnten folgende Nachteile beim Testen festgestellt werden:

- Zuerst konnte beim Testen festgestellt werden, dass das Materiallager AUS (OutStore) nur drei bearbeitete Werkstücke aufnehmen kann, obwohl in der Schnittstellendefinition zur 3D-Animationssoftware eine solche Kapazitätsbeschränkung des OutStores nicht spezifiziert ist. Aufgrund dieser mangelden Funktionalität der Tcl/Tk 3D-Animationssoftware konnte der Tages-Soll nicht erreicht werden und somit ist keine Beendigung der Fertigung möglich.
- Ein anderer Nachteil ist, dass das zur 3D-Animationssoftware gehörende Dokument mehrere falsche Angaben zur Funktionalität der Animationsbefehlen enthält. Im Dokument ist zum Beispiel präzisiert, dass ein Transporter bei der Ausführung der Befehle „agvPerformMoveAbs“ und „agvPerformMoveRel“ vorzeitig anhält, wenn eine Kollision mit anderen Objekten droht (anhand der Ultraschallsensoren). So kann laut dieses Dokuments eine Kollision vermieden werden und die Statusmeldung „stoppedMoving“ wird zurückgegeben. Diese Statusmeldung wird aber nur bei einer Kollision mit einer Station zurückgegeben. Wenn Kollisionen mit anderen HTS drohen, kann das fahrende HTS nicht angehaltet werden und solche Kollisionen wurden beim jeden Test festgestellt. Trotz unzähliger Kollisionen kann die Animation des Fertigungssystems weiter durchgeführt werden. Hierbei gab die Animationssoftware nur die Statusmeldung „collisionDetected“ zurück und nicht „stoppedMoving“ wie im Dokument beschrieben ist.
- Im Dokument ist spezifiziert, dass der verwendete Bereich der Animationsumgebung durch die Koordinaten (0,0) und (4000, 4000) bestimmt wird. Tatsächlich konnte festgestellt werden, dass dies nicht zutrifft. Wenn man z.B. die Koordinaten (250,250) wählt, konnte kein Animationsobjekt dargestellt werden. Genauso

konnten keine Objekte dargestellt werden, deren Koordinaten sich in der Nähe der oberen Schranke wie z.B. (3750,3750) befinden.

8 Entwicklungsumgebung

Die eingesetzten Werkzeugen für die Entwicklung werden in diesem Kapitel gennant und ihre konkrete Verwendung kurz erläutert.

JCreator

JCreator [5] ist eine vollständige integrierte Entwicklungsumgebung (IDE, Integrated Development Environment). Sie enthält mehrere Werkzeuge wie: Editor, Compiler, Debugger, etc., Ant- und CVS-Unterstützung sowie andere Features.

JCreator kam während dieses Individuellen Projekts zur Programmierung zum Einsatz. Es wurde verwendet, da es dem Entwickler eine komfortable Arbeitsumgebung durch Eigenschaften wie die automatische Vervollständigung von Codefragmenten und die einfache Suche im gesamten Projekt bietet.

CVS

CVS steht für Concurrent Versions System. Es bezeichnet ein Software-Programm zur Versionsverwaltung von Quelltext. Der primäre Verwendungszweck war die Sicherstellung der Verfügbarkeit von Quellcode an verschiedenen Orten [1].

Zum Zugriff auf den CVS-Server wurde TortoiseCVS [11] eingesetzt. Dabei handelt es sich um ein CVS-Plugin für den Microsoft Windows Explorer, mit dem komfortabel auf einen CVS-Server zugegriffen werden kann. Besonders vorteihaft beim diesen Werkzeug ist, dass alle gängigen CVS-Aktionen per Mausclick auf Dateien und Verzeichnisse direkt aus dem Windows Commander heraus ausgeführt werden konnten. Der Status der Dateien ist durch die veränderten Datei–Icons ersichtlich und somit hat man einen besseren Überblick über die veränderten Dateien.

Dia

Dia [2] ist ein frei verfügbares Diagrammerstellungsprogramm, das unter der GPL (GNU General Public License) veröffentlicht wird. Dia bietet einfache Grundelemente verschiedener Typen an, mit denen komfortabel Diagramme erstellt werden können. Dia wurde bei der Erstellung von UML Diagrammen verwendet. Das Wergzeug bietet die Möglichkeit, die erstellten Diagrammen in .eps zu exportieren. Somit werden die Bilder direkt in LaTeXeingebunden.

MikTex

MikTeX ist eine TeX-Distribution für Microsoft Windows. Sie enthält alle zur Arbeit mit dem Textsatzprogramm TeX nötigen Programme [6].

Zur Erstellung der Dokumentation wurde das von MikTeX mitgelieferte *pdflatex* genutzt.

The GIMP

Zur Erstellung und Bearbeitung sonstiger Bilder wurde The GIMP [10] eingesetzt.

The GIMP (The GNU Image Manipulation Program) ist ein Bildbearbeitungsprogramm. Der Funktionsumfang zur Bildbearbeitung ist vergleichbar mit dem professioneller kommerzieller Programme.

TeXDoclet

Die im Anhang befindete Beschreibung des Java API wurde mit einen TeXDoclet[9] erstellt.

Ein TeXDoclet ist eine Programm, das die Java Kommentare des Quelltextes in LaTeX-Code erstellt.

9 Schlussbemerkungen

In diesem Abschnitt sollen zum Abschluss dieser Arbeit ein Fazit des Individuellen Projekts gezogen sowie ein kurzer Ausblick über weitere Möglichkeiten gegeben werden.

9.1 Fazit

Im Rahmen dieses Individuellen Projekts wurde die vorhandene Java-Implementierung des HMS an die im DFG-Schwerpunktprogramm entwickelte 3D-Animation angeschlossen. Die auf Basis der im Projekt ForMooS CSP-OZ Systemspezifikation realisierte HMS-Implementierung kann jetzt mittels Tcl/Tk 3D-Animation animiert werden.

Die Aufgabenstellung, die vorhandene Java-Implementierung des HMS an die im DFG-Schwerpunktprogramm entwickelte Tcl/Tk 3D-Animation angeschlossen zu werden, hat sich als machbar herausgestellt.

Schwächen hinsichtlich der zur Implementierung notwendigen Schnittstellendefinition zur Tcl/Tk 3D-Animation [8] sind festzuhalten. Diese hatten zur Folge, dass die dort spezifizierten Befehle unvollständig waren und nicht ausgeführt werden konnten. Die 3D-Animationssoftware gab auf solche eingehende Nachrichten Fehlerantworten zurück und das Suchen der Fehler hat bei der Implementierung Zeit in Anspruch genommen.

Im Nachhinein lässt sich feststellen, dass die Einhaltung des anfanglich festgelegten Abgabetermins des Individuellen Projekts sich als schwierig erwies. Leider traten familiäre Probleme auf, die eine Verschiebung der selbstgestellten Zeitpläne zur Folge hatten. So konnte festgestellt werden, dass die Einhaltung eines Meilensteinplans von großer Bedeutung ist. Wird ein Termin einmal überschritten, wird die Einhaltung auch aller weiteren Meilensteine erschwert.

Eine am Anfang angedachte Erweiterung der Funktionalität der Schnittstelle konnte aufgrund von oben beschriebenen Gründen nicht mehr erfolgen.

Alles in allem ist das Individuelle Projekt eine gute Vorbereitung auf selbstständiges Arbeiten. Mehr noch als in einer Projektgruppe ist man dazu gezwungen, Probleme selbstständig in Angriff zu nehmen und zu lösen sowie eigene Zeitpläne zu erstellen. Auf das Anfertigen einer Diplomarbeit kann sich das hier Gelernte nur positiv auswirken.

Betreuung

Dieses Individuelle Projekt wurde durch Michael Möller betreut. Er hielt sich nicht nur in wöchentlichen Treffen über dem aktuellen Stand des Projektes auf dem Laufenden, sondern stand auch in der übrigen Zeit unterstützend zur Verfügung.

9.2 Ausblick

Die durch dieses Individuelle Projekt erstellte Schnittstellenfunktionalität ist in sich weitestgehend abgeschlossen. In ihrer jetzigen Form ist sie aber bereits gut nutzbar. Erweiterungen wären in diesem Bereich noch denkbar:

- Momentan kann die HMS-Implementierung animiert werden. Die Schnittstelle kann um zusätzliche Funktionalität erweitert werden, die nicht in der HMS-Implementierung benutzt wird. In der Ausbaustufe [7] sind zusätzlich eine Batterieladestation (engl.: Battery station, BS) und ein Parkbereich (engl.: Garage, G) vorgesehen, die in der HMS-Implementierung nicht implementiert sind. Hierbei muss erwähnt werden, dass in diesem Fall eine Erweiterung der abstrakten Schnittstelle um einige Methoden sowie die Implementierung dieser Methoden für die Tcl/Tk 3D-Animation notwendig sind, wie in Abbildung 4.1 dargestellt ist.
- Schon bei der Verwendung von drei Transportern kann nicht mehr von Kollisionsfreiheit der Transporter ausgegangen werden. Um Kollisionen zu vermeiden, gäbe es die Möglichkeit, ein Gitternetz mit Blockstreken einzubringen, so wie in der Ausbaustufe [7] des Fertigungssystems beschrieben ist.
- Es wäre denkbar, dass die in der Spezifikation festgelegte Konfiguration (vor allem die Position der Stationen und Materiallager in der Animationsumgebung) durch entsprechende Änderungen flexible ausgewählt werden könnte. Diese Erweiterung würde jedoch eine größere Änderung der Systemspezifikation und der bestehenden Funktionen nach sich ziehen.
- Die in der HMS-Implementierung entwickelte Funktionalität könnte auch mittels einer anderen Animationssoftware visuallisiert werden. Interessant wäre eine andere 3D-Animationssoftware zur Verfügung bereitzustellen und die in diesem Individuellen Projekt entwickelte Schnittstelle für diese 3D-Animationssoftware zu implementieren. So könnte die entwickelte Schnittstelle wiederverwendet werden.

Die Schnittstelle kann also in der Zukunft noch in einigen Bereichen erweitert werden. Die HMS-Implementierung des holonischen Fertigungssystems kann dadurch auch erweitert werden.

Literaturverzeichnis

[1] CVS. http://www.cvshome.org/.

[2] Dia. http://www.gnome.org/projects/dia/.

[3] Einbettung einer objekt-orientierten formalen Methode in einen objekt-orientierten Software-Entwicklungsprozess (ForMooS). http://csd.informatik.uni-oldenburg.de/projects/formoos.html.

[4] Integration von Techniken der Software-Spezifikation für ingenieurwissenschaftliche Anwendungen. http://tfs.cs.tu-berlin.de/projekte/indspec/SPP/.

[5] JCreator. http://www.jcreator.com/.

[6] Miktex. http://www.miktex.org/.

[7] Referenzfallstudie Produktionstechnik (PA) v2.0. http://tfs.cs.tu-berlin.de/SPP/RefPAv2.doc.

[8] Schnittstelledefinition zur 3D-Animation eines holonischen Fertigungssystems. http://tfs.cs.tu-berlin.de/SPP/3DSchnittstelle.ps.

[9] TeXDoclet. http://texdoclet.stefanmarx.de/.

[10] The GIMP. http://www.gimp.org/.

[11] Tortoise. http://www.wincvs.org/TortoiseCVS/.

[12] C. Fischer. *Combination and Implementation of Processes and Data: From CSP-OZ to Java*. PhD thesis, Bericht Nr. 2/2000, University of Oldenburg, April 2000.

[13] Michael Möller. Implementierung der HMS-Fallstudie. http://csd.informatik.uni-oldenburg.de/~eagle/hms.zip.

[14] M. Möller, E.-R. Olderog, H. Rasch, and H. Wehrheim. Linking CSP-OZ with UML and Java: A Case Study. In *Integrated Formal Methods*, number 2999 in Lecture Notes in Computer Science, pages 267–286, March 2004.

Abbildungsverzeichnis

2.1 Socket-Schnittstelle der HMS-Animation 11
2.2 Die Kommunikation über die Schnittstelle 12
2.3 Das Fertigungssystem 14
2.4 Das Beladen und Endladen von Werkstücken 14

4.1 Realisierungsansatz der Schnittstelle 20

5.1 Überblick über die verwendeten Java-Interfaces 23

6.1 Das vereinfachte Klassendiagramm der HMS-Implementierung 29
6.2 Das Klassendiagramm der implementierten Schnittstelle 40
6.3 Der Kommunikationsablauf bei der Erzeugung eines Transporters 41
6.4 Überblick über die gesamte Architektur 45

Index

$\LaTeX$, 44, 52

abstrakte Schnittstelle, 20, 54
Abstraktion, 21
Agv, 34, 73
Agv(String, CommunicationManager), 75
agvGetPosition(), 67, 75
agvId, 74
agvPerformHandOver(Consumer), 67, 75
agvPerformHandOver(Producer), 68, 75
agvStart(), 68, 75
Aktoren, 13
animate(), 99
Animation, 22
Animation, 63
API, 28, 44
arguments, 82
Aufgabenstellung, 8
autonom, 7

Beschreibung der Java-Interfaces, 22

c, 93
channelIn, 78
channelOut, 78
comm, 74, 89, 92, 95
command, 82
CommunicationManager, 32, 77
communicationManager, 85
CommunicationManager(String, int), 78
CommunicationManager.Instruction, 82
CommunicationManager.Instruction(String, String, String[]), 83
connect(), 86
Consumer, 24
Consumer, 65
CREATE_INSTORE_CMD, 85
CREATE_OUTSTORE_CMD, 85
createHts(Point, short), 64, 86
createInStore(Point, short), 64, 87
createMachine(Point, short), 65, 87
createOutStore(Point, short), 65, 87
createStore(String, Point, short), 88
CSP_{jassda}, 7
CSP-OZ, 7, 46
CVS, 51

Das Paket hms.animation
  Klassen, 46
Das Paket hms.animation.i4d
  Klassen, 46
Das Paket hms.animation.test
  Klassen, 46
Das Paket hms.impl
  Klassen, 46
Das Paket hms.spec
  Klassen, 46
Deadlock, 28
defaultHost, 85
defaultSocket, 85
dezentrale Steuerung, 7, 9
DFG SPP SoftSpec, 7
Dia, 51
Die gesamte Architektur, 45
Distribution, 52

Entwurf der abstrakten Schnittstelle, 21
Entwurf der Umsetzung, 32
Entwurfsentscheidungen, 21

equals(Object), 98

flexible, 7
Funktionale Anforderungen, 15

getI4dAnimation(), 88
getInputBufferPosition(), 66, 93, 96
getNewId(), 79
getOutputBufferPosition(), 71, 90, 93
getStationId(), 94
GPL, 51

handleAnswer(String), 79
HMS, 9
HMS-Implementierung, 10, 20, 31, 32
HMS-Implementierungsaspekte, 28
Holonische Fertigungssysteme, 7, 54
host, 85
Hts, 24
Hts, 66
hts, 100

I4dAnimation, 34, 83
I4dAnimation(), 86
I4dAnimation(String, int), 86
ID, 32, 35
id, 82
Implementierung der Umsetzung, 42
In, 24
In, 68
inPerformSupplyNewItem(), 69, 90
instance, 85
InStore, 35, 88
InStore(CommunicationManager), 90
itemType, 89, 93, 97

Java, 7
Java-Interfaces, 19, 27
  Bestandteile, 22
Java-Threads, 27
JavaDoc, 17
JCreator, 51
JML, 7, 46

Kapselung der Klassen, 19
Klasse
  Workpiece, 30
Klassen
  Acquisition, 30
  ActiveMachine, 29
  Driver, 30
  Hms, 30
  Hts, 29
  HtsCtrl, 30
  InStore, 29
  Machine, 28
  OutStore, 29
  Wzm, 29
Kommunikation, 7, 27, 32
Kommunikation mehrerer Threads, 27
kooperativ, 7

loading, 74

Machine, 24
Machine, 69
Machinetool, 36, 91
Machinetool(String, CommunicationManager), 93
Main, 98
Main(), 99
main(String[]), 99
Main.HtsControl, 99
Main.HtsControl(Animation, Point[]), 100
messageCounter, 78
MikTex, 52
Monitorkonzept, 27
moveTo(Point), 68, 76
mt, 93
MT_TYPES, 85

nebenläufige Kommunikation, 27
nextMtType, 85
nextPointPos, 74

OutStore, 36, 94
OutStore(CommunicationManager), 96

p, 90, 93
Parallelität, 7
pending, 78

performMove(Point[]), 68, 76
performWork(), 70, 94
points, 100
Polymorphie, 21
Producer, 23
Producer, 71
Produktionsautomatisierung, 7
Programmiersprache, 16

Qualität, 16
Quelltext, 17, 45, 51, 52

reaktive Systeme, 7
Realisierungsansatz, 19
RequestReply-Konstrukt, 32
result, 82
route(Point, Point), 76
run(), 79, 100
Runnable, 27

sendAndReceive(String, String[]), 76, 79, 91, 94, 96
Sensoren, 13
SEPARATOR, 83
socket, 78, 85
Socket-Verbindung, 11, 31, 33
start(), 66, 71, 91, 94, 96
started, 74, 89, 92, 95
stationId, 74, 92
Synchronisation paralleler Threads, 27

Tcl/Tk, 16
Tcl/Tk 3D-Animation, 17, 32, 47, 53
Tcl/Tk 3D-Animationssoftware, 11, 31
TeX, 52
TeXDoclet, 44, 52
The GIMP, 52
Thread-Klasse, 27
Threads, 27
Tortoise, 51
toString(), 83, 98

UML, 7, 21
Umsetzungsentwurf, 30, 31
unloading, 74, 92

Verbindungsaufbau, 33

Werkstückvarianten, 37
working, 92
Workpiece, 25
Workpiece, 72
WorkpieceSim, 37, 97
workpieceSim, 90
WorkpieceSim(String), 97

A Java API Dokumentation

A.1 Package hms.animation

Package Contents *Page*

Interfaces

Animation 63

Das Interface zum Erzeugen von Objekten in der Animationsumgebung.

Consumer 65

Das Interface zur Repräsentation von Stationen mit einem Eingangspuffer.

Hts 66

Das Interface stellt die Schnittstellendefinition für Transporter bereit.

In 68

Das Interface In ist vom Interface Producer abgeleitet.

Machine 69

Das Interface Machine ist von den Interfaces Producer und Consumer abgeleitet.

Producer 71

Das Interface zur Repräsentation von Stationen mit einem Ausgangspuffer.

Workpiece 72

Das Interface Workpiece zur Visualisierung von Werkstücke als Objekte in der Animationsumgebung.

A.1.1 Interface Animation

Das Interface zum Erzeugen von Objekten in der Animationsumgebung. Das Interface Animation kann von verschiedenen Animationsklassen implementiert werden. Das Interface legt mehrere Typen von Objekten fest, die zum Fertigungssystem gehören, damit alle Klassen, die das Interface implementieren, diese Typen von Objekten verwenden können. Diese Schnittstelle dient zur Ausführung der Darstellung diverser Objekten in einer (beliebigen) Animationsumgebung und gibt Auskunft über die erfolgreiche oder misslungene Ausführung dieser Darstellung.

Declaration

public interface Animation

All known subinterfaces

I4dAnimation (in A.2.4, page 83)

All classes known to implement interface

I4dAnimation (in A.2.4, page 83)

Method summary

createHts(Point, short) Mit dieser Methode kann ein Transporter als Objekt in einer Animation visualisiert werden.
createInStore(Point, short) Mit dieser Methode kann eine Werkzeugmaschine als Objekt in einer Animation visualisiert werden.
createMachine(Point, short) Mit dieser Methode kann eine Werkzeugmaschine als Objekt in einer Animation visualisiert werden.
createOutStore(Point, short) Mit dieser Methode kann ein Materiallager OutStore als Objekt in einer Animation visualisiert werden.

Methods

- **createHts**
  `Hts` **createHts(** `java.awt.Point` **postion**, `short` **direction**)
  - **Description**
    Mit dieser Methode kann ein Transporter als Objekt in einer Animation visualisiert werden.
  - **Parameters**
    * `postion` – die Position eines Transporters.
    * `direction` – die Ausrichtung eines Transporters.
  - **Returns** – liefert ein Transporter mit einer Position und einer Ausrichtung.
- **createInStore**
  `In` **createInStore(** `java.awt.Point` **postion**, `short` **direction**)
  - **Description**
    Mit dieser Methode kann eine Werkzeugmaschine als Objekt in einer Animation visualisiert werden.

- **Parameters**
  - `postion` – die Position eines Materiallagers InStore.
  - `direction` – die Ausrichtung eines Materiallagers InStore.
- **Returns** – liefert ein Materiallager InStore mit einer Position und einer Ausrichtung.

• **createMachine**
`Machine` **createMachine**`( java.awt.Point` **postion**`, short` **direction** `)`

- **Description**
  Mit dieser Methode kann eine Werkzeugmaschine als Objekt in einer Animation visualisiert werden.
- **Parameters**
  - `postion` – die Position einer Werkzeugmaschine.
  - `direction` – die Ausrichtung einer Werkzeugmaschine.
- **Returns** – liefert eine Werkzeugmaschine mit einer Position und einer Ausrichtung.

• **createOutStore**
`Consumer` **createOutStore**`( java.awt.Point` **postion**`, short` **direction** `)`

- **Description**
  Mit dieser Methode kann ein Materiallager OutStore als Objekt in einer Animation visualisiert werden.
- **Parameters**
  - `postion` – die Position eines Materiallagers OutStore.
  - `direction` – die Ausrichtung eines Materiallagers OutStore.
- **Returns** – liefert ein Materiallager OutStore mit einer Position und einer Ausrichtung.

A.1.2 Interface Consumer

Das Interface zur Repräsentation von Stationen mit einem Eingangspuffer. Unter dem Begriff Consumer sind alle Stationen in der Animationsumgebung zusammengefasst, die einen Eingangspuffer haben, also das Materiallager OutStore und die Werkzeugmaschinen (machine tools). Eine implementierende Klasse dieses Interface liefert Methoden zum Starten eines Consumers sowie zum Abfragen der Anlaufposition am Eingangspuffer eines Consumers.

Declaration

public interface Consumer

All known subinterfaces

Machine (in A.1.5, page 69), OutStore (in A.2.7, page 94), Machinetool (in A.2.6, page 91)

All classes known to implement interface

OutStore (in A.2.7, page 94)

Method summary

getInputBufferPosition() Mit dieser Methode lässt sich die Anlaufposition am Eingangspuffer eines Consumers abfragen.
start() Mit dieser Methode kann ein Consumer innerhalb der Animationsumgebung gestartet werden.

Methods

- **getInputBufferPosition**
  `java.awt.Point` **getInputBufferPosition**`( )`
  - **Description**
    Mit dieser Methode lässt sich die Anlaufposition am Eingangspuffer eines Consumers abfragen.
  - **Returns** – liefert die Anlaufposition am Eingangspuffer eines Consumers.
- **start**
  `void` **start**`( )`
  - **Description**
    Mit dieser Methode kann ein Consumer innerhalb der Animationsumgebung gestartet werden.

A.1.3 Interface Hts

Das Interface stellt die Schnittstellendefinition für Transporter bereit. Es definiert die nötigen Methoden, um die Bewegung eines Transporters in der Animationsumgebung durchführen zu können.

Declaration

public interface Hts

All known subinterfaces

Agv (in A.2.1, page 73)

All classes known to implement interface

Agv (in A.2.1, page 73)

Method summary

agvGetPosition() Mit dieser Methode kann die aktuelle Position eines Transporters in der Animationsumgebung abgefragt werden.
agvPerformHandOver(Consumer) Diese Methode veranlasst das Beladen einer Station vom Typ Consumer.
agvPerformHandOver(Producer) Diese Methode veranlasst das Entladen einer Station vom Typ Producer.
agvStart() Mit dieser Methode kann ein Transporter innerhalb der Animationsumgebung gestartet werden.
moveTo(Point) Diese Methode bekommt als Parameter den neuen Wegpunkt, den ein Transporter anfahren soll.
performMove(Point[]) Diese Methode bekommt als Parameter die Wegpunkte einer Strecke, die ein Transporter anfahren soll.

Methods

- **agvGetPosition**
  `java.awt.Point` **agvGetPosition**`( )`
  - **Description**
    Mit dieser Methode kann die aktuelle Position eines Transporters in der Animationsumgebung abgefragt werden.
  - **Returns** – die aktuelle Position eines Transporters in der Animationsumgebung.
- **agvPerformHandOver**
  `void` **agvPerformHandOver**`( Consumer` **c** `)`
  - **Description**
    Diese Methode veranlasst das Beladen einer Station vom Typ Consumer.

- **Parameters**
  - `c` – eine Station vom Typ Consumer.

- **agvPerformHandOver**
  `void` **agvPerformHandOver**`( Producer` **p** `)`
  - **Description**
    Diese Methode veranlasst das Entladen einer Station vom Typ Producer.
  - **Parameters**
    - `p` – eine Station vom Typ Producer.

- **agvStart**
  `void` **agvStart**`( )`
  - **Description**
    Mit dieser Methode kann ein Transporter innerhalb der Animationsumgebung gestartet werden.

- **moveTo**
  `void` **moveTo**`( java.awt.Point` **newPos** `)`
  - **Description**
    Diese Methode bekommt als Parameter den neuen Wegpunkt, den ein Transporter anfahren soll.
  - **Parameters**
    - `newPos` – den neuen Wegpunkt, den ein Transporter anfahren soll.

- **performMove**
  `void` **performMove**`( java.awt.Point[]` **route** `)`
  - **Description**
    Diese Methode bekommt als Parameter die Wegpunkte einer Strecke, die ein Transporter anfahren soll.
  - **Parameters**
    - `route` – ein Array mit den Wegpunkten einer Strecke, die ein Transporter anfahren soll.

A.1.4 Interface In

Das Interface In ist vom Interface Producer abgeleitet. Es erbt alle Methodendefinitionen des Basis-Interfaces Producer und erweitert dieses um die Methode inPerformSupplyNewItem().

Declaration

public interface In
extends Producer

All known subinterfaces

InStore (in A.2.5, page 88)

All classes known to implement interface

InStore (in A.2.5, page 88)

Method summary

inPerformSupplyNewItem() Diese Methode liefert ein neues Werkstück im Eingangslager InStore.

Methods

- **inPerformSupplyNewItem**
  `Workpiece` **inPerformSupplyNewItem( )**
  - **Description**
    Diese Methode liefert ein neues Werkstück im Eingangslager InStore.
  - **Returns** – liefert ein Werkstück am Eingangslager des InStore.

Members inherited from class `hms.animation.Producer` (in A.1.6, page 71)

- `public Point` **getOutputBufferPosition( )**
- `public void` **start( )**

A.1.5 Interface Machine

Das Interface Machine ist von den Interfaces Producer und Consumer abgeleitet. Es definiert die Eigenschaften von Werkzeugmaschinen. Diese haben einen Ausgangspuffer und einen Eingangspuffer. Aus diesem Grund vereinigt das Interface alle Methodendefinitionen der oben genannten Basis-Interfaces und stellt dazu die Funktion performWork() zur Verfügung.

Declaration

public interface Machine
extends Producer, Consumer

All known subinterfaces

Machinetool (in A.2.6, page 91)

All classes known to implement interface

Machinetool (in A.2.6, page 91)

Method summary

performWork() Diese Methode veranlasst das Holen eines Werkstücks aus dem Eingangspuffer und dessen Bearbeitung.

Methods

- **performWork**
  `void` **performWork**`( )`
  - **Description**
    Diese Methode veranlasst das Holen eines Werkstücks aus dem Eingangspuffer und dessen Bearbeitung.

Members inherited from class `hms.animation.Consumer` (in A.1.2, page 65)

- `public Point` **getInputBufferPosition**`( )`
- `public void` **start**`( )`

Members inherited from class `hms.animation.Producer` (in A.1.6, page 71)

- `public Point` **getOutputBufferPosition**`( )`
- `public void` **start**`( )`

A.1.6 Interface Producer

Das Interface zur Repräsentation von Stationen mit einem Ausgangspuffer. Unter dem Begriff Producer sind alle Stationen in einer Animationsumgebung zusammengefasst, die einen Ausgangspuffer haben, also das Materiallager InStore und die Werkzeugmaschinen (machine tools). Das Basis Interface Producer wurde bereitgestellt, um allgemeine Eigenschaften von Stationen mit einem Ausgangspuffer zu repräsentieren. Dieses Interface kann von verschiedenen Klassen implementiert werden. Eine implementierende Klasse liefert Methoden zum Starten eines Producers sowie zum Abfragen der Anlaufposition am Ausgangspuffer eines Producers.

Declaration

public interface Producer

All known subinterfaces

Machine (in A.1.5, page 69), In (in A.1.4, page 68), Machinetool (in A.2.6, page 91), InStore (in A.2.5, page 88)

Method summary

getOutputBufferPosition() Mit dieser Methode lässt sich die Anlaufposition am Ausgangspuffer eines Producers abfragen.
start() Mit dieser Methode kann ein Producer innerhalb der Animationsumgebung gestartet werden.

Methods

- **getOutputBufferPosition**
  `java.awt.Point` **getOutputBufferPosition**`( )`
  - **Description**
    Mit dieser Methode lässt sich die Anlaufposition am Ausgangspuffer eines Producers abfragen.
  - **Returns** – liefert die Anlaufposition am Ausgangspuffer eines Producers.
- **start**
  `void` **start**`( )`
  - **Description**
    Mit dieser Methode kann ein Producer innerhalb der Animationsumgebung gestartet werden.

A.1.7 Interface Workpiece

Das Interface Workpiece zur Visualisierung von Werkstücke als Objekte in der Animationsumgebung. Das Interface selbst stellt keine Methoden zur Verfügung, es wurde vielmehr aus Implementierungssgründen erstellt worden. Somit bietet es nur die Möglichkeit, Werkstücke zu initialisieren. Eine Klasse, die das Interface implementiert, kann hauptsächlich verschiedene Werkstückvarianten instanziieren.

Declaration

public interface Workpiece

All known subinterfaces

WorkpieceSim (in A.2.8, page 97)

All classes known to implement interface

WorkpieceSim (in A.2.8, page 97)

A.2 Package hms.animation.i4d

Package Contents *Page*

Classes

Agv 73

Die Klasse Agv implementiert die Schnittstelle Hts.

CommunicationManager 77

Die Klasse CommunicationManager ist von der Thread-Klasse abgeleitet.

CommunicationManager.Instruction 82

Diese Klasse wird verwendet, um eine Nachricht bei der Kommunikation darzustellen.

I4dAnimation 83

Diese Klasse implementiert die Schnittstelle Animation und realisiert die HMS-Animation mittels der Tcl/Tk 3D-Animationssoftware.

InStore 88

Die Klasse InStore implementiert das Interface In.

Machinetool 91

Die Klasse Machinetool implementiert das Interface Machine.

OutStore 94

Die Klasse OutStore implementiert das Interface Consumer.

WorkpieceSim 97
Die Klasse WorkpieceSim implementiert das Interface Workpiece.

A.2.1 Class Agv

Die Klasse Agv implementiert die Schnittstelle Hts. Ein Objekt dieser Klasse repräsentiert einen Transporter in der Animationsumgebung. Die Klasse Agv kapselt die Agv-Operationen, die von der HMS-Implementierung abgefragt werden können und stellt Methoden zur Verfügung, um ein Agv in der Animation zu starten, die aktuelle Position eines Transporters abzufragen, zum Be- oder Entladen einer Station, das Fahren eines Transporters innerhalb der Animation zu veranlassen.

Declaration

public class Agv
extends java.lang.Object
implements hms.animation.Hts

Field summary

agvId Die eindeutige Identifikation eines Transporters innerhalb der Animationsumgebung.
comm Der CommunicationManager zur Kommunikation.
loading Gibt an, ob der Transporter ein Werkstück beladen hat.
nextPointPos Die Position des Transporters innerhalb der Animationsumgebung.
started Gibt an, ob der Transporter gestartet ist.
stationId Die eindeutige Identifikation einer Station innerhalb der Animationsumgebung.
unloading Gibt an, ob der Transporter ein Werkstück entladen hat.

Constructor summary

Agv(String, CommunicationManager) Der Konstruktor zum Erstellen eines neuen Agv.

Method summary

agvGetPosition() Mit dieser Methode lässt sich die aktuelle Position des Agvs erfragen.

agvPerformHandOver(Consumer) Diese Methode veranlasst das Beladen einer Station vom Typ Consumer.
agvPerformHandOver(Producer) Diese Methode veranlasst das Entladen einer Station vom Typ Producer.
agvStart() Mit dieser Methode kann ein Agv gestartet werden.
moveTo(Point) Die Methode berechnet den neuen Wegpunkt (innerhalb einer Strecke), den ein Transporter anfahren soll.
performMove(Point[]) Die Methode bekommt als Parameter die Wegpunkte einer Strecke, die ein Transporter fahren soll.
route(Point, Point) Diese Methode liefert ein Array mit den Wegpunkten einer Strecke, die ein Transporter zwischen zwei Stationen fahren soll.
sendAndReceive(String, String[]) Die Methode liefert einen String mit der Animationsantwort auf die eingehende Nachricht zurück.

Fields

- protected java.lang.String **agvId**
  - Die eindeutige Identifikation eines Transporters innerhalb der Animationsumgebung.
- protected CommunicationManager **comm**
  - Der CommunicationManager zur Kommunikation.
  - See also
    - `CommunicationManager` (in A.2.2, page 77)
- protected boolean **started**
  - Gibt an, ob der Transporter gestartet ist.
- protected java.lang.String **stationId**
  - Die eindeutige Identifikation einer Station innerhalb der Animationsumgebung.
- protected boolean **loading**
  - Gibt an, ob der Transporter ein Werkstück beladen hat.
- protected boolean **unloading**
  - Gibt an, ob der Transporter ein Werkstück entladen hat.
- public java.awt.Point **nextPointPos**
  - Die Position des Transporters innerhalb der Animationsumgebung.

Constructors

- **Agv**
  Agv(`java.lang.String` **agvId**, `CommunicationManager` **comm**)
  - **Description**
    Der Konstruktor zum Erstellen eines neuen Agv.
  - **Parameters**
    * `agvId` – die eindeutige Identifikation eines Transporters innerhalb der Animationsumgebung.
    * `comm` – der CommunicationManager zur Kommunikation mit der 3D-Animation.

Methods

- **agvGetPosition**
  `public java.awt.Point` **agvGetPosition**( )
  - **Description**
    Mit dieser Methode lässt sich die aktuelle Position des Agvs erfragen.
  - **Returns** – die aktuelle Position eines Transporters in der Animationsumgebung.
- **agvPerformHandOver**
  `public void` **agvPerformHandOver**(`hms.animation.Consumer` **c**)
  - **Description**
    Diese Methode veranlasst das Beladen einer Station vom Typ Consumer.
  - **Parameters**
    * `c` – eine Station vom Typ Consumer.
- **agvPerformHandOver**
  `public void` **agvPerformHandOver**(`hms.animation.Producer` **p**)
  - **Description**
    Diese Methode veranlasst das Entladen einer Station vom Typ Producer.
  - **Parameters**
    * `p` – eine Station vom Typ Producer.
- **agvStart**
  `public void` **agvStart**( )

– **Description**
Mit dieser Methode kann ein Agv gestartet werden.

- **moveTo**
`public void` **moveTo**(`java.awt.Point` **newPos**)
  – **Description**
  Die Methode berechnet den neuen Wegpunkt (innerhalb einer Strecke), den ein Transporter anfahren soll.
  – **Parameters**
    * `newPos` – die neue Position des Transporters (innerhalb einer Strecke), die er anfahren soll.

- **performMove**
`public void` **performMove**(`java.awt.Point[]` **route**)
  – **Description**
  Die Methode bekommt als Parameter die Wegpunkte einer Strecke, die ein Transporter fahren soll.
  – **Parameters**
    * `route` – die Strecke mit den Wegpunkten.

- **route**
`private java.awt.Point[]` **route**(`java.awt.Point` **from**, `java.awt.Point` **to**)
  – **Description**
  Diese Methode liefert ein Array mit den Wegpunkten einer Strecke, die ein Transporter zwischen zwei Stationen fahren soll.
  – **Parameters**
    * `from` – die aktuelle Position des Transporters.
    * `to` – die Zielposition des Transporters.
  – **Returns** – liefert ein Array mit den Wegpunkten einer Strecke, die ein Transporter fahren soll, um innerhalb der Animationsumgebung zur nächsten Station anzukommen.

- **sendAndReceive**
`protected java.lang.String` **sendAndReceive**(`java.lang.String` **command**, `java.lang.String[]` **arguments**)
  – **Description**
  Die Methode liefert einen String mit der Animationsantwort auf die eingehende Nachricht zurück. Wenn der Befehl nicht oder nur teilweise ausgeführt werden konnte, wird null zurückgeliefert.

– **Parameters**
  * `command` – ein String mit dem Befehlnamen.
  * `arguments` – ein String-Array mit den restlichen Argumente des Befehls.
– **Returns** – liefert ein String mit der Animationsantwort auf die eingehende Nachricht, falls der Befehl erfolgreich durchgeführt werden konnte. Ansonsten wird null zurückgegeben.

A.2.2 Class CommunicationManager

Die Klasse CommunicationManager ist von der Thread-Klasse abgeleitet. Sie verwaltet die Kommunikation über eine Socket-Schnittstelle zwischen der HMS-Implementierung und der Tcl/Tk 3D-Animation. Der Verbindungsaufbau erfolgt im Konstruktor dieser Klasse durch Öffnen eines Sockets zur Tcl/Tk 3D-Animation und Erzeugen eines PrintWriters und eines Readers.

Declaration

public class CommunicationManager
extends java.lang.Thread

Field summary

channelIn Ein Reader für den Socket.
channelOut Ein PrintWriter für den Socket.
messageCounter Ein messageCounter, um eindeutige Befehle zu realisieren.
pending Alle Nachrichten der Kommunikation werden zusammen mit ihren IDs in einer HashMap-Liste festgehalten.
socket Ein Socket zur 3D-Animationssoftware.

Constructor summary

CommunicationManager(String, int) Der Konstruktor zum Initialisierung mit den übergebenen Werte.

Method summary

getNewId()
handleAnswer(String) Diese Methode wird innerhalb der run()-Methode aufgerufen.

run() Die Klasse CommunicationManager ist vom Thread abgeleitet.

sendAndReceive(String, String[]) Methode zur Synchronisation der verschiedenen Threads aus beiden Richtungen - von der HMS-Implementierung zur 3D-Animationssoftware einerseits und von der 3D-Animationsoftware zur HMS-Implementierung andererseits - zu erreichen.

Fields

- protected java.net.Socket **socket**
  - Ein Socket zur 3D-Animationssoftware.
  - See also
    - `java.net.Socket`
- protected java.io.PrintWriter **channelOut**
  - Ein PrintWriter für den Socket.
  - See also
    - `java.io.PrintWriter`
- protected java.io.BufferedReader **channelIn**
  - Ein Reader für den Socket.
  - See also
    - `java.io.BufferedReader`
- protected java.util.Map **pending**
  - Alle Nachrichten der Kommunikation werden zusammen mit ihren IDs in einer HashMap-Liste festgehalten. Wenn eine Nachricht vom Client (HMS-Implementierung) zum Tck/Tk 3D-Animation gesendet wird, wird das Nachrichten-Objekt (zusammen mit ihr zugehörigen ID) zu dieser Liste hinzugefügt. Wenn Antworten von der Animationssoftware mit derselben ID empfangen werden, werden die Einträge aus der Liste gelöscht.
- protected int **messageCounter**
  - Ein messageCounter, um eindeutige Befehle zu realisieren.

Constructors

- **CommunicationManager**
  CommunicationManager(`java.lang.String` **host**, `int` **port**) `throws java.io.IOException`

– **Description**
  Der Konstruktor zum Initialisierung mit den übergebenen Werte.
– **Parameters**
  * `host` – der default Host für die Tcl/Tk 3D-Animation.
  * `port` – der default Port für die Tcl/Tk 3D-Animation.
– **Throws**
  * `java.io.IOException` – wenn ein I/O Fehler auftritt.

Methods

- **getNewId**
  `synchronized java.lang.String` **getNewId**`( )`
- **handleAnswer**
  `void` **handleAnswer**`( java.lang.String` **answer** `)`
  – **Description**
    Diese Methode wird innerhalb der run()-Methode aufgerufen. Hierbei wird die Animationsantwort aufgeteilt, um das Instruction-Objekt zu suchen, das zu dieser Animationsantwort gehört, da jede eingehende Nachricht eine eindeutige Nachrichten-Identifikation ID hat, die bei Antworten unverändert an den Absender zurückgeschickt wird. Wenn eine solche Nachricht gefunden wird, wird die Animationsantwort in einem Array festgehalten und der Thread, der auf diese Antwort wartet, wird durch notify() aufgeweckt.
  – **See also**
    * `CommunicationManager.run()` (in A.2.2, page 79)
- **run**
  `public void` **run**`( )`
  – **Description**
    Die Klasse CommunicationManager ist vom Thread abgeleitet. Daher wird diese Methode parallel mit der start()-Methode ausgeführt. In dieser Methode erfolgt das Lesen der Animationsantwort. Dies wird zeilenweise durchgeführt. Danach wird die HashMap durchgelaufen und die Liste aktuallisiert.
  – **See also**
    * `CommunicationManager` (in A.2.2, page 77)
- **sendAndReceive**
  `java.lang.String[]` **sendAndReceive**`( java.lang.String` **command**`, java.lang.Strin`
  arguments `)`

– **Description**
Methode zur Synchronisation der verschiedenen Threads aus beiden Richtungen - von der HMS-Implementierung zur 3D-Animationssoftware einerseits und von der 3D-Animationsoftware zur HMS-Implementierung andererseits - zu erreichen.

– **Parameters**
  * `command` – ein String mit dem Nachrichtnamen.
  * `arguments` – ein String-Array mit den restlichen Argumenten der Nachricht.

– **Returns** – ein String-Array mit der Antwort der Tcl/Tk 3D-Animation.

Members inherited from class `java.lang.Thread`

- `static void ( )`
- `public static int` **activeCount**`( )`
- `private void` **blockedOn**`( sun.nio.ch.Interruptible )`
- `private volatile` **blocker**
- `public final void` **checkAccess**`( )`
- `private` **contextClassLoader**
- `public native int` **countStackFrames**`( )`
- `public static native Thread` **currentThread**`( )`
- `private` **daemon**
- `public void` **destroy**`( )`
- `public static void` **dumpStack**`( )`
- `private` **eetop**
- `public static int` **enumerate**`( Thread[] )`
- `private void` **exit**`( )`
- `public ClassLoader` **getContextClassLoader**`( )`
- `public final String` **getName**`( )`
- `public final int` **getPriority**`( )`
- `public final ThreadGroup` **getThreadGroup**`( )`
- `private` **group**
- `public static native boolean` **holdsLock**`( Object )`
- **inheritableThreadLocals**
- `private` **inheritedAccessControlContext**
- `private void` **init**`( ThreadGroup, Runnable, String, long )`
- `public void` **interrupt**`( )`
- `private native void` **interrupt0**`( )`
- `public static boolean` **interrupted**`( )`
- `public final native boolean` **isAlive**`( )`
- `public final boolean` **isDaemon**`( )`

- public boolean **isInterrupted**( )
- private native boolean **isInterrupted**(boolean)
- public final void **join**( ) throws InterruptedException
- public final synchronized void **join**(long) throws InterruptedException
- public final synchronized void **join**(long, int) throws InterruptedException
- public static final **MAX_PRIORITY**
- public static final **MIN_PRIORITY**
- private **name**
- private static synchronized int **nextThreadNum**( )
- public static final **NORM_PRIORITY**
- private **priority**
- private static native void **registerNatives**( )
- public final void **resume**( )
- private native void **resume0**( )
- public void **run**( )
- public void **setContextClassLoader**(ClassLoader)
- public final void **setDaemon**(boolean)
- public final void **setName**(String)
- public final void **setPriority**(int)
- private native void **setPriority0**(int)
- private **single_step**
- public static native void **sleep**(long) throws InterruptedException
- public static void **sleep**(long, int) throws InterruptedException
- private **stackSize**
- public synchronized native void **start**( )
- private **stillborn**
- public final void **stop**( )
- public final synchronized void **stop**(Throwable)
- private native void **stop0**(Object)
- private static **stopThreadPermission**
- public final void **suspend**( )
- private native void **suspend0**( )
- private **target**
- private static **threadInitNumber**
- **threadLocals**
- private **threadQ**
- public String **toString**( )
- public static native void **yield**( )

A.2.3 Class CommunicationManager.Instruction

Diese Klasse wird verwendet, um eine Nachricht bei der Kommunikation darzustellen. Sie wird von CommunikationManager benutzt, um die Nachrichten bei der Kommunikation über die Socket-Schnittstelle als Objekte zu verwenden.

Declaration

class CommunicationManager.Instruction
extends java.lang.Object

Field summary

arguments Ein String-Array mit den restlichen Argumenten der Nachricht.
command Ein String mit dem Nachrichtnamen.
id Ein String mit der eindeutigen Identifikation einer Nachricht.
result Ein String-Array mit der Antwort der Animation.
SEPARATOR Eine Konstante zur Begrenzung des Nachrichtnamens von den anderen Argumenten der Nachricht.

Constructor summary

CommunicationManager.Instruction(String, String, String[]) Konstruktor zum Initialisierung mit den übergebenen Werte.

Method summary

toString() Diese Methode überschreibt die von der Klasse Objekt geerbte Methode toString().

Fields

- java.lang.String **command**
  - Ein String mit dem Nachrichtnamen.
- java.lang.String **id**
  - Ein String mit der eindeutigen Identifikation einer Nachricht.
- java.lang.String **arguments**
  - Ein String-Array mit den restlichen Argumenten der Nachricht.
- java.lang.String **result**

– Ein String-Array mit der Antwort der Animation.

- static final java.lang.String **SEPARATOR**
  – Eine Konstante zur Begrenzung des Nachrichtnamens von den anderen Argumenten der Nachricht.

Constructors

- **CommunicationManager.Instruction**
  CommunicationManager.Instruction(java.lang.String **command**, java.lang.String **id**, java.lang.String[] **arguments**)
  – **Description**
    Konstruktor zum Initialisierung mit den übergebenen Werte.
  – **Parameters**
    * `command` – ein String mit dem Nachrichtnamen.
    * `id` – ein String mit der eindeutigen Identifikation einer Nachricht.
    * `arguments` – ein String-Array mit den restlichen Argumenten einer Nachricht.

Methods

- **toString**
  `public java.lang.String` **toString**( )
  – **Description**
    Diese Methode überschreibt die von der Klasse Objekt geerbte Methode toString().
  – **Returns** – liefert eine geeignete String-Repräsentation zurückt.
  – **See also**
    * `java.lang.StringBuffer`

A.2.4 Class I4dAnimation

Diese Klasse implementiert die Schnittstelle Animation und realisiert die HMS-Animation mittels der Tcl/Tk 3D-Animationssoftware. Um der Zugriff von verschiedenen Threads auf die Tcl/Tk 3D-Animation steuern zu können, ist sie als Singleton implementiert. Die Klasse I4dAnimation stellt damit eine globale Zugriffsmöglichkeit auf das einzige Objekt zur Verfügung und instanziert es beim ersten Zugriff automatisch.

Declaration

public class I4dAnimation
extends java.lang.Object
implements hms.animation.Animation

Field summary

communicationManager Ein CommunicationManager zur Verwaltung der Kommunikation über eine Socket-Schnittstelle.
CREATE_INSTORE_CMD Eine String-Konstante mit dem Befehlnamen.
CREATE_OUTSTORE_CMD Eine String-Konstante mit dem Befehlnamen.
defaultHost Der default Host für die Tcl/Tk 3D-Animation.
defaultSocket Der default Port für die Tcl/Tk 3D-Animation.
host Der Host für die Kommunikation mit der Tcl/Tk 3D-Animation.
instance Die einzige Instanz der Klasse I4dAnimation.
MT_TYPES Ein String-Array mit den verwendeten Werkzeugmaschinetypen.
nextMtType Der im Array nachfolgende Werkzeugmaschinetyp.
socket Der Port für die Kommunikation mit der Tcl/Tk 3D-Animation.

Constructor summary

I4dAnimation() Der Default-Konstruktor.
I4dAnimation(String, int) Der Copy-Konstruktor zur Initialisierung mit einem existierenden Objekt.

Method summary

connect() Methode zum Erzeugen eines CommunicationManager mit dem Host und den Port zur Animation.
createHts(Point, short) Mit dieser Methode kann ein Transporter als Objekt in einer Animation visualisiert werden.
createInStore(Point, short) Mit dieser Methode kann ein Materiallager InStore als Objekt in einer Animation visualisiert werden.
createMachine(Point, short) Mit dieser Methode kann eine Werkzeugmaschine als Objekt in einer Animation visualisiert werden.
createOutStore(Point, short) Mit dieser Methode kann ein Materiallager OutStore als Objekt in einer Animation visualisiert werden.
createStore(String, Point, short) Mit dieser Methode kann überprüft werden, ob ein Materiallager erfolgreich visualisiert werden konnte.

getI4dAnimation() Mit der Methode getI4dAnimation() kann auf die einzige Instanz dieser KLasse zugegriffen werden.

Fields

- protected static final java.lang.String **defaultHost**
  - Der default Host für die Tcl/Tk 3D-Animation.
- protected java.lang.String **host**
  - Der Host für die Kommunikation mit der Tcl/Tk 3D-Animation.
- protected static final int **defaultSocket**
  - Der default Port für die Tcl/Tk 3D-Animation.
- protected int **socket**
  - Der Port für die Kommunikation mit der Tcl/Tk 3D-Animation.
- protected CommunicationManager **communicationManager**
  - Ein CommunicationManager zur Verwaltung der Kommunikation über eine Socket-Schnittstelle.
  - See also
    - `CommunicationManager` (in A.2.2, page 77)
- private static I4dAnimation **instance**
  - Die einzige Instanz der Klasse I4dAnimation.
- protected static final java.lang.String **CREATE_INSTORE_CMD**
  - Eine String-Konstante mit dem Befehlnamen.
- protected static final java.lang.String **CREATE_OUTSTORE_CMD**
  - Eine String-Konstante mit dem Befehlnamen.
- protected static final java.lang.String **MT_TYPES**
  - Ein String-Array mit den verwendeten Werkzeugmaschinetypen.
- protected int **nextMtType**
  - Der im Array nachfolgende Werkzeugmaschinetyp.

Constructors

- **I4dAnimation**
  `private` **I4dAnimation**`( )`
  - **Description**
    Der Default-Konstruktor.
- **I4dAnimation**
  `public` **I4dAnimation**`( java.lang.String` **host**`, int` **socket** `)`
  - **Description**
    Der Copy-Konstruktor zur Initialisierung mit einem existierenden Objekt.
  - **Parameters**
    * `host` – der Animationshost.
    * `socket` – der Animationsport.

Methods

- **connect**
  `private void` **connect**`( )`
  - **Description**
    Methode zum Erzeugen eines CommunicationManager mit dem Host und den Port zur Animation. Wenn die Kommunikation erfolgreich über die Socket-Schnittstelle war, die Verbindung zwischen dem Client und der Animation konnte aufgebaut werden. Ansonsten wird eine Fehlermeldung mit Hilfe der printStackTrace() erzeugt und ausgegeben.
- **createHts**
  `public hms.animation.Hts` **createHts**`( java.awt.Point` **position**`, short` **direction** `)`
  - **Description**
    Mit dieser Methode kann ein Transporter als Objekt in einer Animation visualisiert werden.
  - **Parameters**
    * `position` – die Position eines Transporters in der Animationsumgebung.
    * `direction` – die Ausrichtung eines Transporters in der Animationsumgebung.
  - **Returns** – liefert ein Transporter mit einer eindeutigen Identifikation, einer Position und einer Ausrichtung in der Animationsumgebung, wenn der Befehl erfolgreich durchgeführt werden konnte. Ansonsten wird null ausgegeben.

- **createInStore**
  `public hms.animation.In` **createInStore**`( java.awt.Point` **position**`, short` **direction** `)`
  - **Description**
    Mit dieser Methode kann ein Materiallager InStore als Objekt in einer Animation visualisiert werden.
  - **Parameters**
    * `position` – die Position des Materiallagers InStore in der Animationsumgebung.
    * `direction` – die Ausrichtung des Materiallagers InStore in der Animationsumgebung.
  - **Returns** – liefert das Materiallager InStore innerhalb der Animationsumgebung, wenn der Befehl erfolgreich durchgeführt werden konnte. Ansonsten wird null ausgegeben.
- **createMachine**
  `public hms.animation.Machine` **createMachine**`( java.awt.Point` **position**`, short` **direction** `)`
  - **Description**
    Mit dieser Methode kann eine Werkzeugmaschine als Objekt in einer Animation visualisiert werden.
  - **Parameters**
    * `position` – die Position einer Werkzeugmaschine in der Animationsumgebung.
    * `direction` – die Ausrichtung einer Werkzeugmaschine in der Animationsumgebung.
  - **Returns** – liefert eine Machine mit einer eindeutigen Identifikation, einer Position und einer Ausrichtung in der Animationsumgebung, wenn der Befehl erfolgreich durchgeführt werden konnte. Ansonsten wird null ausgegeben.
- **createOutStore**
  `public hms.animation.Consumer` **createOutStore**`( java.awt.Point` **position**`, short` **direction** `)`
  - **Description**
    Mit dieser Methode kann ein Materiallager OutStore als Objekt in einer Animation visualisiert werden.
  - **Parameters**
    * `position` – die Position des Materiallagers OutStore in der Animationsumgebung.

    * `direction` – die Ausrichtung des Materiallagers OutStore in der Animationsumgebung.
  - **Returns** – liefert das Materiallager OutStore innerhalb der Animationsumgebung, wenn der Befehl erfolgreich durchgeführt werden konnte. Ansonsten wird null ausgegeben.

- **createStore**
  `private boolean` **createStore**`( java.lang.String` **command**`, java.awt.Point` **position**`, short` **direction** `)`
  - **Description**
    Mit dieser Methode kann überprüft werden, ob ein Materiallager erfolgreich visualisiert werden konnte.
  - **Parameters**
    * `command` – ein String mit dem Nachrichtnamen.
    * `position` – die Position eines Materiallagers in der Animationsumgebung.
    * `direction` – die Ausrichtung eines Materiallagers in der Animationsumgebung.
  - **Returns** – liefert eine Animationsantwort auf die eingehende Nachricht bzgl. der erfolgreichen Erzeugung eines Materiallagers.

- **getI4dAnimation**
  `public static I4dAnimation` **getI4dAnimation**`( )`
  - **Description**
    Mit der Methode getI4dAnimation() kann auf die einzige Instanz dieser KLasse zugegriffen werden.
  - **Returns** – liefert die einzige Instanz der Klasse.

A.2.5 Class InStore

Die Klasse InStore implementiert das Interface In. Da das Interface In von Producer abgeleitet ist, erbt es alle Methodendefinitionen des Basis-Interfaces Producer. Daher implementiert diese Klasse alle Methoden des Interfaces Producer und wird sie um einige Methoden erweitert. .

Declaration

public class InStore
extends java.lang.Object
implements hms.animation.In

Field summary

comm Ein CommunicationManager zur Verwaltung der Kommunikation über eine Socket-Schnittstelle.
itemType Ein String mit der Werkstückvariante, die von den Werkzeugmaschinen bearbeitet werden soll.
p Eine Station vom Typ Producer in der Animationsumgebung.
started Gibt an, ob der Materiallager InStore gestartet ist.
workpieceSim Ein Werkstück innerhalb der Animationsumgebung.

Constructor summary

InStore(CommunicationManager) Dies ist der Konstruktor zum Erstellen eines Objektes der Klasse InStore.

Method summary

getOutputBufferPosition() Die Methode wird benutzt, um die Anlaufposition am Ausgangspuffer des InStore abzufragen.
inPerformSupplyNewItem() Diese Methode lässt ein neues Werkstück im Eingangslager InStore erscheinen.
sendAndReceive(String, String[]) Die Methode liefert einen String mit der Animationsantwort auf die eingehende Nachricht zurück.
start() Mit dieser Methode kann ein Materiallager InStore gestartet werden.

Fields

- protected CommunicationManager **comm**
  - Ein CommunicationManager zur Verwaltung der Kommunikation über eine Socket-Schnittstelle.
  - See also
    - `CommunicationManager` (in A.2.2, page 77)
- protected boolean **started**
  - Gibt an, ob der Materiallager InStore gestartet ist.
- java.lang.String **itemType**
  - Ein String mit der Werkstückvariante, die von den Werkzeugmaschinen bearbeitet werden soll. Diese Werkstückvariante ist dadurch gekennzeichnet, dass sie von allen Werkzeugmaschinen bearbeitet wird -zuerst von der Werkzeugmaschine „mill“, danach von der „drill“ und abschießend von der „wash“.

- protected WorkpieceSim **workpieceSim**
  - Ein Werkstück innerhalb der Animationsumgebung.
- protected hms.animation.Producer **p**
  - Eine Station vom Typ Producer in der Animationsumgebung. Producer sind alle Stationen in der Animationsumgebung, die einen Ausgangspuffer haben, also das Materiallager InStore und die Werkzeugmaschinen.

Constructors

- **InStore**
  `public` **InStore**`( CommunicationManager` **comm** `)`
  - **Description**
    Dies ist der Konstruktor zum Erstellen eines Objektes der Klasse InStore. Der Konstruktor dieser Klasse liefert nach dem Starten des Materiallagers InStore ein neues Werkstück vom Typ „eng“.
  - **Parameters**
    - `comm` – ein CommunicationManager zur Verwaltung der Kommunikation über die Socket-Schnittstelle.

Methods

- **getOutputBufferPosition**
  `public java.awt.Point` **getOutputBufferPosition**`( )`
  - **Description**
    Die Methode wird benutzt, um die Anlaufposition am Ausgangspuffer des InStore abzufragen.
  - **Returns** – liefert die Anlaufposition am Ausgangspuffer des InStore, wenn der Befehl erfolgreich durchgeführt werden konnte. Ansonsten wird null zurückgegeben.
- **inPerformSupplyNewItem**
  `public hms.animation.Workpiece` **inPerformSupplyNewItem**`( )`
  - **Description**
    Diese Methode lässt ein neues Werkstück im Eingangslager InStore erscheinen.
  - **Returns** – liefert ein Werkstück zurück, wenn der Befehl erfolgreich durchgeführt werden konnte. Ansonsten wird null zurückgegeben.

- **sendAndReceive**
  `protected java.lang.String` **sendAndReceive**`( java.lang.String` **command**`, java.lang.String[]` **arguments** `)`
  - **Description**
    Die Methode liefert einen String mit der Animationsantwort auf die eingehende Nachricht zurück. Wenn der Befehl nicht oder nur teilweise ausgeführt werden konnte, wird null zurückgeliefert.
  - **Parameters**
    * `command` – ein String mit dem Befehlnamen.
    * `arguments` – ein String-Array mit den restlichen Argumente des Befehls.
  - **Returns** – liefert ein String mit der Animationsantwort auf die eingehende Nachricht, falls der Befehl durchgeführt werden konnte. Ansonsten wird null zurückgegeben.
- **start**
  `public void` **start**`( )`
  - **Description**
    Mit dieser Methode kann ein Materiallager InStore gestartet werden.

A.2.6 Class Machinetool

Die Klasse Machinetool implementiert das Interface Machine.

Declaration

public class Machinetool
extends java.lang.Object
implements hms.animation.Machine

Field summary

c Eine Station vom Typ Consumer in der Animationsumgebung.
comm Der CommunicationManager zur Kommunikation.
itemType Ein String mit der Werkstückvariante, die von den Werkzeugmaschinen bearbeitet werden soll.
mt Eine Station vom Typ Werkzeugmaschine in der Animationsumgebung.
p Eine Station vom Typ Producer in der Animationsumgebung.
started Gibt an, ob eine Werkzeugmaschine gestartet ist.
stationId Die eindeutige Identifikation einer Werkzeugmaschine innerhalb der Animationsumgebung.

unloading Gibt an, ob eine Werkzeugmaschine ein Werkstück am Eingangspuffer zur Verarbeitung hat.
working Gibt an, ob eine Werkzeugmaschine ein Werkstück bearbeitet.

Constructor summary

Machinetool(String, CommunicationManager) Der Konstruktor initialisiert eine Werkzeugmaschine.

Method summary

getInputBufferPosition() Anfrage über die Anlaufposition am Eingangspuffer der Werkzeugmaschine.
getOutputBufferPosition() Anfrage über die Anlaufposition am Ausgangspuffer der Werkzeugmaschine.
getStationId() Methode zum Holen einer Werkzeugmaschine.
performWork() Diese Methode veranlasst das Holen eines Werkstücks aus dem Eingangspuffer und dessen Bearbeitung.
sendAndReceive(String, String[]) Liefert die Antwort für einen gesendeten Befehl.
start() Mit dieser Methode kann eine Werkzeugmaschine gestartet werden.

Fields

- private java.lang.String **stationId**
  - Die eindeutige Identifikation einer Werkzeugmaschine innerhalb der Animationsumgebung.
- protected CommunicationManager **comm**
  - Der CommunicationManager zur Kommunikation.
  - See also
    - `CommunicationManager` (in A.2.2, page 77)
- protected boolean **started**
  - Gibt an, ob eine Werkzeugmaschine gestartet ist.
- protected boolean **working**
  - Gibt an, ob eine Werkzeugmaschine ein Werkstück bearbeitet.
- protected boolean **unloading**
  - Gibt an, ob eine Werkzeugmaschine ein Werkstück am Eingangspuffer zur Verarbeitung hat.

- java.lang.String **itemType**
  - Ein String mit der Werkstückvariante, die von den Werkzeugmaschinen bearbeitet werden soll.
- public hms.animation.Producer **p**
  - Eine Station vom Typ Producer in der Animationsumgebung. Producer sind alle Stationen in der Animationsumgebung, die einen Ausgangspuffer haben, also das Materiallager InStore und die Werkzeugmaschinen.
- public hms.animation.Consumer **c**
  - Eine Station vom Typ Consumer in der Animationsumgebung. Consumer sind alle Stationen in der Animationsumgebung, die einen Eingangspuffer haben, also das Materiallager OutStore und die Werkzeugmaschinen.
- public hms.animation.Machine **mt**
  - Eine Station vom Typ Werkzeugmaschine in der Animationsumgebung.

Constructors

- **Machinetool**
  Machinetool(`java.lang.String` **stationId**, `CommunicationManager` **comm**)
  - **Description**
    Der Konstruktor initialisiert eine Werkzeugmaschine.
  - **Parameters**
    * `stationId` – die eindeutige Identifikation einer Werkzeugmaschine innerhalb der Animationsumgebung.
    * `comm` – der CommunicationManager zur Kommunikation.

Methods

- **getInputBufferPosition**
  `public java.awt.Point` **getInputBufferPosition**( )
  - **Description**
    Anfrage über die Anlaufposition am Eingangspuffer der Werkzeugmaschine.
  - **Returns** – liefert die Position, deren Koordinaten der Anlaufposition am Eingangspuffer der Werkzeugmaschine entsprechen.
- **getOutputBufferPosition**
  `public java.awt.Point` **getOutputBufferPosition**( )

- **Description**
  Anfrage über die Anlaufposition am Ausgangspuffer der Werkzeugmaschine.
- **Returns** – liefert die Position, deren Koordinaten der Anlaufposition am Ausgangspuffer der Werkzeugmaschine entspricht.

• **getStationId**
`java.lang.String` **getStationId**`( )`

- **Description**
  Methode zum Holen einer Werkzeugmaschine.
- **Returns** – liefert einen String mit der eindeutigen Werzeugtmaschine .

• **performWork**
`public void` **performWork**`( )`

- **Description**
  Diese Methode veranlasst das Holen eines Werkstücks aus dem Eingangspuffer und dessen Bearbeitung.

• **sendAndReceive**
`protected java.lang.String` **sendAndReceive**`( java.lang.String` **command**, `java.lang.String[]` **arguments** `)`

- **Description**
  Liefert die Antwort für einen gesendeten Befehl.
- **Parameters**
  * `command` – String mit dem Befehlnamen.
  * `arguments` – ein String-Array mit den restlichen Argumente des Befehls.
- **Returns** – liefert einen String mit der Animationsantwort auf die eingehende Nachricht oder null, wenn der Befehl (oder ein Teil davon) nicht ausgeführt werden konnte.

• **start**
`public void` **start**`( )`

- **Description**
  Mit dieser Methode kann eine Werkzeugmaschine gestartet werden.

A.2.7 Class OutStore

Die Klasse OutStore implementiert das Interface Consumer.

Declaration

public class OutStore
extends java.lang.Object
implements hms.animation.Consumer

Field summary

comm Ein CommunicationManager zur Verwaltung der Kommunikation über eine Socket-Schnittstelle.
started Gibt an, ob der Materiallager OutStore gestartet ist.

Constructor summary

OutStore(CommunicationManager) Dies ist der Konstruktor zum Erstellen eines Objektes der Klasse OutStore.

Method summary

getInputBufferPosition() Die Methode wird benutzt, um die Anlaufposition am Eingangspuffer des OutStore abzufragen.
sendAndReceive(String, String[]) Die Methode liefert einen String mit der Animationsantwort auf die eingehende Nachricht zurück.
start() Mit dieser Methode kann ein Materiallager OutStore gestartet werden.

Fields

- protected CommunicationManager **comm**
  - Ein CommunicationManager zur Verwaltung der Kommunikation über eine Socket-Schnittstelle.
  - See also
    - `CommunicationManager` (in A.2.2, page 77)
- protected boolean **started**
  - Gibt an, ob der Materiallager OutStore gestartet ist.

Constructors

- **OutStore**
  `public` **OutStore**`( CommunicationManager` **comm** `)`
  - **Description**
    Dies ist der Konstruktor zum Erstellen eines Objektes der Klasse OutStore.
  - **Parameters**
    * `comm` – ein CommunicationManager zur Verwaltung der Kommunikation über die Socket-Schnittstelle.

Methods

- **getInputBufferPosition**
  `public java.awt.Point` **getInputBufferPosition**`( )`
  - **Description**
    Die Methode wird benutzt, um die Anlaufposition am Eingangspuffer des OutStore abzufragen.
  - **Returns** – liefert die Anlaufposition am Eingangspuffer des OutStore, wenn der Befehl erfolgreich durchgeführt werden konnte. Ansonsten wird null zurückgegeben.
- **sendAndReceive**
  `protected java.lang.String` **sendAndReceive**`( java.lang.String` **command**`, java.lang.String[]` **arguments** `)`
  - **Description**
    Die Methode liefert einen String mit der Animationsantwort auf die eingehende Nachricht zurück. Wenn der Befehl nicht oder nur teilweise ausgeführt werden konnte, wird null zurückgeliefert.
  - **Parameters**
    * `command` – ein String mit dem Befehlnamen.
    * `arguments` – ein String-Array mit den restlichen Argumente des Befehls.
  - **Returns** – liefert ein String mit der Animationsantwort auf die eingehende Nachricht, falls der Befehl erfolgreich durchgeführt werden konnte. Ansonsten wird null zurückgegeben.
- **start**
  `public void` **start**`( )`
  - **Description**
    Mit dieser Methode kann ein Materiallager OutStore gestartet werden.

A.2.8 Class WorkpieceSim

Die Klasse WorkpieceSim implementiert das Interface Workpiece. Realisiert die Darstellung von Werkstücken innerhalb der Animation, ihren String-Repräsentation, sowie den Vergleich zweier String-Objekte.

Declaration

public class WorkpieceSim
extends java.lang.Object
implements hms.animation.Workpiece

Field summary

itemType Ein String mit der Werkstückvariante („eng“), die von den Werkstückmaschinen bearbeitet werden soll.

Constructor summary

WorkpieceSim(String) Der Konstruktor zum Erstellen von Werkstücken.

Method summary

equals(Object) Überschreibt die von der Klasse Object geerbte Methode equals.
toString() Die String-Konvertierung des WorkpieceSim-Objektes.

Fields

- private static java.lang.String **itemType**
  - Ein String mit der Werkstückvariante („eng“), die von den Werkstückmaschinen bearbeitet werden soll.

Constructors

- **WorkpieceSim**
  `public` **WorkpieceSim**`( java.lang.String` **itemType** `)`
  - **Description**
    Der Konstruktor zum Erstellen von Werkstücken.
  - **Parameters**

    * `itemType` – ein String mit dem das neue Werkstück initialisiert werden soll.

Methods

- **equals**
  `public boolean equals( java.lang.Object other )`
  - **Description**
    Überschreibt die von der Klasse Object geerbte Methode equals.
  - **Parameters**
    * `other` – ein bereits existierendes WorkpieceSim-Objekt.
  - **Returns** – liefert true, wenn beiden Objekte denselben Wert repräsentieren.
- **toString**
  `public java.lang.String toString( )`
  - **Description**
    Die String-Konvertierung des WorkpieceSim-Objektes.
  - **Returns** – liefert eine String-Repräsentation des Werkstückes.

A.3 Package hms.animation.test

Package Contents *Page*

Classes

Main 98

Diese Klasse ist zum Testen der Simulation erstellt worden.

Main.HtsControl 99

A.3.1 Class Main

Diese Klasse ist zum Testen der Simulation erstellt worden. Hiermit werden Objekte in der Animationsumgebung dargestellt. Danach werden perform-Befehle zur 3D-Animation gesendet, um die HMS-Implementierung zu testen.

Declaration

public class Main
extends java.lang.Object

Constructor summary

Main()

Method summary

animate()
main(String[])

Constructors

- **Main**
  `public` **Main**`( )`

Methods

- **animate**
  `private static void` **animate**`( )`
- **main**
  `public static void` **main**`( java.lang.String[]` **args** `)`

A.3.2 Class Main.HtsControl

Declaration

static class Main.HtsControl
extends java.lang.Thread

Field summary

hts
points

Constructor summary

Main.HtsControl(Animation, Point[])

Method summary

run()

Fields

- java.awt.Point **points**
- hms.animation.Hts **hts**

Constructors

- **Main.HtsControl**
  `public` **Main.HtsControl**(`hms.animation.Animation` **anim**, `java.awt.Point[]` **points**)

Methods

- **run**
  `void` **run**( )

Members inherited from class `java.lang.Thread`

- `static void ( )`
- `public static int` **activeCount**( )
- `private void` **blockedOn**(`sun.nio.ch.Interruptible`)
- `private volatile` **blocker**
- `public final void` **checkAccess**( )
- `private` **contextClassLoader**
- `public native int` **countStackFrames**( )
- `public static native Thread` **currentThread**( )
- `private` **daemon**
- `public void` **destroy**( )
- `public static void` **dumpStack**( )
- `private` **eetop**
- `public static int` **enumerate**(`Thread[]`)
- `private void` **exit**( )
- `public ClassLoader` **getContextClassLoader**( )
- `public final String` **getName**( )
- `public final int` **getPriority**( )
- `public final ThreadGroup` **getThreadGroup**( )
- `private` **group**
- `public static native boolean` **holdsLock**(`Object`)
- **inheritableThreadLocals**
- `private` **inheritedAccessControlContext**
- `private void` **init**(`ThreadGroup, Runnable, String, long`)

- public void **interrupt**( )
- private native void **interrupt0**( )
- public static boolean **interrupted**( )
- public final native boolean **isAlive**( )
- public final boolean **isDaemon**( )
- public boolean **isInterrupted**( )
- private native boolean **isInterrupted**(boolean)
- public final void **join**( ) throws InterruptedException
- public final synchronized void **join**(long) throws InterruptedException
- public final synchronized void **join**(long, int) throws InterruptedExcepti-on
- public static final **MAX_PRIORITY**
- public static final **MIN_PRIORITY**
- private **name**
- private static synchronized int **nextThreadNum**( )
- public static final **NORM_PRIORITY**
- private **priority**
- private static native void **registerNatives**( )
- public final void **resume**( )
- private native void **resume0**( )
- public void **run**( )
- public void **setContextClassLoader**(ClassLoader)
- public final void **setDaemon**(boolean)
- public final void **setName**(String)
- public final void **setPriority**(int)
- private native void **setPriority0**(int)
- private **single_step**
- public static native void **sleep**(long) throws InterruptedException
- public static void **sleep**(long, int) throws InterruptedException
- private **stackSize**
- public synchronized native void **start**( )
- private **stillborn**
- public final void **stop**( )
- public final synchronized void **stop**(Throwable)
- private native void **stop0**(Object)
- private static **stopThreadPermission**
- public final void **suspend**( )
- private native void **suspend0**( )
- private **target**
- private static **threadInitNumber**
- **threadLocals**
- private **threadQ**
- public String **toString**( )
- public static native void **yield**( )