Fachgebiet Simulation und Systemoptimierung
Fachbereich Informatik
Technische Universität Darmstadt

RoboFrame - Softwareframework für mobile autonome Robotersysteme

RoboFrame - Softwareframework for mobile autonomous robotic systems

Diplomarbeit

von

Sebastian Petters und Dirk Thomas

Darmstadt, Juli 2005

Aufgabenstellung: Prof. Dr. Oskar von Stryk
Betreuer: Dipl.-Inform. Martin Friedmann
Dipl.-Math. Jutta Kiener
Dipl.-Tech. Math. Maximilian Stelzer

Erklärung zur Diplomarbeit

Hiermit versichern wir, dass die vorliegende Diplomarbeit ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt wurde. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, Juli 2005

Sebastian Petters   Dirk Thomas

Zusammenfassung

Das Fachgebiet Simulation und Systemoptimierung der Technischen Universität Darmstadt beschäftigt sich mit der Forschung an mobilen autonomen Robotersystemen. Neben der Hardware und robusten Algorithmen zur Robotersteuerung kommt auch der zugrunde liegenden Softwarearchitektur eine zunehmend wichtige Bedeutung zu.

In dieser Arbeit wurde ein plattformunabhängiges Framework entwickelt, mit dessen Hilfe eine Robotersteuerung und eine graphische Oberfläche zu Entwicklungs- und Debugzwecken entworfen werden kann. Als beispielhafte Implementierung wurden zusammen mit Praktikumsgruppen zwei Anwendungen entwickelt, die im Frühjahr 2005 an der *Internationalen RoboCup GermanOpen* und im Sommer 2005 an der *RoboCup Weltmeisterschaft* in der *Humanoid League* erfolgreich teilnahmen.

Abstract

The Simulation and Systems Optimization Group of the Department of Computer Science of the Technische Universität Darmstadt concerns itself with the research of mobile autonomous robotic systems. Apart from the hardware and durable algorithms for robotic control the basic software architecture becomes more and more important.

In this work a platform-independent framework was developed, which assists to implement a robot control system and a graphic surface for assisting in the development process and for debugging purposes. Two example application were implemented based on that framework together with groups of practical courses, which participated successfully in the *Humanoid League* at the *International RoboCup GermanOpen* in spring 2005 and at the *RoboCup World championship* in summer 2005.

Inhaltsverzeichnis

Abbildungsverzeichnis v

Tabellenverzeichnis vii

1. Vorwort 1
1.1. Ziel der Arbeit . . . 1
1.2. Aufbau der Arbeit . . . 1
1.3. Quellcode . . . 2
1.4. Autoren . . . 2

2. Motivation 3

3. Grundlagen 5
3.1. Frameworks . . . 5
3.2. Existierende Architekturen für Robotersteuerungen . . . 6
3.2.1. GT200x . . . 6
3.2.2. Miro . . . 7
3.2.3. ARIA / Saphira . . . 9
3.2.4. Zusammenfassung . . . 10

4. Anforderungen 11
4.1. Anwendungen . . . 11
4.2. Vorhandene Plattformen . . . 12
4.2.1. iXs iHs04 . . . 12
4.2.2. Kondo KHR-1 . . . 13
4.2.3. Pioneer 2-DX . . . 14
4.2.4. Entwicklungsplattformen . . . 14
4.2.5. Weitere Plattformen . . . 15
4.2.6. Zusammenfassung . . . 15
4.3. Debugging . . . 15
4.4. Softwarequalitätskriterien . . . 16
4.5. Entwicklungsphasen . . . 18

5. Konzept 19
5.1. RoboApp 19
5.2. RoboGui 21

6. Realisierung 23
6.1. RoboApp 23
6.1.1. Plattformabstraktion 23
6.1.2. Schlüssel 26
6.1.3. Module 27
6.1.4. Modul-Kommunikation 29
6.1.5. Datenobjekte und Nachrichtenschlangen 33
6.1.6. Prozesse 36
6.1.7. Kommunikationsverwaltung 38
6.1.8. Router und Konnektoren 40
6.1.9. Netzwerkkommunikation 42
6.1.10. Konfigurationsdateien 44
6.1.11. Anwendungsklasse 44
6.1.12. Logging 45
6.2. RoboGui 48
6.2.1. Dialoge 50
6.2.2. Dialog LogRecorder 51
6.3. RoboCup-spezifische Erweiterungen 52
6.3.1. RoboCup-Anwendung 52
6.3.2. GameController-Konnektor 52
6.3.3. Team-Konnektor 53
6.3.4. Farbtabellen-Dialog 54
6.3.5. XABSL-Dialog 56
6.3.6. CopyFiles-Anwendung 57

7. Ergebnisse 59
7.1. Test-Anwendung 59
7.2. Performanz 61
7.3. RoboCup-Anwendungen 65
7.3.1. RoboCup GermanOpen 2005 65
7.3.2. RoboCup Weltmeisterschaft 2005 66

8. Zusammenfassung und Ausblick 69
8.1. Zusammenfassung 69
8.2. Mögliche Erweiterungen 70
8.2.1. Eigene Container- und Stringklassen 70
8.2.2. Ausnahmebehandlung im Framework 70
8.2.3. Portierung der GUI auf QT 4 71
8.2.4. Mathematische Funktionsbibliothek 72

8.2.5. Speichern der GUI-Einstellungen 72
8.2.6. Automatische Tests für Plattformkompatibilität 73
8.2.7. Sammlung wiederverwendbarer Module und Dialoge 74

A. Verwendung des Frameworks 75
A.1. Kompilieren der Bibliotheken 75
A.2. Das Test-Projekt 75
A.3. Neues Projekt erstellen 76
A.3.1. Datenstrukturen definieren 77
A.3.2. Module realisieren 78
A.3.3. Schlüssel spezifizieren 81
A.3.4. Applikation zusammenfügen 82
A.3.5. Projektdateien für die Applikation 84
A.3.6. Logging 87
A.3.7. Dialoge anlegen 88
A.3.8. GUI-Applikation 89
A.3.9. Projektdateien für die GUI 90

B. Verwendete Software 91
B.1. Unix und Linux 91
B.2. Mipsel Linux Cross-Compiler 92
B.3. Windows 2000 / XP 92
B.4. Pocket PC 92

C. Code-Konventionen 93
C.1. Klassen, Variablen und Konstanten 93
C.2. Dateien und Verzeichnisse 94
C.3. Includes und Namensräume 94
C.4. Formatierung 95
C.5. Implementierung 95
C.6. Dokumentation 95
C.7. Beispielcode 96

D. Subversion-Repository 99
D.1. Zugriff auf das Repository 99
D.2. Repository-Struktur 99

E. Abkürzungen 101

F. Literaturverzeichnis 103

Abbildungsverzeichnis

3.1. Module von GT2004 . . . 7
3.2. Sony Aibo ERS 7 . . . 8
3.3. RobotControl . . . 8
3.4. Grobstruktur von CORBA . . . 9
3.5. Systemkomponenten von ARIA . . . 10

4.1. Roboter iXs iHs04 . . . 12
4.2. Roboter Kondo KHR-1 . . . 12
4.3. Roboter Pioneer 2-DX . . . 14

5.1. Grundstruktur einer vereinfachten Roboter-Steuerung . . . 21
5.2. Komponenten einer Framework-Instanz . . . 22

6.1. Plattformabstraktionsschicht . . . 24
6.2. Lebenszyklus von Modulen . . . 28
6.3. Trennung von Puffern und Ringpuffern . . . 30
6.4. Klassendiagramm der Puffer . . . 31
6.5. Struktur des geteilten Speicherbereichs . . . 32
6.6. Struktur innerhalb einer Nachrichtenschlange . . . 35
6.7. Verschiedene Aufrufvarianten eines Prozesses . . . 37
6.8. Aufrufzeitpunkte eines Prozesses bei langer Modullaufzeit . . . 38
6.9. Sequenzdiagramm einer Socket-basierten Netzwerkverbindung . . . 43
6.10. Übersicht aller Komponenten einer Applikation . . . 46
6.11. Applikations-Dialog unter Windows CE . . . 47
6.12. Hauptfenster von RoboGui . . . 49
6.13. LogRecorder-Dialog . . . 51
6.14. RoboCup Windows CE-Dialog . . . 53
6.15. Dialog zum Erstellen einer Farbtabelle . . . 55
6.16. XABSL-Dialog . . . 57
6.17. CopyFiles-Anwendung . . . 58

7.1. Berechnung der Latenz $\Delta\tilde{t}$ . . . 61
7.2. Messergebnisse für $\Delta\tilde{t}$ unter Windows XP . . . 62
7.3. Messergebnisse für $\Delta\tilde{t}$ unter Windows CE . . . 63

7.4. Messergebnisse für $\Delta\tilde{t}$ unter FreeBSD 64
7.5. Messergebnisse für $\Delta\tilde{t}$ unter Windows XP mit verschiedenen Chunk-Größen 65
7.6. Geplante Struktur für den RoboCup 2005 67

A.1. Komponenten der Beispielanwendung 83

D.1. Verzeichnisstruktur im Subversion-Repository 100

Tabellenverzeichnis

4.1. Wechselwirkungen der Softwarequalitätskriterien 17
4.2. Auswirkungen der Qualitätskriterien auf die wirtschaftlichen Kosten . 17

6.1. Beschränkungen beim Nachrichtenaustausch 41

7.1. Kommandozeilenoptionen für die Test-Anwendung 60

A.1. Übersicht über die Log-Level 87

1. Vorwort

1.1. Ziel der Arbeit

Ziel dieser Arbeit ist die Entwicklung eines *plattformunabhängigen Frameworks* zur Erstellung von *Steuerungsprogrammen* für mobile autonome Robotersysteme.

Im Einzelnen beinhaltet dies die auf dem Roboter laufende Anwendung mit entsprechenden Debug-Schnittstellen sowie eine graphische Oberfläche zur einfachen Entwicklung von funktionalen Komponenten. Als Plattformen sollen zwar die zurzeit am *Fachgebiet Simulation und Systemoptimierung* vorhandenen Roboter dienen, eine Erweiterung auf weitere Systeme soll aber trotzdem leicht möglich sein.

Den Anwendern soll die Entwicklung neuer Steuerungsprogramme so einfach wie möglich gemacht werden, ohne durch das Design des Frameworks unnötig eingeschränkt zu werden. Dazu sind für die Module feste Schnittstellen zu entwickeln, die dennoch einen beliebigen Datenfluss erlauben.

1.2. Aufbau der Arbeit

Diese Arbeit dokumentiert zum einen die getroffenen Entwurfsentscheidungen, sie dient zum anderen aber auch als Handbuch für die Verwendung und Erweiterung des Frameworks.

Zum besseren Überblick werden im Folgenden die Inhalte der weiteren Kapitel kurz dargestellt:

Kapitel 2 Beweggründe für die Erstellung des Frameworks.

Kapitel 3 Beschreibung der Grundlagen für das Framework und kurzer Überblick über die bereits existierenden Lösungen.

Kapitel 4 Vorstellung der geplanten Anwendungen und Roboterplattformen und den sich daraus ergebenden weiteren Anforderungen an das Framework.

Kapitel 5 Überblick über die Struktur des Frameworks.

Kapitel 6 Detaillierte Vorstellung der Komponenten.

Kapitel 7 Beschreibung der mit Hilfe des Frameworks entwickelten Anwendungen und Ergebnisse der Performanz-Messungen.

Kapitel 8 Zusammenfassung der Ergebnisse und Ausblick auf mögliche Erweiterungen.

Anhang A Einführung in die Entwicklung von Anwendungen auf Basis des Frameworks anhand eines einfachen Beispiels.

Anhang B Überblick über die verwendete Software.

Anhang C Vorstellung der Code-Konventionen.

Anhang D Überblick über die Verzeichnisstruktur des Repository.

Anhang E Liste der verwendeten Abkürzungen.

Bei der Verwendung dieser Arbeit als reines Handbuch zur Nutzung des Frameworks wird die Lektüre von Kapitel 5 und Anhang A empfohlen. Das zur Erweiterung benötigte Wissen wird in Kapitel 6 vermittelt und durch die API-Dokumentation ergänzt.

1.3. Quellcode

Der im Rahmen dieser Arbeit entstandene Quellcode liegt auf einer CD bei. Darauf ist neben dem mittlerweile mehr als 33.000 Zeilen umfassenden C++ Quellcode auch die daraus generierbare, vollständige API-Dokumentation enthalten. Zusätzlich ist der gesamte Code in dem in Anhang D beschriebenen SVN-Repository zu finden.

1.4. Autoren

Da diese Diplomarbeit zu zweit erstellt wurde, gilt es darzulegen, wer welche Teile der Arbeit im Wesentlichen verfasst hat.
Sebastian Petters hat die Kapitel 2, 3, 4, Abschnitt 6.3, Kapitel 7 und Abschnitt 8.2,
Dirk Thomas die Abschnitte 6.1, 6.2, 8.1 und Anhang A geschrieben.
Die restlichen Teile wurden gemeinsam verfasst.

2. Motivation

Die Entwicklung einer Robotersteuerungssoftware ist eine sehr komplexe Aufgabe. Neben dem Design der für Robotersysteme notwendigen Hardware muss vor allem eine darauf angepasste Software implementiert werden. Diese besteht hauptsächlich aus Algorithmen, zum Beispiel für eine Bildverarbeitung oder Lokalisierung in bekannter Umgebung, aber auch aus der nötigen Infrastruktur, um mit der Hardware und/oder dem Betriebssystem zu interagieren. Bei einem Wechsel der Roboterplattform müssen meist auch die implementierten Algorithmen angepasst werden, um den geänderten Umgebungsbedingungen Rechnung zu tragen.

Falls die Konstruktion der Roboter-Hardware nicht selbst Teil der Forschungsarbeit ist, wird häufig auf fertige Systeme oder Bausätze zurückgegriffen. Dies hat den Vorteil, dass schneller ein funktionsfähiges und erprobtes System eingesetzt werden kann. Der Entwickler wird dadurch von der Auslegung der Aktorik, Sensorik, Energieversorgung, Ansteuerung etc. entlastet.

Auch die Entwicklung von Algorithmen muss nicht jedes Mal aufs Neue geschehen. Für die einzelnen Teilaufgaben existieren nämlich bereits viele Standardalgorithmen, die in Grenzen an die gegebenen Bedingungen angepasst werden können. Neben zahlreicher Literatur zu diesem Thema, insbesondere [13], existieren auch viele fertige Bibliotheken, zum Beispiel CMVision[1].

Das Zusammenfügen der einzelnen Algorithmen zu einer einheitlichen Anwendung muss dagegen für jeden Anwendungsfall neu vorgenommen werden. Bei Robotersteuerungen müssen die Algorithmen meist auf mehrere, asynchron zueinander ablaufende Prozesse aufgeteilt werden, weswegen die Kommunikation komplexer wird und zwingend eine Synchronisation erfordert.

Die Erfahrungen des GermanTeams[2] beim *RoboCup* in der *Sony Four-Legged Robot League*[3] zeigen, dass einfache Schnittstellen ein effektiveres und effizienteres Arbeiten unterstützen [16]. Dadurch wird die Entwicklung komplexer Algorithmen gefördert und Probleme der Integration in das Gesamtsystem treten in den Hintergrund. Allerdings steht zurzeit noch keine plattformunabhängige Softwarearchitektur für mobile autonome Robotersysteme zur Verfügung, die die Erstellung beliebiger Anwendungen

[1] CMVision Homepage: `http://www-2.cs.cmu.edu/~jbruce/cmvision/`

[2] GermanTeam Homepage: `http://www.germanteam.org`

[3] Sony Four-Legged Robot League Homepage: `http://www.tzi.de/4legged/`

ermöglicht.

Motivation für diese Arbeit war daher die Bereitstellung eines plattformunabhängigen Frameworks, welches die Aggregation bestehender Komponenten zu neuen Anwendungen, ähnlich einem Baukastenprinzip, ermöglicht. Darüber hinaus soll dem Entwickler ein Rahmen für eine graphische Oberfläche zur Verfügung gestellt werden, mit der das Testen und Debuggen der Komponenten erheblich erleichtert wird.

3. Grundlagen

3.1. Frameworks

Frameworks (engl. für Rahmenwerk, Fachwerk, Gerüst) sind heutzutage ein wichtiger Bestandteil beim Software Engineering. Als Generatoren für Anwendungen in einem bestimmten Bereich werden sie häufig eingesetzt, um beispielsweise bestehenden Quellcode wiederzuverwenden und damit die Entwicklungszeiten zu verkürzen. Frameworks geben die zugrundeliegende Architektur von Systemen, die auf ihrer Basis entwickelt werden, vor. Sie stellen also eine Menge von bereits getroffenen Designentscheidungen dar. Beispiele für bekannte Frameworks sind *JUnit*[1] von Erich Gamma und Kent Beck für Regressionstests für Java-Anwendungen, Microsofts *.NET*[2] Technologie oder *Struts*[3] der Apache Software Foundation (ASF) für Web-Anwendungen.

Ein Framework an sich ist nicht lauffähig, es muss an sogenannten *hot spots* erweitert werden, um die eigentlichen, applikationsspezifischen Aufgaben zu erfüllen. Meist wird dazu das *objektorientierte Programmierparadigma* und damit *Vererbung* für die Erweiterung eingesetzt. Eine Anwendung besteht demnach aus dem Framework und weiteren, von den Framework-Basisklassen abgeleiteten Klassen.

Die vom Framework bereitgestellte Funktionalität wird von *frozen spots* geliefert. Im Gegensatz zu einer Bibliothek oder einem API ist die Richtung des Aufrufes genau umgekehrt. Nicht der Anwendercode ruft das Framework auf, sondern das Framework wird instanziert und ruft dann über definierte Punkte den erweiterten, neuen Code auf. Die Instanzierung kann dabei auf verschiedene Arten erfolgen, zum Beispiel durch Konfigurationsdateien oder das Entwurfsmuster *Abstrakte Fabrik* [5].

Generell werden bei objektorientierten Frameworks häufig Entwurfsmuster verwendet, da sie die Wiederverwendung bereits erprobter Techniken vereinfachen. Zusätzlich werden die entstandenen Systeme durch die programmiersprachen-unabhängige und einheitliche Benennung leichter verständlich. Durch die in der Literatur dokumentierten Vor- und Nachteile eines Musters sind die Auswirkungen der gewählten Realisierung für die Anwender direkt ersichtlich.

[1] JUnit Homepage: `http://www.junit.org`
[2] .NET Homepage: `http://www.microsoft.com/net/`
[3] Struts Homepage: `http://struts.apache.org`

Je nach Verwendung eines Frameworks spricht man von einem *White-Box-* beziehungsweise von einem *Black-Box-Framework*. Bei einem White-Box-Framework muss der Entwickler einer Anwendung die Basisklassen des Frameworks kennen und erweitern, bei einem Black-Box-Framework wird die Anwendung durch Instanzierung von Frameworkklassen konfiguriert, jedoch ohne diese zu erweitern. Der Übergang von einem White- zu einem Black-Box-Framework ist fließend, d.h. es sind beliebig viele Zwischenstufen möglich. Je weiter entwickelt ein Framework ist, umso stärker tendiert es in Richtung Black-Box-Framework.

Wie bei allen Technologien, bei denen Vererbung zur Erweiterung eingesetzt wird, kommt auch bei Frameworks das *Fragile Base Class Problem* zum Tragen. Hierunter versteht man, dass bei Erweiterung der Basisklassen bestehende Erweiterungen eventuell nicht mehr lauffähig sind, obwohl die vorgeschriebenen Konventionen von beiden Seiten eingehalten wurden. Details dazu finden sich unter [12].

3.2. Existierende Architekturen für Robotersteuerungen

3.2.1. GT200x

Das *GermanTeam* (GT), der Zusammenschluss der vier Universitäten Humboldt Universität zu Berlin, Universität Bremen, Technische Universität Darmstadt und Universität Dortmund, nimmt seit 2001 an den internationalen *RoboCup* Weltmeisterschaften in der *Sony Four-Legged Robot League* teil. Dabei handelt es sich um eine Liga fußballspielender, vierbeiniger Laufroboter von Sony. Die zugrundeliegende Architektur GT200x [15] mit modularer Struktur und festgelegten Komponenten ermöglicht den Austausch von funktionalen Einheiten (Modulen), indem jedem Modul eine feste Schnittstelle vorgegeben ist. So existieren zum Beispiel für die Bildverarbeitung mittlerweile mehrere Lösungen, die alle mit unterschiedlichen Algorithmen arbeiten, aber dieselben Ein- und Ausgaben haben (siehe Abbildung 3.1).

Als Roboterplattform kommt die *Sony Aibo-Familie* zum Einsatz, seit GT2005 wird nur noch das Modell ERS 7 (siehe Abbildung 3.2) unterstützt. Neben dem Roboter existiert auch ein unter Windows 2000/XP laufender Simulator und eine Simulation innerhalb der Anwendung *RobotControl*. Der Simulator und RobotControl sind die primären Debugging-Tools, mit denen es möglich ist, alle Module zu überwachen, zu konfigurieren und zu testen (siehe Abbildung 3.3).

Die Stärken des GermanTeam liegen zum einen in der Größe des Teams (deutschlandweit arbeiten ca. 50 Studenten, wissenschaftliche Mitarbeiter und Assistenten am GT-Quellcode), zum anderen an den guten Debugging-Tools. Durch die modulare Ar-

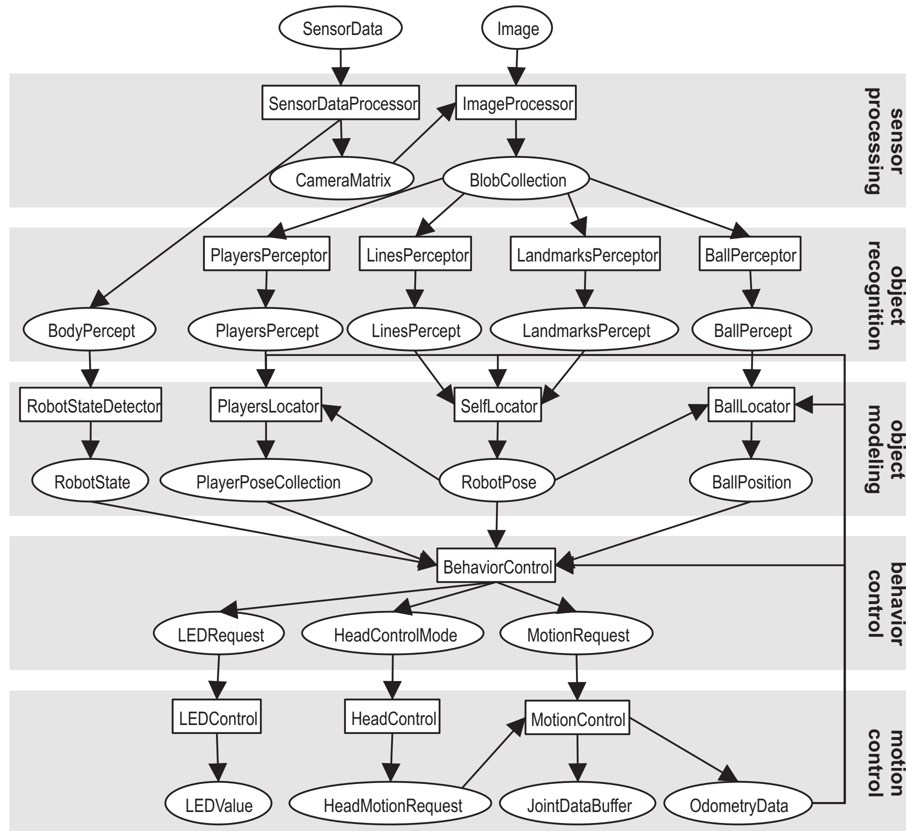

Abbildung 3.1.: Module von GT2004
Übersicht über die in GT2004 verwendeten Module (rechteckig) und die ausgetauschten Daten (oval).

chitektur ist das GermanTeam in der Lage, parallel zahlreiche Algorithmen zu entwickeln.

Bei der GT200x Architektur handelt es sich allerdings noch nicht um ein echtes Framework. Eine Änderung der Modulschnittstellen wirkt sich direkt auf alle Implementierungen aus. Die Datenstrukturen, die zwischen den Modulen ausgetauscht werden, sind stark an die Kommunikationsmechanismen gekoppelt. Zum Entwickeln neuer Anwendungen, die zum Beispiel einen anderen Datenfluss erfordern, sind umfangreiche Änderungen an den Basisklassen nötig.

3.2.2. Miro

Miro (MIddleware for RObots) [2] der Universität Ulm wurde als verteiltes, objektorientiertes Framework für die Steuerung mobiler Roboter entworfen und basiert auf

Abbildung 3.2.: Sony Aibo ERS 7

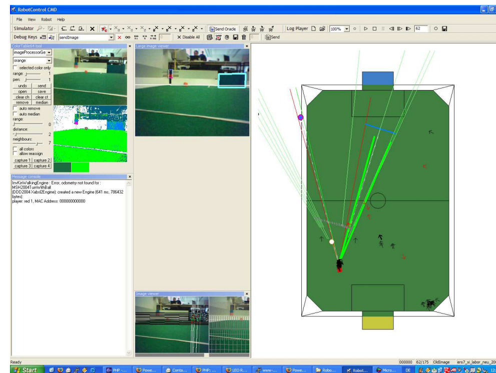

Abbildung 3.3.: RobotControl

der *Common Object Request Broker Architecture* (CORBA)[4]. Motivation hierzu war, dass zur Ansteuerung moderner Sensoren und Aktoren viel Rechenleistung nötig ist und daher meist verteilte Mehrprozessorsysteme zum Einsatz kommen.

CORBA wurde von der *Object Management Group* (OMG)[5] entwickelt und bildet eine objektorientierte Middleware für verteilte Anwendungen in heterogenen Umgebungen. Sie ist nicht an eine bestimmte Programmiersprache gebunden. Die Schnittstellen von Objekten werden mit Hilfe der *Interface Definition Language* (IDL) spezifiziert. Aus den IDL-Definitionen werden für die verwendete Programmiersprache durch spezielle Compiler sogenannte *Stubs* und *Skeletons* generiert, welche die eigentlichen Objekte kapseln. Anfragen eines Objekts werden über den *Object Request Broker* (ORB) entweder an das entsprechende lokale Skeleton oder per *Internet Inter-ORB Protocol* (IIOP) an einen anderen ORB geleitet. Dieser Kommunikationsmechanismus ist in Abbildung 3.4 dargestellt. Zum Ansprechen eines Objekts über einen Namen steht der *COS Naming Service* zur Verfügung. Aufgrund der Spezifikation durch die OMG ist seit CORBA 2.0 nun auch die Kommunikation zwischen ORB-Implementierungen verschiedener Hersteller möglich.

Als Programmiersprache für Miro wird C++, als ORB das plattformunabhängige TAO/ACE[6] verwendet, eine schnelle CORBA Implementierung, die für Echtzeitanwendungen entwickelt wurde. Zurzeit werden als Plattform der RWI B21, die Pioneer Serie von ActivMedia sowie die Sparrow-Architektur der Universität Ulm unterstützt. Nach Aussagen der Entwickler sind Portierungen auf andere Roboter leicht möglich.

Trotz der Wahl von TAO/ACE als ORB bringt die Verwendung von CORBA bei Nicht-Mehrprozessorsystemen einen gewissen Overhead mit sich. Des Weiteren ist die Implementierung von Services mit den zugehörigen IDL-Stubs komplizierter als eine reine C++ Programmierung.

[4] CORBA Homepage: `http://www.corba.org`
[5] OMG Homepage: `http://www.omg.org`
[6] TAO Homepage: `http://www.theaceorb.com`

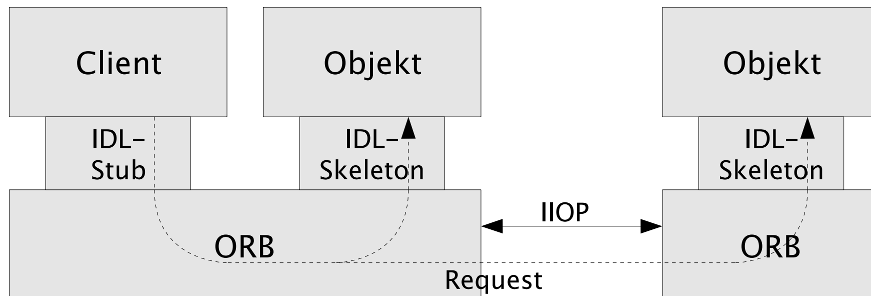

Abbildung 3.4.: Grobstruktur von CORBA

3.2.3. ARIA / Saphira

Beim *ActivMedia Robotics Interface for Applications* (ARIA) [20] handelt es sich um ein von der Firma ActivMedia[7] entwickeltes API für Robotersteuerungen. Es ist objektorientiert und in C++ als Client-Server-Architektur implementiert. Die Roboter-Hardware verfügt über ein eigenes Embedded-System mit dem proprietären Betriebssystem P2OS, das über eine serielle RS-232 Verbindung mit einem weiteren Rechner kommuniziert, auf dem die eigentliche Steuerungssoftware läuft. Die Systemstruktur von ARIA ist in Abbildung 3.5 dargestellt.

ARIA bildet den hardwarenahen Teil der Steuerungssoftware und stellt ein Gateway zur Kommunikation mit der Roboter-Hardware bereit. Zusätzlich enthält es eine *Action-Klasse*, mit deren Hilfe leicht eine reaktive Verhaltenssteuerung realisiert werden kann. Jede Aktion repräsentiert eine *Fuzzy-Regel* [18] und gibt eine *Fuzzy-Menge* aus. Mehrere Aktionen können parallel ablaufen, aus ihren Ausgaben wird von einem sogenannten *Resolver* eine einzelne Fuzzy-Menge durch Interferenz ermittelt. Diese wiederum wird von ARIA defuzzifiziert und die resultierenden Steuerungskommandos über das Gateway an die Hardware geschickt. Den reaktiven Verhaltensweisen kann ein deliberatives Modul übergeordnet werden, um längerfristiges Planen zu ermöglichen.

Auf ARIA setzt Saphira [17] auf und bietet ein einfacheres API und eine graphische Oberfläche. Zusätzlich enthält Saphira Algorithmen zur Lokalisierung und Navigation und bietet mit *Colbert* eine C++ ähnliche Sprache zur einfachen Erstellung von Aktionen.

ARIA unterstützt jedoch nur die Roboter der Firma ActivMedia. Da der Quellcode nicht frei zugänglich ist, ist eine Erweiterung auf andere Plattformen nicht möglich.

[7] ActivMedia Homepage: `http://www.activmedia.com`

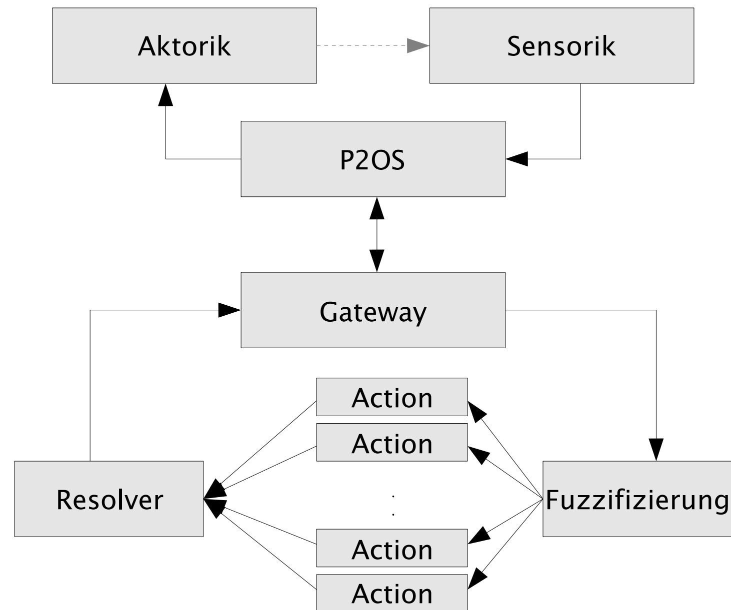

Abbildung 3.5.: Systemkomponenten von ARIA

3.2.4. Zusammenfassung

Im Folgenden werden noch einmal die Vor- und Nachteile der vorgestellten, existierenden Lösungen zusammengefasst.

GT200x
- ⊕ gute Visualisierungs-Mechanismen
- ⊕ Austauschbare Module mit festen Schnittstellen
- ⊖ nur in engen Grenzen flexibel
- ⊖ spezialisiert auf Sony Aibo

Miro
- ⊕ leicht erweiterbar
- ⊕ leicht portierbar
- ⊖ ausgelegt auf Mehrprozessorsysteme
- ⊖ hohe Hardwareanforderungen
- ⊖ durch die Verwendung von CORBA weitere Spezifikationen nötig

ARIA
- ⊕ einfaches API
- ⊕ Graphische Oberfläche
- ⊖ nur Unterstützung von ActivMedia-Robotern
- ⊖ Quellcode nicht verfügbar

4. Anforderungen

4.1. Anwendungen

Die zu entwickelnde Architektur soll die Erstellung von unterschiedlichen Robotersteuerungen ermöglichen. Dabei sollen alle möglichen Einsatzbereiche von mobilen autonomen Robotersystemen abgedeckt werden. Aufgrund der aktuellen Forschung am *Fachgebiet Simulation und Systemoptimierung* wird das Framework wohl hauptsächlich für die Teilnahme am *RoboCup* genutzt werden. Des Weiteren sollen aber auch andere Anwendungen realisiert werden können, wie beispielsweise das Erstellen von Karten oder die Führung von Touristen. Diese Szenarien verlangen, dass mehrere Roboter miteinander kommunizieren oder Daten mit externen Systemen austauschen können.

Weiterhin soll durch das Framework kein spezielles Steuerungsparadigma vorgegeben werden, da damit nur unnötige Einschränkungen vorgegeben würden. Die Wahl des passenden Paradigmas soll, anders als zum Beispiel bei Saphira, ausschließlich durch die Anwendungslogik geschehen.

Die funktionalen Komponenten, die die eigentlichen Algorithmen implementieren, im Folgenden *Module* genannt, sollen weitest möglich plattformunabhängig realisiert werden können. Dazu muss im Framework von der Hardware abstrahiert und diese Schicht den Modulen zugreifbar gemacht werden. Dies beinhaltet unter anderem die Ausgabe von Nachrichten auf der Systemkonsole, das Öffnen von Dateien und Netzwerkverbindungen sowie das Starten von Threads.

Die realisierten Anwendungen an sich sollten aus einzelnen Modulen, ähnlich einem Baukastensystem, zusammengesetzt sein. Die Koppelung der Module untereinander soll nur durch die ausgetauschten Nachrichten definiert werden. Damit wird der Ersatz von Modulen oder die Umstrukturierung der ganzen Anwendung wesentlich leichter ermöglicht. Ähnlich der GT200x Architektur können so verschiedene Algorithmen parallel entwickelt und getestet werden. Dabei soll jedoch eine flexiblere Lösung bei den Modulschnittstellen angestrebt werden.

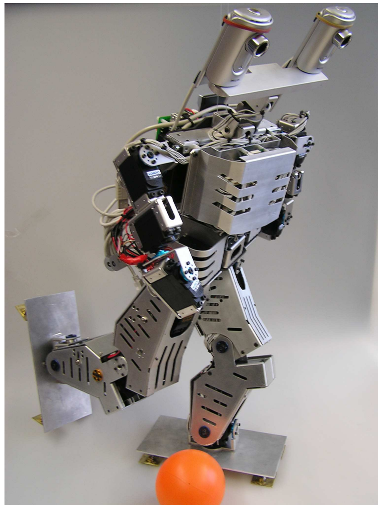

Abbildung 4.1.: Roboter iXs iHs04

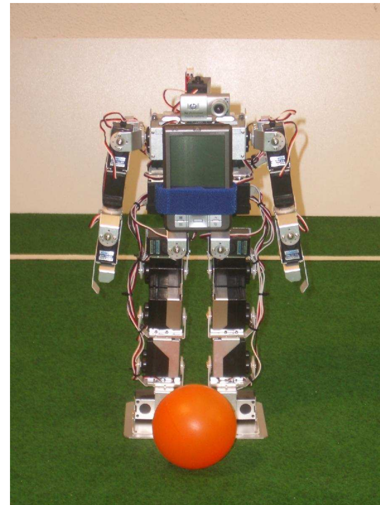

Abbildung 4.2.: Roboter Kondo KHR-1

4.2. Vorhandene Plattformen

Mit Hilfe des zu entwickelnden Frameworks sollen auf verschiedenen Plattformen Anwendungen entwickelt werden können. Hauptaugenmerk liegt dabei auf den zur Zeit am Fachgebiet vorhandenen *zweibeinigen Robotern*, daneben soll das Framework aber auch auf andere Hardware möglichst einfach portiert werden können.

4.2.1. iXs iHs04

Bei dem *iHs04* der iXs Research Corporation [7] handelt es sich um einen humanoiden Roboter mit 24 Freiheitsgraden (DOF) und einer Höhe von 53 cm, welcher in Abbildung 4.1 dargestellt ist.

Das Controller-Board verfügt über einen 64 Bit NEC Vr4181A MIPS Prozessor mit 133 MHz. Dieser Prozessor wurde ursprünglich für hand-held PCs entworfen und kommt beim iHs04 auf einem speziellen Board mit 32 MB Flash- und 32 MB Arbeitsspeicher zum Einsatz. Über eine CompactFlash Wireless LAN Karte sind Netzwerkverbindungen möglich, die Aktorik ist per USB mit dem Controller-Board verbunden.

Das Betriebssystem ist ein spezielles *Mipsel-Linux* mit LinuxThreads als Ersatz für POSIX. Als Compiler wird die GNU Compiler Collection (GCC) in Version 2.96 verwendet. Diese erlaubt das Erstellen von Anwendungen für den MIPS Prozessor von Intel-Systemen aus und ist weitestgehend ISO/EC 14882 kompatibel. Exceptions werden jedoch nicht unterstützt.

4.2.2. Kondo KHR-1

Der *Kondo KHR-1* wird als Bausatz mit handelsüblichen Servos geliefert und besitzt insgesamt 21 DOF. Anders als bei der Standard-Konfiguration wurden pro Bein sechs statt fünf DOF verwendet, um zusätzliche Möglichkeiten für die Bewegung des Fußes zu haben. Diese Bauform ist in Abbildung 4.2 dargestellt. Die Servos werden über zwei Controllerboards RCB-1 im 40-ms-Takt angesteuert, die 40 Trajektorien speichern und abspielen können. Daneben können über ein serielles RS-232C Interface auch Gelenkstellungen vorgegeben werden. Die verwendeten Servos erlauben auch ein Auslesen der aktuellen Gelenkstellung. Diese Daten können über den Controller abgefragt werden [9, 10].

Als Controller kommt ein Fujitsu Siemens Pocket LOOX 420 BTWL zum Einsatz, der über die serielle Schnittstelle mit den RCB-1 verbunden ist. Dabei handelt es sich um einen PDA mit 400 MHz Intel PXA255 Risc-Prozessor mit 32 Bit und ARM/XScale Architektur. Dieser Prozessortyp zeichnet sich durch hohe Leistung bei niedrigem Energieverbrauch aus. Intern verfügt der Pocket LOOX 420 über 32 MB ROM und 64 MB RAM, USB, eine Wireless LAN Netzwerkkarte, Bluetooth und ein Farbdisplay mit einer Auflösung von 240x320 Pixeln mit Touchscreen. Erweiterbar ist er durch einen MMC/SD-Steckplatz, über den eine CMOS Kamera mit einer Auflösung von 240 x 160 Pixeln bei 10 fps angesprochen werden kann.

Das Betriebssystem ist *Microsoft Windows Mobile 2003 Second Edition*, also Windows CE 4.2. Dabei handelt es sich um ein Echtzeitbetriebssystem. Es unterstützt jedoch nicht das volle Win32 API wie Windows XP oder Windows embedded. Der genaue Funktionsumfang ist herstellerspezifisch, der Gerätehersteller kann zum Beispiel aus Speicherplatzgründen auf Subsysteme verzichten. Die Programmierung kann sowohl nativ in C++ oder in .NET erfolgen. Bei Verwendung von .NET wird die entstandene Anwendung innerhalb der Common Language Runtime (CLR), der virtuellen Maschine von .NET, ausgeführt. Die CLR schränkt den Zugriff auf die Hardware und die Betriebssystemfunktionen aus Sicherheitsgründen ein. Für die Entwicklung von hardwarenahen Anwendungen ist daher die Nutzung von nativem Code, zumindest für die unteren Schichten, empfehlenswert, da nur hiermit auf das Echtzeitsystem von Windows CE zugegriffen werden kann.

Als Compiler für nativen Code muss Microsoft eMbedded Visual C++ 4.0 (eVC) mit ServicePack 3 oder höher und Pocket PC SDK verwendet werden (siehe auch Anhang B.4). Dieser ist nicht vollständig ISO/EC 14882 kompatibel, sondern nutzt einen C++ Dialekt, der ähnlich dem von Microsoft Visual Studio 5/6 ist. Die C++ Standard Stream Bibliothek steht nicht zur Verfügung, RTTI kann mit Hilfe eines Patches[1] nachgerüstet werden.

[1] Pocket PC RTTI Patch: `http://support.microsoft.com/kb/830482/`

Abbildung 4.3.: Roboter Pioneer 2-DX

4.2.3. Pioneer 2-DX

Beim *Pioneer 2-DX* (Abbildung 4.3) der Firma ActivMedia handelt es sich um einen dreirädrigen, vielseitig einsetzbaren, modularen Roboter. Das am Fachgebiet Simulation und Systemoptimierung verwendete Modell besitzt Rad-Encoder, ein Gyroskop, eine Kamera, 16 Ultraschall-Abstandssensoren und taktile Sensoren für die Kollisionserkennung.

Der Controller ist ein embedded-System mit P2OS Betriebssystem, der über eine serielle Schnittstelle mit einem weiteren Onboard-Rechner verbunden ist. Dieser Steuerungsrechner ist ein Standard-PC mit Suse Linux oder Windows 2000. Zur Kommunikation nach außen stehen sowohl ein LAN Anschluss als auch Wireless LAN zur Verfügung.

4.2.4. Entwicklungsplattformen

Neben den Robotern soll das Framework auch auf den am Institut vorhandenen Entwicklungsrechnern kompilierbar und lauffähig sein, um die Funktionsfähigkeit auch durch Simulationen oder Testfälle überprüfen zu können.

Unter Windows 2000/XP ist aufgrund des einheitlichen Win32 APIs keine differenzierte Betrachtung nötig. Unter Linux/Unix sind dagegen verschiedene Umgebungen anzutreffen. Je nach verwendeter Linux- (Suse, Debian, Gento, etc.), BSD-Distribution (FreeBSD, OpenBSD, etc.) oder Unix-Derivat (Solaris, AIX, etc.) kommen neben unterschiedlichen Compilern eventuell auch unterschiedliche Bibliotheken zum Einsatz, die unter Umständen einen unterschiedlichen Funktionsumfang implementieren können.

4.2.5. Weitere Plattformen

Ein weiterer potenzieller Anwendungsbereich ist der *Sony Aibo*. Dabei handelt es sich um einen ursprünglich als Spielzeug entwickelten vierbeinigen Roboter mit 64 Bit RISC Prozessor, eingebauter Kamera, Positionsencodern in jedem der 18 Gelenke, mehreren Tastern, Infarot-Abstandssensoren und Funknetzwerkkarte.

Als Betriebssystem kommt *Aperios* zum Einsatz, ein proprietäres objektorientiertes Echtzeitbetriebssystem. Zur Programmierung wird OPEN-R[2] verwendet. Dabei handelt es sich um eine Kombination eines seriellen Bussystems, an das alle Aktoren und Sensoren angeschlossen sind, und ein frei verfügbares API zum Ansprechen und Abfragen derselben.

Zum Kompilieren werden einige Unix-Kommandos, Perl und GNU make benötigt, was eine Unix-Umgebung voraussetzt. Soll Windows als Entwicklungsplattform verwendet werden, sind geeignete Emulatoren, zum Beispiel Cygwin[3], zu verwenden.

4.2.6. Zusammenfassung

Durch die Vielzahl an zu unterstützenden Plattformen muss das Framework sowohl für die interne Verwendung als auch für die eigentlichen Anwendungen zwingend eine Abstraktion der Plattform bieten. Dabei sollte nicht der kleinste gemeinsame Nenner bereitgestellt werden, sondern die nötige und sinnvolle Funktionalität. Nicht verfügbare Funktionalität soll dabei emuliert werden.

4.3. Debugging

Bei den funktionalen Einheiten (Bildverarbeitung, Selbstlokalisierung, etc.) kommen meist Algorithmen zum Einsatz, bei denen Parameter experimentell optimiert werden müssen, oder deren Struktur komplex und fehleranfällig ist. Gerade bei der Entwicklung von Steuerungssystemen für mobile Roboter wird viel Zeit für das Debuggen der Algorithmen verwendet.

Ein großer Teil des Erfolges des GermanTeams in der *Sony Four-Legged Robot League* ist auf die guten Entwicklungstools zurückzuführen. Durch die Möglichkeiten von *RobotControl*, mit dem alle Module auf dem Roboter überwacht und teilweise konfiguriert werden können, wird die Entwicklung neuer Algorithmen wesentlich beschleunigt.

[2]OPEN-R Homepage: `http://openr.aibo.com`
[3]Cygwin Homepage: `http://www.cygwin.com`

Neben der eigentlichen Anwendung soll daher auch die Erstellung einer *plattformunabhängigen graphischen Benutzeroberfläche* (GUI) möglich sein. Für neue Module soll es möglich sein, eigene Visualisierungsroutinen zu implementieren und diese in die GUI einzubinden. Standardmechanismen, wie zum Beispiel das Aufbauen einer Netzwerkverbindung, sollen dabei schon vorgefertigt sein, um den Programmierer zusätzlich zu entlasten.

4.4. Softwarequalitätskriterien

Gerade bei einem Framework sind neben den rein funktionalen Anforderungen auch andere Kriterien, wie zum Beispiel Erweiterbarkeit, Verständlichkeit und Wartbarkeit, wichtig. Das Qualitätsmodell ISO 9126, das als DIN 66272 in Deutschland umgesetzt ist, definiert sechs Merkmale, die jeweils weiter untergliedert sind. Diese Merkmale müssen schon während des Entwurfs des Systems beachtet und während der Implementierung durch geeignete Vorgehensmodelle und -methoden, wie zum Beispiel *Code-Reviews* und *Refaktorisierung*, umgesetzt werden.

Für das Framework sind, da es sich noch in einem frühen Entwicklungsstadium befindet, primär eine leichte Änderbarkeit, Erweiterbarkeit, Portabilität und Erlernbarkeit wichtig. So bleibt gewährleistet, dass das Framework leicht an zukünftig veränderte Anforderungen angepasst werden kann.

Die nach Pomberger identifizierten zehn Qualitätskriterien beeinflussen sich gegenseitig (Tabelle 4.1) in unterschiedlicher Intensität. Daneben wirken sich diese Kriterien zusätzlich auch auf die wirtschaftlichen Kosten (Tabelle 4.2) aus. Gut zu erkennen ist, dass sich gerade die Effizienz auf fast alle anderen Merkmale negativ auswirkt. Diese Beobachtung wurde bereits 1974 von Donald Knuth in [8] gemacht. Danach sind nur in 3% des Quellcodes Performanceoptimierungen sinnvoll, in den restlichen 97% erschweren sie dagegen die Wartung und eventuell nötige Fehlersuche. Optimierungen sollten darüber hinaus nur nach einer genauen Lokalisierung der kritischen Stellen vorgenommen werden, solange diese nicht gefunden sind gilt: *„Premature optimization is the root of all evil“* (Ebd. S.268).

	Korrektheit	Zuverlässigkeit	Adäquatheit	Erlernbarkeit	Robustheit	Lesbarkeit	Erweiterbarkeit	Testbarkeit	Effizienz	Portabilität
Korrektheit		+	0	0	+	0	0	0	0	0
Zuverlässigkeit	0		0	0	+	0	0	0	-	0
Adäquatheit	0	0		+	0	0	0	0	+	-
Erlernbarkeit	0	0	0		0	0	0	0	-	0
Robustheit	0	+	+	0		0	0	+	-	0
Lesbarkeit	+	+	0	0	+		+	+	-	+
Erweiterbarkeit	+	+	0	0	+	0		+	-	+
Testbarkeit	+	+	0	0	+	0	+		-	+
Effizienz	-	-	+	-	-	-	-	-		-
Portabilität	0	0	-	0	0	0	+	0	-	

Tabelle 4.1.: Wechselwirkungen der Softwarequalitätskriterien
Horizontale Merkmale wirken sich positiv (+), negativ (-) oder neutral (0) auf vertikale Merkmale aus (nach Pomberger).

	Entwicklungszeit	Lebenszeit	Entwicklungskosten	Betriebskosten	Wartungskosten	Übertragungskosten
Korrektheit	-	+	-	+	+	0
Zuverlässigkeit	-	+	-	+	+	0
Adäquatheit	-	0	-	+	+	-
Erlernbarkeit	-	0	-	+	0	0
Robustheit	-	+	-	+	+	0
Lesbarkeit	+	+	+	0	+	+
Erweiterbarkeit	-	+	+	0	+	+
Testbarkeit	+	+	+	0	+	+
Effizienz	-	+	-	+	-	-
Portabilität	-	+	-	-	+	+

Tabelle 4.2.: Auswirkungen der Qualitätskriterien auf die wirtschaftlichen Kosten
Die Merkmale (horizontal) wirken sich positiv (+), negativ (-) oder neutral (0) auf die Kosten (vertikal) aus (nach Pomberger).

4.5. Entwicklungsphasen

Aufgrund der vorgesehenen Verwendung des Frameworks bereits bei der *Internationalen RoboCup GermanOpen* im April 2005 wurde es schon während der Entwicklung genutzt. Daher mussten die wichtigsten Schnittstellen sehr früh festgelegt werden und sollten sich nach Möglichkeit nicht ändern. Hierbei wurde aus Zeitgründen die Realisierung des Steuerungsteils gegenüber der graphischen Oberfläche für das Debugging vorgezogen.

Für die im Juli 2005 stattfindende *RoboCup Weltmeisterschaft* wurden diese beiden Teile fertig gestellt und zusätzliche, RoboCup-spezifische Funktionalität hinzugefügt.

Als erste Plattform sollte der Robotertyp iXs iHs04 unterstützt werden, eine Portierung auf Windows CE für den Robotertyp Kondo KHR-1 stand zu Beginn der Arbeit noch nicht fest.

5. Konzept

Das Konzept zur Erstellung des Frameworks orientiert sich direkt an den in Kapitel 4 beschriebenen Anforderungen. Es ermöglicht dem späteren Entwickler, sich vollständig auf die eigentliche Anwendung konzentrieren zu können, ohne sich mit plattformspezifischen Details auseinandersetzen zu müssen. Das Framework *RoboFrame* ist in zwei Teile untergliedert: den Applikationsteil *RoboApp* und die graphische Benutzeroberfläche *RoboGui*.

Die funktionalen Komponenten, welche die datenverarbeitenden Schritte der Anwendung durchführen, werden im weiteren Verlauf *Module* genannt. Die Kommunikationswege zwischen den Modulen sollen durch das Framework nicht fest vorgegeben sein, wie das bei der Architektur des GermanTeams der Fall ist. Vielmehr soll jedes Modul nur definieren, welche Ein- und Ausgaben es hat, nicht aber explizit mit welchen anderen Modulen es kommuniziert. Die Kommunikation zwischen den einzelnen Modulen soll für diese völlig transparent abgewickelt werden.

Dadurch wird die Implementierung der Module unabhängig von der späteren Zuordnung auf eventuell mehrere Prozesse oder gar Computer. Diese logische Trennung erlaubt den flexiblen Austausch von Modulen mit gleichem Ein- und Ausgabe-Interface. Ebenso ist mit diesem Ansatz die geforderte Inter-Roboter-Kommunikation oder die Steuerung von außen ohne weitere Besonderheiten möglich.

Da die vorhandene Rechenleistung ohnehin sehr begrenzt ist und primär auch keine Mehrprozessorsysteme zum Einsatz kommen, wird weder auf CORBA noch auf andere Middleware zurückgegriffen.

5.1. RoboApp

Aus den Zielsetzungen in Kapitel 4 ergeben sich eine Reihe von konkreten Anforderungen an den Applikationsteil.

Die Kommunikationsverwaltung wird nicht zentral definiert, sondern flexibel durch die Ein- und Ausgabedaten der Module selbst spezifiziert. Dazu benötigt das Framework von jedem Modul Informationen über die angeforderten und bereitgestellten Daten.

Diese *deskriptiven Informationen* werden zur Laufzeit vom Framework von den Modulen abgefragt. Auf eine Spezifikation in externen Dateien wird, anders als bei CORBA, verzichtet, um eine zusätzliche Vorverarbeitung und Indirektion zu vermeiden. Die *dynamischen Kommunikationsverbindungen* erlauben es, die zugrunde liegende Architektur logisch von der eigentlichen Anwendung zu trennen. Gerade diese Kopplung stellt in der GT-Architektur eine erhebliche Einschränkung dar.

Einem Modul stehen grundsätzlich zwei verschiedene Konzepte für die Kommunikation zur Verfügung, nämlich entweder die Verwendung von Puffern oder die von Blackboards.

Über die *Puffer* können Nachrichten, welche durch einen eindeutigen Schlüssel identifiziert werden, von einem Modul zu einem oder mehreren anderen übertragen werden. Diese müssen im Deskriptor des Moduls dem Framework bekannt gemacht werden, daraufhin stellt das Framework zur Laufzeit die eigentliche Kommunikationsverbindung zwischen diesen Puffern her.

Beim Einfügen eines Elements in einen Puffer wird dieses automatisch mit einem *Zeitstempel* versehen, der bei der weiteren Verarbeitung allen Modulen zur Verfügung steht. Dies ermöglicht zum Beispiel die spätere Fusionierung von verschiedenen Sensorwerten.

Dagegen stellen die *Blackboards* einen globalen Speicherplatz bereit, auf welchen mehrere Module wechselseitig zugreifen können. Auch Einträge im Blackboard müssen über den Deskriptor beim Framework registriert werden.

Die Module selbst werden einem *Prozess* hinzugefügt, welcher durch das Framework bereitgestellt wird. Dieser ist in der Lage mehrere Module aufzunehmen und führt diese immer seriell aus. Dagegen laufen mehrere Prozesse generell asynchron zueinander und können dadurch auch mit unterschiedlichen Taktraten aufgerufen werden.

Zur Abwicklung der Kommunikation müssen beliebige, bei der Programmierung des Frameworks noch unbekannte Datenstrukturen in ein einheitliches Format konvertiert und wieder zurückkonvertiert werden können.

Als Beispiel betrachte man eine stark vereinfachte Standard-Roboter-Steuerung wie in Abbildung 5.1 dargestellt.

Eine mögliche Realisierung dieses Grundprinzips ist in Abbildung 5.2 dargestellt. Da die Bildverarbeitung und die Bewegungssteuerung unterschiedliche Anforderungen an den Aufruftakt haben, ist es sinnvoll, beide Module jeweils getrennten Prozessen hinzuzufügen. Des Weiteren wird das Modul der Verhaltenssteuerung jedoch demselben Prozess hinzugefügt, zu welchem auch die Bildverarbeitung gehört. Diese Entscheidung basiert auf der Annahme, dass eine Verhaltensentscheidung nur nach der Verarbeitung neuer Bilder notwendig beziehungsweise sinnvoll ist. Damit können

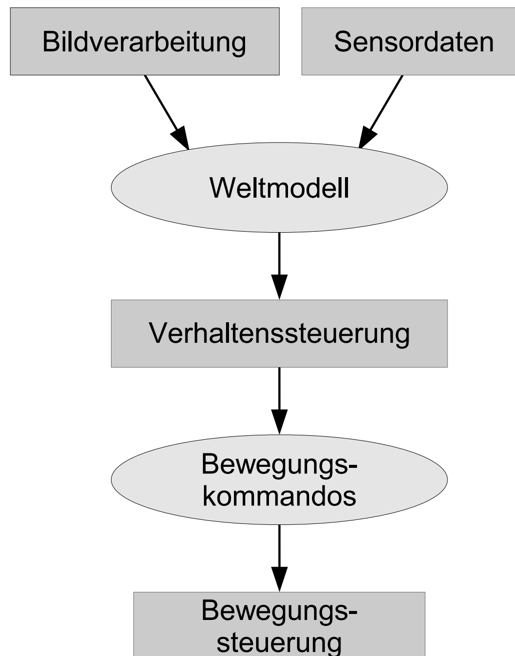

Abbildung 5.1.: Grundstruktur einer vereinfachten Roboter-Steuerung
Rechtecke bezeichnen die *funktionalen Komponenten*, Ellipsen stehen für die *ausgetauschten Informationen*.

beide Prozesse in verschiedenen Intervallen aufgerufen werden, was den unterschiedlichen Anforderungen der Module gerecht wird.

Der zentrale Bestandteil einer Anwendung auf Basis des Frameworks ist der *Router*, welcher die Prozesse verwaltet und außerdem für die gesamte Kommunikationsverwaltung zuständig ist. Des Weiteren stellt dieser Schnittstellen zur Verfügung, um mit anderen Routern via Netzwerk zu kommunizieren.

5.2. RoboGui

Eine weitere Anforderung an das Framework stellt die Möglichkeit dar, mittels einer graphischen Anwendung mit einem oder auch mehreren Anwendungen auf Robotern zu kommunizieren. Aufgrund der geforderten Plattformunabhängigkeit der graphischen Benutzeroberfläche wurde QT[1] der Firma Trolltech als Basis ausgewählt. Dabei handelt es sich um ein für mehrere Plattformen verfügbares GUI-Toolkit.

[1] QT Homepage: `http://www.trolltech.com`

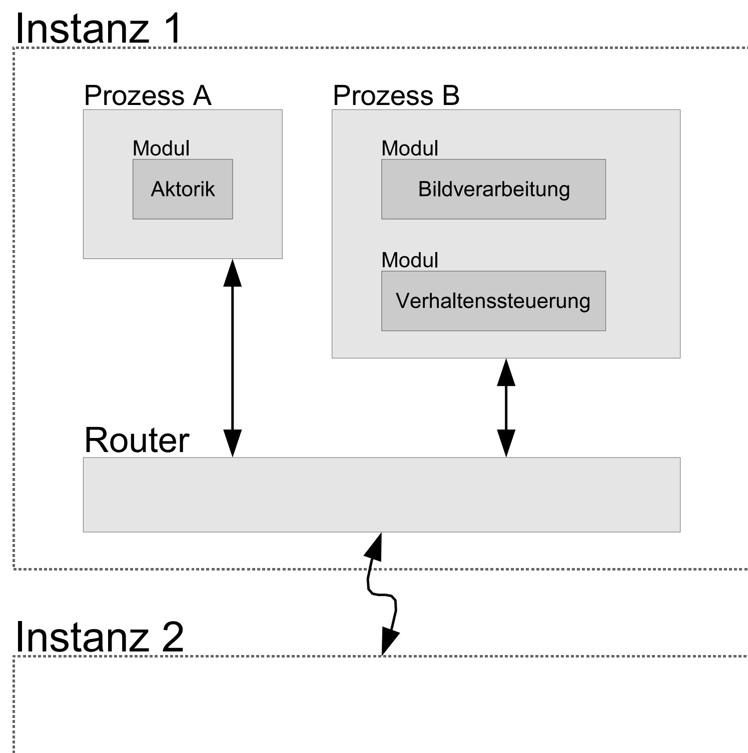

Abbildung 5.2.: Komponenten einer Framework-Instanz

Da einige Instanzen zum Beispiel auf dem Roboter selbst keine graphische Oberfläche benötigen, wurde RoboGui als Erweiterung auf Basis von RoboApp konzeptioniert. Dadurch ist auch bei der GUI die flexible Kommunikationsverwaltung integraler Bestandteil.

RoboGui stellt hierbei die Funktionalität zur Verfügung, sich mit beliebigen Instanzen verbinden zu können und mit diesen Nachrichten auszutauschen. Insbesondere können der GUI benutzerdefinierte Dialoge hinzugefügt werden. Damit ist es möglich, die graphische Benutzeroberfläche individuell mit anwendungsspezifischen Lösungen zu erweitern.

6. Realisierung

Die Implementierung des Frameworks ist in C++ und vollständig objektorientiert realisiert. Dabei wird sowohl von Templates und als auch der Standard-Template-Library (STL) reger Gebrauch gemacht. Außerdem wird Run-Time Type Information (RTTI) und Mehrfachvererbung eingesetzt, soweit dies erforderlich ist oder der Quellcode dadurch besser strukturiert wird. Auf den Einsatz externer Bibliotheken wurde bewusst verzichtet, um sich beim Design nicht einschränken zu müssen.

Die gesamte API des Frameworks ist direkt im Quellcode vollständig dokumentiert. Diese Dokumentation kann mit *doxygen*[1] generiert werden.

6.1. RoboApp

6.1.1. Plattformabstraktion

Das Framework soll aufgabengemäß auf verschiedenen, heterogenen Plattformen zum Einsatz kommen. Um sowohl eine plattformunabhängige Benutzung als auch die Portierung auf weitere Plattformen zu vereinfachen, wurde sämtlicher plattformabhängiger Quellcode an nur einer Stelle zusammengefasst, der sogenannten *Plattformabstraktionsschicht*. Alle Systemaufrufe und plattformabhängige Bibliotheken sind dazu an dieser Stelle gekapselt. Der Quellcode der Plattformabstraktionsschicht befindet sich im Framework im Verzeichnis *platform*. Die gesamte restliche Implementierung stützt sich ausschließlich auf die entsprechenden Klassen in diesem Verzeichnis, um Systemfunktionen aufzurufen.

Diese Schicht ist in einige funktionale Gruppen unterteilt wie in Abbildung 6.1 dargestellt. Dabei ist nur ein Teil der Plattformabstraktionsschicht plattformabhängig. Alle weiteren Komponenten setzen auf dieser Schicht auf und nutzen keinerlei plattformspezifischer Funktionalität. Die beiden plattformabhängigen Komponenten *BaseSystem* und *BaseThread* existieren zurzeit in jeweils drei Ausprägungen.

Die Klasse `SystemCall` ist für Systemaufrufe zuständig. Darunter fällt neben dem Ermitteln der Systemzeit und der Plattform auch die Ausgabe auf der Konsole. Da-

[1] doxygen Homepage: `http://www.doxygen.org`

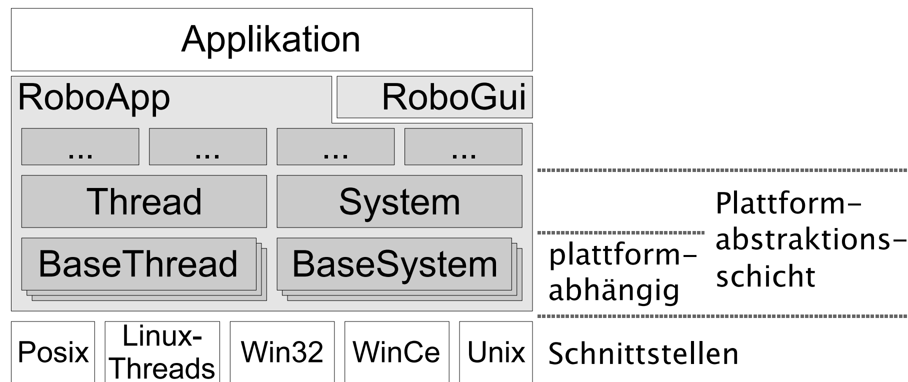

Abbildung 6.1.: Plattformabstraktionsschicht

bei ist zu beachten, dass es unter Windows CE keine Konsole im eigentlichen Sinne gibt. Daher besteht die Möglichkeit, bei diesem Objekt eine entsprechende Klasse `Console` zu registrieren, welche zum Beispiel unter Windows CE die Ausgaben in einen Dialog umleitet. Der plattformspezifische Quellcode wird dadurch eingebunden, dass abhängig von der verwendeten Plattform unterschiedliche Varianten der Klasse `BaseSystemCall` eingebunden werden, welche wiederum als Elternklasse von SystemCall dient. Dabei legt das Interface `IBaseSystemCall` fest, welche Methoden eine plattformabhängige Implementierung zur Verfügung stellen muss.

Falls eine solche plattformabhängige Klasse statische Methoden bereitstellen muss, sind diese als Kommentar im entsprechenden Interface vermerkt, auch wenn die Implementierung dort nicht vorgeschrieben werden kann.

Zur Verwaltung von Threads und den entsprechenden Synchronisationsmethoden wird auf die von modernen Betriebssystemen bereitgestellten Mechanismen [22] zurückgegriffen.

Die Klasse `Thread` ermöglicht es, Programmcode in eigenen Threads auszuführen. Unter Unix und Linux wird dies durch Posix-Threads realisiert und unter Windows wird die Windows-eigene Thread-API genutzt. Bei Mipsel-Linux kommen dafür spezielle LinuxThreads zum Einsatz, welche einen Vorgänger der Posix-Threads darstellen. Auch hier wird durch das Interface `IBaseThread` festgelegt, was die plattformabhängigen Klassen `BaseThread` implementieren müssen. Das Interface sieht es vor, eine eindeutige Kennung des aktuellen Threads zu ermitteln. Da diese Beschreibung plattformabhängig ist, wird in der `Thread`-Klasse innerhalb einer LookUp-Tabelle für jeden Thread eine synthetische Nummer verwaltet.

Des Weiteren abstrahiert das Thread-Interface auch von den unterschiedlichen Prioritäten der verschiedenen Thread-Bibliotheken und definiert dazu mehrere Konstanten für unterschiedliche Stufen. Die Umsetzung dieser plattformunabhängigen Konstanten in plattform-spezifische Thread-Prioritäten wird in der jeweiligen Implementierung ab-

gebildet. Damit können mehrere Threads mit unterschiedlichen Prioritäten ausgeführt werden, um den verschiedenen Anforderungen in einzelnen Anwendungsteilen gerecht zu werden.

Sobald Threads verwendet werden, ist es notwendig, diese entsprechend zu synchronisieren. Die Klasse `Condition` dient dazu, die Ausführung eines Threads anzuhalten und nach Empfang eines entsprechenden Signals fortzufahren. Dadurch wird während des Wartens eines Threads auf ein Ereignis keinerlei Rechenzeit verbraucht.

Die Klassen `Mutex` und `ReadWriteLock` ermöglichen es, den Zugriff auf geteilte Speicherbereiche zwischen mehreren Threads zu synchronisieren. Dies ist aufgrund der Nebenläufigkeit von Threads nötig, um zum Beispiel gleichzeitige Zugriffe auf gemeinsame Variablen zu unterbinden.

Ein Mutex muss durch die `lock`-Methode angefordert werden. Dies kann immer nur von einem Thread gleichzeitig geschehen, weshalb der Aufruf blockierend wirkt, falls bereits ein anderer Thread diese Methode aufgerufen hat. Nach dem zu synchronisierenden Abschnitt muss ein Mutex mit der Methode `unlock` wieder freigegeben werden.

Ein `ReadWriteLock` ermöglicht dagegen noch weitere Möglichkeiten der Nutzung. Anders als bei einem Mutex gibt es zwei verschiedene Methoden `acquireReadLock` und `acquireWriteLock`, um einen Lese- bzw. Schreibzugriff anzufordern. Dabei können zur gleichen Zeit mehrere Threads einen Lesezugriff gewährt bekommen oder lediglich einen Schreibzugriff. Die Semantik der Methode `releaseLock` entspricht der Methode des Mutexes, um den Zugriff wieder freizugeben.

Da das Win32-API keine ReadWriteLocks vorsieht, werden diese für diese Plattform innerhalb des Frameworks durch die in [1] vorgestellte Implementierung, der schreibende Zugriffe bevorzugt, realisiert. Dieser verwendet mehrere Mutexe und Zähler, um die Lese- und Schreibzugriffe zu synchronisieren.

Des Weiteren gibt es noch eine Klasse `Socket`, welche eine Netzwerkschnittstelle zur TCP- und UDP-basierten Kommunikation realisiert. Dabei unterscheidet sich die Art der Implementierung von den bisherigen Klassen der Plattformabstraktionschicht. Bei einem Socket gibt es eine ziemlich genaue Vorgabe, in welcher Reihenfolge einzelne Aufrufe erfolgen dürfen. Zum Beispiel muss ein Socket immer zuerst erstellt werden oder darf erst nach dem Binden an einen Port von diesem Nachrichten entgegen nehmen oder versenden. Diese Ablaufstruktur ist in einer *Zustandsmaschine* realisiert. Um diese nicht unnötigerweise in jeder plattformspezifischen Implementierung zu wiederholen, ist die Zustandsmaschine bereits in der plattformunabhängigen Klasse implementiert.

Um ein aktives Warten auf neue Daten zu verhindern, können Sockets im *Multiplexed-IO Modus* [21] betrieben werden. In diesem Modus wird der entsprechende Thread während des Wartens blockiert, beim Eintreffen neuer Daten oder nach einem

Timeout wird die Ausführung des Threads fortgesetzt. Dies reduziert, im Gegensatz zu aktivem Warten im nicht-blockierenden I/O Modus, die Prozessorlast.

Im paketorientierten UDP Betrieb eines Sockets muss diesem das Versenden von Broadcast-Nachrichten explizit durch Aufruf der Methode `setBroadcast` erlaubt werden. Danach ist über eine entsprechende Zieladresse der Versand eines Datenpakets an mehrere Empfänger möglich.

Die letzte Komponente der Plattformabstraktionsschicht ist die Klasse `File`, welche den Zugriff auf Dateien ermöglicht. Dabei wird zwischen dem lesenden, schreibenden und anhängenden Zugriff auf Text- und Binärdateien unterschieden. Neben den eigentlichen Methoden für die Dateizugriffe, welche auf den unterschiedlichen Plattformen nahezu gleich realisiert werden konnten, stellt diese Klasse allerdings auch Funktionalität zur Verfügung, um mit relativen und absoluten Pfaden im Dateisystem arbeiten zu können.

Dabei ist zu beachten, dass ein *aktuelles Verzeichnis* unter Windows CE nicht definiert ist. Deshalb müssen relative Pfadangaben bei dieser Plattform immer in absolute Angaben umgewandelt werden, indem der absolute Pfad der aktuell ausgeführten Anwendung ermittelt wird und von dort aus eine relative Adressierung erfolgt.

6.1.2. Schlüssel

Schlüssel dienen dazu, Nachrichten einer Anwendung eindeutig zu identifizieren. Dieser Schlüssel ist eine eindeutige Zahl vom Typ `Key` und muss zu Beginn bei der `KeyRegistry` registriert werden. Das Framework ermöglicht daraufhin, dass Nachrichten einer bestimmten Datenquelle zu allen Datensenken mit demselben Schlüssel weitergeleitet werden. Die KeyRegistry ermöglicht an zentraler Stelle die Prüfung sämtlicher registrierter Schlüssel auf Eindeutigkeit.

Zusätzlich wird zu jedem Schlüssel auch die dahinter stehende Datenstruktur mit angegeben. Dadurch kann bei der weiteren Benutzung sichergestellt werden, dass sich hinter einem Schlüssel auch die eine korrekte Datenstruktur befindet. Bei der Realisierung dieser Funktionalität ergibt sich die Problematik, dass die Angabe eines *Typnamens* als Funktionsparameter in C++ nicht möglich ist. Daher wird nicht die Klasse selbst, sondern ein *Wrapper-Template* mit der Klasse als einzigem Argument übergeben. Somit lässt sich der Typ der Template-Klasse später mit einem anderen vergleichen, indem die Laufzeit-Typinformationen beider Objekte genutzt wird. Sollte ein Schlüssel in Verbindung mit einer anderen Datenstruktur, als er zuvor bei der KeyRegistry registriert wurde, verwendet werden, erkennt das Framework diese fehlerhafte Nutzung und gibt eine entsprechende Warnung aus.

Bei dem Festlegen dieser Schlüssel sollte man nicht auf die nahe liegende Definition als `static const` zurückgreifen, denn der Compiler für Windows CE erlaubt keine

direkte Deklaration der Konstanten in Header-Dateien. Ein Deklarieren der Konstanten innerhalb der dazugehörigen *cpp*-Datei ist allerdings nicht mehr ausreichend, um diese Schlüssel als konstante Ausdrücke in zum Beispiel Template-Parametern zu benutzen. Deshalb bleibt aus Gründen der Plattformunabhängigkeit nur die Möglichkeit, die Schlüssel mittels `#define` als symbolische Konstante vom Präprozessor ersetzen zu lassen.

Sowohl RoboApp als auch RoboGui benutzen diese Schlüssel und daher müssen diese sowohl in der Applikation als auch in der GUI vorab bei der KeyRegistry registriert werden. Deshalb ist es sinnvoll, sämtliche Schlüssel innerhalb einer Datei zu definieren, wodurch diese an den entsprechenden Stellen einfach eingebunden werden kann.

6.1.3. Module

Modul-Klassen bilden die eigentlichen funktionalen Komponenten einer Anwendung auf Basis des Frameworks und erben von der Klasse `Module`. Das Framework benötigt für die Kommunikationsverwaltung von einem Modul Informationen über die auszutauschenden Datenstrukturen, unter anderem die Schlüssel aller Datenquellen und Datensenken des Moduls. Diese liefert das Modul durch die Methode `getModuleDescriptor`, indem dort sämtliche Klassenvariablen vom Typ `In-/OutBuffer` und `BlackBoard` übergeben werden. Auf diese Strukturen und ihre Bedeutung in der Kommunikationsverwaltung wird in Abschnitt 6.1.4 genauer eingegangen. Zusätzlich kann in dieser Methode auch der Name des Moduls festgelegt werden, um eine textuelle Beschreibung zu ermöglichen.

Des Weiteren gibt die Elternklasse einige Methoden vor, welche im *Lebenszyklus* eines Moduls durch den umgebenden Prozess aufgerufen werden.

In Abbildung 6.2 ist der Lebenszyklus eines Moduls dargestellt. Da Klassenvariablen erst während der Konstruktion des Moduls angelegt werden, kann der Modul-Deskriptor durch den Prozess erst danach abgefragt werden, da er Referenzen auf einige der Variablen verwendet. Das Framework benötigt diese Informationen allerdings, um die Puffer und Blackboards vollständig zu initialisieren, was jedoch innerhalb des Konstruktors noch nicht gewährleistet werden kann. Deshalb dient die `init`-Methode dazu, die Initialisierung des jeweiligen Moduls nach dem Aufruf des Konstruktors abzuschließen, denn erst zu diesem Zeitpunkt ist durch das Framework sichergestellt, dass alle Klassenvariable vom Modul verwendet werden können.

Nach der Erstellung einer Modul-Instanz wird `init` vor sämtlichen anderen Methoden außer `getModuleDescriptor` aufgerufen. Das Gegenstück dazu ist die Methode `cleanUp`, welche vor dem Destruktor ausgeführt wird, um zum Beispiel allokierte Speicherbereiche wieder freizugeben. Die beiden Methoden `start` und `stop`

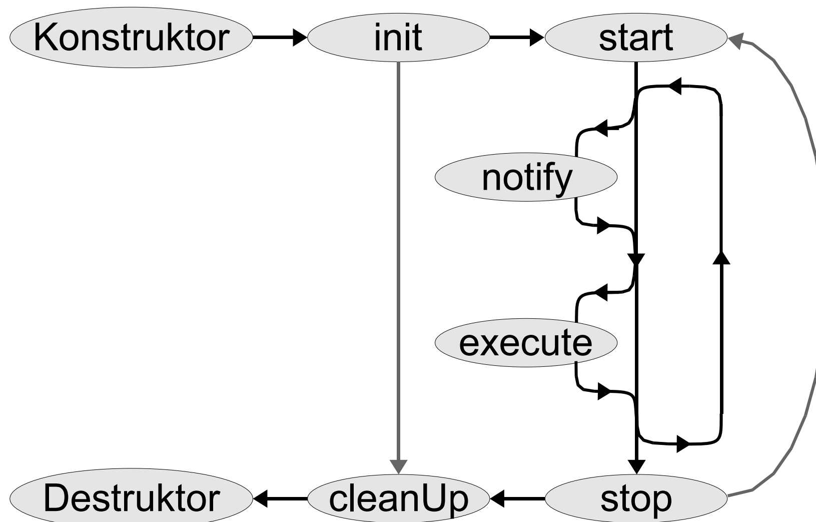

Abbildung 6.2.: Lebenszyklus von Modulen

werden vom umgebenden Prozess aufgerufen, bevor der Prozess gestartet bzw. nachdem dieser gestoppt wurde. Dabei kann ein Modul nach der Initialisierung entweder *gar nicht*, *genau einmal* oder auch *mehrmals* gestartet und wieder gestoppt werden.

Über die Methode `notify(Key)` wird das Modul darüber informiert, dass ein neues Datenobjekt mit dem angegebenen Schlüssel in dem dazugehörigen Puffer gespeichert wurde. Der Rückgabewert der Methode legt fest, ob im Anschluss die Methode `execute` des Moduls aufgerufen wird (`true`) oder nicht (`false`). Innerhalb dieser Methode sollten keine langwierigen Berechnungen durchgeführt werden, damit ein entsprechender Aufruf durch das Framework schnell durchgeführt werden kann. Sofern `notify` nicht überschrieben wird, liefert die Implementierung der Elternklasse stets `true` zurück.

Die wichtigste Methode eines Moduls lautet `execute`. Dort werden die eigentlichen datenverarbeitenden Schritte durchgeführt. Die Elternklasse sieht hierbei einen Parameter vom Typ `ExecuteFlags` vor, welches dem Modul signalisiert, weshalb es aufgerufen wurde. Die ExecuteFlags sehen dabei drei Unterscheidungen für die Ursache des `execute`-Aufrufes vor:

- Das Modul (oder aber ein anderes Modul innerhalb desselben Prozesses) hat neue Daten erhalten, auf welche in der `notify`-Methode mit `true` reagiert wurde.
- Für den Prozess wurde ein zeitbasiertes Intervall ausgewählt und dieser Zeitpunkt ist gerade eingetreten.

- Für den Prozess wurde ein zeitbasiertes Intervall ausgewählt und dieser Zeitpunkt liegt bereits in der Vergangenheit. Dies kann geschehen, wenn eine vorherige Ausführung der Module dieses Prozesses länger gedauert hat als durch dessen Zeitintervall festgelegt.

6.1.4. Modul-Kommunikation

Die `Buffer`- und `BlackBoard`-Klassen stellen die Schnittstellen für die Modul-Kommunikation dar. Damit auf die einzelnen Datenelemente typsicher zugegriffen werden kann, was ein sehr wichtiges Kriterium bei der Benutzung darstellt, sind diese Komponenten als Templates realisiert. Neben der Datenklasse haben diese Klassen auch den Schlüssel als Template-Parameter. Diese Kombination muss, wie bereits erwähnt, der Registrierung des Schlüssels bei der KeyRegistry entsprechen.

Beide Komponenten dienen dem Austausch von Datenobjekten, aber sie sind für zwei unterschiedliche Einsatzszenarien ausgelegt.

Puffer

Die Puffer stellen eine Struktur zur Verfügung, in der Datenobjekte in einem Ringpuffer vorgehalten werden. Daher muss hierbei zusätzlich zu den beiden genannten Parametern noch die maximale Anzahl der Elemente mit angegeben werden. Bei den Puffern werden zwei Varianten unterschieden: zum einen die `OutBuffer`, auf welche ausschließlich schreibend zugegriffen werden kann, und zum anderen die `InBuffer`, welche dagegen nur lesenden Zugriff erlauben. Dies ermöglicht den Nachrichtenaustausch eines Senders mit mehreren Empfängern.

Die Datenelemente liegen dabei nicht direkt in den `In-/OutBuffern` selber. Beide Puffervarianten sind lediglich Proxies [5], welche sämtliche Zugriffe an einen darunterliegenden `RingBuffer` weiterleiten. Somit können mehrere Puffer mit dengleichen Schlüsseln innerhalb eines Prozesses denselben Ringpuffer nutzen. Dies ist möglich, da sämtliche Module eines Prozesses sequentiell ablaufen und damit niemals nebenläufig auf den Ringpuffern arbeiten. Auch Ringpuffer mit einer unterschiedlichen Anzahl an Elementen werden hierbei zusammengeführt, indem der verwendete Speicher auch für den größten Puffer ausreichend dimensioniert wird. Dennoch liefert jeder Proxy eine auf seine Größe abgestimmte Sicht auf den Ringpuffer. Durch dieses Zusammenführen der Datencontainer wird unnötiges Kopieren der Datenobjekte im Speicher vermieden. Der umgebende Prozess trägt dafür Sorge, dass die von den Proxies benutzten Ringpuffer bereitgestellt werden. Abbildung 6.3 zeigt die Trennung der Ringpuffer von den in Modulen verwendeten Proxies.

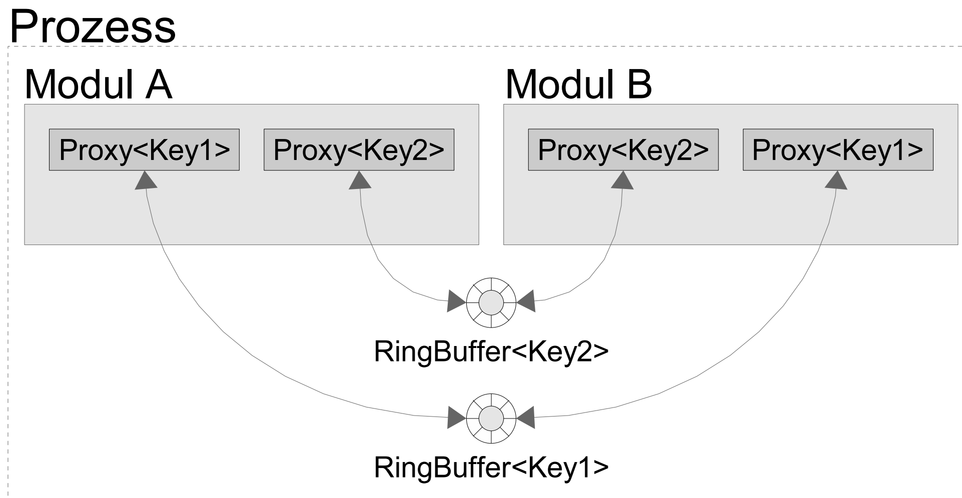

Abbildung 6.3.: Trennung von Puffern und Ringpuffern

Liegen zwei Puffer allerdings in verschiedenen Prozessen, müssen die Datenstrukturen bei der Übertragung durch das Framework von einem OutBuffer zu einem anderen InBuffer kopiert werden. Dabei werden die Datenobjekte aus einem Ringpuffer serialisiert, über Queues zwischen den Prozessen übertragen und in einen anderen Ringpuffer wieder deserialisiert. Durch die getrennte Verwaltung der Daten in unterschiedlichen Prozessen kann durch die Puffer ohne aufwendige Synchronisierung jederzeit auf sämtliche Pufferelemente zugegriffen werden. Eine entsprechende Beispielanwendung ist in Abbildung A.1 dargestellt.

Der `InBuffer` stellt zusätzlich noch Funktionen zur Verfügung, um zu ermitteln, welche Datenelemente neu sind und noch nicht abgefragt wurden. Um dies auch zu ermöglichen, wenn derselbe Ringpuffer von mehreren Puffer-Proxies benutzt wird, werden diese Informationen in den Proxy-Instanzen selbst verwaltet. Dazu registrieren sich die Proxies bei den darunterliegenden Ringpuffern als Observer [5] und werden von diesen über neue Datenelemente notifiziert, um ihre eigenen Zustandsdaten aktualisieren zu können. Dieser Notifizierungsmechanismus wird ebenfalls dazu genutzt, um beim Hinzufügen neuer Daten zum Ringpuffer die entsprechenden Module per `notify`-Aufruf darüber zu informieren.

Das zugehörige Klassendiagramm der Puffer ist in Abbildung 6.4 dargestellt. Im linken Teil sind alle Proxyklassen dargestellt und im rechten Teil sieht man die abgebildeten Datencontainer, welche von den Proxies benutzt werden. Ein Modul benutzt dabei direkt die `In-/OutBuffer`, während ein Prozess die dazugehörigen Interfaces oder die Klasse `IStreamBuffer` verwendet.

Sämtliche Pufferklassen, welche von den Modulen selbst benutzt werden, sowie der darunterliegende Ringpuffer, sind Template-Klassen, um einen typsicheren Zugriff zu ermöglichen. Innerhalb des Frameworks selbst kann dagegen nicht mit den typisier-

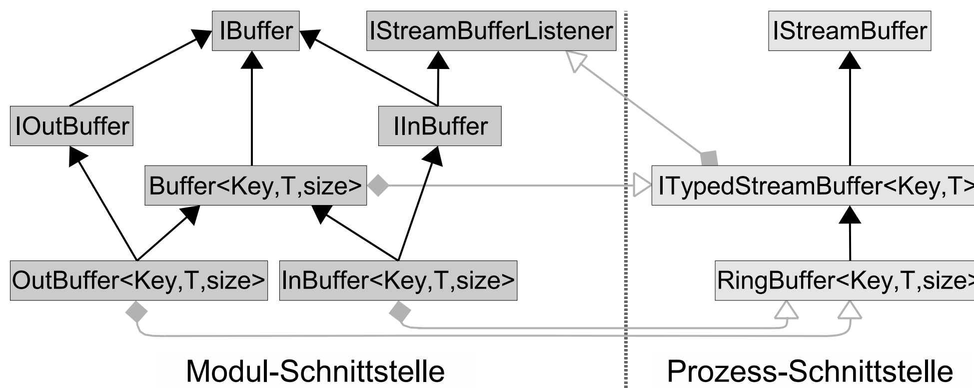

Abbildung 6.4.: Klassendiagramm der Puffer

ten Puffern gearbeitet werden, da deren Klassen zum Kompilier-Zeitpunkt des Frameworks noch nicht bekannt sind. Daher gibt es zu jeder Template-basierten Pufferklasse noch eine Elternklasse, welche den Zugriff aus Sicht des Prozesses ermöglicht jedoch kein Template darstellt. Des Weiteren wird ein Teil der Implementierung der beiden Pufferklassen in einer weiteren Elternklasse `Buffer` zusammengefasst, um Codeduplikationen zu vermeiden.

Als Zugriffsmethoden sieht der `OutBuffer` zwei verschiedene Varianten vor, um neue Datenobjekte hinzuzufügen. Bei der Methode `add(const T&)` wird das hinzuzufügende Datenobjekt als konstante Referenz übergeben. Allerdings erfordert das Speichern innerhalb des Ringpuffers das Kopieren des Datenobjektes. Als optionalen zweiten Parameter kann man der Methode noch einen Zeitstempel mitgeben, welcher den Zeitpunkt beschreibt, an dem das Datenobjekt erzeugt wurde. Falls auf diese Angabe verzichtet wird, wird als Zeitstempel die aktuelle Systemzeit verwendet.

Die zweite Methode mit der Signatur `T* add()` ermöglicht das Hinzufügen eines Datenobjektes ohne dieses dabei kopieren zu müssen. Durch diese Methode erhält man einen Verweis auf ein Datenobjekt, welches vom Modul entsprechend beschrieben werden kann. Dabei ist zu beachten, dass sich das referenzierte Objekt in einem beliebigen Zustand befinden kann und daher sämtliche Klassenvariablen gesetzt werden sollten. Im Anschluss daran muss die `finish()`-Methode des Puffers aufgerufen werden, um zu signalisieren, dass der Zugriff auf das Datenobjekt beendet ist. Auch bei dieser Methode besteht die Möglichkeit, durch einen optionalen Parameter einen eigenen Zeitstempel zu setzen. Nach dem Aufruf der `finish`-Methode darf der Verweis auf das Datenobjekt nicht mehr verwendet werden.

Der `InBuffer` stellt mit der Methode `const T& get(unsigned short)` die Möglichkeit bereit, Elemente aus dem Puffer abzurufen. Durch die Rückgabe einer Referenz wird das zeitaufwändige Duplizieren des Datenobjektes vermieden. Der optionale Parameter der Methode bezeichnet den Index des Elements in dem Puffer, wobei der Wert 0 das neuste Datenobjekt adressiert. Analog kann mit der Methode

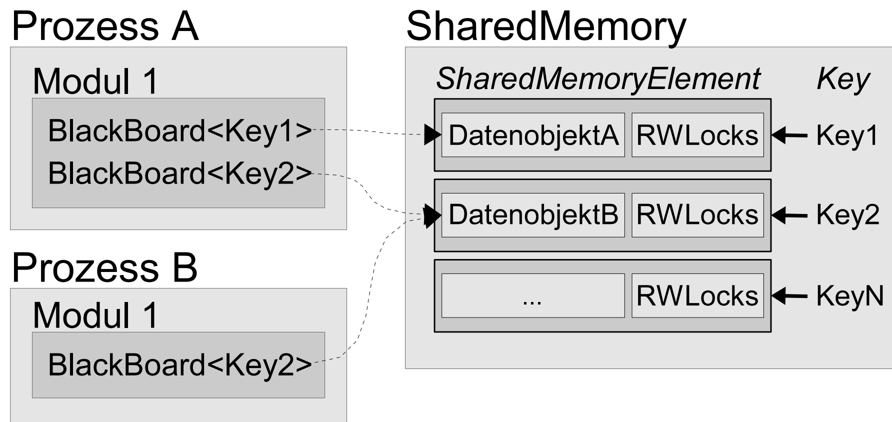

Abbildung 6.5.: Struktur des geteilten Speicherbereichs

`bool isNew(unsigned short)` abgefragt werden, ob ein Element bereits abgerufen wurde. Zusätzlich stehen Methoden bereit, um über alle neuen Datenobjekte iterieren zu können.

Geteilte Speicherbereiche

Im Gegensatz zum Puffer dient ein `BlackBoard` dazu, mehreren Modulen asynchron zueinander lesenden und schreibenden Zugriff auf ein Datenobjekt zu ermöglichen. Dabei handelt es sich, wie schon bei den Puffern, um einen Proxy, der allerdings auf einen geteilten Speicherbereich zugreift, der von allen Prozessen genutzt wird. Dieser wird durch die Klasse `SharedMemory` bereitgestellt und ermöglicht es, auf dieselben Daten von unterschiedlichen Threads aus zuzugreifen und damit jegliches Kopieren der Datenobjekte zu vermeiden. Die Struktur ist aus Abbildung 6.5 ersichtlich.

Dieses Konzept erfordert, dass Lese- und Schreibzugriffe entsprechend synchronisiert werden müssen und macht einen wesentlich höheren Aufwand beim Zugriff durch die Module notwendig. Daher beinhaltet jedes `SharedMemoryElement` neben dem eigentlichen Datenobjekt noch eine Variable vom Typ `ReadWriteLock`. Diese ermöglicht, dass mehrere Aufrufer gleichzeitig lesend oder einer schreibend auf das Datenobjekt zugreifen können. Allerdings muss vor dem Zugriff auf das Datenobjekt zuerst vom Modul selbst der Zugriff mittels der Methoden `acquireReadLock` oder `acquireWriteLock` angemeldet werden. Diese Aufrufe sind blockierend, bis der angeforderte Zugriff gewährt werden kann. Entsprechend existiert eine Methode `releaseLock`, welche nach der Interaktion mit dem Datenobjekt den Zugriff wieder freigibt. Um die Wartezeit beim Zugriff auf ein Element zu minimieren, sollten die Bereiche zwischen den Aufrufen von `acquireReadLock` beziehungsweise `acquireWriteLock` und `releaseLock` möglichst schnell abgearbeitet werden können.

Vom `SharedMemoryElement` selbst kann programmiertechnisch nicht sichergestellt werden, ob die beschriebenen Zugriffsrichtlinien eingehalten werden. Ein Modul kann das Datenobjekt auch nach der Freigabe weiterverwenden, was zu inkonsistenten Speicherinhalten führen könnte. Da sich lesende und schreibende Zugriffe gegenseitig ausschließen, darf man niemals beide `acquire`-Methoden in Folge aufrufen, ohne dazwischen den Zugriff wieder freizugeben. Dies würde zwangsläufig zu einem endlos blockierenden Programmteil - *Deadlock* [22] genannt - führen.

Bei der Verwendung von BlackBoards in mehreren Instanzen kann der Austausch der Datenobjekte durch das Framework nicht realisiert werden. Da in beiden Instanzen auf ein Datenobjekt sowohl lesend als auch schreibend zugegriffen werden kann, müssten die Datenobjekte instanzübergreifend synchronisiert werden.

Ein möglicher Ansatz dies zu realisieren wäre, die Änderungen der Datenobjekte an sich als sogenannte *Deltas* zu verschicken. Diese müssten dann auch bei den anderen Instanzen auf dem Datenobjekt durchgeführt werden. Dieses Konzept kann durch das Framework allerdings nicht umgesetzt werden, da es dazu detaillierte Informationen über die Struktur der zugrunde liegenden Datenobjekte benötigen würde. Auf Modulebene könnte dagegen darauf geachtet werden, dass sämtliche Operationen auf einer Datenstruktur, welche in einem BlackBoard gespeichert ist, durch ein Datenobjekt selbst repräsentiert werden. Damit wäre es dann möglich, die Datenobjekte der durchgeführten Operationen mit anderen Instanzen auszutauschen. Diese müssten dann sämtliche Operationen auf das Datenobjekt im BlackBoard anwenden, um alle verteilt durchgeführten Schreiboperationen zu integrieren.

6.1.5. Datenobjekte und Nachrichtenschlangen

Die `StreamedData`-Klasse bildet die Framework-interne Datenstruktur für Nachrichten und repräsentiert ein serialisiertes Datenobjekt. Ein `StreamedData`-Objekt enthält neben den serialisierten Daten auch Informationen über den Schlüssel, den Zeitstempel, die Herkunft und die Größe des Datenobjektes. Diese sind im sogenannten StreamedData-Header zusammengefasst.

Um benutzerdefinierte Datenstrukturen übertragen zu können, müssen diese zunächst in StreamedData-Objekte konvertiert werden und auch daraus zurückkonvertiert werden können. Daher müssen benutzerdefinierte Datenstrukturen in dieses Format überführt und auch daraus zurückkonvertiert werden. Anders als in Java ist es in C++ nicht inhärent möglich, Objekte zu serialisieren.

Das Konzept der Serialisierung und Verwaltung der serialisierten Daten ist an die Vorgehensweise der Boost-Bibliothek[2] angelehnt.

[2]Boost Homepage: `http://www.boost.org`

Die Funktionalität zur Konvertierung wird durch den Operator << sowie durch den Operator >> bereitgestellt. Das Framework erfordert, dass sich jede auszutauschende Datenstruktur sowohl aus einem StreamedData-Objekt - welches einen `InStream` darstellt - deserialisieren, als auch in einen `OutStream` serialisieren lässt. Dabei muss die Datenstruktur in der Methode `operator >> (OutStream&) const` sämtliche Klassenvariablen in den Stream schreiben beziehungsweise in der Methode `operator << (InStream&)` aus dem Stream lesen. Um den Aufwand möglichst gering zu halten, stellt das Framework innerhalb der Stream-Klassen Streamingoperatoren zur Verfügung, um sowohl alle primitive Datentypen als auch einige STL-Container in Streams zu schreiben und daraus zu lesen.

Nachdem eine Datenstruktur in diese generische Form transformiert wurde, kann diese durch sogenannte Nachrichtenschlangen, welche durch die Klasse `StreamedDataQueues` realisiert werden, durch das Framework verschickt werden. Diese *Indirektion der Nachrichtenweiterleitung* ist notwendig, da die einzelnen Komponenten des Frameworks in eigenen Threads laufen und somit nicht unmittelbar per Methodenaufruf miteinander interagieren können. In der Regel verbindet eine Queue zwei Komponenten aus verschiedenen Threads. Dabei ist die Datenübertragung unidirektional, d.h. es gibt einen dedizierten Sender und entsprechend einen Empfänger.

Eine StreamedDataQueue kann mehrere StreamedData-Instanzen aufnehmen. Um den Speicherbedarf zu minimieren, wird der Speicher nicht am Stück allokiert, sondern erst bei Bedarf in einzelnen Teilen. Diese Speicherblöcke werden durch die Klasse `Chunk` dargestellt und befinden sich in der Regel nicht zusammenhängend im Speicher. Diese Struktur ist in Abbildung 6.6 kurz skizziert.

Durch diese Fragmentierung kann es beim Befüllen einer Queue mit einem Datenobjekt ebenfalls dazu kommen, dass die serialisierten Daten in mehr als einem Chunk zu liegen kommen. Um die Daten auch bei fragmentierter Speicherung wieder deserialisieren zu können, werden auch hier, wie bei einer Queue, zusätzlich Verweise auf den oder die Speicherbereiche gespeichert, in denen sich die serialisierten Daten befinden. Auch hierbei wird, wie bei der Queue selbst, die `Chunk`-Klasse genutzt. In dieser werden Pointer auf den verwiesenen Speicherbereich und die Größe gespeichert. Dabei ist es möglich, den Speicher entweder extern zu allokieren und nur an einen Chunk zu übergeben, oder aber diese Aufgabe dem Chunk selbst zu überlassen. Dadurch wird die Verwaltung der Speichersegmente innerhalb von der StreamedData-Klasse und einer Queue vereinheitlicht.

Falls die Datenobjekte auf der Empfängerseite nicht schnell genug aus der Queue entnommen werden, wächst diese nicht beliebig an, sondern nur bis zu einer maximalen Größe. Die notwendigen Verwaltungsinformationen über die aktuell enthaltenen Daten zählen hierbei nicht zur maximalen Größe, sondern belegen zusätzlichen Speicher. Um einen reibungslosen Ablauf zu gewährleisten und die Queue in einem konsistenten Zustand zu belassen, auch wenn der benötigte Speicher die maximale Kapazität

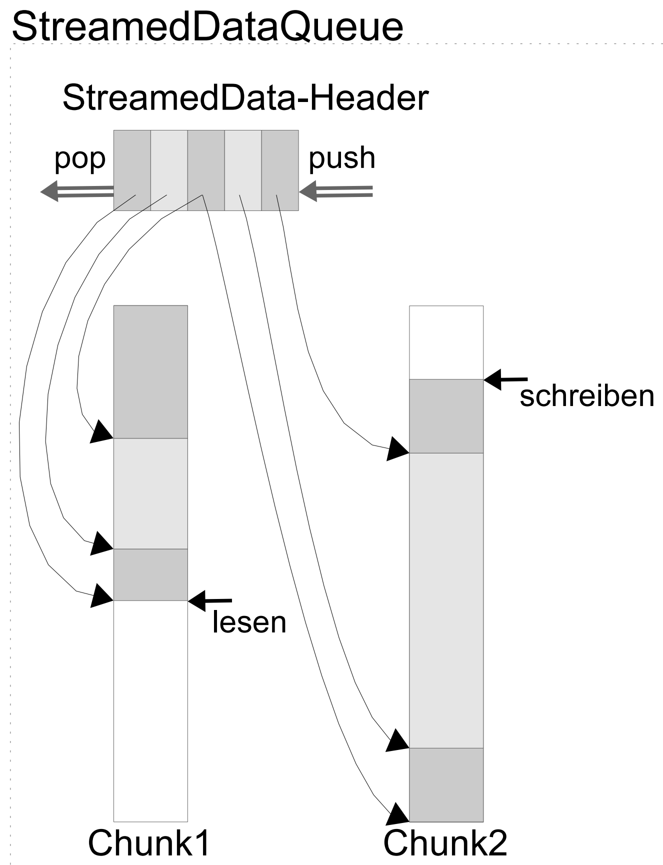

Abbildung 6.6.: Struktur innerhalb einer Nachrichtenschlange
Jeder StreamedData-Header innerhalb der Queue hält Verweise auf seine Datenbereiche in den Chunks

einer Queue übersteigt, sind *Schreiboperationen* auf einer StreamedDataQueue immer *transaktionssicher*. Das heißt, dass bereits begonnene Schreiboperationen rückgängig gemacht werden, wenn diese nicht vollständig durchgeführt werden konnten. Dadurch geht das Datenobjekt, welches der Queue hinzugefügt werden sollte, verloren. Dieses Ereignis bewirkt eine entsprechende Warnung als Ausgabe.

Damit der Empfänger einer Queue nicht aktiv auf neue Elemente in der Nachrichtenschlange warten muss, kann sich eine Klasse vom Typ `QueueListener` bei der Queue als Observer [5] registrieren. Sobald neue Daten in die Queue geschrieben wurden, wird der registrierte Empfänger durch einen Methodenaufruf benachrichtigt.

6.1.6. Prozesse

Die `Process`-Klasse bildet die Laufzeitumgebung für Module. Dabei können einem Prozess beliebig viele Module hinzugefügt werden. Die Prozess-Klasse erbt von der in Abschnitt 6.1.1 beschriebenen Klasse `Thread`, womit sämtlicher Code dieser Klasse und auch alle dazugehörigen Module innerhalb eines eigenen Threads laufen. Daher kommuniziert ein Prozess mit dem Router über zwei Nachrichtenschlangen, welche in Abschnitt 6.1.5 beschrieben wurden.

Wie auch die Klasse `Module` besitzt ein Prozess die Methoden `init`, `start`, `stop` und `cleanUp`, welchen dieselbe Semantik eines Lebenszykluses zugrunde liegt, und durch den in Abschnitt 6.1.8 beschriebenen Router aufgerufen werden.

Der Prozess benötigt von jedem Modul Informationen über die verwendeten Puffer und BlackBoards, welche durch den entsprechenden Moduldeskriptor abgefragt werden können. Diese Daten werden vom Prozess innerhalb der Methode `init` angefordert und dazu verwendet, die notwendigen Ringpuffer für die Puffer-Proxies anzulegen. Da der Ringpuffer eine Template-Klasse ist und, wie bereits zuvor erwähnt, das Framework keine Kenntnisse von benutzerdefinierten Datenstrukturen hat, kann der Prozess diese Datenspeicher nicht selbst anlegen. Daher benutzt dieser eine Methode des Puffer-Proxies, um einen Ringpuffer anlegen zu lassen. Damit die Größe des erzeugten Ringpuffers ausreichend ist, um allen Proxies gerecht zu werden, wird der jeweils größte Puffer eines bestimmten Schlüssels vom Prozess für die Erstellung des Ringpuffers ausgewählt. Danach übergibt der Prozess allen anderen Proxies desselben Schlüssels einen Verweis auf den angelegten Ringpuffer. Analog dazu wird auch die Initialisierung der BlackBoards durchgeführt.

Zur Laufzeit wird der Prozess bei eingehenden Nachrichten vom Router und gegebenenfalls zusätzlich zeitgesteuert getriggert. Bei einem zeitbasierten Aufruf gibt es drei verschiedene Varianten:

Festes Intervall: Der Aufruf erfolgt nach einem festen Intervall. Sollte die vorherige Ausführung noch nicht beendet sein, wenn das Intervall abgelaufen ist, wird der Aufruf direkt im Anschluss nachgeholt.

Intervall seit Beginn der letzten Ausführung: Der Aufruf erfolgt, wenn seit dem Beginn der letzten Ausführung eine festgelegte Zeitspanne abgelaufen ist. Auch hier gilt, dass der Aufruf direkt im Anschluss nachgeholt wird, falls die vorherige zeitbasierte Ausführung noch nicht beendet ist.

Intervall seit Beendigung der letzten Ausführung: Der Aufruf erfolgt, wenn seit dem Ende der letzten Ausführung die angegebenen Zeitspanne vergangen ist.

Die Aufrufvarianten werden in Abbildung 6.7 auf einem Zeitstrahl dargestellt.

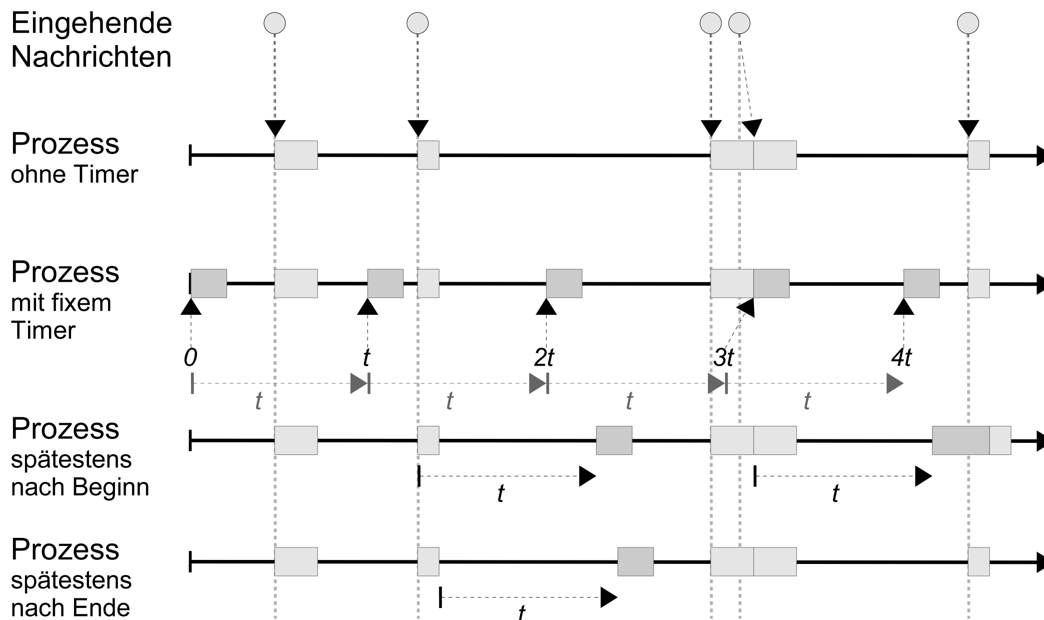

Abbildung 6.7.: Verschiedene Aufrufvarianten eines Prozesses

Falls das feste Zeitintervall eines Prozesses kleiner gewählt wird als die benötigte Laufzeit der Module, stauen sich die Aufrufe immer weiter an. Dies wird durch die Abbildung 6.8 verdeutlicht. Dieser Effekt tritt auch dann auf, wenn man innerhalb eines Prozesses mit dem Debugger die Ausführung dieses Threads zum Beispiel mittels eines Breakpoints anhält. Da sämtliche anderen Threads unabhängig davon weiter ausgeführt werden, sammeln sich auch hier die Aufrufe für den angehaltenen Prozess immer weiter an.

Um diesem Effekt entgegenzuwirken sollten sämtliche Module die an die Methode `execute` übergebenen `ExecuteFlags` berücksichtigen. Wenn ein Modul signalisiert bekommt, dass die letzte Ausführung zu viel Zeit beansprucht hat, sollte es falls möglich den Arbeitsaufwand eines Durchlaufes reduzieren oder aber die Ausführung überspringen. Damit ist es einem Modul möglich, die zur Verfügung stehende Rechenzeit optimal auszunutzen, indem zum Beispiel die Berechnungstiefe den aktuell vorhandenen Ressourcen angepasst werden kann.

Sobald der Prozess getriggert wurde, werden folgende Aktionen ausgeführt. Zunächst werden alle eingehenden Datenobjekte vom Prozess in die jeweiligen Ringpuffer weitergeleitet. Falls auch nur eines der Module auf die Mitteilung über neue Datenobjekte signalisiert, dass eine Ausführung erwünscht ist, oder das Triggern des Prozesses zeitgesteuert erfolgte, wird der Reihe nach die `execute`-Methode von allen Modulen ausgeführt. Die Module werden dabei seriell in der Reihenfolge ihres Hinzufügens zum Prozess aufgerufen. Ein Modul kann daher auch aufgerufen werden, obwohl es in der eigenen notify-Methode `false` zurückgeliefert hat, da ein anderes

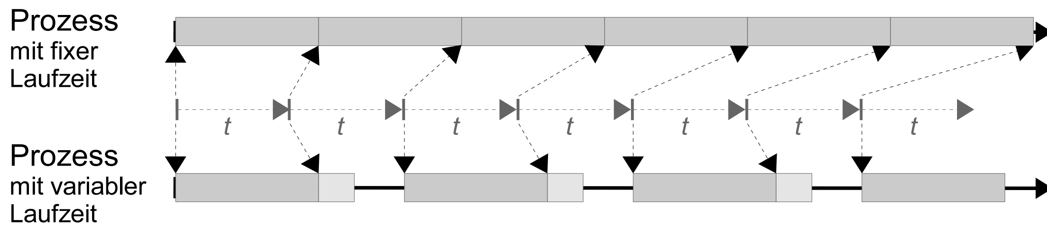

Abbildung 6.8.: Aufrufzeitpunkte eines Prozesses bei langer Modullaufzeit

Modul innerhalb dieses Prozesses eine Ausführung gewünscht hat. Nach jedem einzelnen Modulaufruf wird überprüft, ob neue Daten in den Ringpuffern hinzugefügt wurden, welche an andere Komponenten gesendet werden müssen, und diese gegebenenfalls verschickt. An dieser Stelle berücksichtigt die Prozess-Klasse, ob angeforderte Daten wirklich außerhalb benötigt oder nur innerhalb des Prozesses weiterverarbeitet werden.

6.1.7. Kommunikationsverwaltung

Damit das Framework in der Lage ist, dynamisch sämtliche Kommunikationswege zu verwalten, müssen Informationen darüber ausgetauscht und gespeichert werden, welche Daten wo benötigt werden und welche Quelle diese zur Verfügung stellt. Dazu gibt es zwei Datenstrukturen: zum einen die Klasse `KeySupply`, welche von Datenquellen generiert wird, und zum anderen die Klasse `KeyRequest`, welche von Datensenken generiert wird. Beide Klassen enthalten hierbei den Schlüssel der betreffenden Datenstruktur und werden im Weiteren als `KeyExchange`-Datenstrukturen zusammengefasst. Diese werden nach der Generierung automatisch als Nachrichten durch das gesamte Framework weitergeleitet. Da sämtliche datenverarbeitenden Komponenten miteinander verbunden sind, ist sichergestellt, dass diese Informationen nach kurzer Übertragungslatenz überall zur Verfügung stehen. Für das Bereitstellen von Daten kommt aus Sicht des Frameworks zum Beispiel ein OutBuffer in Frage, wohingegen Daten von InBuffern angefordert werden.

Die KeyRequest-Klasse ermöglicht, neben der optionalen Spezifikation der Datenherkunft, vor allem die Festlegung des Intervalls, in welchem die Daten angefordert werden. Neben dem Anfordern sämtlicher Datenpakete ist es auch möglich, eine festgelegte Anzahl an Datenpaketen in Folge anzufordern. Des Weiteren kann man eine Intervall-basierte Anfrage auswählen, um jedes n-te Datenobjekt oder jeweils nach einem Zeitintervall Daten geliefert zu bekommen. Da in jedem Datenpaket, welches durch das Framework behandelt wird, ohnehin die Herkunft gespeichert wird, muss diese Information in diesen Klassen nicht redundant gespeichert werden.

Bei der Kommunikation soll möglichst keine unnötige Datenübertragung stattfinden und diese trotzdem unbeschränkt flexibel gestaltet werden können. Damit das Framework die Kommunikationsverwaltung zur Laufzeit vollständig dynamisch verwalten kann, wird an jedem Knoten des Kommunikationsnetzes entschieden, wohin Datenpakete jeweils weitergeleitet werden sollen und müssen. Wenn alle Anfragen auch in der Datenquelle selbst vorliegen, kann im Optimalfall bereits dort entschieden werden, ob überhaupt eine entsprechende Datensenke an den Daten interessiert ist. Nur in diesem Fall werden die Daten durch das Framework verschickt.

Um diese *Routing-Entscheidungen* in den Knotenpunkten durchführen zu können, benötigt das Framework eine Informationsstruktur, welche sämtliche ausgetauschten KeyExchange-Objekte verwaltet. Dazu dient die Klasse `KeyTable`. In dieser Struktur werden die KeySupplies und KeyRequests getrennt voneinander gespeichert und nach ihren Schlüsseln und der Herkunft sortiert. An jedem Kommunikationsknoten wird dabei eine KeyTable vorgehalten, um entsprechende Rountinginformationen aufgrund der zur Laufzeit gesammelten Daten bereitzustellen.

Damit einmal verschickte KeyExchange-Objekte auch wieder rückgängig gemacht werden können, besitzen diese noch ein sogenanntes Status-Attribut. Damit ist es möglich, vorher versendete KeyRequests zu widerrufen.

Eine besondere Schwierigkeit entsteht bei der Kommunikationsverwaltung durch die zusätzliche Wahl des erwähnten Intervalls bei KeyRequests. Bei der Anforderung von n Datenpaketen eines bestimmten Schlüssels, sollten idealerweise genau n davon erstellt und verschickt werden. Zu diesem Zweck steht innerhalb der Module die Methode `isRequested(Key)` zur Verfügung, um vorab zu ermitteln, ob zum Beispiel zu berechnende Daten überhaupt benötigt werden. Damit diese Art der Filterung möglich ist, werden Einträge in der `KeyTable` aufgrund der übertragenen Daten bearbeitet, um den Zustand des Intervalls zu aktualisieren.

Würde man diese Operation allerdings an jedem Kommunikationsknoten durchführen, wäre es nicht möglich zum Beispiel nur jedes zweite Datenpaket anzufordern. In diesem Falle würde an jedem beteiligten Knotenpunkt auf dem Kommunikationspfad jedes zweite Paket nicht weitergeleitet, was nicht zu dem gewünschten Ergebnis führen würde. Daher darf diese Bearbeitung nur einmal bei der Übergabe der Daten an das Framework durchgeführt werden, nicht aber bei der Weiterleitung in den einzelnen Knoten.

Dieses Konzept führt allerdings zu einem nicht direkt ersichtlichen Effekt, wenn es für eine Datenstruktur m Quellen gibt, aber keine explizit ausgewählt wurde. Wenn in dieser Situation n Datenpakete angefordert werden, werden von jeder passenden Datenquelle die entsprechende Anzahl verschickt. Dadurch erhält die Senke $m * n$ Datenpakete.

6.1.8. Router und Konnektoren

Jeder Instanz der Software liegt genau ein `Router` zugrunde. Dieser kann mit sogenannten `Connector`-Objekten interagieren, welche zur Laufzeit hinzugefügt und auch entfernt werden können. Der Router ist dafür zuständig, sämtliche Datenpakete entsprechend der dynamischen Anforderung zwischen allen beteiligten Konnektoren auszutauschen. Er hat ähnliche Methoden für den Lebenszyklus wie Module und Prozesse.

Ein Konnektor ist eine abstrakte Klasse, welche Daten anfordert und auch Daten zur Verfügung stellen kann. Jeder Konnektor bekommt durch den Router eine lokal eindeutige Nummer zugewiesen, die sogenannte `ConnectorId`. Bei der Kommunikation vom Konnektor zum Router werden generell StreamedDataQueues benutzt. Dabei wird der Router entsprechend als QueueListener registriert, womit er, wie in Abschnitt 6.1.5 beschrieben, bei eingehenden Daten automatisch benachrichtigt wird. In der umgekehrten Richtung ist es der konkreten Konnektor-Realisierung überlassen, ebenfalls eine Queue zu nutzen oder direkt per Methodenaufruf Daten vom Typ StreamedData entgegenzunehmen.

Ein Beispiel für einen Konnektor ist die Prozess-Klasse, welche die benutzerdefinierten Module enthält. Da dieser selbst in einem eigenen Thread läuft, ist ein direkter Methodenaufruf vom Router zum Konnektor nur mit Synchronisierung möglich. Deshalb wird auch in dieser Richtung eine Queue eingesetzt, wobei in diesem Fall der Prozess den QueueListener darstellt.

Bei der dynamischen Verwaltung der Konnektoren zur Laufzeit muss der Router darauf achten, dass zum Beispiel Konnektoren aufgrund der Nebenläufigkeit nicht unmittelbar auf Anfrage entfernt werden können. Dazu muss ein Thread-basierter Konnektor zunächst angehalten werden und die Queues zwischen den beiden Klassen sollten vollständig abgearbeitet sein. Erst daraufhin sollte sich der Thread selbständig beenden und nicht von außerhalb durch entsprechende `cancel`-Methoden. Daher können solche Operationen nur asynchron durchgeführt werden und wirken sich dadurch erst nach einer kurzen Latenz aus. Dabei muss zusätzlich vom Router sichergestellt werden, dass beim Entfernen eines Konnektors die von ihm angeforderten Schlüssel vorab widerrufen werden, um obsolete Kommunikationskanäle nicht weiterzuführen.

Der Router wickelt die Kommunikation zwischen den Konnektoren in mehreren einzelnen Schritten ab. Sobald dieser über neue eingehende Nachrichten notifiziert wurde, wird sein Thread getriggert. Er beginnt nun solange alle eingehenden Daten weiterzuvermitteln, bis sämtliche eingehende StreamedDataQueues leer sind. Der Router entnimmt dabei das eingehende StreamedData-Objekt der Queue und versieht es zunächst mit der lokal gültigen Absenderadresse. Dazu verwendet er die Nummer des Konnektors, welcher das Datenpaket geliefert hat. Sofern das Paket aus derselben Instanz stammt, wird die Konnektor-Id der `DataSource`-Klasse gesetzt. Anders ist

Herkunft	Ziel		
	lokal	lokal beschränkt	remote
lokal	ja	ja	ja
lokal beschränkt	ja	ja	nein
remote	ja	ja	nein

Tabelle 6.1.: Beschränkungen beim Nachrichtenaustausch

dies bei StreamedData-Objekten, welche per Netzwerk aus einer anderen Instanz verschickt wurden. Dabei ist bereits von der anderen Instanz eine Konnektor-Id gesetzt worden und es wird stattdessen die Instanz-Id der DataSource-Klasse mit der lokalen Konnektor-Id versehen. Dadurch ist es möglich, die Herkunft eines Datenpaketes aus *lokaler* Sicht eindeutig zu identifizieren, ohne einen globalen Adressraum etablieren zu müssen.

Nach dieser Vorverarbeitung wird das konkrete Datenobjekt geprüft, ob es sich um eine Framework-eigene Nachricht handelt. Dazu zählen zum Beispiel die in Abschnitt 6.1.7 beschriebenen KeyExchange-Objekte. Diese werden gesondert behandelt und danach nicht weiter verschickt.

Abhängig von der Art des Absender-Konnektors werden die Datenpakete daraufhin an andere Konnektoren weitergeleitet. Dabei wird das Weiterleiten an den Sender selbst direkt unterbunden. Das Framework schränkt allerdings zusätzlich den Austausch von Datenpakten zwischen bestimmten Konnektor-Typen explizit ein.

Dazu werden folgende drei Typen von Konnektoren unterschieden:

Lokale Konnektoren sind keiner Beschränkung beim Datenaustausch unterworfen.

Remote-Konnektoren realisieren die Kommunikation mit einer anderen Instanz, welche in der Regel per Netzwerk abgewickelt wird. Darauf wird in Abschnitt 6.1.9 eingegangen.

Lokal beschränkte Konnektoren sind ähnlich den lokalen Konnektoren. Allerdings werden ausgehende Nachrichten nicht an andere Instanzen - also Remote-Konnektoren - verschickt.

Die Beschränkungen beim Nachrichtenaustausch zwischen diesen verschiedenen Konnektor-Typen sind in Tabelle 6.1 dargestellt. Durch diese Unterscheidung wird es möglich, dass Daten zwischen *Remote-Konnektoren* nicht ausgetauscht werden und somit sogenanntes *Relaying* unterbunden wird.

Die häufigste Verwendung findet der *lokale Konnektor*. Die bereits in Abschnitt 6.1.6 beschriebenen Prozesse sind von diesem Konnektor-Typ. Aber auch *Appender*

und *Dialoge*, auf welche in Abschnitt 6.1.12 und Abschnitt 6.2.1 näher eingegangen wird, gehören dieser Gruppe an.

Der Typ der *lokal beschränkten Konnektoren* dient einem speziellen Anwendungsfall, dass mehrere lokale Konnektoren miteinander Daten austauschen, diese aber anderen Instanzen nicht zugänglich gemacht werden sollen. Diese Art wird für den in Abschnitt 6.3.2 beschriebenen GameController benötigt.

6.1.9. Netzwerkkommunikation

Neben der lokalen Weiterleitung von Objekten durch die Klasse StreamedDataQueue ist eine *Weiterleitung per Netzwerk* über Instanzgrenzen hinweg ebenfalls möglich. Dazu dient die Klasse `SocketConnector`, welche von `Connector` erbt. Sie verzichtet dabei auf eine zweite Queue und versendet die per Methodenaufruf übergebenen StreamedData-Objekte unmittelbar über einen Socket.

Damit das Framework mit einer anderen Instanz Informationen austauschen kann, muss vorher eine Netzwerkverbindung etabliert werden. Diese wird im Framework mittels TCP-Sockets realisiert. Dabei wird in der Regel bei jeder Instanz eine Klasse `SocketListener` gestartet, um eingehende Verbindungen entgegennehmen zu können. Diese wartet auf einem festgelegten Port auf eingehende Verbindungen und läuft dazu in einem eigenen Thread, um die Anwendung unterdessen nicht zu blockieren.

Eine Verbindung wird aufgebaut, indem ein SocketConnector gestartet wird und diesem eine IP-Adresse mitgeteilt wird, mit der eine Verbindung hergestellt werden soll. Daraufhin wird eine TCP-Verbindung über einen festgelegten Port mit der entsprechenden anderen Instanz hergestellt. Auf der Gegenseite wird durch den Socket-Listener als Gegenstück ebenfalls ein entsprechender SocketConnector angelegt. Ab diesem Zeitpunkt ist eine bidirektionale Datenübertragung möglich. Dieser Ablauf ist in Abbildung 6.9 dargestellt.

Sobald zwei Systeme mit nicht-synchronisierten Uhren miteinander kommunizieren und dabei Zeitstempel austauschen, muss man die zeitliche Verschiebung zueinander beachten. Daher wird unmittelbar nach dem Aufbau der Netzwerkverbindung von beiden Seiten die jeweilige *Zeitdifferenz* zur Gegenseite ermittelt. Dazu werden, wie in Abbildung 6.9 gezeigt, jeweils zwei Zeitstempel ausgetauscht und unter Berücksichtigung der Übertragungszeit die zeitliche Differenz zu der jeweiligen Gegenstelle ermittelt. Von nun an wird der Zeitstempel im Header jedes empfangenen StreamData-Objektes in die lokale Zeit umgerechnet.

Danach ist diese Netzwerkverbindung genauso nutzbar wie eine Queue. Man kann Daten per Methodenaufruf an den SocketConnector übergeben, woraufhin dieser die StreamedData-Objekte über die Socket-Verbindung verschickt. Auf der Gegenseite

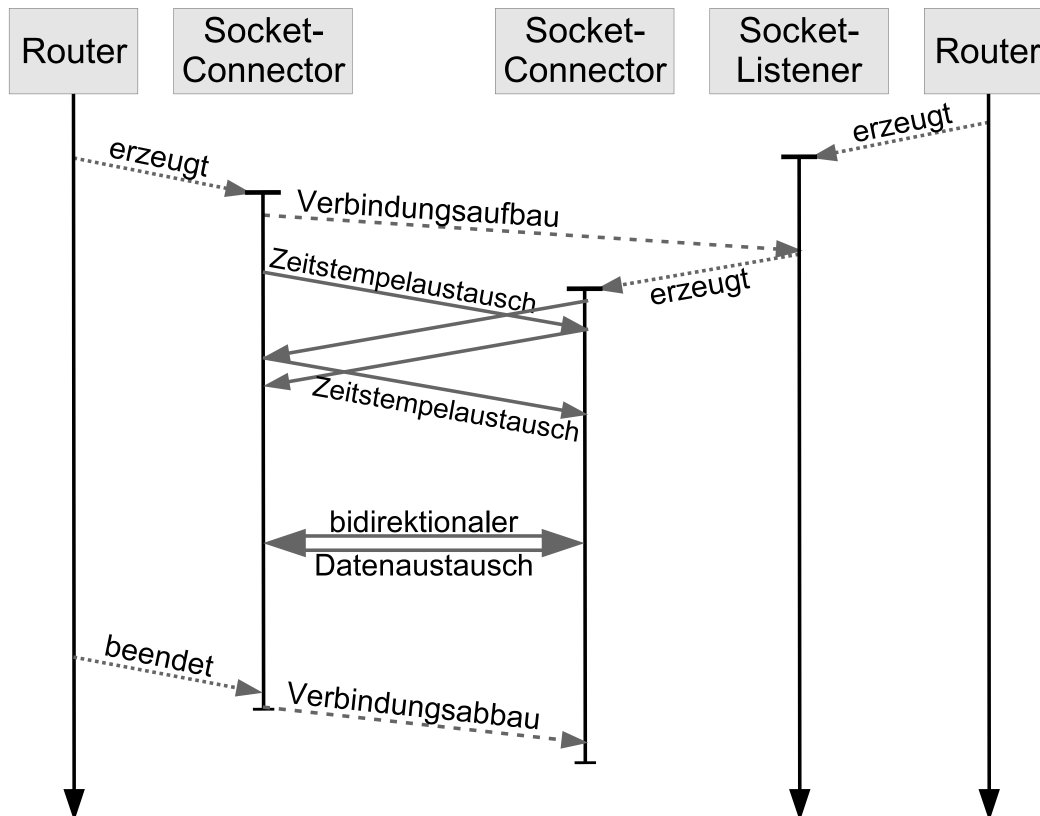

Abbildung 6.9.: Sequenzdiagramm einer Socket-basierten Netzwerkverbindung

werden die Daten empfangen und in einer Queue bereitgestellt.

Falls einer der beiden Kommunikationspartner die Verbindung auflöst, muss der Socket-Konnektor nicht nur beendet werden, sondern er muss sich selbst beim Router deregistrieren.

Dadurch ist sichergestellt, dass keine weiteren Nachrichten an die dazugehörige Queue gesendet werden. Zu diesem Zweck verschickt der SocketConnector nach erfolgreicher Trennung und Schließung des Sockets eine Nachricht mit dem Schlüssel `removeConnector` an den Router. Dadurch wird der Router benachrichtigt und dieser kann den Konnektor entfernen und sicherstellen, dass sämtliche KeyRequests der entsprechenden KeyTable widerrufen werden.

Da die Übertragung über ein Netzwerk potenziell fehleranfälliger ist als die lokale Übermittlung, sieht das Framework hierbei eine *Prüfsumme* vor. Diese Funktionalität steht bei Bedarf zur Verfügung und muss explizit mit der Präprozessordefinition DATA_DIGEST während des Kompilierens aktiviert werden. Daraufhin wird von jedem StreamedData-Objekt vor dem Verschicken ein Hashwert erstellt und dieser mitübertragen. Das ermöglicht der Gegenseite, die empfangenen Daten mit dem Hashwert

zu vergleichen und somit die Integrität der übertragenen Daten sicherzustellen. Fehlerhafte Datenpakete führen zu der Ausgabe einer Warnung und werden anschließend verworfen.

Bei durchgeführten Tests kam es allerdings nie zu Übertragungsfehlern, was jedoch auf die Verwendung des Kommunikationsprotokolls TCP zurückzuführen ist, welches bereits die Integrität der übertragenen Daten sicherstellt. TCP-basierte Verbindungen werden zum Debuggen und zur Anbindung der GUI verwendet. Für die Kommunikation zwischen mehreren Instanzen ist es dagegen sinnvoller, diese über UDP durchzuführen, um die Vorteile bei der Übertragungsgeschwindigkeit gegenüber TCP auszunutzen. Da hierbei das Kommunikationsprotokoll die Aufgabe der Integritätsprüfung nicht erfüllt, kann auf die Framework-eigenen Prüfsummen zurückgegriffen werden.

6.1.10. Konfigurationsdateien

Das Interface der Klasse `File` der Plattformabstraktionsschicht basiert auf den durch C++ bereitgestellten Funktionen, um auf Dateien zuzugreifen. Das Lesen und Schreiben ist für den Benutzer alles andere als intuitiv. Daher werden durch das Framework noch weitere Klassen bereitgestellt, um den Umgang mit speziellen Dateiformaten zu vereinfachen.

Die Klasse `TextFile` dient dem Zugriff auf Textdateien und ermöglicht einen zeilenbasierten Zugriff. Dabei wird die Datei zum Beispiel beim Lesen nicht vollständig in den Arbeitsspeicher geladen sondern nur einzelne Teile, um jeweils die Daten bis zum nächsten Zeilenumbruch zwischenzuspeichern. Dies ermöglicht einen komfortablen und gleichzeitig Ressourcen-schonenden Zugriff.

Basierend auf diesen Textdateien wurde die Klasse `ConfigurationFile` realisiert. Diese Klasse ermöglicht das Laden und Speichern von Werten zu bestimmten Namen innerhalb einer Konfigurationsdatei. Des Weiteren sind diese Schlüssel-Wert-Paare in Bereiche unterteilt, um sie entsprechend ihres Kontextes gruppieren zu können. Dieses Format ist dem von ini-Dateien nachempfunden. Dadurch sind die Inhalte der Konfigurationsdateien im Gegensatz zu Binärdateien auch für Menschen lesbar und ermöglichen ein manuelles Anpassen von Werten mittels eines einfachen Texteditors.

6.1.11. Anwendungsklasse

Die Klasse `Application` dient als Elternklasse dazu, eine einfache Schnittstelle für das Erstellen einer Anwendung zur Verfügung zu stellen. Sie sieht Methoden vor, neue Prozesse anzulegen, die Anwendung zu starten und zu beenden sowie sich mit anderen Instanzen zu verbinden.

Eine konkrete Anwendung kann in der Methode `initKeys` ihre eigenen Schlüssel registrieren. Eine besondere Bedeutung kommt hierbei der `init`-Methode zu, in der die Prozesse mit `createProcess` angelegt und die Module instanziert und zugeordnet werden müssen. Dadurch wird erst die eigentliche Struktur der Anwendung festgelegt.

In Abbildung 6.10 ist eine umfassende Darstellung einer Applikation abgebildet, welche sämtliche vorgestellten Komponenten enthält.

Unter Windows CE ist allerdings eine Konsolenanwendung nicht direkt ausführbar, sondern es wird eine Anwendung auf Basis der Microsoft Foundation Classes (MFC) benötigt. Um eine Duplikation der Konfiguration des Prozesslayouts zu vermeiden und einen einheitlichen Dialog zum Ausführen von Applikationen unter Windows CE zu ermöglichen, stellt das Framework bereits einen generischen Dialog zur Verfügung, welcher die Möglichkeit bietet, die Anwendung zu starten und zu stoppen. Die Klasse `WinCeApp` ist als Template realisiert, um die eigentliche Applikationsklasse als Parameter zu verwenden. Dabei wird sämtlicher MFC-Code bereits vom Framework bereitgestellt. Der entsprechende Dialog ist in Abbildung 6.11 dargestellt und umfasst ein Listenfeld, in welchem die Meldungen der Konsole aufgelistet werden. Die Ausgabe der Konsolenmeldungen wird ermöglicht, indem sich das Listenfeld bei der Klasse SystemCall als eigenständige Konsole registriert, auf welche sämtliche Ausgaben umgeleitet werden.

6.1.12. Logging

Um zu Debug- oder Informationszwecken Zustandsdaten oder Warnungen bzw. Fehlermeldungen ausgeben zu können, existiert eine Klasse `Logger`, welche an das Konzept von Log4J[3] angelehnt ist.

Zum einen ist einer Log-Nachricht ein `LogLevel` zugeordnet, was die unterschiedlichen Wichtigkeitsgrade beschreibt. Diese lauten `Debug`, `Info`, `Warn`, `Error` und `Fatal`. Dabei soll es beim Kompilieren als Release-Version durch zahlreiche Debugausgaben nicht zu Performancebeeinträchtigungen kommen. Deshalb werden Log-Nachrichten durch einen Makroaufruf realisiert, dadurch können beim Kompilieren ohne die Präprozessordirektive `DEBUG` entsprechende häufig verwendete Makros vom Präprozessor entfernt werden.

Zum anderen wird ein sogenannter *Pfad* verwendet, um die Quelle der Ausgabe zu identifizieren. Dieser wird in der Regel durch die folgenden, mit Punkten voneinander getrennten Attribute gebildet: der Namensraum, Teile des Verzeichnisses und der Klassenname. Dadurch ist es möglich, die Ausgaben, abhängig von diesem Pfad, unterschiedlich zu behandeln.

[3] Log4J Homepage: `http://www.log4j.org`

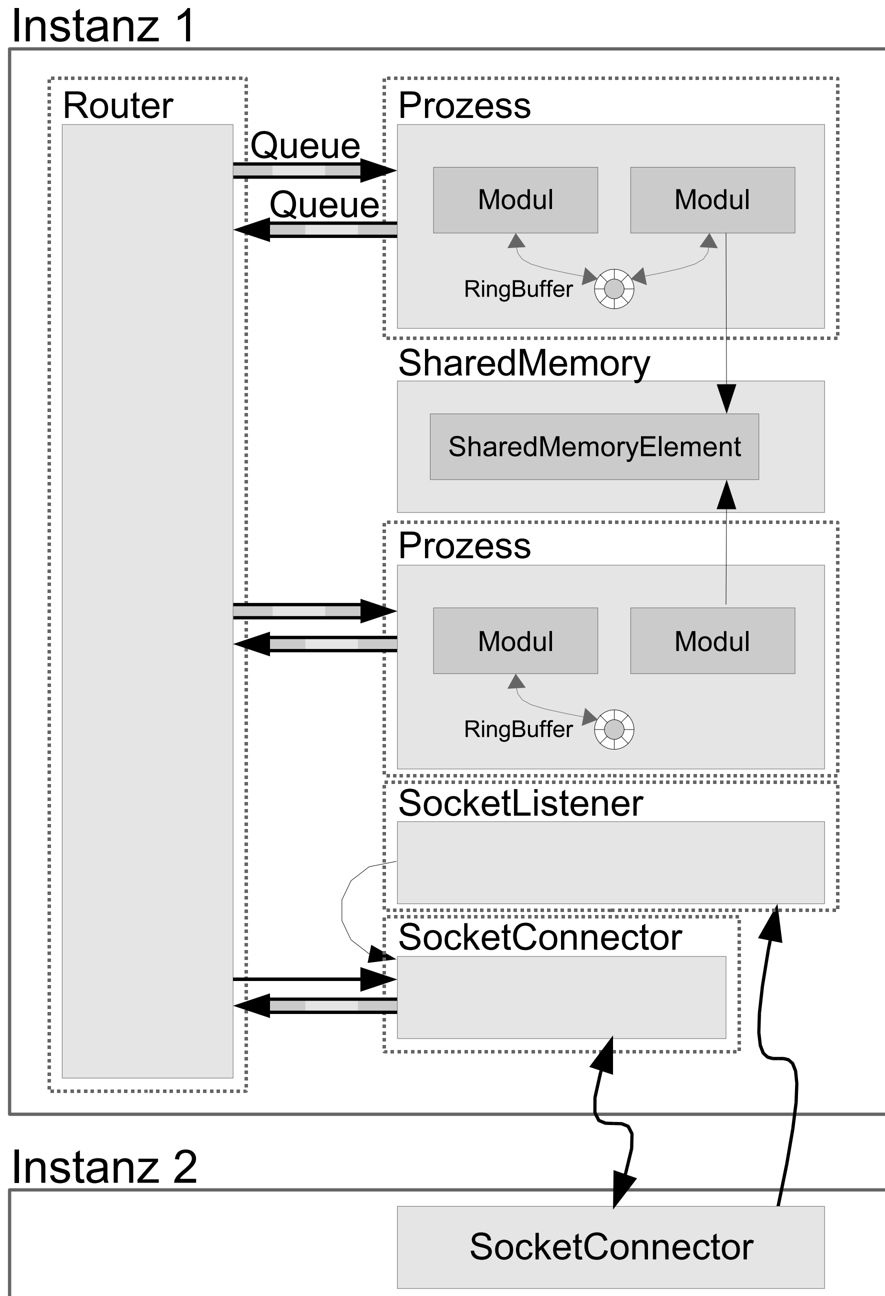

Abbildung 6.10.: Übersicht aller Komponenten einer Applikation
Die einzelnen Threads sind durch gestrichelt umrandete Bereiche hervorgehoben.

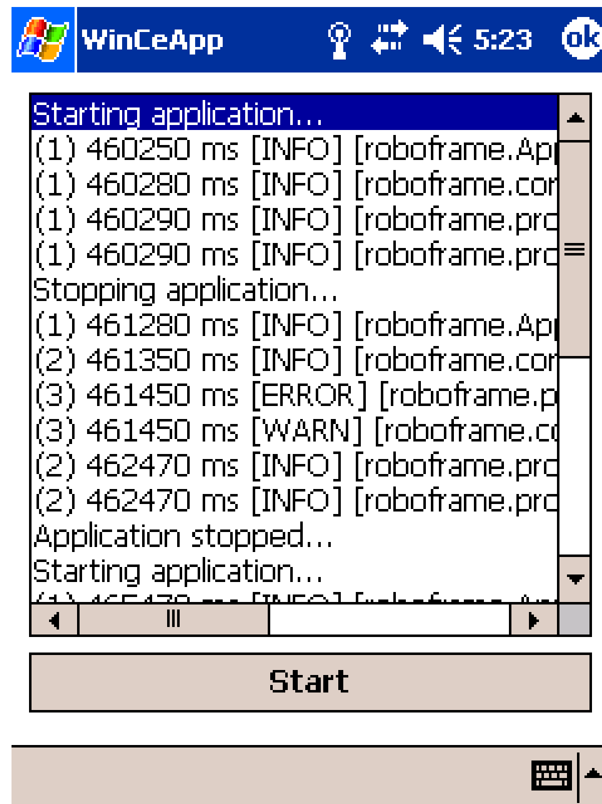

Abbildung 6.11.: Applikations-Dialog unter Windows CE

Die Nachricht selbst kann dabei ähnlich wie bei der Funktion `printf` durch einen Formatierungstext und eine variable Anzahl von Argumenten zusammengestellt werden.

Um die Log-Nachrichten auszugeben, existieren sogenannte `Appender`, welche bei der `Logger`-Klasse registriert werden müssen. Die Klasse `ConsoleAppender` gibt die Nachrichten auf der Konsole aus. Dagegen versendet der `NetworkAppender` die Textmeldungen als Datenstruktur `LogMessage` an andere Instanzen, um zum Beispiel von einem anderen System aus die Meldungen analysieren zu können.

Jeder dieser Appender kann dahingehend konfiguriert werden, nur bestimmte Log-Level anzuzeigen oder nach einem Pfad zu filtern. Dabei können durchaus mehrere Appender des gleichen Typs bei der Logger-Klasse registriert werden, wobei darauf geachtet werden sollte, dass diese so konfiguriert sind, eine Meldung nicht mehrfach auf demselben Ausgabemedium darzustellen.

Jede Nachricht wird vom Logger an sämtliche registrierte Appender weitergereicht, sofern deren Filterkriterien zutreffen.

Zur besseren Darstellung der Funktionsweise wird noch ein kurzes Beispiel angesprochen. Wird beim Logger lediglich ein ConsoleAppender mit einem leeren Pfad und dem Level LOG_THRESHOLD_WARN registriert, werden sämtliche Meldungen vom Typ Warn, Error und Fatal - egal mit welchem Pfad - auf der Konsole ausgegeben. Wenn allerdings zusätzlich noch ein ConsoleAppender mit dem Pfad beispiel.pfad und dem Level LOG_THRESHOLD_ALL registriert wird, gibt es für Meldungen, deren Pfad mit beispiel.pfad beginnt, nun zwei potenzielle Appender. Wenn das Log-Level einer Meldung einer Warnung oder höher entspricht, treffen die Filterkriterien beider Appender zu, woraufhin die Meldung auch doppelt auf der Konsole ausgegeben wird. Daher wäre es sinnvoller, das Log-Level des zweiten Appenders mit LOG_LEVEL_DEBUG | LOG_LEVEL_INFO anzugeben, damit die Filterkriterien der beiden ConsoleAppender keine Schnittmenge haben, um Mehrfachausgaben zu vermeiden. Damit werden zusätzlich zu allen Warnungen und Fehlern auch die Debug- und Info-Ausgaben des angegebenen Pfades auf der Konsole dargestellt.

6.2. RoboGui

Die graphische Benutzeroberfläche basiert auf den Komponenten von RoboApp und nutzt Klassen der QT-Bibliothek für die eigentliche Darstellung der GUI. In QT werden die einzelnen Teilkomponenten der graphischen Benutzeroberfläche *Widgets* genannt.

Zahlreiche Subsysteme, wie zum Beispiel das *Logging* oder die *Netzwerkkommunikation*, können ohne Anpassungen wiederverwendet werden. Es wurden entsprechende Oberflächen erstellt, um zum Beispiel zur Laufzeit neue Verbindungen zu anderen Instanzen per Netzwerk herzustellen oder diese auch wieder zu trennen. Das Hauptfenster von RoboGui ist in Abbildung 6.12 dargestellt und zeigt oben rechts die verbundenen Instanzen an.

Um mit den GUI-spezifischen Komponenten interagieren zu können, wurden an einigen Stellen Erweiterung der vorhandenen Komponenten durchgeführt. Für das Logging existiert zusätzlich ein QtAppender, der es ermöglicht, Log-Einträge innerhalb eines Widgets auszugeben. Dieses ist im unteren Teil des Hauptfensters dargestellt.

Bei der Verwendung von QT ist hierbei ein wichtiger Aspekt zu beachten: bei einer multi-threaded Anwendung sollte ausschließlich innerhalb der *QT-Eventloop* manipulierend auf Widgets zugegriffen werden. Ansonsten kann es zu Fehlern durch asynchrone Zugriffe kommen. Falls verschiedene Threads Änderungen an GUI-Elementen vornehmen wollen, sollten diese zwischengespeichert werden, und daraufhin das entsprechende Widget durch den QT-spezifischen Mechanismus eines CustomEvents benachrichtigt werden. Innerhalb der Methode customEvent, welche daraufhin von

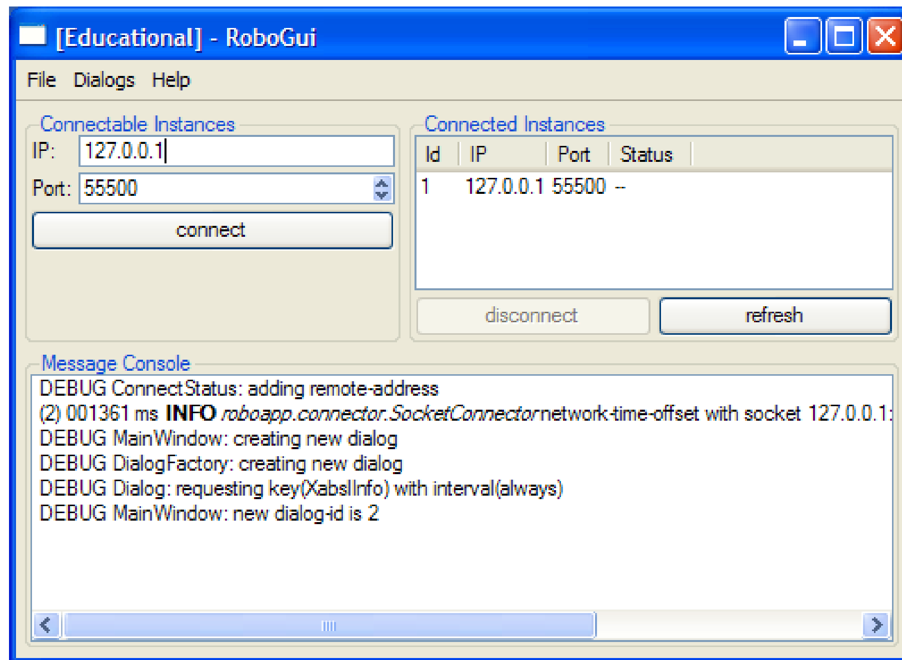

Abbildung 6.12.: Hauptfenster von RoboGui

der QT-Eventloop aufgerufen wird, können dann die anstehenden Änderungen durchgeführt werden.

Um die Kommunikation mit Dialogen zu ermöglichen, wird für die Kommunikationsverwaltung eine abgewandelte Art des Routers verwendet, welcher durch die Klasse `GuiRouter` realisiert wird. Diese führt eine besondere Behandlung beim Deregistrieren von Dialogen durch Notwendig ist dies wegen einer besonderen Eigenart von QT: wenn ein GUI-Element nicht mehr verwendet wird, kann es nicht einfach gelöscht werden, da es innerhalb der QT-Bibliothek zum Beispiel noch für Ereignisse registriert ist oder mit anderen Widgets in Verbindung steht. Daher muss die Methode `deleteLater` verwendet werden, welche es QT ermöglicht, das GUI-Element erst zu löschen, nachdem sämtliche Verweise darauf entfernt wurden.

Die graphische Benutzeroberfläche ist nicht auf eine Sprache festgelegt, sondern es wird die QT-spezifische Methode verwendet, um Textbausteine in die jeweilige Sprache zu übersetzen. Für eine genaue Beschreibung des Verfahrens wird hier nur auf die QT-Dokumentation[4] verwiesen. Für sämtliche GUI-Komponenten, die vom Framework bereitgestellt werden, liegt sowohl eine englische als auch eine deutsche Übersetzung aller Textbausteine vor. Diese Sprachdateien befinden sich im Verzeichnis *roboframe/robogui/resource*. Die GUI versucht beim Start anhand von Umgebungsvariablen automatisch zu ermitteln, welche Sprache auf dem System verwendet wird, um die entsprechende Sprachdatei einzubinden.

[4] QT-Dokumentation: `http://doc.trolltech.com`

6.2.1. Dialoge

Dialoge ermöglichen es, die graphische Benutzeroberfläche mit eigener Funktionalität zu erweitern und müssen dazu von der Klasse `Dialog` erben. Das Konzept für den Nachrichtenaustausch unterscheidet sich von dem eines Moduls deutlich. Da keinerlei Puffer verwendet werden, muss ein Dialog innerhalb der Methode `addRepresentationWrappers` selbst festlegen, mit Daten welches Schlüssels er arbeiten kann. Dabei ist er nicht auf eine Schlüssel-basierte Auswahl festgelegt, sondern kann auch alle vorhandenen Schlüssel spezifizieren, welche mit einer bestimmten Datenstruktur bei der KeyRegistry registriert wurden. Dies wird durch die statische Methode `getKeys` der Klasse KeyRegistry ermöglicht, welche eine beliebige Datenstruktur als Parameter erwartet. Dieser Parameter muss dazu, wie in Abschnitt 6.1.2 beschrieben, innerhalb eines Templates vom Typ `RepresentationWrapper` gekapselt werden. Dadurch wird es möglich, eine Datenstruktur zu visualisieren, auch wenn sie in Zukunft mit einem neu hinzugekommen Schlüssel verwendet wird.

Zur Laufzeit sollte der Dialog die Auswahl ermöglichen, welche Nachrichten er von welcher verbundenen Instanz beziehen will. Damit dies nicht für jeden Dialog neu implementiert werden muss, realisiert die Methode `chooseRequestedKeys` bereits ein modales Fenster, in welchem die angeforderten Nachrichten aufgrund des Schlüssels und sämtlicher möglicher Parameter ausgewählt werden können.

Um die eingehenden Datenpakete zu behandeln, muss die `handle`-Methode mit dem Parameter StreamedData implementiert werden. Diese wird für jede eingehende Nachricht aus dem Kontext der QT-Eventloop aufgerufen. Die Daten können dabei in Abhängigkeit des Schlüssels der empfangenen Nachricht mittels der Streaming-Operatoren in eine entsprechende Datenstruktur gelesen werden.

Um die eingehende Nachricht auch dann behandeln zu können, wenn lediglich die Datenstruktur aber nicht der Schlüssel bekannt ist, existiert in der KeyRegistry-Klasse die statische Methode `matchKeyStreamable`. Diese ermöglicht den Vergleich eines Schlüssels mit einer beliebigen Datenstruktur und liefert zurück, ob diese Kombination entsprechend bei der KeyRegistry registriert wurde.

Vom Framework selbst werden für die Benutzeroberfläche lediglich zwei konkrete Dialoge bereitgestellt, welche ausschließlich zum Debugging der Anwendung gedacht sind.

Der Dialog `LoggingDlg` dient dazu, Log-Meldungen von verbunden Instanzen anzuzeigen und entsprechende Filteroptionen für die LogLevel und den Pfad bereitzustellen. Die Nachrichten selbst werden dabei innerhalb einer Liste dargestellt und sind nach den einzelnen Spalten sortierbar. Die Datenquelle für diesen Dialog bildet der in Abschnitt 6.1.12 beschriebene `NetworkAppender`, welcher entsprechend konfiguriert sein muss, damit diese Informationen in der GUI zur Verfügung stehen.

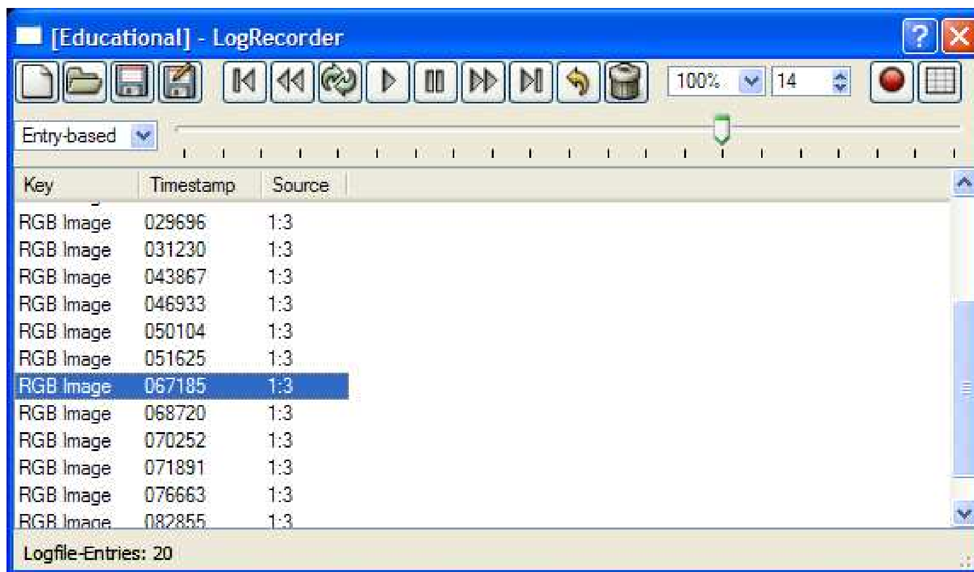

Abbildung 6.13.: LogRecorder-Dialog

Des Weiteren wurde der Dialog `StreamableDlg` erstellt, welcher alle registrierten Schlüssel unabhängig von der dahinter stehenden Datenstruktur behandeln kann. Aufgrund der nicht bekannten enthaltenen Datenstruktur kann dieser Dialog lediglich die allgemeinen Kopfdaten des StreamedData-Objektes auswerten. Allerdings kann er mit allen benutzerdefinierten Schlüsseln umgehen und ermöglicht damit das Überprüfen, ob bestimmte Datenpakete empfangen werden, und kann neben dem Schlüssel auch die Herkunft und den Zeitstempel anzeigen.

6.2.2. Dialog LogRecorder

Neben den bereits beschriebenen Dialogen zum rudimentären Debuggen wurde ein neuer Dialog realisiert, welcher die angeforderten Nachrichten in einer Binär-Datei, einem sogenannten `LogFile`, archiviert. Der `LogRecorder`-Dialog stellt neben der Funktionalität der Archivierung sämtlicher übertragenen Datenpakete auch die Möglichkeit zur Verfügung, diese im Nachhinein wiederzugeben. Dies ermöglicht das Analysieren von Modulen auf Basis von realen Eingabedaten, welche beliebig oft wiederholt werden können. Der Dialog sieht hierbei zahlreiche Methoden zum Abspielen der aufgezeichneten Daten vor. Man kann sowohl in einer übersichtlichen Liste aller Datenpakete navigieren und einzelne manuell auswählen oder aber eine automatische Wiedergabe-Funktion nutzen. Dabei kann man die Geschwindigkeit beim Abspielen frei skalieren zwischen *verlangsamt* über *Echtzeit* bis hin zu *beschleunigt*. Des Weiteren kann man auch einzelne Datenpakete nachträglich aus dem Log-File entfernen. Der Dialog ist in Abbildung 6.13 dargestellt.

Intern werden sämtliche Datenpakete seriell in einem binären Log-File gespeichert. Dabei nutzt die `LogFile`-Klasse die Framework-internen Sreaming-Operatoren, um beliebige Datenformate speichern und wieder laden zu können. Somit kann dieser Dialog auch zukünftig mit neuen Datenobjekten ohne Anpassung verwendet werden.

6.3. RoboCup-spezifische Erweiterungen

Zurzeit des Schreibens dieser Arbeit läuft die Vorbereitung auf die *RoboCup Weltmeisterschaft 2005* in Osaka, bei der die oben genannten Roboter, sowie eventuell neue Modelle eingesetzt werden sollen. Zu diesem Zweck erstellte Anwendungen sind mit leicht geänderten Anforderungen konfrontiert. Um diese zu erfüllen, wurden einige Erweiterungen für das Framework implementiert, die im Folgenden vorgestellt werden.

6.3.1. RoboCup-Anwendung

Beim *RoboCup* agieren mehrere Roboter als ein Team. Um eine Identifizierung eines Roboters sowohl durch die Schiedsrichter als auch untereinander zu ermöglichen, verfügt jeder Roboter über eine Spielernummer. Allen Roboter eines Teams wird eine eindeutige Teamnummer zugewiesen. Die Spielrichtung der Roboter wird durch die Teamfarbe definiert. Blaue Roboter spielen auf das gelbe, rote auf das blaue Tor.

Um diese Informationen bereitzustellen, wurde die `Application`-Klasse durch die Klasse `RoboCupApplication` erweitert. Diese liest beim Starten eine Konfigurationsdatei, in der die oben genannten Werte enthalten sind, ein und stellt Zugriffsmethoden zur Abfrage bereit. Zusätzlich enthält die Konfiguration weitere Parameter, wie zum Beispiel die Ports für die in den folgenden Abschnitten beschriebenen GameController- und Team-Konnektoren.

Als Windows CE-Oberfläche wurde ein weiterer MFC-Dialog implementiert, der diese Werte und Informationen über den Spielstand anzeigt (siehe Abbildung 6.14). Über die vorhandenen Schaltflächen kann der Spielstatus geändert werden. Dazu registriert sich der Dialog als Konnektor beim Router und versendet Nachrichten vom Typ `GameState`.

6.3.2. GameController-Konnektor

In der Sony Four-Legged Robot League hat sich die Verwendung einer externen Anwendung zur Steuerung des Spiels, der sogenannte *GameController*, etabliert. Dieser wird von dem Assistenten des Schiedsrichters bedient und sendet Informationen über

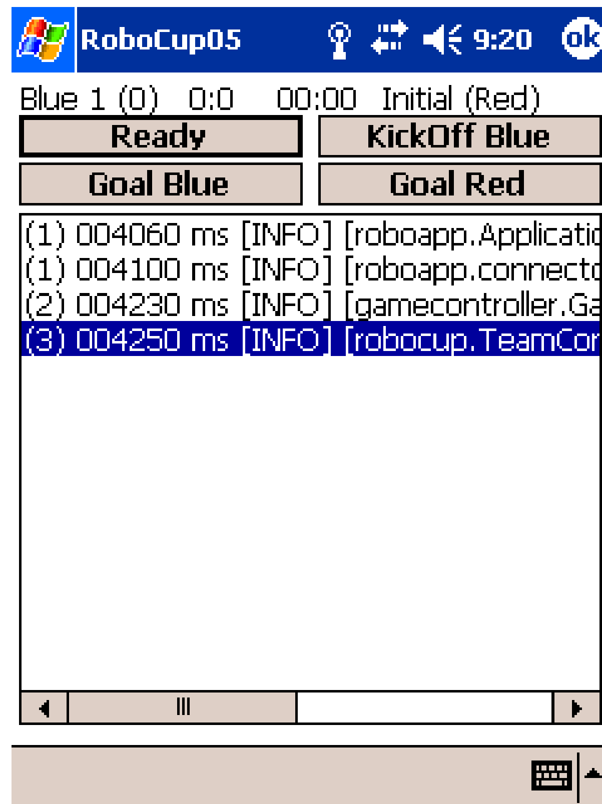

Abbildung 6.14.: RoboCup Windows CE-Dialog

den aktuellen Spielzustand per UDP-Broadcast an die Roboter. Die Eingabe der Daten erfolgt über eine graphische Oberfläche.

Um diese Daten zu empfangen wurde ein von der Klasse `Connector` abgeleiteter GameController-Konnektor implementiert. Dieser empfängt über einen UDP-Socket die Daten des GameControllers und konvertiert sie in eine `GameState` Nachricht.

6.3.3. Team-Konnektor

Um eine Kommunikation der Roboter eines Teams untereinander zu ermöglichen, wurde ein *Team-Konnektor* entwickelt. Der Austausch von Nachrichten geschieht über einen UDP-Broadcast, da statt einer zuverlässigen Übertragung eher eine schnelle Übermittlung gefordert ist.

Auf die Angabe des Typs der auszutauschenden Nachricht kann verzichtet werden. Der Team-Konnektor muss nur mit den Schlüsseln der zu versendenden und zu empfangenden Nachricht konfiguriert werden. Da das Framework die Daten bereits in serialisierter Form bereitstellt, kann intern direkt mit diesen gearbeitet werden.

Die Pakete werden mit einem Nachrichtenkopf versehen, der die Spieler- und Teamnummer des Absenders enthält. Beim Empfang werden diese Informationen überprüft und das Paket bei unterschiedlicher Team- oder gleicher Spielernummer verworfen. Damit werden die zwangsläufig durch die Verwendung von Broadcasts empfangenen eigenen Nachrichten nicht an die restliche Anwendung weitergeleitet.

Auf eine Verschlüsselung wurde bisher aus Performancegründen verzichtet. Denkbar wäre hier jedoch eine einfache, symmetrische exklusiv-oder Verknüpfung (XOR) mit einem Team-Schlüssel, im einfachsten Fall mit der Teamnummer. Dadurch könnte ein Mitlesen der Kommunikation durch andere Teams leicht unterbunden werden.

6.3.4. Farbtabellen-Dialog

Um *Farbtabellen* erstellen zu können, welche für die *Segmentierung* von Kamerabildern benötigt werden, wurde ein zusätzlicher, anwendungsspezifischer Dialog (Abbildung 6.15) erstellt. Hauptaugenmerk lag dabei auf der Weiterführung des Grundgedankens, welcher auch dem Framework an sich zugrunde liegt. Mehrere Farbräume, verschiedene Bildauflösungen wie auch unterschiedliche Bittiefen der Farbkanäle sollten hierbei völlig unabhängig von dem eigentlichen Dialog und dessen Funktionalität unterstützt werden. Dieses Konzept erlaubt zum einen, die graphische Oberfläche mit den Interaktionsmöglichkeiten lediglich einmal implementieren zum müssen, und ermöglicht zum anderen, auch die einfache Erweiterung mit weiteren Farbräumen, Bildauflösungen oder sonstigen Kamera-spezifischen Eigenschaften. Dabei ist der `ColorTable`-Dialog als Template realisiert. Dieser erhält folgende Klassen als Template-Parameter, wobei diese Klassen teilweise selbst als Templates realisiert sind:

Eine Pixel-Klasse stellt für jeden Farbraum die Methode zur Verfügung, einen RGB-Farbwert zu liefern, um einen Bildpunkt in der GUI darstellen zu können.

Eine Bild-Klasse definiert die Breite und Höhe des Bildes und liefert für eine Bildkoordinate das entsprechende Pixel-Objekt zurück. Des Weiteren kann sich diese Klasse aus einem StreamedData-Objekt deserialisieren. Diese Bild-Klasse hat ebenfalls eine Pixel-Klasse als Template-Parameter, um den eigentlichen Farbraum zu definieren.

Eine ColorClassCollection-Klasse legt die Farben fest, die klassifiziert werden können, und definiert neben einem korrespondierenden RGB-Farbwert für die Anzeige auch einen beschreibenden Namen für jede Farbklasse.

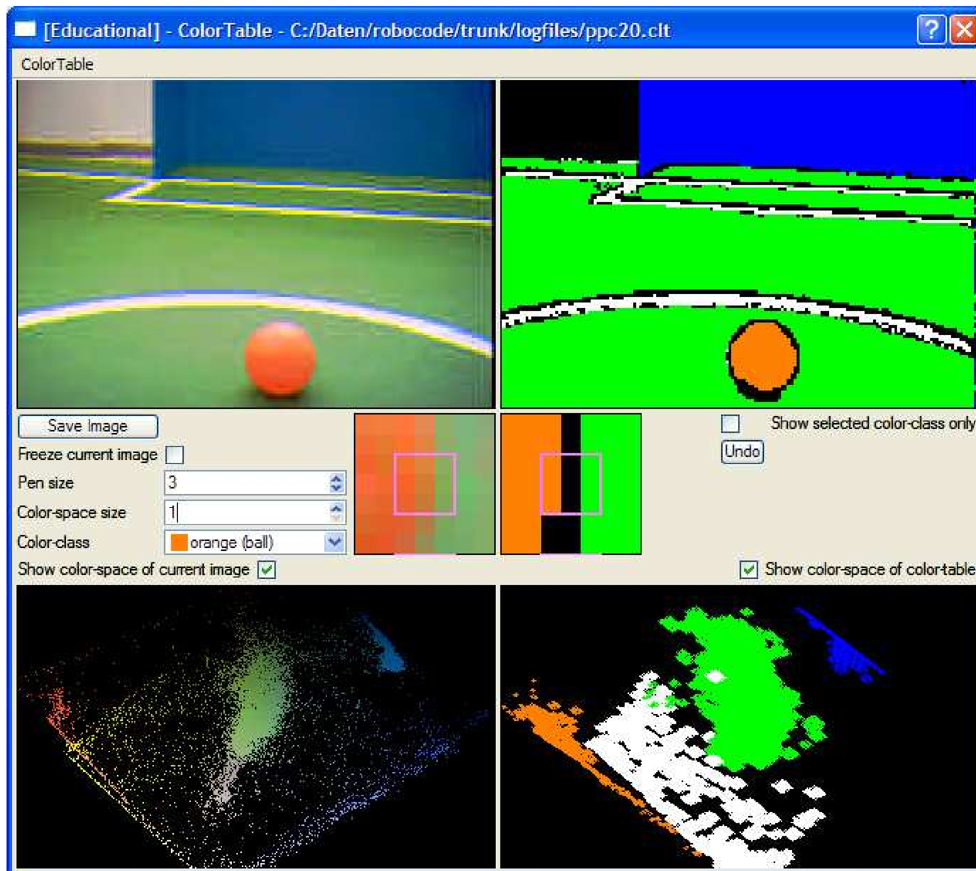

Abbildung 6.15.: Dialog zum Erstellen einer Farbtabelle

Eine ColorTable-Klasse kann sich aus einer Datei lesen beziehungsweise in einer speichern. Außerdem kann man die Farbklasse von einem Farbwert abfragen und setzen. Auch eine Farbtabelle erhält eine Pixel-Klasse als Template-Parameter, da nur diese zum Beispiel auch die Nachbarschaftsbeziehung von Farbwerten im Farbraum ermitteln kann.

Für sämtliche dieser Template-Parameter existieren entsprechende Interfaces, welche abstrakt die zu implementierenden Methoden definieren. Durch diese strikte objektorientierte Modellierung kann der eigentliche Dialog mit zusätzlicher Funktionalität erweitert werden, welche nicht für zum Beispiel neue Farbräume angepasst werden müssen. Dabei könnten die für die Pixel- und Bilddarstellung verwendeten Klassen zukünftig auch in der Bildverarbeitung Verwendung finden, um auch dort von der erweiterten Flexibilität zu profitieren.

Neben der Möglichkeit, mit der Maus sowohl in das Rohbild als auch in das segmentierte Bild zu klicken, um Pixel einer entsprechend ausgewählten Farbklasse zuzuordnen, existiert zusätzliche eine vergrößerte Darstellung der Bildausschnitte um den

Mauszeiger herum. Dies ermöglicht das genauere Auswählen der betroffenen Bildpunkte. Um den Aufwand bei der Erstellung der Farbtabelle zu reduzieren, kann sowohl die Mauszeiger-Größe verändert werden als auch die Größe des zu modifizierenden Bereichs im Farbraum selbst.

Außerdem wurde das Problem der fälschlichen Auswahl eines Bildpunktes berücksichtigt. Dazu wird jegliche Modifikation der Farbtabelle durch eine entsprechende Klasse `ColorTableCommand` umgesetzt. Diese Realisierung entspricht dem sogenannten Command-Pattern [5]. Durch die Kapselung an dieser einen Stelle, welche die Änderung durchführt, war es möglich, den vorherigen Stand der Farbtabelle zu sichern und eine Funktion anzubieten, um diese Aktion rückgängig zu machen. Der Dialog ist durch die Speicherung der Command-Objekte in der Lage beliebig, viele Aktionen rückgängig zu machen.

Um dem Benutzer eine Visualisierung anzubieten, in wie weit sowohl das Robbild als auch die Farbtabelle den entsprechenden *Farbraum* ausnutzen, sind im unteren Bereich des Dialoges optional noch zwei dreidimensionale Ansichten integriert. Diese zeigen den entsprechenden Farbraum mit den aktuell verwendeten Pixeln und Farbklassen in einer rotierenden Ansicht und basieren auf dem *OpenGL-Widget* von QT. Aufgrund dieser Anzeige ist es einem Benutzer wesentlich besser möglich, die Qualität einer Farbtabelle zu beurteilen.

6.3.5. XABSL-Dialog

Von Christian Fuest wurde die *eXtensible Agent Behavior Specification Language (XABSL) Engine* von Martin Lötzsch [11] als Modul für das Framework portiert. Dabei handelt es sich um eine Beschreibungssprache für Verhalten für autonome Agenten und einen Interpreter. Ein Verhalten besteht bei XABSL aus mehreren verschachtelten Zustandsmaschinen, *Optionen* genannt. Die durchzuführende Aktion wird von sogenannten *BasicBehaviors* definiert. Für jeden Zustand kann eine Option oder ein BasicBehavior als nächste Ebene angegeben werden.

Um das Verhalten zur Laufzeit analysieren zu können, wurde für RoboGui der XABSL-Dialog (Abbildung 6.16) entwickelt. Dieser zeigt den aktuellen Aktivierungspfad, das ausgewählte BasicBehavior und dessen Parameter an.

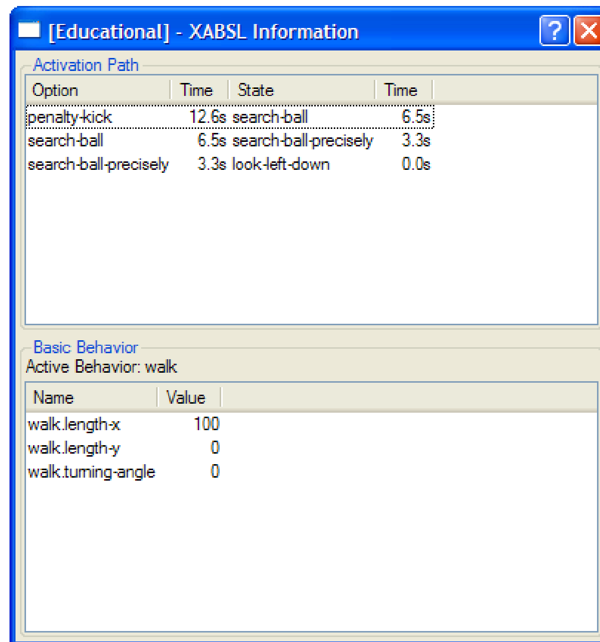

Abbildung 6.16.: XABSL-Dialog

6.3.6. CopyFiles-Anwendung

Zum Ausführen der für den Pocket PC kompilierten Programme müssen diese erst auf das Gerät kopiert werden. Zusätzlich werden weitere Dateien, wie beispielsweise Parametersätze, Farbtabellen und Konfigurationsdateien zur Ausführung benötigt. Um diese Arbeiten zu erleichtern und damit weniger fehleranfällig zu machen, wurde auf Basis des Frameworks die CopyFiles Anwendung erstellt, die den Kopier- und Konfigurationsvorgang automatisiert. Über eine graphische Oberfläche (siehe Abbildung 6.17) können die entsprechenden Dateien ausgewählt und die Konfiguration vorgenommen werden. Für jeden Spieler existiert eine Schaltfläche, um die entsprechende Konfiguration auf den Pocket PC zu überspielen. Die Übertragung der Daten geschieht mit Hilfe des *Remote Application Programming Interface* (RAPI) und Microsoft ActiveSync.

Um den Vorgang weiter zu vereinfachen, kann CopyFiles über das Optionen-Menü so konfiguriert werden, dass es automatisch gestartet wird, falls ein Pocket PC per ActiveSync verbunden wird. Hier kann ebenfalls der sogenannte Gast-Modus erzwungen werden, bei dem kein Profil für den Pocket PC auf dem Desktop eingerichtet werden muss. Dazu werden undokumentierte Windows-Registry-Schlüssel verwendet, die mit den ActiveSync Versionen 3.x funktionieren.

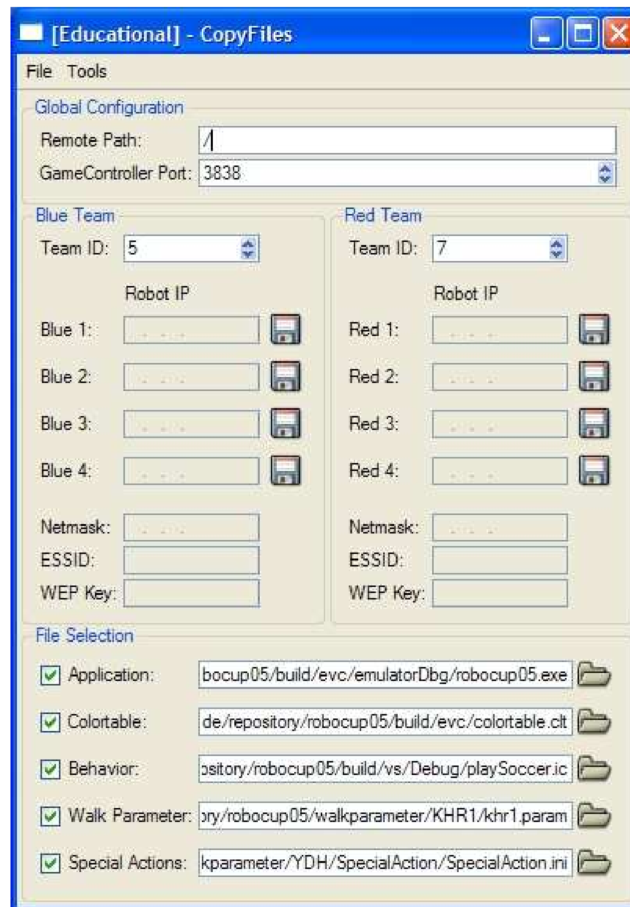

Abbildung 6.17.: CopyFiles-Anwendung

Sollte eine Instanz der CopyFiles-Anwendung geöffnet sein, wenn ein Pocket PC verbunden wird, so würde eine weitere Instanz der Anwendung gestartet. Aus diesem Grund stellt die Klasse `SingleInstance` einen benannten Mutex bereit, der beim Start gesperrt wird, um eine mehrfache Ausführung zu verhindern. Eine zweite Instanz kann dies feststellen und dann sofort terminieren.

Die im Dialog eingestellten Werte werden in einer Konfigurationsdatei im Home-Verzeichnis des Benutzers gespeichert.

7. Ergebnisse

Wie schon in Kapitel 3.1 aufgeführt, sind Frameworks für sich allein nicht lauffähig, sondern bilden lediglich ein Grundgerüst für eine Anwendung. Um ein Framework zu evaluieren, muss dieses also instanziert werden. Im Folgenden werden die auf Basis von *RoboFrame* entwickelten Anwendungen und die durchgeführten Performanz-Messungen näher beschrieben.

7.1. Test-Anwendung

Um die Funktionsfähigkeit des Frameworks zu überprüfen, wurde zunächst eine Test-Anwendung implementiert, die einfache Daten zwischen zwei Modulen, Prozessen oder Instanzen verschickt. Zusätzlich wurden die Debugging-Mechanismen und die GUI getestet.

Die Test-Anwendung läuft auf allen vom Framework unterstützten Betriebssystemen ohne eine spezielle Roboterhardware zu erfordern. Über Kommandozeilenoptionen können der gewünschte Test und weitere Einstellungen bestimmt werden (siehe Tabelle 7.1).

Als Module wurden implementiert:

`BlackBoardWriter`: Schreibt in das Blackboard und sperrt dieses dazu für alle Zugriffe.

`BlackboardReader`: Liest aus dem Blackboard und sperrt dieses für Schreibzugriffe.

`BlobSender`: Implementiert den in Kapitel 7.2 beschriebenen Sender zur Messung der Performanz.

`BlobReceiver`: Implementiert den Empfänger für die Performanz-Messungen.

`TypeCollectionSender`: Generiert eine `TypeCollection`-Instanz, welche alle Datentypen, die vom Framework serialisiert werden können, enthält.

`TypeCollectionReceiver`: Validiert eine `TypeCollection`-Instanz.

Option	Parameter	Bedeutung	Standardwert
−−help		Zeigt Verwendungshinweise	
−−portoffset, −p	Zahl	Wird auf den Standard Netzwerkport addiert	0
−−debug, −d		Aktiviert Debug-Ausgaben	
−−test, −t	blackboard, timing, stream	Wählt den durchzuführenden Test aus	timing
−−layout, −l	sync, async, separate, sender, receiver	Wählt das Prozesslayout aus	async
−−senderip, −s	IP-Adresse	Sender falls *layout=receiver* gewählt	127.0.0.1
−−reader, −r	Zahl	Legt für den Blackboard-Test Anzahl der Leser fest	5
−−writer, −w	Zahl	Legt für den Blackboard-Test Anzahl der Schreiber fest	5

Tabelle 7.1.: Kommandozeilenoptionen für die Test-Anwendung

Als Testfälle wurden folgende Szenarien gewählt:

blackboard: Mehrere BlackBoardReader und -Writer in je einem eigenen Prozess greifen auf das BlackBoard zu, um den Zugriffsschutz zu testen.

timing: Zwei Module tauschen zur Durchführung von Performanz-Messungen große Datenmengen aus.

stream: Zwei Module tauschen alle vom Framework serialisierbaren Datentypen aus, um die Korrektheit der Streaming-Funktionalität zu testen.

Für alle Testfälle außer *blackboard* kann mittels der Kommandozeilenoption *layout* die Aufteilung der Module auf eventuell mehrere Prozesse festgelegt werden:

sync: Beide Module laufen in einem Prozess.

async: Jedes Modul läuft in einem eigenen Prozess.

separate: Die beiden Module laufen in je einem Prozess in je einer Instanz. Die Verbindung zwischen beiden Instanzen wird automatisch über eine lokale Netzwerkverbindung hergestellt.

sender: Es wird nur das sendende Modul gestartet.

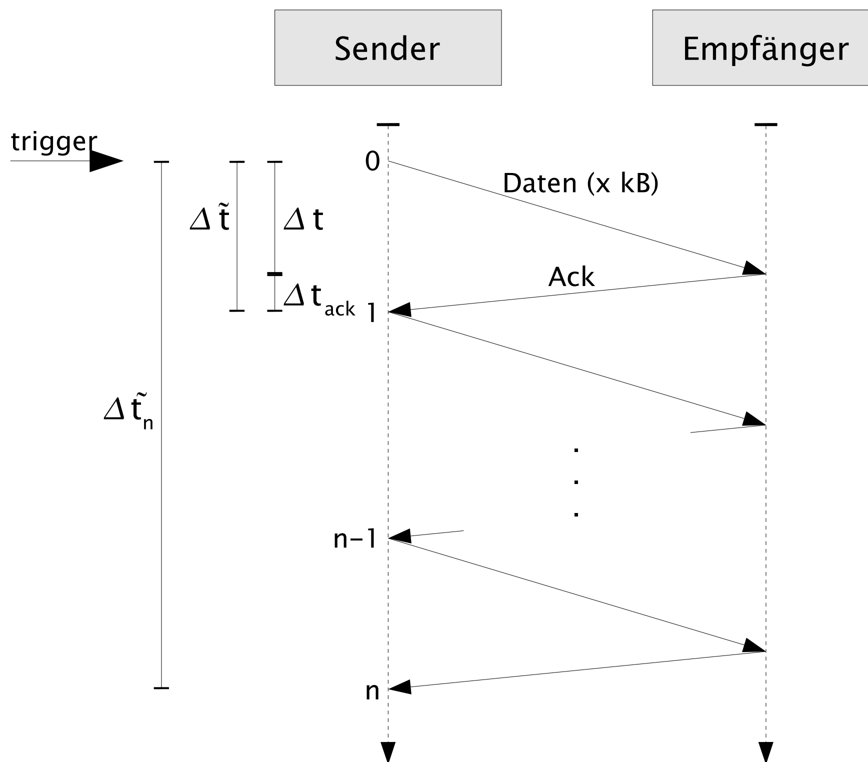

Abbildung 7.1.: Berechnung der Latenz $\Delta\tilde{t}$

receiver: Es wird nur das empfangende Modul gestartet. Es wird versucht, eine Verbindung mit dem Sender herzustellen, dessen IP-Adresse mit Hilfe der Option *senderip* angegeben wird.

Mit Hilfe der Test-Anwendung war es möglich, Funktions- und Performanz-Tests durchzuführen und die grundsätzliche Funktionalität und damit auch die Verwendbarkeit des Frameworks zu überprüfen.

7.2. Performanz

Bei der Verwendung des Frameworks ist die Latenz Δt, also der Zeitunterschied zwischen dem Absenden einer Nachricht und deren Ankunft beim Empfänger, für die Auslegung der Prozessstruktur wichtig (siehe Abbildung 7.1). Δt ist hauptsächlich abhängig davon, ob Sender und Empfänger im selben Prozess liegen oder nicht. Muss eine Nachricht vom Framework über Prozess- oder gar Instanzgrenzen hinweg transportiert werden, so steigt mit ihrer Größe auch die Dauer der Übertragung. Dies ist darauf zurückzuführen, dass die Daten eventuell mehrfach kopiert werden müssen. Aber auch die Streaming-Operatoren können die Latenz erhöhen, falls vom Anwender

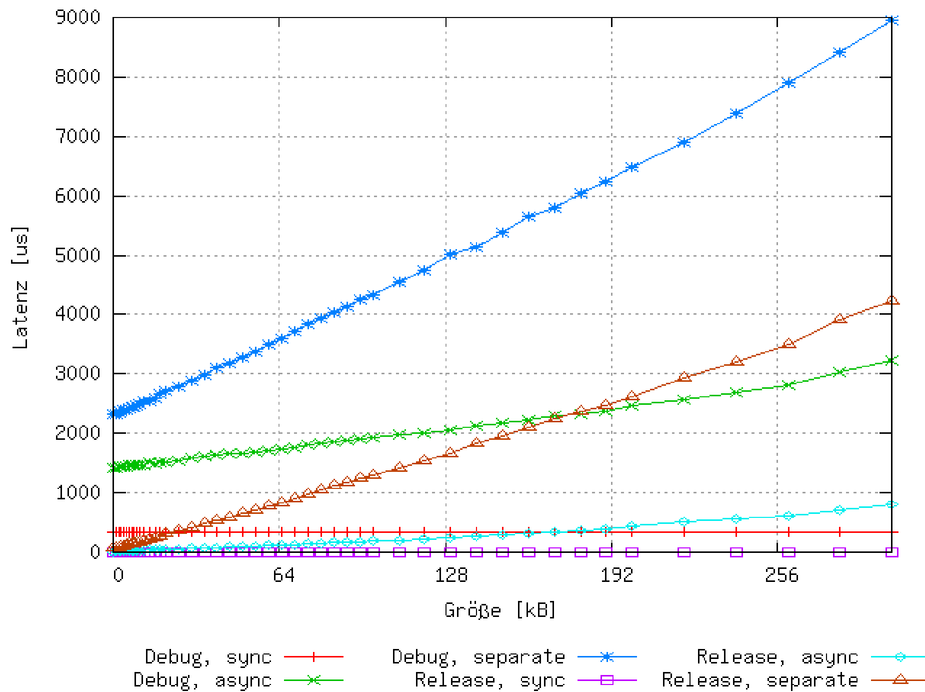

Abbildung 7.2.: Messergebnisse für $\Delta\tilde{t}$ unter Windows XP

eine ineffiziente Implementierung gewählt wurde.

Eine direkte Messung von Δt ist aufgrund der geringen Auflösung der Systemuhren von ca. 10 ms nicht möglich. Stattdessen wurde über mehrere Datenübertragungen gemittelt. Dazu schickt der Sender aufgrund eines externen Signals eine Nachricht an den Empfänger. Dieser antwortet unverzüglich mit einer Quittierungsnachricht (Ack) und der Vorgang wiederholt sich nun mehrmals. Nach n Durchläufen bricht der Sender den Vorgang ab und ermittelt $\Delta\tilde{t}_n$. Dabei wird n so gewählt, dass $\Delta\tilde{t}_n$ ausreichend hoch ist, um Messfehler aufgrund der Auflösung der Systemuhr ausgleichen zu können.

Damit ist es möglich, eine mittlere Zeit $\Delta\tilde{t} = \Delta\tilde{t}_n/n$ für ein Nachricht-Antwort-Paar zu berechnen. Für den Fall, dass die Größe der Quittierungsnachricht der Größe der Nachricht entspricht, gilt $\Delta t = \Delta t_{ack}$, und damit $\Delta t_{ack} = \Delta\tilde{t}/2$.

Als Testsysteme kamen folgende Konfigurationen zum Einsatz:

- Windows XP Professional SP 2 auf Intel Centrino 1,5 GHz, 1024 MB RAM
- Windows CE 4.2 auf Fujitsu Siemens Pocket LOOX 420 BTWL, Intel PXA255 Prozessor mit 400 MHz
- FreeBSD 5.3 auf Intel Centrino 1,5 GHz, 1024 MB RAM

In Abbildung 7.2 sind die Messergebnisse unter der Plattform Windows XP dargestellt. Dabei wird sowohl zwischen den verschiedenen Prozesslayouts als auch zwi-

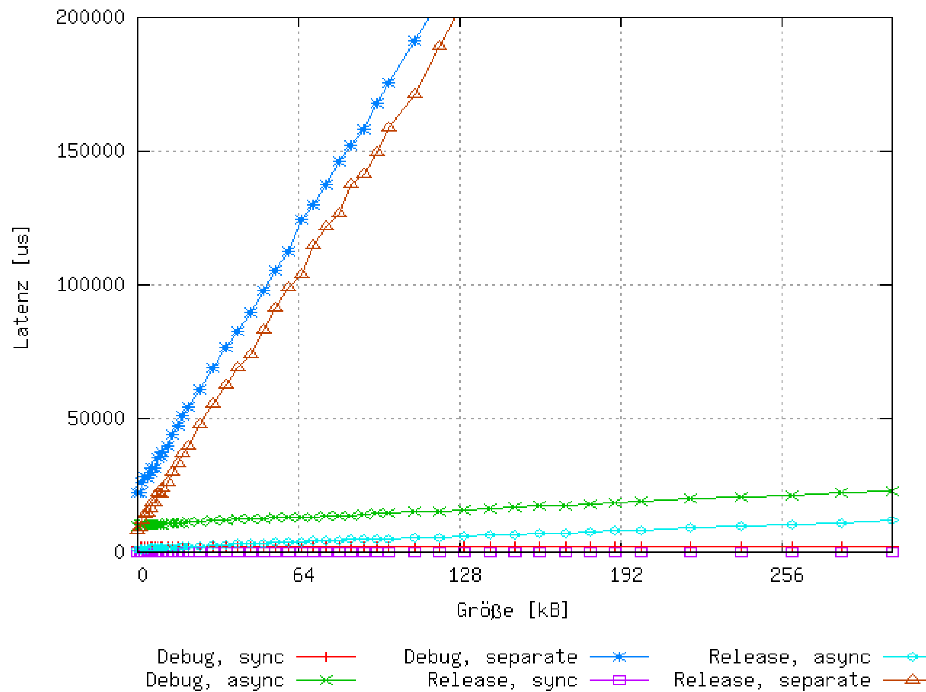

Abbildung 7.3.: Messergebnisse für $\Delta\tilde{t}$ unter Windows CE

schen den Konfigurationen Debug und Release unterschieden. Es wird deutlich, dass, falls Sender und Empfänger im selben Prozess liegen, die Latenz wie erwartet unabhängig von der Größe der Nachricht ist und im Bereich weniger Mikrosekunden liegt. Dies ist darauf zurückzuführen, dass die Daten nicht kopiert werden müssen und kein Kontext-Wechsel zwischen Prozessen stattfinden muss. Sobald Sender und Empfänger in zwei verschiedenen Prozessen einer Instanz liegen, steigt die Latenz mit der Größe der Nachricht leicht an. Dieser Anstieg ist nichtlinear, was auf den erhöhten Verwaltungsaufwand beim Speichern größerer Nachrichten in den zugrundeliegenden Queues zurückzuführen ist. Bei der Netzwerk-basierten Kommunikation über Instanzgrenzen hinweg kommt dieser Effekt noch stärker zum Tragen, da die Datenpakete auf beiden Seiten von den IP-Stacks behandelt werden müssen. Des Weiteren ist aus der Abbildung erkennbar, dass für den produktiven Einsatz des Frameworks unbedingt die Release-Konfiguration verwendet werden sollte. Die Debug-Variante ist aufgrund der abgeschalteten Compileroptimierungen und der zahlreichen Log-Aufrufe um Größenordnungen langsamer und eignet sich daher nur für die Fehlersuche. Dieser Effekt kommt gerade bei kleinen Datenpaketen deutlich zum Tragen.

Die Messergebnisse unter Windows CE sind in Abbildung 7.3 dargestellt. Aufgrund der geringen Rechenleistung sind die gemessenen Latenzen deutlich höher als unter den beiden anderen Plattformen. Allerdings sind die Kurvenverläufe bei den Prozesslayouts *sync* und *async* durchaus mit denen unter Windows XP vergleichbar. Einzig die Latenzen der Nachrichtenübertragung über Instanzgrenzen hinweg sind wesentlich hö-

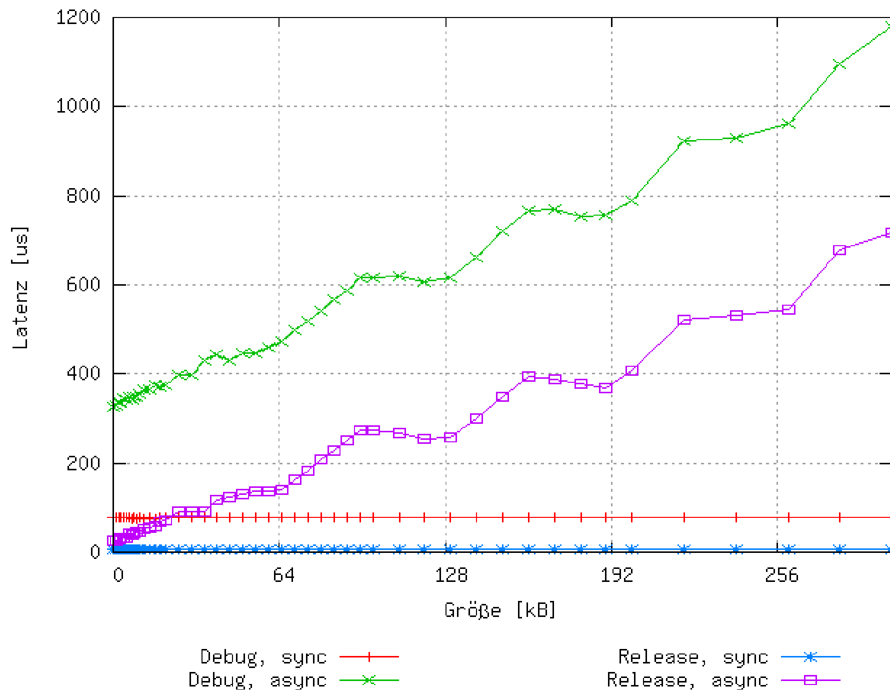

Abbildung 7.4.: Messergebnisse für $\Delta\tilde{t}$ unter FreeBSD

her und steigen mit der Größe der Nachrichten extrem an. Die durch die Übertragung erzeugte zusätzliche Last kann das Laufzeitverhalten der gesamten Anwendung auf dem Pocket PC negativ beeinflussen.

Die unter FreeBSD gemessenen Latenzen sind in Abbildung 7.4 dargestellt und entsprechen sowohl von der Größenordnung als auch vom Verlauf nahezu denen unter Windows XP. Einzig beim Prozesslayout *async* treten deutliche Abweichungen vom gleichmäßigen Ansteigen auf. Offensichtlich werden größere Speicherbereiche bis zu einem gewissen Maß schneller verarbeitet als kleinere. Da sich deren Größe durch die Chunk-Größe definiert, ist ein sich wiederholendes Muster erkennbar. Dies kann nur durch das betriebssystem-spezifische Speichermanagement erklärt werden.

Falls Sender und Empfänger nicht im selben Prozess liegen, spielt die Größe der von den Queues am Stück allokierten Speicherbereichen bei großen Datenpaketen eine nicht unerhebliche Rolle. Daher werden in Abbildung 7.5 die Latenzen unter Windows XP mit dem Prozesslayout *async* und der Release-Konfiguration mit mehreren verschiedenen Chunk-Größen dargestellt. Dabei zeigt sich, dass, falls die Chunk-Größe klein gewählt wird, durch das Anfordern mehrerer Speicherbereiche die Latenz stärker steigt. Der im Framework verwendete Standardwert von 64 kB hat sich als guter Kompromiss herausgestellt.

Die Messergebnisse zeigen, dass die Übertragungslatenzen zwischen zwei Modulen trotz der hohen Abstraktion des Frameworks sehr gering ausfallen.

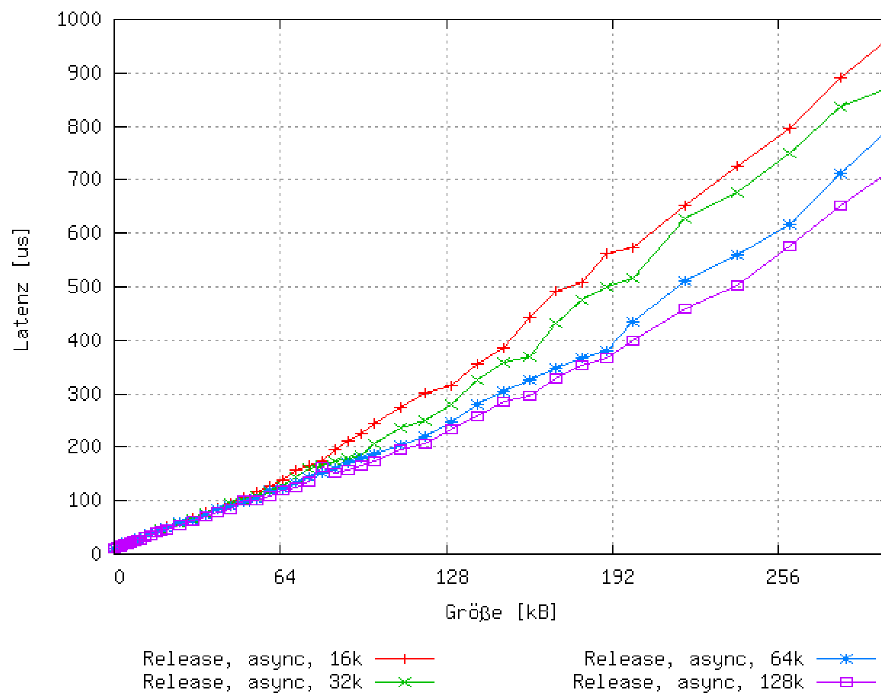

Abbildung 7.5.: Messergebnisse für $\Delta\tilde{t}$ unter Windows XP mit verschiedenen Chunk-Größen

7.3. RoboCup-Anwendungen

Auf Basis des Frameworks wurden von Praktikumsgruppen für die Teilnahme an den *RoboCup*-Meisterschaften Module und Anwendungen entwickelt. Diese werden im Folgenden kurz vorgestellt.

7.3.1. RoboCup GermanOpen 2005

Innerhalb der kurzen Vorbereitungszeit zur *Internationalen RoboCup GermanOpen 2005* in Paderborn war es mit Hilfe des Frameworks möglich, für die beiden Roboterplattformen iHs04 und KHR-1 eine Anwendung zu entwerfen. Diese umfasste eine Bildverarbeitung, eine Bewegungssteuerung und eine rudimentäre Verhaltenssteuerung [4].

7.3.2. RoboCup Weltmeisterschaft 2005

Für die Teilnahme an der *RoboCup Weltmeisterschaft 2005* wurde eine komplexere Struktur, wie in Abbildung 7.6 dargestellt, entworfen. Dabei wurde verstärkt auf einen möglichen Austausch und eine spätere Erweiterbarkeit der Module geachtet.

Der Bildverarbeitung wird ein ImageProvider vorgestellt, der die plattformabhängige Ansteuerung der Kamera beinhaltet. Damit können die bildverarbeitenden Algorithmen sowohl für den iHs04 als auch für den KHR-1 ohne Änderung eingesetzt werden. Die Ausgabe dieses Moduls sind *Perzepte*, also wahrgenommene Objekte, im Roboter-Koordinatensystem (KS). Der Ursprung des Roboter-KS liegt hierbei auf der Spielfeld-Ebene und befindet sich senkrecht unterhalb des Schwerpunktes des Roboters. Die Transformation der Objekte aus dem Kamera-KS wird mit Hilfe einer Kamera-Matrix durchgeführt, die aus den aktuellen Gelenkwinkeln berechnet werden kann.

An diesen Teil der Wahrnehmung schließt sich eine Modellierung an. Hier wird an einer *probablistischen Monte-Carlo Selbstlokalisierung* [3] und einer *Kalman-basierten Ballmodellierung* gearbeitet. Die Informationen des daraufhin aufgebauten Weltmodells werden mit Hilfe des Team-Konnektors (siehe Kapitel 6.3.3) unter den Spielern einer Mannschaft über Wireless LAN ausgetauscht.

Für die Verhaltenssteuerung wurde die bereits in Kapitel 6.3.5 erwähnte XABSL Engine auf die Plattformen portiert. XABSL ermöglicht die Beschreibung von Verhalten für autonome Agenten in XML. Diese XML Dateien werden von XSL-Stylesheets in einen sogenannten Intermediate-Code transformiert, welcher wiederum von der XABSL-Engine interpretiert wird. Neben der Beschreibung in XML Dateien können die Spezifikationen auch in einer menschen-lesbareren Form in *YABSL* angegeben werden. Die dafür nötigen Compiler sind in Ruby[1] geschrieben.

Die bereits bei der GermanOpen eingesetzte Bewegungssteuerung wurde weiter optimiert und mit weiteren Schüssen ausgestattet. Hauptsächlich wurde die erzielte Laufgeschwindigkeit erhöht. Die Soll-Werte der Gelenkstellungen werden von der Bewegungssteuerung nicht nur an die Aktorik gesendet, sondern auch für die Berechnung der Kamera-Matrix bereitgestellt.

Die ursprünglich für eine Win32-Umgebung entwickelten Module konnten durch die Plattformabstraktion des Frameworks leicht auf den iHs04 portiert werden.

[1] Ruby Homepage: `http://www.ruby-lang.org`

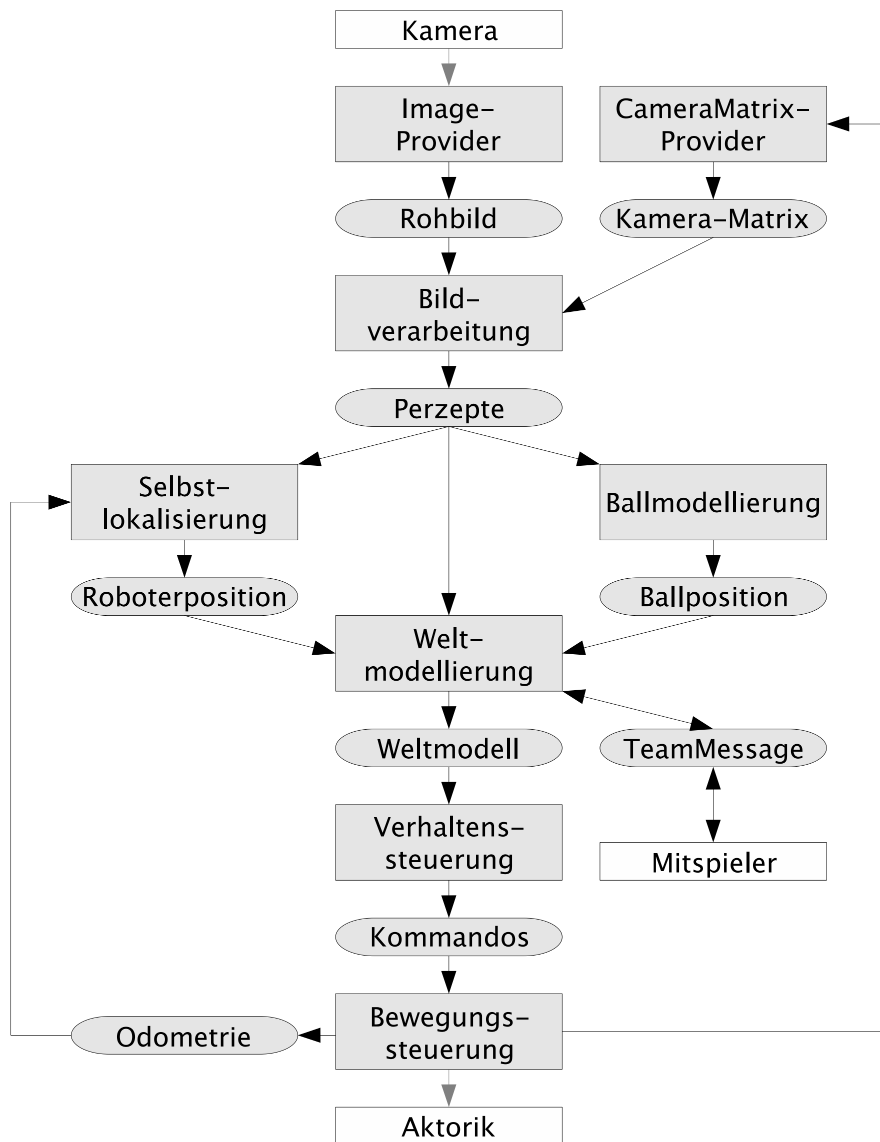

Abbildung 7.6.: Geplante Struktur für den RoboCup 2005
Übersicht über die geplanten Module (rechteckig) und die ausgetauschten Daten (abgerundet).

8. Zusammenfassung und Ausblick

8.1. Zusammenfassung

Das im Rahmen dieser Diplomarbeit entwickelte Framework dient dem Erstellen von Roboter-Steuerungs-Software mit dazugehöriger graphischer Oberfläche zum Testen und Debuggen. Durch die Verwendung des *objektorientierten Programmierparadigmas* und aktueller Software-Engineering-Methoden [19, 6] ist ein flexibles System entstanden, das Anpassungen und auch Erweiterungen leicht ermöglicht. Dadurch ist es möglich, das Framework in Zukunft flexibel auf wechselnde Anforderungen anzupassen, was ein wichtiges angestrebtes Softwarekriterium darstellt. Auch die geforderte *Portierbarkeit* des Frameworks auf weitere Plattformen ist aufgrund der dazwischen geschalteten Plattformabstraktionsschicht mit wenig Aufwand realisierbar, was im Hinblick auf die Anschaffung neuer Roboter zusätzlich für *Investitionssicherheit* sorgt. Bei neuen Plattformen muss nun nicht mehr zwangsläufig mit der Entwicklung wieder von vorne begonnen werden, denn sowohl das Framework als auch entsprechend implementierte Module können unverändert weiterverwendet werden. Trotz der durch das Framework zur Verfügung gestellten Abstraktion und Flexibilität bringt dessen Verwendung nur einen sehr geringen Mehraufwand mit sich.

Auch für einen Entwickler ergeben sich durch die Nutzung des Frameworks eine Reihe von Vorteilen. Der Umfang der durch den Benutzer zu verwendenden Schnittstellen ist überschaubar und ermöglicht eine *schnelle Erlernbarkeit* und Verwendung, ohne dass durch die Struktur des Systems Beschränkungen für die Funktionalität der Anwendung vorgegeben werden. Bereits entwickelte Komponenten lassen sich mit wenig Aufwand wiederverwenden und auch die Entwicklungszeit neuer Komponenten wird durch das Framework reduziert. Die zugrundeliegende Komplexität für die notwendige Kommunikation zwischen den Modulen wird dabei durch das Framework komplett vor dem Benutzer verborgen. Somit kann sich der Anwender ausschließlich auf die funktionalen Komponenten konzentrieren. Davon haben vor allem bereits die Praktikumsgruppen profitiert, welche innerhalb kurzer Zeit auf Basis des Frameworks eine lauffähige Anwendung für die *German Open 2005* und die *RoboCup Weltmeisterschaft 2005* realisiert haben.

8.2. Mögliche Erweiterungen

Da im zeitlich beschränkten Rahmen dieser Diplomarbeit nicht alle denkbaren Aspekte umgesetzt werden konnten, ergeben sich noch einige Möglichkeiten, das Framework funktional zu erweitern. Auch Framework-extern sind Verbesserungen möglich, welche zum einen die Infrastruktur bei der Entwicklung und zum anderen die Programmierung der Module selbst betreffen.

8.2.1. Eigene Container- und Stringklassen

Container, wie zum Beispiel Listen, Queues, Maps oder Strings, werden bei Algorithmen häufig eingesetzt. Unter C++ existiert mit der Standard Template Library (STL) eine stabile Implementierung, die wohl alle möglichen Anwendungsbereiche abdeckt und auf den meisten Plattformen verfügbar ist.

Gerade durch ihre Vielseitigkeit ist die STL aber auch nicht sehr einfach in der Benutzung und in Verbindung mit multi-threaded Anwendungen aufgrund der Synchronisierung nicht sehr schnell. Für Module garantiert das Framework, dass sie nicht nebenläufig aufgerufen werden, daher wird hier die Synchronisation nicht benötigt.

Die in der STL enthaltene String-Klasse zur Verwaltung von Zeichenketten lässt einige Funktionen, wie Vergleiche ohne Berücksichtigung der Groß- und Kleinschreibung oder Konvertierungen, vermissen. Hier ist eine Klasse wünschenswert, die ähnlich der Java String-Klasse diese Funktionalität objektorientiert kapselt.

Da neben der STL auch viele weitere Implementierungen von Containerklassen in C++ existieren, müssen diese eventuell nicht neu entwickelt werden. Bei der Auswahl einer geeigneten Bibliothek müssen jedoch einige Punkte, wie Plattformunabhängigkeit und Typsicherheit, berücksichtigt werden. Einige Containerimplementierungen, wie zum Beispiel die in QT 4 enthaltene *Tulip*-Bibliothek, enthalten auch aus anderen Programmiersprachen übliche Konzepte, wie Java-ähnliche Iteratoren oder `foreach`-Schleifen. Diese sind zum Teil mächtiger als die bestehenden Konzepte und unterstützen so durch einfacheren Quellcode dessen Verständlichkeit.

8.2.2. Ausnahmebehandlung im Framework

Eine *Ausnahme* (engl. Exception) bezeichnet bei modernen Compilern das Weiterreichen von kritischen Programmzuständen zur Behandlung an höhere Ebenen. Dort können dann Maßnahmen ergriffen werden, um den Fehler aufzulösen oder ihn kontrolliert zu übergehen. Ausnahmen ermöglichen dem Programmierer, Fehlerbehandlungscode von Algorithmen getrennt einzubinden und damit die Erstellung flexiblerer

und fehlertoleranterer Anwendungen.

Unter C++ ist eine Unterstützung von Ausnahmen erst in neueren Compilern vorhanden. Der GCC 2.96, der zurzeit noch für den iXs verwendet wird, unterstützt sie nicht.

Um bei der Erstellung von Modulen Ausnahmen benutzen zu können, müsste das Framework zum einen eine Ausnahmebehandlung bereitstellen, zum anderen bei fehlerhaften Zugriffen auf Framework-Methoden Ausnahmen werfen. Dies bedeutet in den meisten Fällen eine Änderung der entsprechenden Methoden-Signaturen, da Fehlersituationen nicht mehr über den Rückgabewert oder zusätzliche Methoden abgefragt, und damit umfangreiche Änderungen an den Framework-Klassen notwendig würden.

Um Ausnahmen überall konsistent einzubauen, muss zusätzlich untersucht werden, welche Unterstützung durch die vorhandenen Compiler gegeben ist, oder welche neuen Werkzeuge benötigt werden.

Außerdem ist zu beachten, dass bei der Ausführung von Programmen mit Ausnahmebehandlung zusätzlicher Code ausgeführt werden muss, um weitere Informationen zu protokollieren. Dies wirkt sich negativ auf die Laufzeit der restlichen Programmteile aus. Dieser Effekt kann bei verschiedenen Compilern unterschiedlich ausfallen. Daher müssen vor einer Integration von Ausnahmen in das Framework auf allen Plattformen Tests durchgeführt werden, um den entstehenden zusätzlichen Aufwand quantifizieren zu können.

8.2.3. Portierung der GUI auf QT 4

Während der Vorbereitung auf die *RoboCup Weltmeisterschaft 2005* ist Version 4 der von RoboGui verwendeten QT Bibliothek erschienen. QT 4 bricht zum Teil mit Konventionen, die unter QT 3 verwendet wurden, um zahlreiche Neuerungen und Verbesserungen realisieren zu können. Zu nennen sind hier hauptsächlich die *Interview* und *Arthur* genannten Technologien, die eine *Model-View-Architektur* und verbesserte 2D-Zeichen API beinhalten. Diese Technologien können die Visualisierung von Algorithmen wesentlich vereinfachen und damit deren Entwicklung beschleunigen.

Für den leichten Umstieg auf Version 4 bringt diese eine Sammlung von QT 3 kompatiblen Klassen mit. Bei Verwendung dieser Klassen können jedoch viele der neuen Merkmale nicht genutzt werden. Bei einer Portierung auf QT 4 sollte deswegen von Anfang an auf die erweiterten Möglichkeiten zurückgegriffen werden, was neben Änderungen an RoboGui auch eine Anpassung der bestehenden Visualisierungen erforderlich macht.

Das QT 4 beiliegende Werkzeug *qt3to4* kann Routineänderungen bei der Portie-

rung automatisch vornehmen. Die Trolltech-Entwickler haben darauf geachtet, dass bei diesem Vorgang nicht gelöste Probleme meistens zu Kompilierfehlern führen und somit leichter identifizierbar sind als Laufzeitfehler. Dennoch ist bei einem Umstieg mit zahlreichen und umfangreichen Änderungen zu rechnen.

Die zurzeit verwendete Version 3.3 wird nach Angaben von Trolltech bis Juni 2007 weiterhin unterstützt[1].

8.2.4. Mathematische Funktionsbibliothek

Viele der für eine Robotersteuerung eingesetzten Algorithmen basieren auf mathematischen Verfahren. Um den Modulentwicklern einen plattformunabhängigen Zugriff auf die benötigten Funktionen zu geben, wäre die Entwicklung einer mathematischen Funktionsbibliothek anzustreben. Diese könnte neben den üblichen Basisfunktionen auch eigene Datentypen wie Vektoren, Matrizen und Fixkommazahlen bereitstellen.

Um Rechenzeit zu sparen, könnte die Berechnung der häufig genutzten trigonometrischen Funktionen durch Verwenden von Tabellen oder Approximationen ersetzt werden. Die zusätzliche Verwendung von Fixkommazahlen statt Gleitkommazahlen würde sich bei Plattformen ohne mathematischen Koprozessor positiv auf die Ausführungszeiten auswirken.

8.2.5. Speichern der GUI-Einstellungen

Bei der Verwendung der GUI muss man nach jedem Start der graphischen Oberfläche zunächst die entsprechenden Dialoge öffnen und innerhalb dieser eventuell noch weitere Einstellungen vornehmen. Um diesen Aufwand zu reduzieren, wäre es sinnvoll, dass beim Schließen der Anwendung die Auswahl der geöffneten Dialoge und deren Einstellungen gespeichert werden können. Dadurch wäre es möglich, nach einer Sitzung beim erneuten Aufruf an derselben Stelle weiterzuarbeiten.

Eine weitere sinnvolle Ergänzung wäre es, mehrere unterschiedliche Sitzungen getrennt zu archivieren, um für unterschiedliche Tätigkeiten verschiedene Konfigurationen auswählen zu können.

[1] Trolltech Support and Maintenance Program: `http://www.trolltech.com/support/`

8.2.6. Automatische Tests für Plattformkompatibilität

Aufgrund der eingesetzten *heterogenen Entwicklungs- und Roboterplattformen* ist es schwierig, wenn nicht gar unmöglich, Änderungen vor dem Bereitstellen auf dem SVN-Server auf allen Systemen zu testen. Dabei ist das Kompilieren alleine nicht hinreichend um zu beurteilen, ob ein Programm korrekt funktioniert. Da es durch die verschiedenen Compiler und die unterschiedlichen Implementierungen der STL gegebenenfalls auf einzelnen Systemen zu Fehlern kommen kann, wäre es daher sinnvoll *Regressionstests* für das Framework bereitzustellen. Dadurch ist es wesentlich einfacher möglich, die Funktionsfähigkeit nach durchgeführten Anpassungen sicherzustellen. Fehler im Quellcode, welche nur auf bestimmten Plattformen auftreten, würden schneller erkannt.

Um eine verlässliche Aussage über die Korrektheit des Quellcodes treffen zu können, muss dieser durch die gewählten Testfälle komplett abgedeckt werden. Um diese Abdeckung zu messen, bieten sich Metriken an, die durch eine sogenannte *Code-Coverage-Analyse* [14] ermittelt werden können. Hierbei wird für jede Zeile während der Ausführung protokolliert, ob und wie oft sie ausgeführt wurde. Bei C++ erfordert dies die Verwendung spezieller Werkzeuge, da zusätzlicher Code in den ursprünglichen Quellcode eingearbeitet werden muss. Auf Basis der ermittelten Metriken können dann die Tests erweitert werden und somit deren Qualität erhöht werden. Dennoch ist eine 100%-ige Abdeckung des Quellcodes nicht realisierbar.

Für die Realisierung würde sich die Verwendung der Bibliothek *CppUnit*[2] anbieten. Diese bietet ebenfalls die Möglichkeit, die QT-basierte graphische Oberfläche zu testen.

Als weitere Ergänzung dieser Regressionstest wäre es denkbar, diese automatisch durchzuführen, sobald Änderungen am Quellcode eingecheckt werden. Der bereits verwendete Subversion-Server ist hierfür eine gute Ausgangsbasis, denn dieser ermöglicht es bereits selbst definierte Aktionen nach dem Einchecken auszuführen.

Durch ein bestehendes Test-Framework würde auch die Erstellung von Tests für Module vereinfacht werden. Diese könnten parallel zu neuen Modulen entwickelt werden und würden auch deren Funktionalität und Stabilität sicherstellen.

[2] CppUnit Homepage: `http://cppunit.sourceforge.net`

8.2.7. Sammlung wiederverwendbarer Module und Dialoge

Um die Wiederverwendbarkeit des entwickelten Quellcodes zu maximieren, sollten möglichst große Teile der Anwendungsprogrammierung so generisch implementiert werden, dass diese Komponenten ohne Anpassung auf zahlreichen Plattformen zum Einsatz kommen können. Daher sollte das Anlegen und Pflegen einer entsprechenden Sammlung angestrebt werden. Diese soll plattformunahängige Module, Datenstrukturen und Dialoge enthalten und damit die Komposition von Anwendungen nach dem Baukastenprinzip fördern.

Eine konkrete Applikation benötigt dann im Idealfall lediglich hardware-spezifische Programmteile zum Zugriff auf Aktorik und Sensorik, welche als eigene Klassen gekapselt sind, und einige der vorgefertigten Komponenten aus dieser Bibliothek.

A. Verwendung des Frameworks

Dieser Anhang soll die Einarbeitung in den kommentierten Quellcode erleichtern. Beispielhaft werden hier die nötigen Einzelschritte zur Erstellung einer einfachen Anwendung auf Basis des Frameworks beschrieben. Nach der Erstellung der notwendigen Bibliotheken werden die zu implementierenden Schnittstellen detailliert vorgestellt und anhand eines Beispiels umgesetzt.

A.1. Kompilieren der Bibliotheken

Vor der Erstellung einer eigenen Anwendung müssen zunächst die beiden Bibliotheken des Frameworks kompiliert werden. Zum einen ist dies *RoboApp* für die eigentliche Applikationen, zum anderen *RoboGui* für die graphische Oberfläche. Dies muss nur einmalig geschehen und ist nur nach Aktualisierungen oder Anpassung des Framework-Quellcodes zu wiederholen.

Dazu muss zunächst der Pfad *trunk* des in Anhang D beschriebenen Repository ausgecheckt werden. Der Quellcode des Frameworks liegt dabei unter *roboframe*. Der eigentliche Vorgang zum Erstellen der Bibliotheken ist natürlich abhängig von der jeweiligen Entwicklungsplattform.

- Unter Linux/Unix ist das Skript *rebuild* im Unterverzeichnis *build/autotools* zu benutzen, welches die notwendigen Aufrufe der Autotools durchführt.
- Unter Windows-Systemen muss in Visual C++ lediglich das jeweilige Projekt erstellt werden. Die Projektmappe befindet sich für Visual Studio unter *build/vs*, für eMbedded Visual C++ unter *build/evc*.

A.2. Das Test-Projekt

Um die Funktionsfähigkeit des Frameworks auf der eingesetzten Plattform zu testen, ist es sinnvoll, das Test-Projekt *demo* auszuchecken, zu kompilieren und testweise auszuführen. Nach der Kompilierung liegen zwei ausführbare Dateien vor: zum einen *demoApp*, die Konsolenanwendung, zum anderen *demoGui*, die dazugehörige graphische Oberfläche. Bei der Konsolenanwendung hat man unter Plattformen, welche dies

unterstützen, noch die Möglichkeit, per Kommandozeilenparameter Argumente mitzugeben, welche in Abschnitt 7.1 näher erläutert sind.

A.3. Neues Projekt erstellen

In der Konzeptionierungsphase für ein neues Projekt muss man sich genau überlegen, in welche Module man die Anwendung aufteilen möchte und mit welchen Datenstrukturen diese miteinander kommunizieren. Dabei sollte, wie bei anderen Softwareprojekte, auf eine niedrige Kopplung mit möglichst kleinen Schnittstellen zwischen Modulen und einer hohen Kohäsion innerhalb von Modulen geachtet werden.

Das folgende Beispiel kann nur einen ersten Einblick geben, was mit dem Framework möglich ist. Auf eine detaillierte Kommentierung wurde im Rahmen dieser Einführung verzichtet. Der Quellcode dient nur der Illustration und erhebt keinen Anspruch auf Vollständigkeit. Zur vollständigen Beschreibung der Funktionalität wird an dieser Stelle auf Kapitel 6 verwiesen.

Der Quellcode des Framework enthält dagegen eine umfangreiche Kommentierung und mittels doxygen[1] kann daraus eine vollständige Dokumentation generiert werden.

Zunächst sollte man ein Verzeichnis auf der Ebene von *roboframe* anlegen entsprechend dem Namen des Projektes. In diesem Verzeichnis gibt es - wie auch beim Projekt *example* - die Unterverzeichnisse *build* und *src*. Der gesamte Quellcode des Projektes liegt hierbei im *src*-Verzeichnis. Dagegen befinden sich alle kompilierten Objekte und Skripte zum Übersetzen der Anwendung getrennt davon unterhalb des *build*-Verzeichnisses.

Die Projektdateien können aus dem Example-Projekt übernommen werden, um unnötigen Aufwand zu vermeiden. Allerdings sind dort in der Regel die projektspezifischen Dateien einzupflegen.

Des Weiteren sollten die anwendungsabhängigen Module, Dialoge und serialisierbaren Datenobjekte aus Übersichtlichkeit und Konsistenz zu den restlichen Projekten, analog zu *example* und *demo*, in die Unterverzeichnisse *module*, *dialog* und *streamable* aufgeteilt werden.

Für diese kurze Einführung dient ein Beispiel mit drei Modulen. Dabei soll das erste Modul ein Bild von der Kamera aufnehmen und durch einen OutBuffer bereitstellen. Das zweite Modul soll die Gelenkwinkel ermitteln und ebenfalls anderen Modulen zur Verfügung stellen. Das dritte und letzte Modul soll sowohl Bilder als auch Gelenkwinkel empfangen und auf der Konsole bei jedem Bild Informationen zu den Gelenkwinkeln ausgeben.

[1] doxygen Homepage: `http://www.doxygen.org`

A.3.1. Datenstrukturen definieren

Zunächst werden die Datenstrukturen für den Informationsaustausch zwischen den Modulen definiert.

Damit eigene Datenobjekte zwischen Modulen ausgetauscht werden und auch zwischen Instanzen verschickt werden können, müssen sie zwei Streaming-Operatoren zur Verfügung stellen. Dazu sollte die Daten-Klassen von IStreamable erben. Damit wird durch das Interface sichergestellt, dass die notwendigen Operator-Funktionen implementiert werden müssen. Ansonsten treten bereits beim Kompilieren entsprechende Fehler auf. Alternativ kann diese Funktionalität auch durch globale Funktionen bereitgestellt werden.

Sollte dies nicht möglich oder gewünscht sein, können diese Operatoren auch außerhalb der Klasse zur Verfügung gestellt werden. Sollten diese Operatoren, die dem Serialisieren und Deserialisieren dienen, fehlen, führt dies beim Linken der Programme zu Fehlern.

Die Klasse für die Repräsentation von Bildern (Listing A.1) enthält in der Klassenvariable `grayValues` für jeden Bildpunkt der Kamera einen Graustufenwert und erbt von `IStreamable`.

```
class Bild : public IStreamable
{
  char[320][240] grayValues;

  InStream& operator << (const InStream& stream) {
    stream.read(&grayValues[0], 320*240);
  }

  OutStream& operator >> (OutStream& stream) const {
    stream.write(&grayValues[0], 320*240);
  }
}
```

Listing A.1: streamable/Bild.h

Die andere Datenstruktur (Listing A.2) enthält in der Klassenvariable `gelenkwerte` für mehrere Gelenke die jeweiligen Werte und implementiert die notwendigen Operatoren außerhalb der Klasse.

```
class Gelenkwerte
{
  vector<float> gelenkwerte;
}


InStream& operator << (Gelenkwerte& obj,
```

```
    const InStream& stream) {^
  obj.gelenkwerte.clear();
  int count;
  float value;
  stream >> count;
  for (i = 0; i < count; i++) {
    stream >> value;
    obj.gelenkwerte.insert(value);
  }
}

OutStream& operator >> (const Gelenkwerte& obj,
    OutStream& stream) {
  int count = gelenkwerte.size();
  stream << count;
  for (i = 0; i < count; i++)
    stream << obj.gelenkwerte[i];
}
```

Listing A.2: streamable/Gelenkwerte.h

In den angesprochenen Streaming-Operatoren sollten alle Klassenvariablen in den Stream geschrieben bzw. daraus gelesen werden. Für die primitiven Datentypen und einige STL-Klassen werden die dafür notwendigen Operatoren bereits durch das Framework bereitgestellt. Bei Pointern ist hierbei Vorsicht geboten, damit nicht die Speicheradresse selbst sondern der eigentliche Inhalt (de-)serialisiert wird.

A.3.2. Module realisieren

Als Nächstes werden die Module realisiert, welche von der Klasse `Module` erben.

Wie bei einigen anderen Klassen auch, gibt es die Methoden `init` und `cleanUp`. Diese können überschrieben werden, um initialisierende und bereinigende Aktionen durchzuführen. Dabei wird sichergestellt, dass `init` immer vor der ersten Benutzung und `cleanUp` vor dem Destruktor des Moduls aufgerufen wird (siehe Abschnitt 6.1.3).

Um mit anderen Modulen oder Dialogen Informationen austauschen zu können, benutzt man sogenannte Puffer oder geteilte Speicherbereiche (siehe Abschnitt 6.1.4). Im Rahmen dieser Beispielanwendung werden allerdings nur Puffer zur Kommunikation genutzt. Diese werden als Klassenvariablen angelegt.

Bei Puffern wird hierbei zwischen zwei Varianten unterschieden: Puffer für eingehende Daten - sogenannte `InBuffer` oder für ausgehende Daten - die `OutBuffer`. Beide Puffervarianten sind Template-Klassen und erwarten entsprechende Template-Parameter. Neben der Datenstruktur-Klasse und der Größe des Puffers wird auch ein sogenannter Schlüssel mit angegeben. Ein Schlüssel wird an einer zentralen Stelle de-

finiert und bezeichnet eine konkrete Nachricht, welche zwischen zwei oder mehreren Modulen ausgetauscht wird. Auf diese Schlüssel wird in Abschnitt A.3.3 nochmals genauer eingegangen. Bei der Größe der OutBuffer ist darauf zu achten, dass sie ausreichend dimensioniert sind, um alle Elemente, die in einem Durchgang angelegt werden, aufnehmen zu können.

Das erste Modul (Listing A.3) sollte Bildobjekte generieren - dies geschieht in der `execute`-Methode - und diese Daten verschicken.

```
class ModulEins : public Module
{
  void getModuleDescriptor(ModuleDescriptor& desc) {
    desc.setName("Modul Eins");
    desc.addBuffer(&outBilder);
  }

  void init() {...} // kamera-treiber initialisieren

  void cleanUp() {...} // kamera-treiber zurücksetzen

  void execute(const ExecuteFlags&) {
    Bild& bild = outBilder.add();
    // bild mit sinnvollen werten befüllen
    ...
    outBilder.finish();
    LOGPATH_DEBUG("ModulEins", "bild verschickt");
  }

  OutBuffer<KEY_BILD, Bild, 1> outBilder;
}
```

Listing A.3: module/ModulEins.h

In der zu implementierenden Methode `getModuleDescriptor` kann das Modul den eigenen Namen angeben. Des Weiteren müssen sämtliche Puffer und Black-Boards beim Deskriptor-Parameter mit den entsprechenden Methoden `addBuffer` und `addBoard` registriert werden. Somit kann dem Framework von jedem Modul die für die Kommunikation notwendigen Daten zur Verfügung gestellt werden.

Beim Hinzufügen des Bildes zum OutBuffer wird die `add`-Methode ohne Argumente benutzt, um entgegen einer Übergabe des Bildobjektes als Parameter das Kopieren der Datenstruktur zu vermeiden.

Das zweite Modul (Listing A.4) sollte die Gelenkwerte ermitteln und ebenfalls verschicken und wird auf gleiche Weise realisiert.

```
class ModulZwei : public Module
{
  void getModuleDescriptor(ModuleDescriptor& desc) {
    desc.setName("Modul Zwei");
    desc.addBuffer(&outGelenkwerte);
  }

  void execute(const ExecuteFlags&) {
    Gelenkwerte gelenkwerte;
    // gelenkwerte mit sinnvollen werten befüllen
    ...
    outGelenkwerte.add(gelenkwerte);
    LOGPATH_DEBUG("ModulEins", "werte verschickt");
  }

  OutBuffer<KEY_WINKEL, Gelenkwerte, 1> outGelenkwerte;
}
```

Listing A.4: module/ModulZwei.h

Das dritte Modul (Listing A.5) sollte beide Datenstrukturen entgegennehmen und bei neu eintreffenden Bildern eine Ausgabe durchführen. Dabei müssen die Schlüssel der InBuffer mit denen der entsprechenden OutBuffer übereinstimmen, damit das Framework diese beiden mit einem Kommunikationsweg verbinden kann.

Die Besonderheit gegenüber den beiden anderen Module besteht darin, dass dieses Modul nur bei eingehenden Daten Aktionen durchführen soll. Für jede eingehende Datenstruktur, die in einem der InBuffer landet, wird die Methode `notify` mit dem entsprechenden Schlüssel als Parameter aufgerufen. Dieser Aufruf findet nach dem Eintragen in den Puffer, aber vor dem eventuellen Aufruf der `execute`-Methode statt. Wenn diese Methode den Rückgabewert `true` liefert, wie dies in der Elternklasse realisiert ist, wird die `execute`-Methode aufgerufen. In diesem Beispiel soll das Modul aber nur nach eingehenden Bildern und nicht bei Gelenkwerten ausgeführt werden. Daher muss die Methode `notify` entsprechend überschrieben werden. Man beachte, dass diese Funktion keine zeitaufwendigen Aktionen durchführen soll - dies sollte ausschließlich in der `execute`-Methode geschehen.

```
class ModulDrei : public Module
{
  void getModuleDescriptor(ModuleDescriptor& desc) {
    desc.setName("Modul Drei");
    desc.addBuffer(&inBilder);
    desc.addBuffer(&inGelenkwerte);
  }

  bool notify(Key& key) {
    if (key == KEY_WINKEL) return false;
    return true;
  }
```

```
  void execute(const ExecuteFlags&) {
    const Bild& bild = inBilder.get();
    const Gelenkwerte& alt = inGelenkwerte.get(1);
    const Gelenkwerte& neu = inGelenkwerte.get(0);
    // abhängig von der differenz der beiden gelenkwerte
    SystemCall::println("Die Kamera wurde während der
        Ausnahme des Bildes schnell/langsam bewegt");
  }

  InBuffer<KEY_BILD, Bild, 1> inBilder;

  InBuffer<KEY_WINKEL, Gelenkwerte, 2> inGelenkwerte;
}
```

Listing A.5: module/ModulDrei.h

Wichtig ist es, die Datenstrukturen beim Abfragen aus dem InBuffer nicht implizit zu kopieren, sondern die Variablen per Referenz zuzuweisen. Neben dem Grund der besseren Performanz kann es auch passieren, dass bei zu großen Datenobjekten der Programmstack überschrieben wird, da kurzfristig viele Daten auf dem Stack abgelegt werden.

Bei der Ausgabe von Textnachrichten auf die Konsole ist die Plattformunabhängigkeit zu beachten. Da nicht alle Plattformen Streams und damit `cout` unterstützen, muss hierfür die Methode `println` der SystemCall-Klasse verwendet werden.

A.3.3. Schlüssel spezifizieren

Die bereits kurz erwähnten Schlüssel sollten in einer separaten Datei (Listing A.6) definiert werden. Dabei handelt es sich um eindeutige Zahlen. Laut Konvention sind Zahlen unterhalb von 1000 für das Framework reserviert. Des Weiteren wird eine statische Methode benutzt, um alle Schlüssel bei der KeyRegistry zu registrieren, damit dies nicht in der Applikation und der GUI dupliziert wird.

```
#define KEY_BILD 1001
#define KEY_WINKEL 1002

class BeispielKeys
{
  static void registerKeys() {
    KeyFactory::registerKey(KEY_BILD, "KeyBild",
      new RepresentationWrapper<Bild>);
    KeyFactory::registerKey(KEY_WINKEL, "KeyGelenkwinkel",
      new RepresentationWrapper<Gelenkwerte>);
    LOGPATH_INFO("BspKeys", "%d schlüssel registriert", 2);
  }
}
```

Listing A.6: BeispielKeys.h

Damit sind die Datenstrukturen und auch die funktionalen Komponenten fertig gestellt. Nun müssen diese Teile noch zu einer vollständigen Applikation zusammengesetzt werden.

A.3.4. Applikation zusammenfügen

Die Anwendungsklasse (Listing A.7) erbt von `Application` und legt dabei die verwendeten Module und deren Aufteilung auf mehrere Prozesse fest. In der `init`-Methode müssen die notwendigen Prozesse angelegt und konfiguriert werden, welche dann die Module aufnehmen. Mehrere Module innerhalb eines Prozesses werden immer seriell in der Reihenfolge ihres Hinzufügens ausgeführt. Wenn Module innerhalb eines Prozesses nicht ausschließlich ausgeführt werden sollen, wenn von extern Daten empfangen werden, kann man den Prozess mit einem Zeitintervall konfigurieren, nach dem automatisch ein Aufruf erfolgt. Diese unterschiedliche Art des Aufrufes von Modulen kann in der `execute`-Methode des Moduls anhand des Parameters abgefragt werden.

```
class BeispielApplikation : public Application
{
  void init() {
    // eltern-methode aufrufen um die
    // standard-appender beim logging zu nutzen
    Application::init()

    // zusammenstellen der module und prozesse
    Process* proc1 = createProcess();
    proc1->setName("Process Bildgenerierung");
    proc1->setTimerMilliseconds(500);
    proc1->addModule(new ModulEins());

    Process* proc2 = createProcess();
    proc2->setName("Process Gelenkwinkel");
    proc2->setTimerMilliseconds(80);
    proc2->addModule(new ModulZwei());

    Process* proc3 = createProcess();
    proc3->setName("Process Auswertung");
    proc3->addModule(new ModulDrei());

    LOGPATH_INFO("BspApp", "%d prozesse angelegt", 3);
  }

  void initKeys() {
    BeispielKeys::registerKeys();
  }
}
```

Listing A.7: BeispielApplikation.h

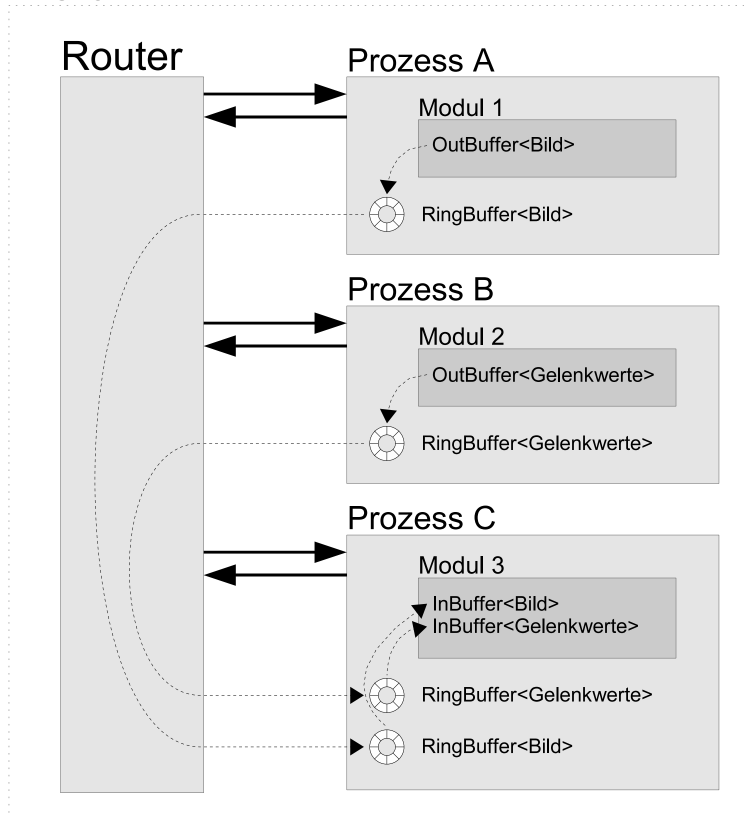

Abbildung A.1.: Komponenten der Beispielanwendung

In unserem Beispiel sollen die Gelenkwinkel öfter verschickt werden als die Bilder. Deshalb werden diese beiden Module nach einem festen Zeitintervall aufgerufen - im Gegensatz zum dritten Modul, welches ohne zeitgesteuerten Aufruf konfiguriert wird. Dadurch ergibt sich eine Prozessstruktur wie in Abbildung A.1 dargestellt. In dieser Abbildung ist sehen, wie die Daten der `In-/OutBuffer` der Module in einem `RingBuffer` des Prozesses gespeichert werden. Von dort werden ausgehende Daten über den Router in andere Ringpuffer weitergeleitet, um diese Daten den Modulen in einem anderen Prozess zur Verfügung zu stellen.

Neben der Prozessstruktur müssen die oben erwähnten Schlüssel beim Framework registriert werden. Dadurch kann das Framework zum einen sicherstellen, dass keine Schlüsselwerte doppelt vorkommen, und zum anderen überprüfen, dass der Schlüssel an allen Stellen im Quellcode mit der korrekten Datenstruktur benutzt wird. Diese Registrierung erfolgt in der Methode `initKeys`.

Zu einer lauffähigen Anwendung fehlt nun noch eine *main-Funktion* (Listing A.8). Diese hat einzig die Aufgabe, von der Applikation die `execute`-Methode aufzurufen. Um diese Anwendung korrekt beenden zu können, wird noch eine Funktion hinzugefügt, welche die Applikation beim Drücken von STRG+C sauber beendet.

```
BeispielApplilation* app;

void catchSignal (int sig) {
  app->stop();
  signal(sig, catchSignal);
}

int main(int argc, char *argv[]) {
  signal(SIGINT, catchSignal);

  app = new BeispielApplikation();
  int retVal = app->exec();
  delete app;
  return retVal;
}
```

Listing A.8: mainApp.h

Um die Applikation unter Windows CE ausführen zu können, wird eine eigene *main*-Datei (Listing A.9) benötigt, welche auf einen vorgefertigten Dialog zurückgreift, in dem die Anwendung ausgeführt wird. Dabei bringt das Framework alle notwendigen MFC-spezifischen Implementierungen bereits mit, damit anwendungsseitig nur ein minimaler Aufwand notwendig ist. Da hierbei die Verwendung der bereitgestellten MFC-Komponenten wichtig ist, werden auch die notwendigen Includes angegeben.

```
#include <wince/StdAfx.h>

#include "BeispielApplikation.h"
#include <wince/WinCeApp.h>

static roboapp::WinCeApp<BeispielApplication> ceApp;
```

Listing A.9: mainCeApp.h

A.3.5. Projektdateien für die Applikation

Bei den Projektdateien wird zwischen drei unterschiedlichen Plattformen unterschieden: Autotools (Linux/Unix), Visual Studio (unter Windows) und eVC (für Windows CE).

Autotools

Bei den *Autotools* sollte für ein eigenes Projekt auf die Dateien von *example* zurückgegriffen und diese entsprechend modifiziert werden. Dadurch wird unter anderem erreicht, dass alle Projekte das Cross-Kompilieren unterstützen und sich allgemeine Einstellungen, wie zum Beispiel das Aktivieren des Debug-Modus, vornehmen lassen.

In der Datei *configure.ac* muss

- im Makroaufruf `AC_INIT` der Programmname und die Versionsnummer eingetragen werden und
- innerhalb von `AC_CONFIG_SRCDIR` der relative Pfad auf eine beliebige Datei des eigenen Quellcodes.

Mehr Änderungen sind dabei in der Datei *Makefile.am* notwendig.

- Zunächst sollte der Projektname bei jedem Vorkommnis angepasst werden.
- Sollte für ein Projekt keine GUI benötigt werden, kann das entsprechende Ziel unter `noinst_PROGRAMS` entfernt werden.
- Die *cpp*-Dateien des Projektes sind in den drei Variablen `projekt_SOURCES`, `projektApp_SOURCES` und `projektGui_SOURCES` anzugeben. Die erste Variable dient dabei lediglich dazu doppelte Angaben für die Applikation und die GUI zu vermeiden.
- Zusätzlich müssen noch die Header-Dateien, welche vom *Meta-Object-Compiler* von QT bearbeitet werden müssen, in der Variable `projektGui_MOC` angegeben werden.

Abschließend ist die Datei *garbage* dahingehend anzupassen, dass sämtliche beim Kompilieren des Projektes generierten Dateien entfernt werden.

Visual Studio .NET

In *Visual Studio* legt man innerhalb einer entsprechenden Projektmappe eine neues Projekt auf Basis der Vorlage *Win32-Konsolenprojekt* an. Sämtliche dabei generierten Dateien können daraufhin entfernt werden. Sowohl die Projektmappe mit der Dateiendung *.sln* als auch die Projektdatei *.vcproj* selbst sollten direkt im Verzeichnis *build/vs* liegen und nicht innerhalb eines Unterverzeichnisses. Bei den nun folgenden Einstellungen sollte darauf geachtet werden, diese für alle Konfigurationen und nicht nur für die aktuelle durchzuführen.

- Zunächst sind sowohl das Ausgabeverzeichnis als auch das Zwischenverzeichnis auf *$(ConfigurationName)* zu setzen, damit sämtliche Konfigurationen getrennt voneinander kompiliert werden.

- Als weiteres Include-Verzeichnis ist neben dem eigenen Quellcode-Verzeichnis (*../../src*) auch das von RoboApp (*../../../roboframe/roboapp/src*) mit anzugeben.

- Bei der Codeerstellung der Laufzeitbibliothek ist eine *Multithreaded-DLL* auszuwählen, wobei gegebenenfalls die Debug-Variante gewählt werden kann. Innerhalb der C++-Sprachoptionen sind die Laufzeit-Typinformationen zu aktivieren.

- *Vorkompilierte Header* sind dagegen nicht zu verwenden.

- Als zusätzliche Eingabe für den Linker dient die Abhängigkeit zu der Bibliothek *WS2_32.Lib* für die Netzwerkkommunikation.

- Des Weiteren sollte auch beim Linker darauf geachtet werden, in der Release-Konfiguration auf das Generieren von Debug-Informationen zu verzichten.

eMbedded Visual C++

In *eVC* wird ähnlich wie bei Visual Studio verfahren. Die Vorlage für das Projekt heißt hierbei *WCE Pocket PC 2003 Application*. Dabei muss im darauf folgenden Assistenten ein *leeres Projekt* ausgewählt werden. Auch hier sollte sowohl der Workspace mit der Dateiendung *.vcw* als auch die Projektdatei *.vcp* selbst im Verzeichnis *build/evc* liegen und nicht innerhalb eines Unterverzeichnisses.

Bei den Einstellungen muss zunächst angegeben werden, dass *MFC mit einer shared DLL* genutzt werden soll. Auch hier ist wieder darauf zu achten, dass diese Einstellungen für alle Konfigurationen und nicht nur für die aktuelle durchgeführt werden.

- Für die C++-Projektoptionen sind die Parameter */GR /GX* hinzuzufügen.

- Wie schon bei Visual Studio sind auch hier die *vorkompilierten Header* zu deaktivieren.

- Für den Präprozessor muss zusätzlich noch *../../../roboframe/roboapp/include* zu den beiden für Visual Studio angegebenen Include-Verzeichnissen hinzugefügt werden.

- Beim Linker müssen die beiden Bibliotheken *Ccrtrtti.lib* und *ws2.lib* angegeben und unterhalb von *Output* noch das Entry-Point-Symbol auf *wWinMainCRTStartup* festgelegt werden.

- Dem Projekt muss abschließend noch die Ressourcen-Datei *WinCeApp.rc* hinzugefügt werden, welche sich im Verzeichnis *roboframe/roboapp/include/wince* befindet.

Log-Level	Verwendung	Makro
LOG_LEVEL_DEBUG	Low-Level Debug Nachrichten	LOGPATH_DEBUG
LOG_LEVEL_INFO	Seltene, wichtigere Debug Nachrichten	LOGPATH_INFO
LOG_LEVEL_WARN	Fehlersituationen, die aufgelöst werden können; Datenverlust ist möglich	LOGPATH_WARN
LOG_LEVEL_ERROR	Ernste Fehlersituationen, die nicht automatisch aufgelöst werden können	LOGPATH_ERROR
LOG_LEVEL_FATAL	Fehlersituationen, die zum Abbruch der Anwendung führen sollten	LOGPATH_FATAL

Tabelle A.1.: Übersicht über die Log-Level

A.3.6. Logging

Um zu Debug- oder Informationszwecken Zustandsdaten ausgeben zu können, wurde ein an das in der Java Welt übliche Log4J[2] angelehnte Logging-API implementiert.

An jeder Stelle im Programmcode kann auf eine hierarchische Struktur von Loggern zugegriffen werden, wobei der sogenannte *Pfad* bei Methoden üblicherweise den Projektnamen, den Datei-Pfad und den Klassennamen enthalten sollte. So ist eine einfache Zuordnung von Ausgaben zu einer Klasse möglich. Daneben existieren fünf Ebenen (Log-Level) mit unterschiedlichem Wichtigkeitsgrad der Log-Nachricht, die in Tabelle A.1 aufgeführt sind.

Durch die Verwendung der `LOGPATH_XXX`-Makros, können Nachrichten abgesetzt werden. Die Parameter dabei sind zuerst der Pfad, danach die Nachricht. Die Nachricht kann über die üblichen `printf`-Parameter formatiert werden. In den beschriebenen Beispiel-Klassen wurden vereinzelt LOG-Makros verwendet, um Informationen über durchgeführte Aktionen auszugeben.

Um Log-Nachrichten auszugeben, muss mindestens ein `LogAppender` registriert werden. Ein Appender wird für einen bestimmten Pfad registriert und gibt dann alle Nachrichten dieses Teilbaumes aus, zusätzlich kann noch ein Log-Level angegeben werden. Zurzeit existieren im Framework Appender zur Ausgabe auf der Konsole (`ConsoleAppender`) und für das Versenden der Nachrichten über das Netzwerk (`NetworkAppender`) an eine GUI . Für die GUI wurde zusätzlich noch der `QtAppender` implementiert, welcher die Ausgabe in einem Widget darstellt. Falls dem Log-

[2]Log4J Homepage: `http://www.log4j.org`

ger keine Appender hinzugefügt werden, erfolgt auch keinerlei Ausgabe oder Versand von Meldungen.

Wird nicht mit der Präprozessor-Direktive `DEBUG` kompiliert, werden alle `DEBUG`-Makros entfernt, um das Laufzeitverhalten der Anwendung nicht zu negativ zu beeinflussen.

A.3.7. Dialoge anlegen

Eigene Dialoge (Listing A.10) müssen von der Klasse `Dialog` erben. Wie bereits bei den Modulen beschrieben, gibt es auch bei Dialogen die Methoden `init` und `cleanUp`. In diesen Methoden kann es sinnvoll sein, notwendige Fensterelemente zu erzeugen, beziehungsweise wieder zu entfernen.

Anders als bei Modulen gibt es bei Dialogen keine Puffer. Da Puffer immer auf einen Schlüssel festgelegt sind, widerspricht dies der Grundidee von Dialogen, dass sie in der Lage sind, sich unabhängig von den Schlüsseln für die Verarbeitung von bestimmten Datenstrukturen zu registrieren. Dies ermöglicht es spezifische Datenstrukturen visualisieren zu können, ohne die in Zukunft dafür vergebenen konkreten Schlüssel zu kennen. Daher können in dem Deskriptor des Dialogs in der Methode `addRepresentationWrappers` auch nur die Datenstrukturen selbst registriert werden.

Für jedes eingehende Datenobjekt wird die `handle`-Methode mit dem Parameter StreamedData aufgerufen. Dort kann dann unterschieden werden, um welche Datenstruktur es sich handelt, falls der Dialog mehrere Datenstrukturen verarbeiten kann. In dieser Methode findet dann auch die eigentliche Datenverarbeitung statt.

```
void BeispielDialog::addRepresentationWrappers(
    std::set<IRepresentationWrapper*>& wrappers) {
  wrappers.insert(new RepresentationWrapper<Bild>);
  wrappers.insert(new RepresentationWrapper<Gelenkwerte>);
}

void BeispielDialog::handle(StreamedData& data)
{
  if (KeyFactory::matchKeyStreamable(data.getKey(),
              new RepresentationWrapper<Bild>)) {
    // data enthält ein bild-objekt
    Bild bild;
    bild << data;
    // ...
  }
  if (KeyFactory::matchKeyStreamable(data.getKey(),
              new RepresentationWrapper<Gelenkwerte>)) {
    // data enthält ein gelenkwerte-objekt
```

```
    Gelenkwerte gelenkwerte;
    gelenkwerte << data;
    // ...
  }
}
```

Listing A.10: dialog/BeispielDialog.h

Damit man für jeden Dialog zur Laufzeit auswählen kann, wann wie viele Daten von welcher anderen Instanz empfangen werden sollen, existiert in der Elternklasse `Dialog` bereits eine Methode, welche einen modalen Dialog öffnet, um die nötige Auswahl zu ermöglichen. Daher ist es sinnvoll, die Methode `chooseRequestedKeys` via QT-Signal mit einer Schaltfläche oder einem Kontextmenü zu verbinden. Durch diese Methode wird ein modaler Dialog geöffnet, in dem man auswählen kann, welche Daten von welcher verbundenen Instanz in welchem Intervall empfangen werden sollen.

A.3.8. GUI-Applikation

Entsprechend der Applikation benötigt man auch für die GUI eine entsprechende Klasse (Listing A.11), welche die eigenen Schlüssel und Dialoge registriert. Da die init-Methode nicht überschrieben wird, werden die von der Elternklasse standardmäßig konfigurierten Appender für das Logging genutzt.

```
class BeispielGui : public GuiApplication
{
  BeispielGui::BeispielGui(int & argc, char ** argv)
    : GuiApplication(argc,argv)
  {}

  void BeispielGui::initKeys()
  {
    BeispielKeys::registerKeys();
  }

  void BeispielGui::registerDialogs()
  {
    // optional die framework-eigenen
    // dialoge registrieren
    GuiApplication::registerDialogs();

    // eigene dialoge registrieren
    dialogFactory->registerDialogWrapper(
        new DialogWrapper<BeispielDialog>);
  }
}
```

Listing A.11: BeispielGui.h

Des Weiteren benötigt man auch wieder eine entsprechende main-Funktion (Listing A.12).

```
int main( int argc, char ** argv )
{
  BeispielGui app(argc, argv);
  app.init();
  return app.exec();
}
```

Listing A.12: mainGui.h

A.3.9. Projektdateien für die GUI

In diesem Teil werden lediglich die Projekteinstellungen für Visual Studio beschrieben. Unter eVC steht keine graphische Oberfläche zur Verfügung und für die Autotools wurde die Beschreibung bereits bei der Applikation gegeben, da dort durch ein zentrales Makefile beide Unterprojekte kompiliert werden können.

Visual Studio .NET

Bei der Erstellung des Projektes geht man in *Visual Studio* genauso vor wie für die Applikation in Abschnitt A.3.5 beschrieben.

- Die Liste der zusätzlichen Include-Verzeichnisse sollte dabei noch um das Quellcode-Verzeichniss von RoboGui (*../../../roboframe/robogui/src*) und die QT-Verzeichnisse *$(QTDIR)/include* und *$(QTDIR)/mkspecs/win32-msvc.net* erweitert werden.
- Beim Linken sind neben dem zusätzlichen Bibliothek-Verzeichnis *$(QTDIR)/lib* noch die beiden Bibliotheken *qt-mtedu333.lib* und *qtmain.lib* anzugeben.

B. Verwendete Software

Das Framework und die Beispielanwendungen wurden unter Verwendung der folgenden Software entwickelt. Zur Erstellung neuer Anwendungen wird die Benutzung derselben Hilfsmittel empfohlen, um Inkompatibilitäten zu vermeiden. Zusätzlich nötige Werkzeuge (Editoren, SVN-Clients, etc.) sind hier nicht angegeben, da ihre Auswahl dem Benutzer ohne Einschränkung freisteht.

B.1. Unix und Linux

- GNU Autoconf 2.59[1]
- GNU Automake 1.9[2]
- GNU Libtool 1.5[3]
- GNU Make 3.80[4]
- GNU Compiler Collection (GCC) 3.4.2[5]
- Trolltech QT 3.3.4[6]

Alternativ zu den Autotools können auch folgende Programme verwendet werden:

- Eclipse 3.0[7]
- Eclipse C/C++ Development Tools (CDT) 2.1 Plugin[8]

[1]GNU Autoconf Homepage: `http://www.gnu.org/software/autoconf/`
[2]GNU Automake Homepage: `http://www.gnu.org/software/automake/`
[3]GNU Libtool Homepage: `http://www.gnu.org/software/libtool/`
[4]GNU Make Homepage: `http://www.gnu.org/software/autoconf/`
[5]GCC Homepage: `http://gcc.gnu.org`
[6]QT Homepage: `http://www.trolltech.com/products/qt/`
[7]Eclipse Homepage: `http://www.eclipse.org`
[8]Eclipse CDT Homepage: `http://www.eclipse.org/cdt/`

B.2. Mipsel Linux Cross-Compiler

- GNU Compiler Collection (GCC) 2.96

B.3. Windows 2000 / XP

- Microsoft Visual Studio .NET
- Trolltech QT 3.3.3 Educational Version

B.4. Pocket PC

- Microsoft ActiveSync 3.8[9]
- Microsoft eMbedded Visual C++ 4 mit ServicePack 4
- Microsoft Pocket PC 2003 SDK
- Pocket PC Run-Time Type Information Patch[10]

[9] Microsoft ActiveSync Homepage: `http://www.microsoft.com/windowsmobile/downloads/activesync38.mspx`

[10] Pocket PC RTTI Patch: `http://support.microsoft.com/kb/830482/`

C. Code-Konventionen

Code-Konventionen sind bei der Programmierung aus mehreren Gründen wichtig:

- Nahezu keine Software wird während der gesamten Einsatzzeit von denselben Programmierern betreut.
- Die Lesbarkeit des Codes wird erhöht und somit auch das Verständnis für den Code verbessert.
- Etwa 80% der Arbeitszeit am Quellcode entfällt auf Wartung und Erweiterung.

Daher ist es wichtig, dass sich alle beteiligten Entwickler an diese Richtlinien halten.

Die hier erwähnten Konventionen sind stark an die *Java Code Conventions*[1] angelehnt und erweitern diese um einige C++-spezifischen Angaben.

C.1. Klassen, Variablen und Konstanten

Klassennamen sollten immer mit einem Großbuchstaben beginnen. Bei zusammengesetzten Namen ist jeweils der erste Buchstabe jedes Teilwortes ebenfalls groß zu schreiben.

Variablennamen sollten dagegen immer mit einem Kleinbuchstaben beginnen. Auch hier gilt bei zusammengesetzten Namen die Kapitalisierung der ersten Buchstaben aller Folgeworte.

Konstanten sollten vollständig mit Großbuchstaben geschrieben werden. Dabei werden mehrere Teilnamen mittels Unterstrich voneinander abgesetzt.

Die Namensgebung sollte in beiden Fällen möglichst selbsterklärend sein und keine Abkürzungen enthalten. Unterstriche sollten sowohl bei Klassen- als auch bei Variablennamen vermieden werden. Für Sonderzeichen oder Umlaute gilt diese Einschränkung für alle Konstrukte.

[1] Code Konventionen für Java Homepage: `http://java.sun.com/docs/codeconv/`

C.2. Dateien und Verzeichnisse

Verzeichnisnamen sind in Kleinbuchstaben zu schreiben. Damit werden Probleme auf Windows-Plattformen umgangen.

Dateien tragen abhängig von ihrem Inhalt die Endung `.h` bei Header-Dateien oder `.cpp` für Source-Dateien. Diese Aufteilung in *Deklaration* und *Definition* sollte immer durchgeführt werden, sofern dies, wie zum Beispiel bei Templates, nicht unmöglich ist. In jeder Datei sollte auch nur eine gleichnamige Klasse deklariert bzw. definiert werden, wobei die Schreibweise der Dateinamen aus der Konvention für Klassennamen folgt.

Ausnahmen bilden hier Dateien, welche keinerlei Klassendefinition enthalten, sondern ausschließlich globale Funktionen. Diese sollten demnach mit einem Kleinbuchstaben beginnen. Als Beispiel wäre die Datei mit der `main`-Funktion zu nennen.

C.3. Includes und Namensräume

Beim Einbinden von Dateien ist darauf zu achten, sie strukturell zu gruppieren. Zunächst sollten die Dateien inkludiert werden, deren Klassen beerbt werden. Danach sollten jeweils abgesetzt die Dateien aus dem eigenen Projekt folgen. Des Weiteren werden Klassen aus dem Framework, Dateien von Bibliotheken wie QT und am Ende C++-Bibliotheken eingebunden. Dabei sollte jeder Block für sich alphabetisch sortiert sein, um eine besser Übersicht zu gewährleisten.

Bei dem Quellcode eines Projektes ist des Weiteren darauf zu achten, dass sowohl in den Header-Dateien als auch in den Implementierungen ein projektspezifischer Namensraum genutzt wird. Ob andere Namensräume global eingebunden oder einzeln vor den entsprechenden Klassen notiert werden, sollte man von der Quantität der notwendigen Auszeichnungen abhängig machen. Von der Verwendung von `using namespace` ist allerdings in Header-Dateien abzusehen, da dies ansonsten implizit auch Auswirkungen auf Dateien hat, welche diesen Header lediglich inkludieren. Wegen solcher *Seiteneffekte* sollte dies vermieden werden.

Sowohl bei den Includes als auch bei den Namensräumen sollte darauf geachtet werden, nur notwendige Auszeichnungen vorzunehmen. Das heißt, in einer `cpp`-Datei kann auf das Inkludieren von Dateien verzichtet werden, welche bereits in der zugehörigen Header-Datei erfolgte. Allerdings sollten keine Einbindungen entfallen, welche implizit durch andere Header-Dateien erfolgen.

C.4. Formatierung

Diese beiden Konventionen zur Textformatierung stehen im Gegensatz zu den *Java Code Konventionen*.

Die Formatierung der eigentlichen Quellcodezeilen muss nicht auf 80 Zeichen pro Zeile beschränkt sein. Aufgrund der aktuelle Arbeitsumgebungen dürfen Zeilen durchaus auch länger werden, können aber der Lesbarkeit wegen auch entsprechend umgebrochen werden.

Des Weiteren sollten Tabulatoren nicht durch Leerzeichen ersetzt werden, um eine individuell passende Einrückungsgröße zu ermöglichen.

C.5. Implementierung

Bei Klassen ist grundsätzlich darauf zu achten, sowohl einen Standardkonstruktor als auch einen Destruktor bereitzustellen. Diese können bei Bedarf auch privat deklariert werden, sollten aber auf jeden Fall vorhanden sein. Beim Destruktor ist es des Weiteren wichtig, diesen `virtual` zu definieren, um mit RTTI Informationen zu dieser Klasse erhalten zu können.

Außerdem sollten im gesamten Quellcode statt C-Casts in Form von runden Klammern ausschließlich die neuen C++-konformen Typwandlungen `dynamic_cast`, `static_cast`, `const_cast` und `reinterpret_cast` verwendet werden. Dies dient neben dem besseren Auffinden vor allem der Vermeidung von falschen Typwandlungen, bei denen zum Beispiel - nicht direkt erkennbar - die `const`-Eigenschaft einer Variablen entfernt wird.

C.6. Dokumentation

Zum einen sollten Codepassagen, welche umfangreich oder komplex sind, durch einen kurzen Kommentar in Form von `//` dokumentiert werden, um das Verständnis der Implementierungsdetails zu vereinfachen. Zum anderen sollten durch JavaDoc-ähnliche Kommentare mittels doxygen vollständige API-Dokumentationen generiert werden können. Dazu ist es notwendig, neben einem Kommentar zu Beginn der Datei jede Klasse, Methode und Klassenvariable mit einer entsprechenden Auszeichnung zu versehen. Gerade bei Methoden ist darauf zu achten, sämtliche Parameter und gegebenenfalls auch den Rückgabewert zu dokumentieren.

Bei Vererbung kann die Methodendokumentation entfallen, wenn ansonsten nur der Kommentar wiederholt werden würde.

Den Anspruch, dass das gesamte Projekt ohne Fehler und möglichst ohne Warnung übersetzt werden kann, sollte man auch an die API-Dokumentation haben.

C.7. Beispielcode

Beispielcode (Listing C.1) einer Header-Datei entsprechend der angegebenen Konventionen.

```
/**
 * @file Bar.h
 * Declaration of the class Foo::Bar.
 */

#ifndef __Bar_h__
#define __Bar_h__

#include [Elternklassen]

#include [Projektklassen]

#include [Bibliotheksklassen]

#include [Systemklassen]

namespace Foo {

/**
 * Forward declaration of class YZ.
 */
class YZ;

/**
 * KlasseBar dient einem Zweck.
 *
 * Detaillierte Beschreibung (auch mehrzeilig).
 */
class Bar
  : public IBar,
    public IFoo
{

public:

  /**
   * Default constructor.
   */
  Bar();
```

```
  /**
   * Destructor.
   */
  virtual ~Bar();

  /**
   * Beschreibung der Methode.
   *
   * @param x   Beschreibung des Parameters x
   * @param yy  Beschreibung des Parameters yy
   * @param z   Beschreibung des Parameters z
   * @return    Beschreibung des Rückgabewertes
   */
  bool methodeA(int x, const AB::Object* yy, std::string& z);

private:

  /** Beschreibung der Klassenvariable */
  unsigned int count;

}

} // namespace

#endif // __Bar_h__
```

Listing C.1: Bar.h

Der dazugehörige Beispielcode der `.cpp`-Datei ist in Listing C.2 zu finden.

```
/**
 * @file Bar.h
 * Declaration of the class Foo::Bar.
 */

#include [Headerdatei]

#include [weitere Projekt-, Bibliotheks- und Systemklassen]

using namespace AB;
using namespace CD;

namespace Foo {

bool Bar::methodeA(int, const Object* yy, std::string& z)
{
  // Der erste Parameter wird nicht benutzt und
  // hat daher keinen Variablennamen
  ...
}

} // namespace
```

Listing C.2: Bar.cpp

D. Subversion-Repository

Der gesamte Quellcode des Frameworks, wie auch alle darauf basierende Projekte, befinden sich in einem Subversion-Repository auf einem Server des *Fachgebiets Simulation und Systemoptimierung*. Dieses ist zugangsbeschränkt unter der Adresse https://svn.sim.informatik.tu-darmstadt.de/robocode/ zu erreichen. Für nähere Informationen zur Benutzung von Subversion ist es empfehlenswert, die Kapitel *2. Basic Concepts* und Kapitel *3. Guided Tour* des Subversion-Buches[1] zu lesen. Dagegen ist das Kapitel *A. Subversion for CVS Users* für Personen gedacht, welche bereits mit CVS, einem ähnlichen Versions-Kontroll-System, gearbeitet haben. Im weiteren Verlauf wird das Grundverständnis von *Subversion* (SVN) vorausgesetzt.

D.1. Zugriff auf das Repository

Um auf das SVN-Repository zugreifen zu können, benötigt man einen SVN-Client. Unter Windows empfiehlt sich die Benutzung von TortoiseSVN, welches sich in den Explorer integriert. Für die Entwicklungsumgebung Eclipse kann die benötigte Funktionalität durch das Plugin *subclipse*[2] eingebunden werden.

D.2. Repository-Struktur

In Abbildung D.1 ist die Verzeichnisstruktur des SVN-Repository abgebildet. SVN-typisch befinden sich in der ersten Ebenen die Verzeichnisse *trunk*, *tags* und *branches*. Unter *trunk* befinden sich die eigentlichen Daten in ihrem aktuellsten Entwicklungsstand. Im Unterverzeichnis *trunk/roboframe* befindet sich der Quellcode für die beiden Bibliotheken RoboApp und RoboGui. Auf der gleichen Ebene befindet sich dann für jedes Projekt ein eigenes Verzeichnis. Zum Testen der Funktionalität des Frameworks gibt es das Projekt *demo* und als Vorlage für neue Projekte kann man sich an *example* orientieren. In jedem Projektverzeichnis befinden sich unterhalb von *build* die verschiedenen Build-Skripte und Projektdateien.

[1] SVN-Buch: `http://svnbook.red-bean.com`
[2] Subclipse Homepage: `http://subclipse.tigris.org`

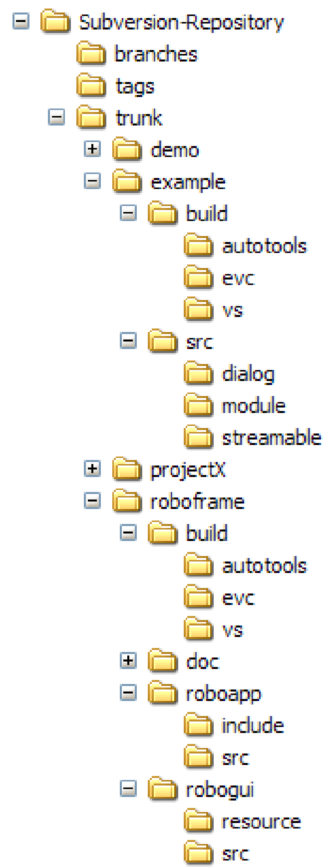

Abbildung D.1.: Verzeichnisstruktur im Subversion-Repository

E. Abkürzungen

API Application Programming Interface
Ein API bezeichnet die Schnittstelle, die ein Softwaresystem (zum Beispiel ein Betriebssystem) anderen Programmen zur Verfügung stellt. Die Schnittstelle besteht meist aus Routinen, Klassen und/oder Protokollen.

CVS Concurrent Versions System[1]
Ein Programm zur Versionsverwaltung von Dateien, hauptsächlich Software-Quellcode.

DOF Degree of Freedom
Freiheitsgrade eines Roboter-Systems.

GUI Graphical User Interface
Die grafische Schnittstelle auf Computern, die eine Interaktion mit dem Benutzer verlangen.

MFC Microsoft Foundation Classes
Eine Sammlung objektorientierter Klassenbibliotheken, die von Microsoft für das Betriebssystem Windows für die Programmierung von windowsbasierten Anwendungen mit C++ entwickelt wurden.

PDA Personal Digital Assistant
Ein tragbarer Kleinst-Computer.

RISC Reduced Instruction Set Computing
Moderne Prozessorarchitektur, die sich durch wenige festverdrahtete, dafür schnellere Befehle auszeichnet.

RTTI Run-Time Type Information
Mechanismus zum Abfragen von Typinformationen zur Laufzeit eines Programms.

TCP Transmission Control Protocol
Ein zuverlässiges, verbindungsorientiertes Transportprotokoll in Computernetzwerken. Es ist Teil der TCP/IP-Protokollfamilie und enthält eine Ende-zu-Ende Datensicherung.

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

UDP User Datagram Protocol
Ein minimales, verbindungsloses Netzwerkprotokoll. Es gehört zur Transportschicht der TCP/IP-Protokollfamilie und ist im Gegensatz zu TCP nicht auf Datensicherung ausgelegt.

SVN Subversion[2]
Eine Open-Source-Software zur Versionsverwaltung von Dateien. Subversion wird häufig als Nachfolger von CVS bezeichnet.

[2]SVN Homepage: `http://subversion.tigris.org`

F. Literaturverzeichnis

[1] COURTOIS, P. J. ; HEYMANS, F. ; PARNAS, D. L.: Concurrent control with readers and writers. In: *Commun. ACM* 14 (1971), Nr. 10, S. 667–668. – ISSN 0001–0782

[2] ENDERLE, Stefan ; SABLATNOEG, Stefan ; UTZ, Hans: *Miro Manual*. Department of Computer Science, University of Ulm, April 2005

[3] FOX, D. ; BURGARD, W. ; DELLAERT, F. ; THRUN, S.: Monte Carlo Localization: Efficient Position Estimation for Mobile Robots. In: *Proc. of the National Conference on Artificial Intelligence*, 1999

[4] FRIEDMANN, Martin ; KIENER, Jutta ; KRATZ, Robert ; LUDWIG, Tobias ; PETTERS, Sebastian ; STELZER, Maximilian ; VON STRYK, Oskar ; THOMAS, Dirk: Darmstadt Dribblers 2005: Humanoid Robot / Simulation and Systems Optimization Group, Department of Computer Science, Technische Universität Darmstadt. 2005. – Forschungsbericht

[5] GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph ; VLISSIDES, John: Design patterns: abstraction and reuse of object-oriented design. (2002), S. 701–717. ISBN 3–540–43081–4

[6] HUNT, Andrew ; THOMAS, David: *Der Pragmatische Programmierer*. Carl Hanser Verlag, 2003. – ISBN 3–446–22309–6

[7] iXs Research Corporation: *iHs04 Instruction Manual Version 1.0*. 2003

[8] KNUTH, Donald E.: Structured Programming with go to Statements. In: *ACM Computing Surveys* 6 (1974), Nr. 4, S. 261–301. – ISSN 0360–0300

[9] Kondo Kagaku Co. Ltd.: *KHR 1 - Hardware Manual*. 2004

[10] Kondo Kagaku Co. Ltd.: *RCB 1 - Software Manual*. 2004

[11] LÖTZSCH, Martin ; BACH, Joscha ; BURKHARD, Hans-Dieter ; JÜNGEL, Matthias: Designing Agent Behavior with the Extensible Agent Behavior Specification Language XABSL. In: *RoboCup 2003: Robot Soccer World Cup VII* Bd. 3020. Padova, Italy : Springer, 2004, S. 114–124

[12] MIKHAJLOV, Leonid ; SEKERINSKI, Emil: A Study of the Fragile Base Class Problem. In: *Lecture Notes in Computer Science 1445*, Springer Verlag, July 1998, S. 355–382

[13] MURPHY, Robin R.: *Introduction to AI Robotics*. MIT Press, 2000. – ISBN 0–262–13383–0

[14] NTAFOS, Simeon C.: A Comparison of Some Structural Testing Strategies. In: *IEEE Trans. Softw. Eng.* 14 (1988), Nr. 6, S. 868–874. – ISSN 0098–5589

[15] RÖFER, Thomas ; BURKHARD, Hans-Dieter ; VON STRYK, Oskar ; DAHM, Ingo ; LAUE, Tim ; HOFFMANN, Jan ; JÜNGEL, Matthias ; GÖHRING, Daniel ; LÖTZSCH, Martin ; DÜFFERT, Uwe ; SPRANGER, Michael ; ALTMEYER, Benjamin ; GOETZKE, Viviana ; BRUNN, Ronnie ; DASSLER, Marc ; KUNZ, Michael ; RISLER, Max ; STELZER, Maximilian ; THOMAS, Dirk ; UHRIG, Stefan ; SCHWIEGELSHOHN, Uwe ; HEBBEL, Matthias ; NISTICÓ, Walter ; SCHUMANN, Carsten ; WACHTER, Michael: *German Team Teamreport 2004*. German Team

[16] RÖFER, Thomas ; VON STRYK, Oskar: Softwarearchitektur: Ein Erfolgsfaktor beim Roboterfußball! In: *OBJEKTspektrum* 2005 (2005), Juli/August, Nr. 4, S. 31–32

[17] SAFFIOTTI, A. ; RUSPINI, E. H. ; KONOLIGE, K.: Blending Reactivity and Goal-Directedness in a Fuzzy Controller. In: *Proc. of the IEEE Int. Conf. on Fuzzy Systems*. San Francisco, California : IEEE Press, 1993, S. 134–139

[18] SAFFIOTTI, A. ; RUSPINI, E. H. ; KONOLIGE, K.: Using Fuzzy Logic for Mobile Robot Control. In: *International Handbook of Fuzzy Sets and Possibility Theory* Bd. 5. Norwell, MA, USA, and Dordrecht, The Netherlands : Kluwer Academic Publishers Group, 1997

[19] STARKE, Gernot: *Effektive Software-Architekturen*. Carl Hanser Verlag, 2002. – ISBN 3–446–21998–6

[20] STENZEL, Roland: *Steuerungsarchitekturen für autonome mobile Roboter*, TH Aachen, Diss., 2002

[21] STEVENS, W. R.: *UNIX Network Programming: Networking APIs: Sockets and XTI*. Upper Saddle River, NJ, USA : Prentice Hall PTR, 1997. – ISBN 0–13–490012–X

[22] TANENBAUM, Andrew S.: *Modern Operating Systems*. Prentice-Hall, International, 1992 (Internals and Design Principles). – TAN a 92:1 P-Ex. – ISBN 0–13–595752–4