Freie Universität Berlin
Fachbereich Mathematik und Informatik
Institut für Informatik

Bachelorarbeit

Client-Server-Kommunikation für humanoide Roboter

Björn Kahlert

Berlin, 19.10.2009

Gutachter

Prof. Dr. Raúl Rojas
Institut für Informatik
Freie Universität Berlin

Betreuer

Hamid Reza Moballegh
Institut für Informatik
Freie Universität Berlin

Eigenständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Bachelorarbeit selbstständig und ohne unerlaubte Hilfe angefertigt habe. Alle verwendeten Hilfsmittel und Quellen sind im Literaturverzeichnis vollständig aufgeführt und die aus den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht.

Berlin, 19.10.2009 Björn Kahlert

Konventionen

Die folgenden Formatierungskonventionen werden in dieser Arbeit verwendet:

- **Fett** wird zur Hervorhebung wichtiger Informationen verwendet.
- *Kursiver* Text kennzeichnet Terminologien.
- `Lucida Console` dient der Kennzeichnung von Bezeichnungen, wie sie in den original Programmquellen verwendet werden.

```
Lucida Console wird Darstellung von Quelltext(-auszügen) benutzt.
Lucida Fax Fett hebt wichtige Quelltextpassagen hervor.
```

- [X] referenziert auf einen Quellennachweis mit der Bezeichnung X im Literaturverzeichnis
- [i] verweist auf eine Fußnote am Ende der Seite

Zusammenfassung

Diese Arbeit beschäftigt sich mit dem Entwicklungswerkzeug FUremote, das ihm Rahmen des studentischen Projekts FUmanoid entstand und behandelt die Schwächen der Vorgängerlösungen sowie eine Abwägung von Vor- und Nachteilen möglicher Lösungsansätze.
Hierbei wird insbesondere der Aufbau und die Funktionsweise der Eclipse Rich Client Platform (RCP) beschrieben.

Ein weiterer Abschnitt befasst sich mit der Struktur von FUremote, die anhand der implementierten Plug-ins und in Hinblick auf wichtige Konzepte der Eclipse RCP wie *Extensions* und *Selection Service* vorgestellt wird.

Der Aufbaubeschreibung von FUremote schließt sich eine Erläuterung der Netzwerkfunktionalitäten an. Letztere umfasst eine ausführliche Erklärung des FUmanoid-Protokolls, wichtiger Designentscheidungen und der automatischen Robotererkennung- und beobachtung.

Die Arbeit schließt mit einem Ausblick auf mögliche Verbesserungen ab.

Inhaltsverzeichnis

1. Einleitung **1**

1.1. Studentisches Projekt FUmanoid 1

1.2. Ausgangssituation 1

2. Programm FUremote **4**

2.1. Architektur 4

2.2. Eclipse RCP 6

2.3. Aufbau 13

2.4. Netzwerk 22

3. Ausblick **37**

3.1. GameController 37

3.2. Automatische Farbkalibrierung 37

3.3. Netzwerk 38

3.4. Motion Editor 38

3.5. Threads 38

4. Fazit **39**

5. Literaturverzeichnis **40**

6. Anhang **41**

6.1. Protokolldefinition 41

6.2. Einrichtung Eclipse 57

6.3. Erstellung eines Plug-ins 62

6.4. Erstellung einer Binary Version 68

1. Einleitung

Diese Arbeit beschäftigt sich mit einer Lösung, welche die Kommunikation zwischen humanoiden Fußball spielenden Robotern und Computern ermöglicht. Die entsprechende Software, wie auch diese Arbeit entstanden im Rahmen der Mitarbeit bei FUmanoid, einem studentischen Projekt der Arbeitsgruppe «Künstliche Intelligenz» der Freien Universität Berlin [1].

1.1. Studentisches Projekt FUmanoid

Das FUmanoid-Projekt entwickelt seit 2006 unter der Aufsicht von Professor Raúl Rojas humanoide Roboter, die in der Lage sind, im Verbund Fußball zu spielen. Die knapp 60cm großen Roboter nehmen seitdem erfolgreich beim alljährlich stattfindenden RoboCup in der KidSize-Liga teil. Während im Jahr 2007 der 3. Platz erkämpft wurde, konnte sich das Team im Jahr 2009 über den 2. Platz freuen.

Im Jahr 2008 gewannen die FUmanoids den 2. Platz bei der German Open.

Die Veranstalter des RoboCups bezeichnen den Wettbewerb als "Forschungs- und Bildungsinitiative" [2] mit dem Ziel, die Forschung in den beiden Gebieten Künstliche Intelligenz und Robotik zu fördern.

Erklärtes Ziel ist es, das im Jahre 2050 amtierende menschliche Fußballteam zu schlagen.

1.2. Ausgangssituation

Damit ein humanoider Roboter Fußball spielen kann, muss er seine Umgebung wahrnehmen und auf sie reagieren können. Die Wahrnehmung umfasst die Kopfkamera, unter den Füßen angebrachte Drucksensoren und sämtliche Motoren[i]. Die Reaktion hingegen findet fast ausschließlich in den Beinen - genauer durch Laufen und Schießen - statt.

---

[i] Zu den Motoren besteht eine bidirektionale Verbindung. Sie ermöglicht es nicht nur, die Motoren auf eine bestimmte zu setzen, sondern auch, Parameter wie Widerstand oder Temperatur zu erfragen.

Der Entwicklungsprozess am Roboter ist zeit- und kostenintensiv. Zum einen sind dafür nicht zu unterschätzende Verschleiß- und Neuanschaffungskosten verantwortlich, zum anderen zeitraubendes Einstudieren von Bewegungsabläufen und ein aufwändiges Nachvollziehen des Roboterverhaltens.
Für den Umgang mit den Robotern wurden vor der Anfertigung dieser Arbeit folgende 3 Programme verwendet:

- **Motion Editor**
  Dieses Programm ermöglichte das Erfassen aller Motorwerte zu einem bestimmten Zeitpunkt. Durch das Anfertigen mehrerer solcher Momentaufnahmen konnten Bewegungsfolgen aufgezeichnet und optimiert werden.

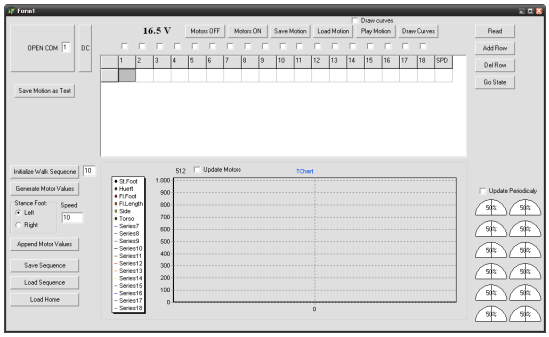


Abbildung 1: Motion Editor

- **Server-Programm**
  Das Server-Programm diente zum Starten und Überwachen eines Spiels. Es ermöglichte die Festlegung der Spieleparameter und den Empfang von Logging-Nachrichten seitens der Roboter.

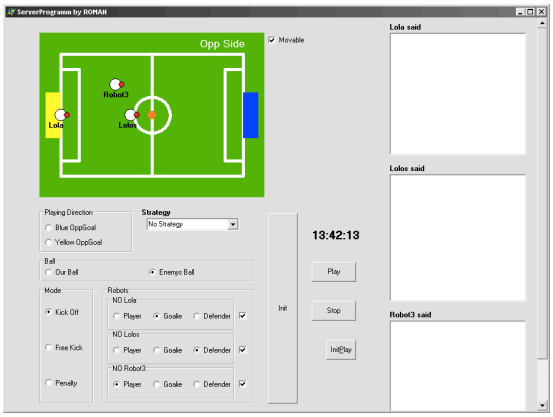


Abbildung 2: Server-Programm

- **Terminal**
  Über einen seriellen RS232-Anschluss konnte eine Verbindung zum Roboter aufgebaut werden. Kaum dokumentierte Befehle erlaubten das Aufspielen von Software und das Ausführen grundlegender Aktionen auf dem Roboter.

Jedes dieser Programme war eigenständig und erforderte die Konfiguration von Parametern bei jedem Verbindungsaufbau mit einem Roboter. Motion Editor und Konsole waren besonders fehleranfällig.
Erschwerend kam an modernen Notebooks das Fehlen einer RS232-Schnittstelle hinzu, mit dem das bis dahin eingesetzte Biolid-Kit [3] - genauer das Biolid Controle Module - verbunden wurde.
Das Problem konnte mit einem USB-auf-RS232-Adapters überbrückt werden; trotzdem blieb die Oberfläche benutzerunfreundlich und die Anbindung kabelgebunden.

Ähnliche Probleme waren auch auf Entwicklungsebene anzutreffen: Die über zwei Jahre gewachsene FUmanoid-Software unterlag Kapazitätsbeschränkungen, vor allem in Bezug auf den Hauptspeicher.
Während der Teilnahme an Wettbewerben wurde außerdem unter hohem Zeit- und Erwartungsdruck an der FUmanoid-Software gearbeitet.

Da das Potential der Roboter weitgehend erschöpft war, fiel 2008 im Anschluss an den RoboCup die Entscheidung, eine neue Robotergeneration zu entwickeln. Neben grundlegenden Änderungen der Hardware und der Portierung von FUmanoid sollte es auch ein Programm geben, welches die 3 Insellösungen vereint, die Möglichkeit für Erweiterungen bietet und alle Funktionen über das Netzwerk bereit stellen kann.

2. Programm FUremote

Die neue in den Jahren 2008/2009 entwickelte Robotergeneration basiert auf der *Computer-on-Module*-Lösung Gumstix. Dabei handelt es sich um ein Kaugummistreifen großes Mainboard mit ARM-Prozessor, 128 MB Arbeitsspeicher, 32 MB Flashspeicher und Erweiterungsschnittstellen.

Der Neuentwurf der Roboter machte auch die Neuimplementierung der peripheren Programme notwendig. Anstelle einer Neuimplementierung nutzte man die Gelegenheit, die Programme auf Grundlage der alten Lösungen neu zu entwerfen.

Neben den im vorangegangenen Kapitel genannten Anforderungen, sollte die Software in Java geschrieben werden, um zum einen die Wartbarkeit und zum andern die Portabilität zu erhöhen. Die alten Programme waren in Delphi geschrieben und erschienen dem Team nicht zukunftssicher.
Die Integration des neuen Codes in die bereits existierende Versionierungslösung sollte es dem Team ermöglichen, gemeinsam an Erweiterungen für das neue Programm FUremote zu arbeiten.

2.1. Architektur

Das Programm FUremote basiert auf den folgenden zwei Architekturmustern:

- **Client-Server**
  Die Client-Server-Architektur wird für die Kommunikation zwischen dem FUremote-Programm und den Robotern verwendet. Hierbei wird ein eigenes auf UDP basierendes Protokoll verwendet. Eine genaue Beschreibung dieses Protokolls befindet sich im Kapitel 2.4 Netzwerk.

- **Plug-in**
  Die Plug-in-Architektur arbeitet nach dem Prinzip der Steuerungsumkehr [4] (engl. «Inversion of Control», IoC) und verlagert die Programmlogik in so genannte *Plug-ins*. Der Programmkern umfasst dabei nur noch den Code zur Arbeit mit den Plug-ins, während sich in letzteren der problemspezifische Code befindet.

Durch die Verwendung von Plug-ins kann eine hohe Modularität im fertigen Programm erzielt werden. Neue Funktionen können von unterschiedlichen Projektteilnehmern dank einheitlicher Schnittstellen und ohne hohes Konfliktpotential entwickelt werden.

Für einen Plug-in-Mechanismus standen mehrere Möglichkeiten zur Wahl:

1. **Eigenentwicklung**
   Eine Eigenentwicklung hätte den augenscheinlichen Vorteil, auf das Problem zugeschnitten zu sein. Allerdings wäre die Aufgabe, einen Plug-in-Mechanismus selbst zu entwickeln anspruchsvoll und zeitraubend. Diese Lösung würde inkl. der Entwicklung von FUremote den zeitlichen Rahmen einer Bachelorarbeit übersteigen. Außerdem wäre es fraglich, ob sich bei diesem Weg ein Mehrgewinn ergeben würde.

2. **Verzicht auf Plug-ins**
   Angesichts des Zeitrahmens und hohen Aufwandes zur Entwicklung von FUremote wäre der bloße Verzicht auf eine Plug-in-Lösung bereits vorteilhafter gewesen.
   Zu Beginn der Programmentwicklung wären mit schnellen Ergebnissen zu rechnen. Allerdings würde mit jeder neuen Funktionalität das Programm komplexer in seiner Weiterentwicklung und die Wartbarkeit verringern.

3. **Existierende Plug-in-Implementierungen**
   Die Verwendung einer bereits existierenden Lösung (z.B. in Form eines Frameworks) hätte einerseits den Vorteil einer stabilen und effektiven Entwicklungsgrundlage, die andererseits aber Einarbeitungszeit erfordern würde. Gleichzeitig müsste dieser Zeitaufwand von jedem zukünftigen Entwickler investiert werden.

Gerade in Hinblick auf die dritte Möglichkeit gab es Bedenken bezüglich des Einarbeitungsaufwandes und der Komplexität einer "großen" Plug-in-Lösung. Es wurde im Team mehrfach die Befürchtung geäußert, dass es sich dabei um "Overkill", also um einen unverhältnismäßig aufwendigen Lösungsvorschlag handele.

Schließlich ist die Entscheidung dennoch auf die Verwendung einer bereits existierenden Plug-in-Lösung gefallen. Die Notwendigkeit eines wartbaren und erweiterungsfähigen Programms rechtfertigte den erhöhten Arbeitsaufwand.

2.2. Eclipse RCP

Technisch basiert FUremote auf der Eclipse Rich Client Platform, einem Framework zur Entwicklung von *Rich Clients*.

Vor- und Nachteile

Die Wahl fiel aus folgenden Gründen auf das Eclipse Framework:

- **Hohe Reputation**
  Nach jahrelanger Erfahrung in der Entwicklung von Entwicklungsplattformen entwickelte IBM schließlich die IDE Eclipse, welche als «Integrationsplattform für Entwicklungswerkzeuge» [5] geplant war.
  Im November 2001 wurde das Framework als Open Source veröffentlicht und die Betreuung an die neu gegründete Eclipse Foundation übergeben.
  Heute wird die Foundation von einem Konsortium namenhafter Softwarehersteller unterstützt [6].

- **Hohe Verlässlichkeit**
  Das seit Ende der 90er Jahre existierende NASA-Projekt Maestro, welches sich mit der Verwaltung entfernter Fahrzeuge und Experimente in Weltraummissionen beschäftigt, verwendet seit 2004 das Eclipse RCP zur Entwicklung [7].
  Das Projekt profitiert somit von den Fähigkeiten des Frameworks, unterschiedlichste Anforderungen erfüllen und sich schnell anpassen zu können.

- **Hohe Kompatibilität**
  Das Framework basiert auf der visuellen Komponenten SWT für das Implementierungen für eine breite Palette von Betriebssystemen und Systemarchitekturen existieren. Die Anforderung des FUmanoid-Projekts, dass FUremote sowohl auf 32- als auch 64Bit-Architekturen wie auch unter Windows, Linux und Mac OS laufen können sollte, konnte damit also erfüllt werden.

- **Hohe Erweiterbarkeit**
  Der von Eclipse verwendete Plug-in-Mechanismus eignet sich inhärent für eine Lösung, die sich durch ihre Erweiterbarkeit auszeichnet. Gleichzeitig verringert die visuelle Integration in Eclipse die Fehleranfälligkeit der Arbeit mit Plug-ins.

Den Vorteilen steht ein hoher Einarbeitungsaufwand gegenüber, denn das Framework bedient sich einer Fülle von Entwurfsmustern, verwendet einen komplexen Plug-in-Mechanismus und basiert auf dem *Standard Widget Kit* [8]. Das SWT allein betrachtet benötigt bereits einen gewissen Zeitbedarf für die Einarbeitung.

Plug-in-Komponentenmodell

Das von Eclipse verwendete Plug-in-Komponentenmodell basiert auf der Implementierung des *OSGi*-Frameworks, welches zuständig für die Definition, Komposition und Ausführung von so genannte *Bundles* ist [7].
Derartige Bundles werden im Eclipse-Sprachgebrauch als Plug-ins bezeichnet. Die folgenden 5 Plug-ins bilden dabei das Eclipse RCP Framework:

- **OSGi**
  Definition, Komposition und Ausführung von Bundles/Plug-ins

- **Runtime**
  Aufbauend auf OSGi Bereitstellung von Erweiterungen (*Extensions*), Anwendungen (*Applications*) und Produkten (*Products*)

- **SWT**
  Einheitlicher Zugriff auf GUI-Elemente verschiedener Betriebssysteme

- **JFace**
  Ermöglichung der vereinfachten Arbeit mit SWT durch Verwendung des Architekturmusters Model-View-Controller

- **UI**
  Aufbauend auf Runtime und JFace Bereitstellung abstrakter Bedienelemente wie Editoren (*Editor*), Perspektiven (*Perspective*) und Arbeitsflächen (*Workbench*)

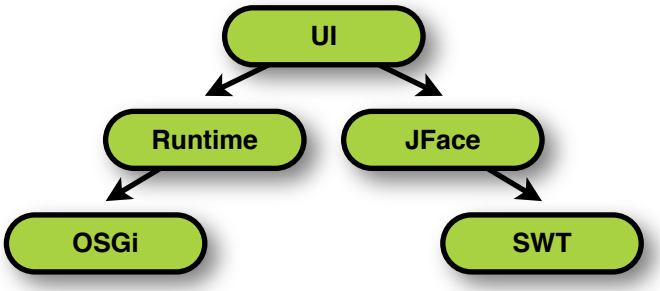


Abbildung 3: Abhängigkeiten der Basis-Plug-ins

Ein minimales Plug-in besteht aus einer *Aktivator*-Klasse und einem *Manifest*.

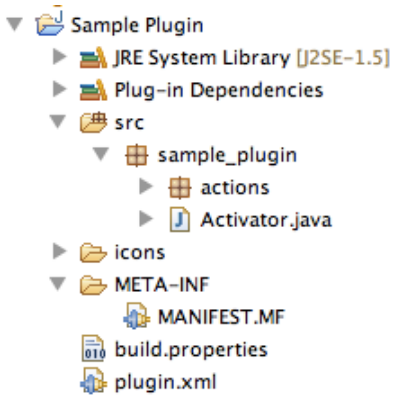


Abbildung 4: Verzeichnisstruktur eines minimalen Plug-ins

Die Aktivator-Klasse regelt den Lebenszyklus des Plug-ins und muss von der Klasse `Plugin`, welche sich im Plug-in Runtime befindet, abgeleitet sein. Zu den wichtigsten Methoden der Klasse gehören `start` und `stop`.

Das Manifest mit Namen `MANIFEST.MF` enthält Metainformationen zum Plug-in. Zu den wichtigsten Angaben gehören der Name des Plug-ins, die Namen der Plug-ins von denen das beschriebene Plug-in abhängt und der Name der Aktivator-Klasse.

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Sample_Plugin
Bundle-SymbolicName: Sample_Plugin;singleton:=true
Bundle-Version: 1.0.0.qualifier
Bundle-Activator: sample_plugin.Activator
Require-Bundle: org.eclipse.ui, org.eclipse.core.runtime
Bundle-ActivationPolicy: lazy
Bundle-RequiredExecutionEnvironment: J2SE-1.5
```

Listing 1: Minimale MANIFEST.MF

Erweiterungen

Die Eclipse RCP verwendet das Konstrukt der Erweiterungen (*Extensions*), um Plug-ins miteinander zu verknüpfen. Dazu muss ein Plug-in einen Erweiterungspunkt (*Extension Point*) definieren, den dann dasselbe typischerweise aber ein anderes Plug-in durch eine Extension erweitern kann.

Informationen zu Extension Points und Extensions werden nicht im Manifest, sondern in einem Deskriptor mit Namen `plugin.xml` abgelegt. Extension Points werden in der `plugin.xml` genannt, die Definitionen werden aber in dedizierten *Extension Point Schemas* festgehalten.

Beispiel

Das UI-Plug-in ist für die Bereitstellung des Hauptfensters zuständig und bietet über den Extension Point `menus` die Möglichkeit, das Menü des Fensters zu erweitern, wobei sich die Definition des Extension Points in der Datei `menus.exsd` befindet.

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.0"?>
<plugin>
[…]
      <extension-point id="menus"
            name="%ExtPoint.menus" schema="schema/menus.exsd"/>
[…]
</plugin>
```

Listing 2: Gekürzte `plugin.xml` des Plug-ins `org.eclipse.ui`, Version 3.5.0

```
<?xml version='1.0' encoding='UTF-8'?>
<schema targetNamespace="org.eclipse.ui"
                              xmlns="http://www.w3.org/2001/XMLSchema">
     […]
     <element name="extension">
          <complexType>
               <sequence>
                    […]
                    <element ref="menuContribution"
                              minOccurs="0" maxOccurs="unbounded"/>
               </sequence>
          </complexType>
     </element>
     […]
     <element name="menuContribution">
          <complexType>
               <sequence>
                    <element ref="separator"
                              minOccurs="0" maxOccurs="unbounded"/>
                    […]
               </sequence>
          </complexType>
     </element>
     […]
</schema>
```

Listing 3: Gekürzte `menus.exsd` des Plug-ins `org.eclipse.ui`, Version 3.5.0

Man kann der Definition entnehmen, dass eine Extension in XML über ein gleichnamiges Tag (`extension`) eingeleitet wird. Als Unterelemente sind beliebig viele `menuContribution`-Elemente erlaubt. In der 2. Hälfte des Schemas ist definiert, dass unter anderem der Tag `separator` als Unterelement von `menuContribution` erlaubt ist und zwar mindestens 0 mal (`minOccurs`). Nach oben ist die Anzahl der Unterelemente nicht beschränkt (`maxOccurs`). Damit existieren keine Beschränkungen bezüglich der Anzahl der untergeordneten Knoten.

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin>
     <extension>
          <menuContribution locationURI="menu:file?before=additions">
               <separator name="MySeparator" visible="true"/>
          </menuContribution>
     </extension>
</plugin>
```

Listing 4: `plugin.xml` eines beliebigen Plug-ins

Listing 4 zeigt eine mögliche Extension, welche in einem Menü mit Namen `file` vor der unsichtbaren Markierung `additions` ein Trennsymbol (`separator`) mit der Bezeichnung `MySeparator` einfügt.

Durch die lose Kopplung ist das Programm zunächst nicht auf das Plug-in angewiesen. Nur wenn das Plug-in vorhanden ist, wird das Trennsymbol im Menü des Hauptfensters angezeigt.

Extension Points haben weit mächtigere Beschreibungsmöglichkeiten. So kann die Menüerweiterung um eine Beschreibung ergänzt werden, die angibt, unter welchen Bedingungen ein Menüpunkt aus- oder eingeblendet oder inaktiv sein soll (*Command Core Expressions*, [16])

Eine weitere Möglichkeit besteht darin, von der Extension zu verlangen, dass sie eine Klasse angibt, die ein bestimmtes Interface implementieren muss. Auf diese Weise kann das Plug-in, das den Extension Point definiert, Methoden der erweiternden Plug-ins aufrufen. Genau hierbei handelt es sich um *Inversion of Control*, das heißt das Hauptprogramm wird auf ein einfaches Gerüst simplifiziert, während erweiternde Plug-ins *Ereignissenken*[ii] nutzen, um an bestimmten Momenten eigenen Code zur Ausführung bringen zu lassen.

[ii] Eine Ereignissenke (engl. "event sink") ist eine Programmroutine, die beim Eintritt eines Ereignisses bestimmte Aktionen ausführt [9]. Die typischste Aktion ist üblicherweise der Aufruf zuvor registrierter ereignissensitiver Listener.

Die Verwendung zum Zeitpunkt der Programmierung noch unbekannter Datentypen wird durch die Verwendung eigener *ClassLoader*[iii] ermöglicht. Hierzu verwendet Eclipse Informationen in der `plugin.xml` um den nachzuladenden Code zu identifizieren und *Reflection*[iv] um die notwendigen Objekte zu instanzieren.

Lebenszyklus

Eine Eclipse RCP-Anwendung besteht im Wesentlichen aus einem plattformabhängigen Startprogramm (*Launcher*), beliebig vielen Plug-ins und einer Konfigurationsdatei.

Im Moment des Starts durch den Launcher liest dieser die Konfigurationsdatei, in der alle Plug-ins aufgelistet sind, die für den erfolgreichen Start der Anwendung benötigt werden.

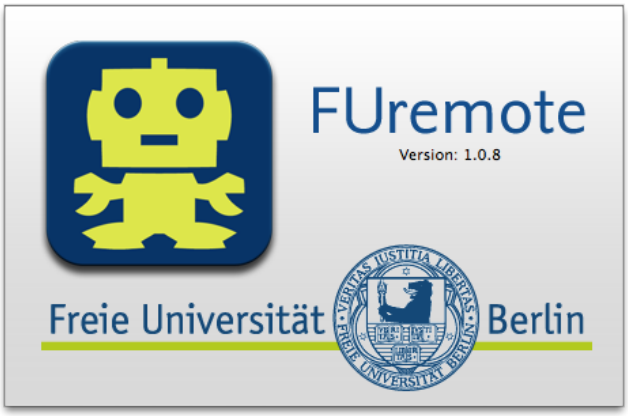


Abbildung 5: Startbildschirm von FUremote
im Moment der Validierung der Abhängigkeiten zwischen den zu ladenden Plug-ins

[iii] ClassLoader sind für das Laden von Klassen zuständig [10]. Für gewöhnlich werden alle notwendigen Klassen automatisch durch einen Standard-ClassLoader geladen. Die Verwendung eines eigenen ClassLoaders ermöglicht das Laden weiterer Klassen, auf die dann ebenfalls Zugriff besteht.

[iv] Bei der Reflexion handelt es sich um die Möglichkeit eines Programms, die eigene Struktur und das eigene Verhalten überwachen und verändern zu können [10]. Java ermöglicht unter anderem den Zugriff auf Metainformationen von Klassen und Objekten, die Umgehung von Sichtbarkeitsmodifikatoren, sowie die Instanzierung neuer Objekte und den Aufruf von Methoden mit beliebigen Parametern.

Anschließend werden die Abhängigkeiten der Plug-ins auf Validität und Erfüllbarkeit überprüft und schließlich die Anwendung gestartet.
Beabsichtigt der Entwickler nicht die Erweiterung einer bestehenden sondern die Erstellung einer eigenen Anwendung, muss er selbst den Extension Point `Application` erweitern. Dieser entspricht in der klassischen Java-Programmierung der `main`-Klasse.

Hat das `Runtime`-Plug-in eine `Application`-Erweiterung finden können, wird auf ihr die `start`-Methode aufgerufen. Es ist die Aufgabe dieser Klasse, die grafische Oberfläche unter Zuhilfenahme der Eclipse RCP-Plug-ins zu erstellen.

Eclipse startet bei Programmstart ausschließlich das Plug-in mit der `Application`-Erweiterung und solche von denen dieses Plug-in abhängig ist. Von den übrigen Plug-ins werden lediglich die Deskriptoren gelesen, um unter anderem darin definierte Menüerweiterungen korrekt darzustellen. Erst wenn beispielsweise ein Menüpunkt eines unaktivierten Plug-ins angeklickt wurde, wird das Plug-in tatsächlich aktiviert.

Dieses Vorgehen bezeichnet Eclipse als *Lazy Start Bundles* [11] und ermöglicht eine hohe Startgeschwindigkeit und Skalierbarkeit.

Bei Programmende wird die `stop`-Methode des Plug-ins mit der `Application`-Erweiterung aufgerufen. In der Regel werden an dieser Stelle vorgenommene Einstellungsänderungen in der Konfigurationsdatei gesichert.

2.3. Aufbau

Durch die Verwendung der Eclipse Rich Client Platform konnte FUremote so entwickelt werden, dass die späteren Komponenten nur eine geringe Kopplung untereinander aufwiesen.

In der Programmierung sind unterschiedlich große Gliederungseinheiten in Verwendung. Während Java viele feingliedrige Einheiten wie Methoden, Klassen und Pakete (*packages*) unterstützt, gibt es keine Entsprechung für Module oder Komponenten.

Man kann sich damit behelfen, Pakete zur Realisierung von Modulen zu verwenden. Um den Zugriff durch andere Programmteile zu schützen, gibt es einen geeigneten Sichtbarkeitsmodifikator (*visibility modifier*) mit der Bezeichnung `package`. Allgemein ist es in Java üblich, größere Programmmodule mithilfe von Paketen zu realisieren. Dieser Ansatz erfordert jedoch eine hohe Disziplin, da der Entwickler sicher im Umgang mit den vier Java-Sichtbarkeitsmodifikatoren sein muss.

Erschwerend kommt hinzu, dass Komponenten bzw. Subsysteme praktisch gar nicht mehr ohne weiteres in Java abgebildet werden können. Zumindest in diesem Punkt bietet die Eclipse RCP durch die Verwendung von Plug-ins Abhilfe. Eclipse verlangt an einer zentralen Stelle die Markierung von Paketen als *exportiert*, damit ein anderes Plug-in darauf Zugriff haben kann und verringert so die Kohäsion zwischen Plug-ins.

Programmkern

FUremote baut auf den Plug-ins der Eclipse RCP auf und ergänzt diese um eine eigene Plug-in-Schicht. Diese soll weiteren Plug-ins sich wiederholende Arbeiten ersparen und zentrale Funktionalitäten bereitstellen.

Die im Folgenden erklärten Core-Plug-ins bilden die Grundlage für problemspezifische Teilapplikationen.

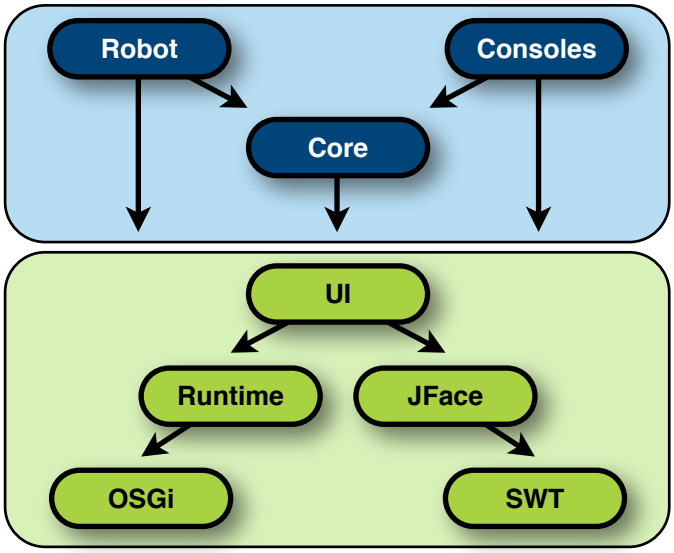


Abbildung 6: Erweiterung der Eclipse RCP-Plug-ins um die 3 Core-Plug-ins

FUremote Core (de.fumanoid.furemote)

Dieses Plug-in ist der eigentliche Programmkern. Es definiert das Skelett und die Funktionslogik des Hauptfensters und implementiert die Grundfunktionalität des Netzwerks zur zustandslosen Kommunikation mit beliebigen Netzwerk-Teilnehmern. Es erweitert als einziges den Extension Point `Application`.

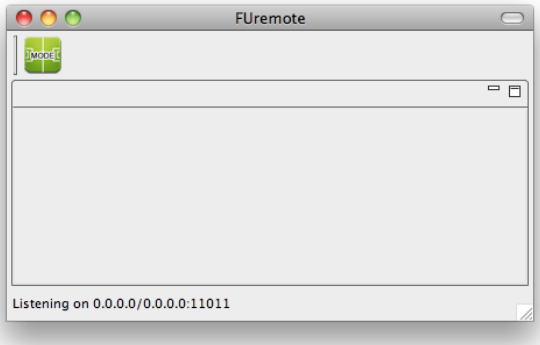


Abbildung 7: FUremote Core ohne weitere Plug-ins

FUremote Robot Plug-in (de.fumanoid.furemote.robot)

Das Robot-Plug-in erkennt und verwaltet automatisch im Netzwerk erreichbare Roboter. Es stellt alle Roboter in einem Fensterbereich (*View*) dar und bietet dem Anwender grundlegende Interaktionsmöglichkeiten wie das An- und Abschalten von Motoren an.

Plug-ins können das Robot-Plug-in benutzen, um - dem *Observer*-Entwurfsmuster entsprechend - über beliebige Zustandsänderungen benachrichtigt zu werden.

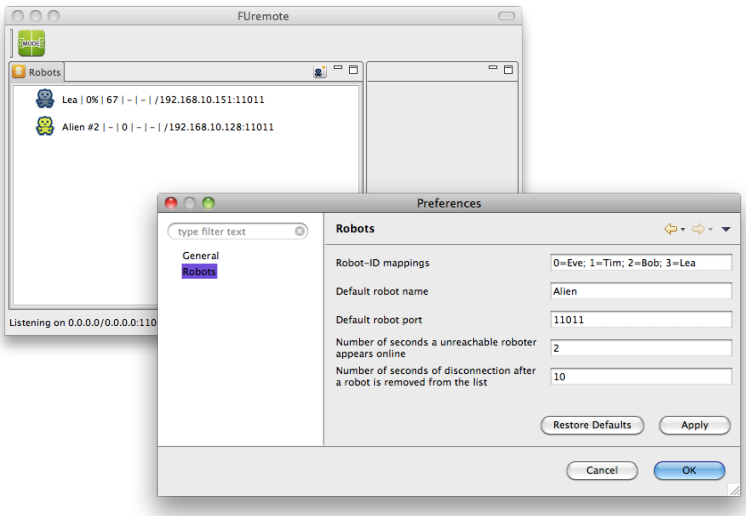


Abbildung 8: Robot Plug-in mit zwei erkannten Robotern und der geöffneten Konfiguration

FUremote Consoles Plug-in (de.fumanoid.furemote.console)

Das Eclipse UI-Plug-in kennt als Anzeigeeinheiten nur *Editoren*[v] und *Views*[v]. Die aus Eclipse bekannte Konsole kann nur in einer Instanz existieren. Um diese Einschränkung zu umgehen, verwendet dieses Plug-in einen modifizierten Quellcode der Konsole.
Weiterhin wird FUremote Core um zwei Konsolen erweitert. Eine Konsole dient der Ausgabe von Nachrichten, die an die Standardausgabe bzw. Standardfehlerausgabe geschickt wurden. Eine zweite Konsole dient der Verfolgung des Netzwerkverkehrs.

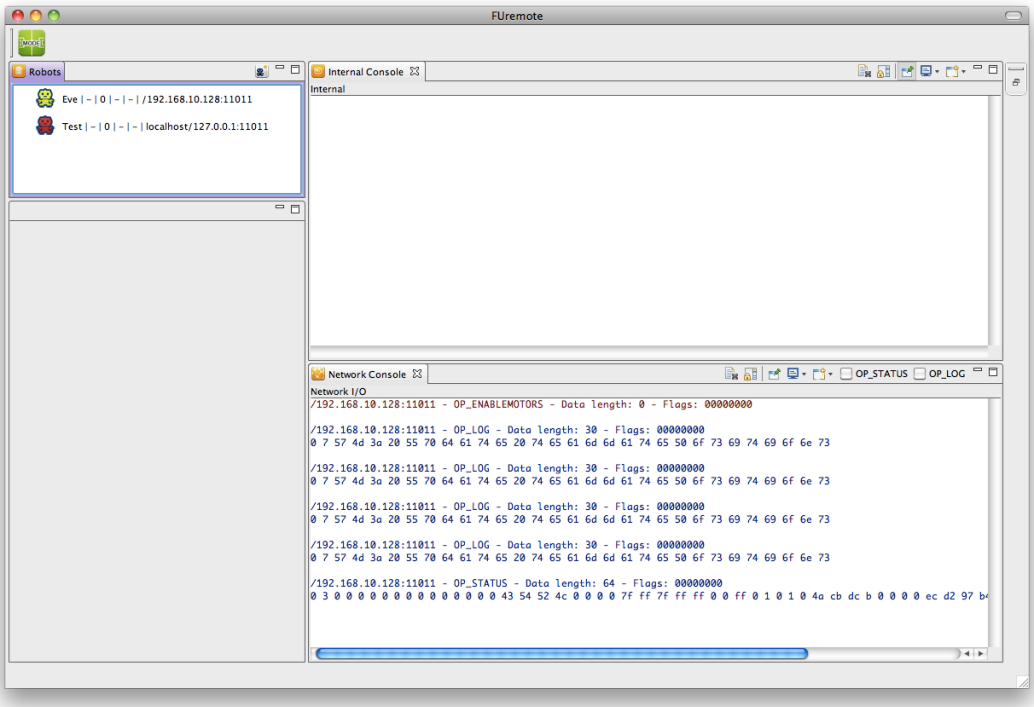


Abbildung 9: FUremote Console Plug-in mit Nachrichten von zwei erkannten Robotern

---

[v] Editoren und Views sind die Elemente mit denen Inhalte in einem Eclipse RCP-Fenster angezeigt werden. Dabei stellt der Editor für den Anwender eine Möglichkeit zur Bearbeitung eines konkreten Inhaltstyps - zum Beispiel eines XML-Dokuments - bereit.

Um den Editor können beliebig viele Views angeordnet werden. Ein View ist für die Bereitstellung zusätzlicher Informationen zuständig. Konventionellerweise bezieht sich ein View dabei auf das im Editor geöffnete Dokument.
Ein View kann aber ebenso zur Programmierung völlig unabhängiger Funktionieren genutzt werden. Auf diese Weise entsteht eine Art Unterprogramm für dedizierte Einsatzzwecke.

Um Editoren und Views zu platzieren, bedient man sich einer Perspective. Sie definiert die Anordnung eines Editors und der dazugehörigen Views. Definiert man mehrere Perspectives, kann der Anwender zwischen diesen nach Belieben wechseln. [7]

Motion Editor

Eine der Hauptarbeiten im Umgang mit den Robotern besteht im Entwurf und in der Optimierung so genannter *Motions*. Dabei handelt es sich um Bewegungsabläufe, die entweder statisch oder dynamisch sein können.

Der Motion Editor ist ein Programm, das ursprünglich in Delphi geschrieben worden ist und bereits mit der Roboter-Vorgängergeneration genutzt wurde. Durch den langen Umgang mit dem Motion Editor waren die Ansprüche an die neue Version besonders klar.

Realisiert wurde der Motion Editor als Plug-in (`de.fumanoid.furemote.motion↵editor`). Verwendet wurde als zentrale visuelle Komponente eine Tabelle - genauer ein *Grid* - um die Motorwerte in der X-Achse und in ihrem zeitlichen Verlauf in der Y-Achse darzustellen.

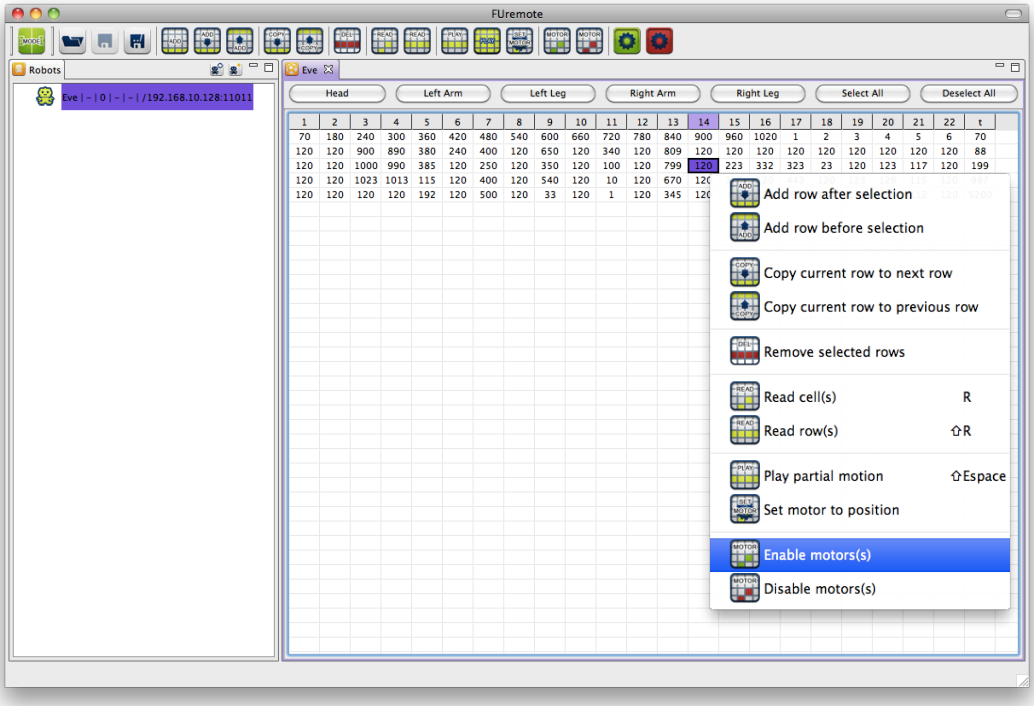


Abbildung 10: Motion Editor bei der Bearbeitung einer Motion

Zum Zeitpunkt des Entwurfs war keine freie ausgereifte Grid-Implementierung ausfindig zu machen. Es musste daher auf die Nebula Grid Widget-Implementierung der Eclipse Foundation zurückgegriffen werden, die sich zum Zeitpunkt der Fertigstellung dieser Arbeit noch immer im Alpha-Status befand.

Kopplungsarme Propagierung von Selektionen

Um dem Anwender einen hohen Bedienkomfort zu ermöglichen und Fehlbedienungsquellen minimieren zu können, müssen auch entfernte Programmteile auf Änderungen des Grids reagieren können.

Für Selektionsereignisse in der Bedienoberfläche bietet das Eclipse Framework - genauer das JFace-Plug-in - einen *Selection Service* an.
Implementiert ein Bedienelement einen *Selection Providern* werden Selektionen innerhalb des Bedienelements an den globalen Selection Service weitergeleitet.
Das Gegenstück sind *Selection Listener*: Tritt eines Selektionsereignis auf, wird dieses an alle registrierten Selection Listener verteilt.

Auf diese Weise erfährt jeder Selection Listener von allen Selektionen im Programm. Es ist die Aufgabe des Selection Listeners, irrelevante Selektionen zu verwerfen.

Durch die Implementierung eines Selection Providers für das Nebula Grid Widget wird die Verteilung der Selektionsereignisse unter allen aktivierten Plug-ins ermöglicht. So können bedarfsweise bestimmte Knöpfe ein- und ausgeblendet oder bewusst deaktiviert werden, ohne die Kopplung zwischen den Plug-ins zu erhöhen.

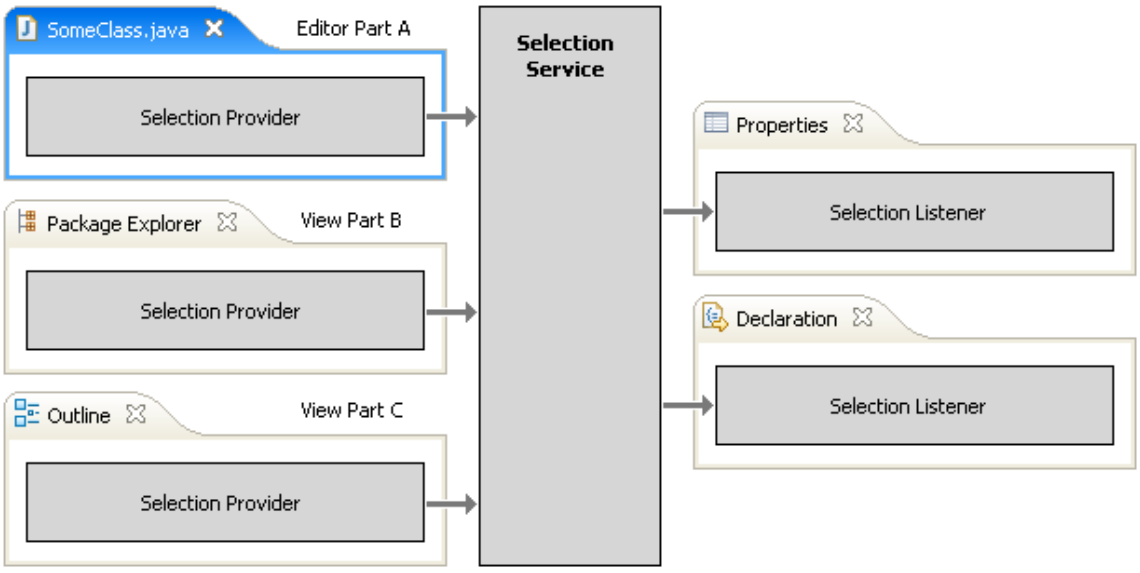


Abbildung 11: Zusammenspiel von Selection Provider, Selection Service und Selection Listener [12]

Listing 5 zeigt einen Selection Listener, der sich für Selektionsereignisse des Motion Editors interessiert. Innerhalb der `init`-Methode registriert sich der Listener beim Selection Service. Dieser ruft beim Auftreten einer Selektion die `selectionChanged`-Methode des Listeners auf und übergibt ihr die Selektion. Schließlich findet im Methodenrumpf mithilfe von `instanceof` ein Überprüfung der Selektionselemente statt.

```
package de.fumanoid.furemote.motioneditor.editors;
[…]
import org.eclipse.ui.ISelectionListener;
import org.eclipse.ui.ISelectionService;

public class MySelectionListener implements ISelectionListener {
     public void init() {
          IWorkbenchWindow workbench = PlatformUI.
               getWorkbench().getActiveWorkbenchWindow();
          ISelectionService service = workbench.getSelectionService();
          service.addSelectionListener(this);
     }

     public void dispose() {
          […]
          service.removeSelectionListener(this);
     }

     public void selectionChanged(IWorkbenchPart part, ISelection selection) {
          if(part instanceof MotionEditor) {
               if(selection instanceof IStructuredSelection) {
                    IStructuredSelection structuredSelection =
                         (IStructuredSelection) selection;
                    for(Object element:structuredSelection.toArray()) {
                         if(element instanceof Point) {
                              /* do something */
                         }
                    }
               }
          }
     }
}
```

Listing 5: Verwendung des `SelectionService`

Eine Erläuterung des Datenformat, mit dem Motions vom Motion Editor gespeichert bzw. an die Roboter gesendet werden, befindet sich im Abschnitt 2.4 Netzwerk.

Weitere Plug-ins

Neben den drei Programmkern-Plug-ins und dem des Motion Editors existieren zum Zeitpunkt der Anfertigung dieser Arbeit folgende fünf weitere Plug-ins:

Plug-in	Beschreibung	
Camera	Das Camera-Plug-in kann Bilder von einem oder mehreren Robotern empfangen. Damit kann der Entwickler "durch die Augen" des Roboters sehen und Probleme leichter beheben.	
Init	Init erlaubt es, Spielanweisungen an einzelne Roboter oder das ganze Team zu schicken. Die Kommandos reichen von einfachen Befehlen wie "Spiel starten" bis zur Zuweisung von Rollen während des Spiels. Der entsprechende Teil des Protokolls wurde so offen gestaltet, dass dieses Plug-in vorab keine Kenntnisse über die dem Roboter bekannten Strategien, Rollen und Verhalten haben muss. Init verwendet einen im Abschnitt 2.4 näher erläuterten Befehlssatz zur Erfragung der auf den Robotern implementierten Strategien, Rollen und Verhalten.	
Log	Das Log empfängt Meldungen der Roboter und ermöglicht deren Filterung auf FUremote-Seite nach Schwere und Quelle der Meldung / des Fehlers.	
Pitch	In diesem Plug-in ist die klassische Fußballfeld-Darstellung implementiert. Der Entwickler kann der Anzeige die gemeldeten Positionen der Roboter und des Balls visuell entnehmen.	
Template	Das Template-Plug-in ist kein Plug-in für ein Produktiv-System sondern veranschaulicht anhand funktionierender Beispiele, wie ein Plug-in-Entwickler am besten bei der Erweiterung von FUremote vorgeht.	

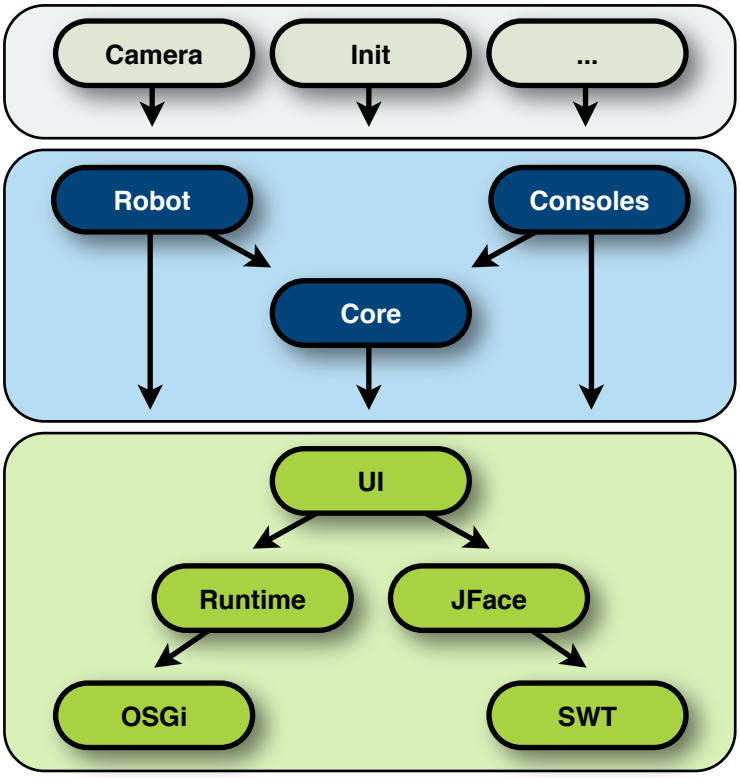


Abbildung 12: Erweiterung der FUremote Kern-Anwendung
(Eclipse RCP, Kern-Plug-ins) um darauf aufbauende Plug-ins wie Camera oder Init.

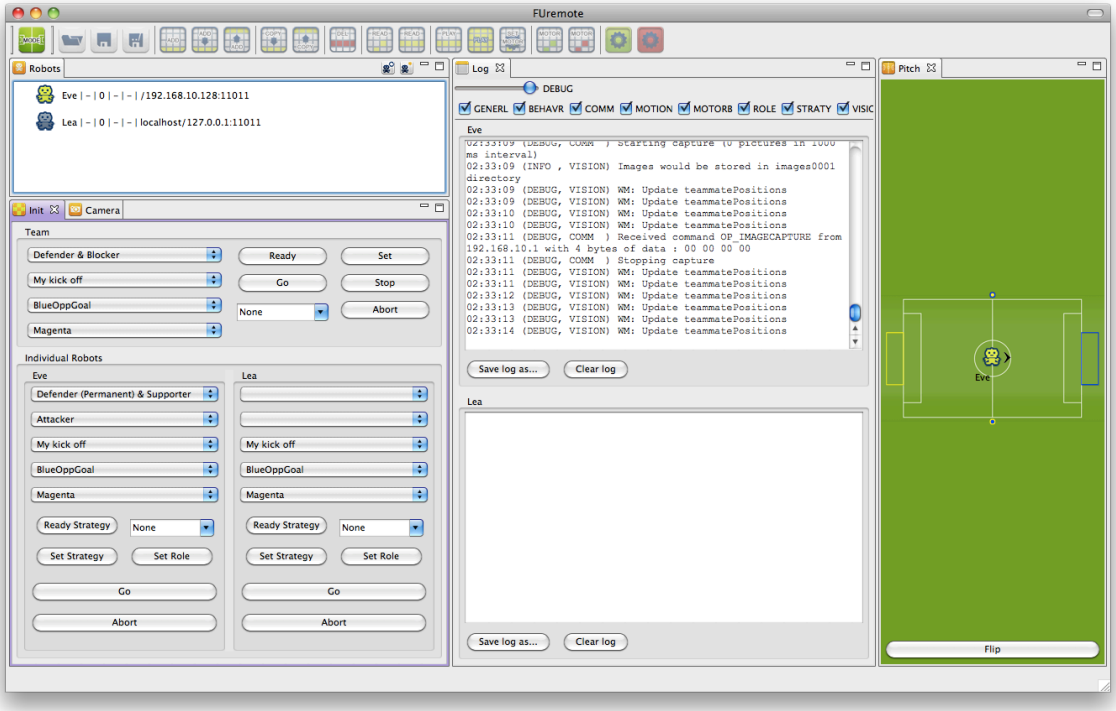


Abbildung 13: FUremote in der Standard-Ansicht

2.4. Netzwerk

Die Kommunikation zwischen der Anwendersoftware FUremote und dem Roboter-Gegenstück FUmanoid verwendet ein auf UDP basierendes Protokoll.
Da die RoboCup-Regeln die Verwendung anderer Protokolle während eines Wettkampfs untersagen, wurde es für das Klügste erachtet, stets auf UDP aufzubauen.

Standardmäßig wird der Port 11011 genutzt. FUremote erlaubt die Angabe eines neuen Listener-Ports, für den Fall, dass ein Konflikt mit einer anderen Mannschaft auftritt.

Dass nur eine Netzwerkverbindung geöffnet und nur ein Protokoll definiert werden musste, war ein zeitsparender Vorteil. Nachteilig ist, dass die Möglichkeit eines Paketverlusts immer besteht. Auch wenn dieser Fall bereits eingetreten ist, konnten signifikante Probleme immer durch erneuten manuellen Versand der Nachricht umgangen werden.

Ein folgenbehafteter Nachteil des Einsatzes von UDP ist in der Tatsache begründet, dass Pakete nicht zwangsläufig in der vorgesehenen Reihenfolge beim Empfänger ankommen müssen.
Der Motion Editor erfragt beispielsweise die Position jedes Motors jeweils mit einer separaten Anfrage. In einer ersten Version des Protokolls enthielt das Antwortpaket nicht die ID des abgefragten Motors.
Wollte der Anwender nun zur Aufzeichnung einer Motion mehrere Motoren zeitgleich abfragen, wurden von FUremote auch mehrere Abfragen an den Roboter gesendet. Mangels Kennzeichnung der Antwortpakete durch eine Motor-ID konnten die Antworten nicht ihrer korrespondierenden Anfrage zugeordnet werden.

Das Problem konnte nur auf zwei Weisen gelöst werden:

1. **Versand mit Pausen**
   Würde FUremote auf jede Motoren-Anfrage die Antwort abwarten, gäbe es kein Zuordnungsproblem. Dieser Lösungsansatz wäre allerdings durch die hohe Anzahl abfragbarer Motoren zu langsam. Außerdem müsste er das Problem der fehlenden Paketverlusterkennung lösen.

2. **Protokollanpassung**
   Würde jedes Antwortpaket die Motor-ID des befragten Motors enthalten, könnte FUremote die Antwort der zugehörigen Anfrage zuordnen.

Abgesehen von der geringfügigen Vergrößerung des Antwortpakets, bringt die zweite Lösung keine weiteren Nachteile mit sich und wurde somit übernommen.

Client-Server

Die Einordnung der Kommunikation zwischen den Netzteilnehmern im Falle der FUmanoid-Software ist nicht ganz eindeutig.

Klar ist, dass es sich bei der Kommunikation zwischen den Robotern um ein *peer-to-peer*-Netzwerk handelt. Weniger offensichtlich ist aber, wer bei der Kommunikation zwischen Workstation (FUremote-Software) und Roboter (FUmanoid-Software) der Client und wer der Server ist.

Der Name "Serverprogramm" des FUremote-Vorgängers würde nahe legen, dass es sich bei der Workstation um den Server und bei den Robotern um Clients handelt. Andererseits würde dies dem Ziel widersprechen, autonome Roboter zu entwickeln.

Tatsächlich ist es nicht die Workstation sondern sind es vielmehr die Roboter, die Ressourcen bereitstellen und die von der Workstation gestellten Anfragen beantworten [vi].

Ein weiterer Einordnungsversuch wäre das *Master-Slave*-Modell [13], bei dem einseitig vom *Master* Befehle an den *Slave* geschickt werden. In diesem Fall wäre die Workstation der Master und die Roboter Slaves. Da Roboter untereinander nur Informationen, aber keine Befehle austauschen, wäre das Modell nicht verletzt.

[vi] In der sich im Anhang befindlichen Protokolldefinition wird Server für die Bezeichnung der FUremote-Software und Client für die Roboter verwendet.

Protokollaufbau

Das FUmanoid-Protokoll ist paketorientiert. Ein Paket besteht aus zwei Teilen: dem stets 4 Byte großen Header und einem Teil für Nutzdaten (siehe Abbildung 14, hellgrauer Bereich).

Eingebettet werden FUmanoid-Pakete in UDP. Ein UDP-Paket verwendet einen 8 Byte großen Header, in dem unter anderem der Absender- und der Zielport gespeichert sind. FUremote und FUmanoid verwenden jeweils den Port 11011.

UDP basiert auf dem Internet Protokoll (IP), welches in der Version 4 einen Header mit einer Größe von 20-60 Byte verwendet. Die maximale Paketgröße ist auf 65.536 Byte [14, 15] beschränkt. Abzüglich der Header von IP, UDP und FUmanoid stehen für die Nutzdaten also 65.464 Byte zur Verfügung.

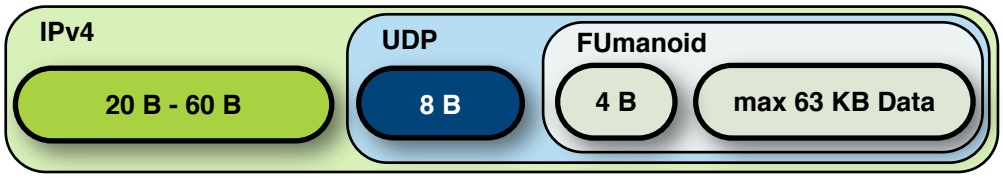


Abbildung 14: Aufbau des FUmanoid-Protokolls
unter Verwendung von UDP und IPv4

Der Header hat folgenden Aufbau:

- `unsigned short`**: operation id**
  Die *operation id* ist eine Zahl, welche die auszuführende Operation identifiziert. Wichtige IDs sind beispielsweise die 1, welche für Status-Pakete und die 7, welche für Motorabfragen verwendet werden.

- `unsigned byte`**: flags**
  Flags beschreiben das Paket detaillierter. Derzeit existiert nur das Flag 1 (`FLAG_IS_ANSWER`), mit dem ausgedrückt wird, dass es sich bei dem Paket um eine Antwort handelt.

- `unsigned byte`**: unused**
  Dieses Feld wird gegenwärtig nicht verwendet.

Das Datenfeld ist abhängig von der verwendeten Operation.
So verlangt die Operation `OP_SETWALKINGSPEED` beispielsweise 3 Parameter vom Typ `unsigned byte` zur Beschreibung von Vorwärts-, Seitwärts- und Rotationsgeschwindigkeit.

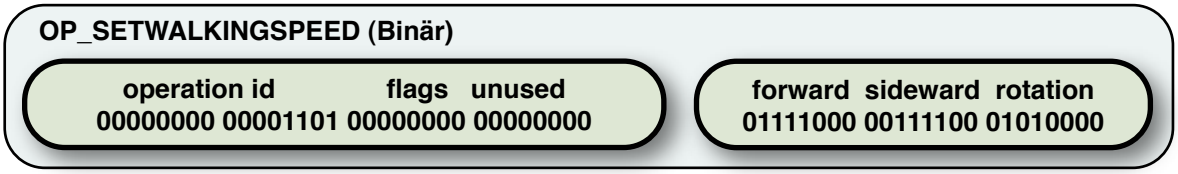


Abbildung 15: Beispiel-Paket zum Setzen der Laufgeschwindigkeit
mit den dezimalen Werten 120, 60 und 80.

Operationen

Derzeit sind 25 Operationen definiert, von denen bereits 21 in FUremote implementiert worden sind. Zu den implementierten Operationen gehören unter anderem:

- **OP_LOG**
  Dieser Befehl schickt eine Log-Nachricht vom Roboter zur Workstation.
  Neben der in ASCII kodierten Nachricht, enthält ein `OP_LOG` die Quelle der Meldung und eine Einstufung, die von `DEBUG` bis `ERROR` reicht.

- **OP_STATUS**
  Alle 500ms wird von jedem Roboter ein Status-Paket als Broadcast versendet. Den beteiligten Robotern dient es zur Ortung der Teammitglieder und des Balls.
  Neben der automatischen Erkennung von eingeschalteten Robotern entnimmt FUremote dem Status-Paket ebenfalls Informationen zu den Positionen von Roboter und Ball, der aktuell gespielten Rolle usw.

- **OP_GETSTRATEGIES / OP_GETROLES / OP_GETBEHAVIORS**
  Das Init-Plug-in ermöglicht das Setzen der zu verwendenden Strategie, Rolle und des Verhaltens. Um die Auswahlmöglichkeiten des Plug-ins aktuell zu halten, müssen die entsprechenden Implementierungen auf Roboterseite mit einer Bezeichnung versehen werden. Mittels der parameterlosen Anfrage `OP_GETSTRATEGIES` (analog `OP_GETROLES`, `OP_GETBEHAVIORS`) kann eine Liste der aktuell auf dem Roboter verfügbaren Strategien angefordert werden.

Die Antwort enthält eine NULL-terminierte Liste von <ID, Bezeichnung>-Paaren. FUremote verwendet diese Liste z.B. zur besseren Auswertung von Status-Paketen, welche nur die IDs nicht aber die Bezeichnungen der gesetzten Strategien, Rollen und Verhalten enthalten.

- **OP_PLAYMOTION**
  Mit Hilfe dieser Operation kann FUremote dem Roboter eine Motion zur Ausführung schicken. Eine Motion besteht aus Y *Moves* (siehe Abbildung 16). Ein Move besteht aus N Motorwerten und der Verzögerung (*Delay*) zum nächsten Move, wobei jeder dieser Werte durch 2 Byte dargestellt wird.
  Zur Darstellung einer Motion werden die Moves miteinander konkateniert.
  Da Moves durch kein spezielles Symbol voneinander getrennt sind, werden den Moves deren Anzahl - also Y - vorangestellt.

  Das Format, mit dem der Motion Editor Motions speichert (siehe Listing 6), ähnelt dem Paket-Format stark. Allerdings werden bei der Speicherung Kommata zur Separierung von Motorwerten, NEWLINE zur Trennung von Moves und `-1` zur Auszeichnung des Motionendes verwendet.

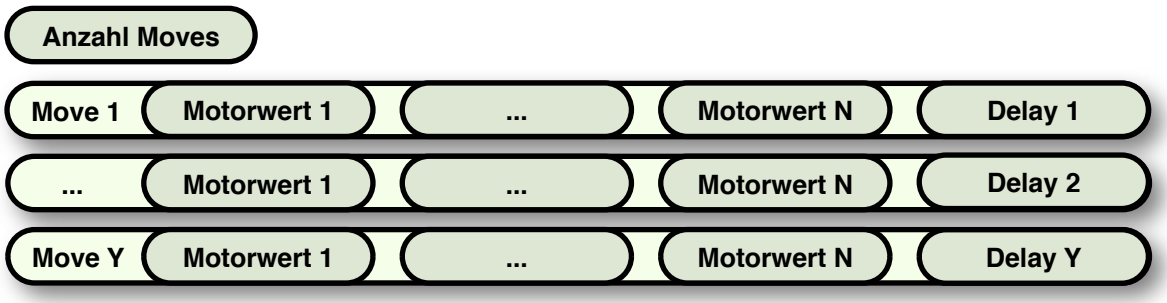


Abbildung 16: Schematischer Aufbau eines OP_PLAYMOTION-Pakets
bzw. einer vom Motion Editor gespeicherten Motion

```
101,102,103,14,15,16,17,18,19,110,111,112,113,114,81,11,11,18,19,20,421,2,100,
77,130,133,54,60,70,80,85,85,77,120,130,132,120,120,30,11,18,25,42,461,10,150,
-1
```

Listing 6: Gespeicherte Motion mit 2 Moves und 22 Motoren

Zu den nicht implementierten Operationen gehören OP_LIMITTEAM zur Gruppierung von Robotern, OP_TIMESYNC zur Synchronisierung der Zeit zwischen Roboter und Workstation und OP_GETMOTOROFFSETS zur Abfrage der Motorabweichungen vom Nominalwert.

Netzwerkschnittstelle

Der Netzwerkzugriff ist im Plug-in FUmanoid Core (de.fumanoid.furemote) implementiert und befindet sich dort im Package de.fumanoid.furemote.network.

Für die Erstellung eines Sockets und für die Verarbeitung ein- und ausgehender Pakete ist die Klasse UDPConnection zuständig. Sie implementiert das Runnable-Interface und wird als eigenständiger Thread bei Applikationsstart ausgeführt.

Die UDPConnection-Klasse wurde streng nach dem *Observer*-Entwurfsmuster entworfen. Zudem werden innerhalb von UDPConnection mehrere Interface definiert. Interessiert sich eine Klasse für den Netzwerkverkehr, muss sie mindestens eins der Interface implementieren und sich als Listener bei UDPConnection registrieren. Abhängig vom Typ des Listeners wird dieser dann über den ein- oder ausgehenden Datenverkehr benachrichtigt.

Listing 7 enthält einen Ausschnitt aus dem Programmteil der Klasse UDPConnection, der für den Empfang von UDP-Paketen zuständig ist.

```
this.socket.receive(this.packet);

// Make copy of the data and notify all listeners for incoming data
byte[] dataCopy = new byte[this.packet.getLength()];
System.arraycopy(this.packet.getData(), 0, dataCopy, 0, dataCopy.length);
this.firePacketReceived(this, packet.getAddress(), packet.getPort(),
                                                                dataCopy);
try {
     // Create message from packet and notify listeners for incoming Messages
     this.fireMessageReceived(Message.createFrom(this, this.packet));
} catch (InvalidMessageFormatException e) { }
```

Listing 7: Code zur Verarbeitung eingehender UDP-Pakete

Nachrichtenverarbeitung

Unterschieden werden generell Pakete (`Packet`) und Nachrichten (`Message`). Pakete beschreiben die unbehandelten Daten, die ein UDP-Paket enthält, während Nachrichten den Inhalt eines UDP-Pakets prüfen und im Erfolgsfall als FUmanoid-Nachricht darstellen. Für beide Typen stehen Listener bereit, über die eine Klasse über einen Dateneingang informiert werden kann.

Die Netzwerk-Konsole des Consoles-Plug-ins registriert einen so genannten `PacketListener` und erhält damit unmittelbar nach Empfang eines UDP-Pakets eine Datenkopie. Dies ermöglicht das Debuggen der Netzwerkkomponenten mit minimaler Beeinflussung durch mögliche Störfaktoren. Alle übrigen Programmteile registrieren `MessageListener`.
Registriert eine Klasse einen `MessageListener`, erhält erstere nach Dateneingang nur dann eine Benachrichtigung, wenn es sich bei dem Paket um eine FUmanoid-Nachricht im Datenbereich des UDP-Pakets handelt. Ist dies der Fall, wird ein `Message`-Objekt generiert und an den Listener übergeben.

Intern besteht eine `Message` aus dem dazugehörigen `UDPConnection`-Objekt, der IP-Adresse und des Ports des Absenders und der *Payload* des empfangenen Pakets. Für die Verarbeitung der Payload ist die von `Message` komponierte Klasse `MessageBody` zuständig.

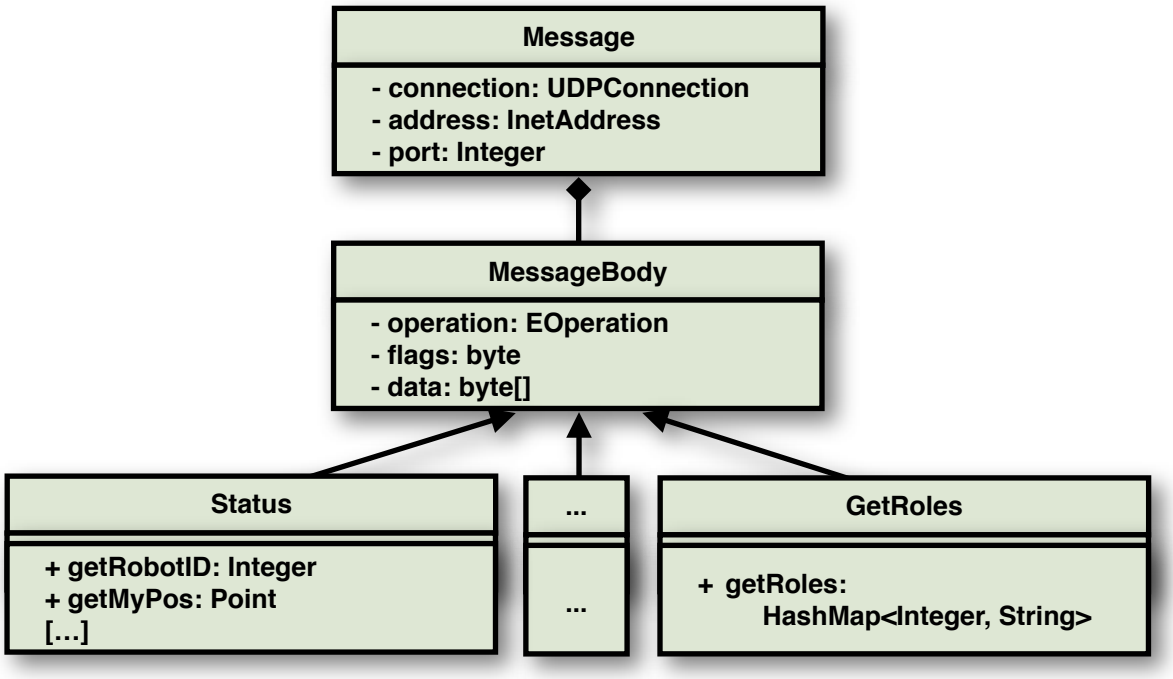


Abbildung 17: UML-Klassendiagramm wichtiger Nachrichten-Klassen

Der Konstruktor der `MessageBody`-Klasse benötigt dazu den empfangenen Payload als Byte-Array und die übermittelte Größe. Die Klasse ist generisch und prüft nur, ob bestimmte Bedingungen - wie die Mindestlänge des Pakets - nicht verletzt wurden.

```
public MessageBody(byte[] data, int length)
          throws InvalidMessageFormatException {

     if(data.length < length) throw new ArrayIndexOutOfBoundsException(
               "The provided length is greater than the array's length!");

     if(data.length < MessageBody.MIN_LENGTH || length < MIN_LENGTH)
          throw new InvalidMessageFormatException(
"The passed message must be at least of " + MIN_LENGTH + " bytes length!");

     // Extracts the eOperation by the first 2 bytes
     this.operation = EOperation.byOrdinal(
                              ByteUtils.bigEndianComposition(data[0], data[1]));

     // Extracts the flags from the 3rd byte
     this.flags = data[2];

     // If there are enough bytes the data part gets copied to the private
     // data field.
     // Instead of cp the full byte array only the needed length is copied.
     if(length == MessageBody.MIN_LENGTH) this.data = new byte[0];
     else {
          this.data = new byte[length-MessageBody.MIN_LENGTH];
          System.arraycopy(data, MessageBody.MIN_LENGTH,
               this.data, 0, length-MessageBody.MIN_LENGTH);
     }
}
```

Listing 8: Prüfung einer Payload auf Validität

Unterklassen von MessageBody stellen komfortable Methoden zum Zugriff auf die Felder des FUmanoid-Pakets bereit. Die `Status`-Klasse ist beispielsweise zuständig für Status-Pakete und definiert unter anderem die Methode `public Point getBallPos()`, welche die Koordinaten zurück gibt, unter denen der Roboter den Ball glaubt (siehe Listing 9).

```
public Point getMyPos() {
      int x = ByteUtils.bigEndianComposition(
            this.getPayload().getByte(20), this.getPayload().getByte(21));
      int y = ByteUtils.bigEndianComposition(
            this.getPayload().getByte(22), this.getPayload().getByte(23));
      return (x == 32767 || y == 32767) ? null new Point(x, y);
}
```

Listing 9: Zugriff auf Felder eines FUmanoid-Pakets

Roboterüberwachung

Das Robot-Plugin erkennt automatisch verfügbare Roboter und listet diese in einer Liste auf. Dazu registriert das Plug-in - konkreter die `Activator`-Klasse - einen `MessageListener` bei dem bereits gestarteten `UDPConnection`-Thread. Von nun an wird jede eingehende Nachricht an den Listener weitergeleitet. Anhand der Absender-IP-Adresse der Nachricht ermittelt das Plug-in, ob der erkannte Roboter sich bereits in der Liste befindet. Ist dies nicht der Fall, wird er der Liste hinzugefügt, indem eine Instanz der Klasse `Robot` erstellt wird.

```
synchronized public static Robot genFromMessage(Message msg) {
      // Get an already known robot
      Robot robot = Robot.getFromMessage(msg);

      // If robot not known create one
      if(robot == null) {
            robot = new Robot(msg.getName(), msg.getConnection(),
                                          msg.getAddress(), msg.getPort());
            Robot.robots.add(robot);
            Robot.fireRobotAdded(robot);
            robot.startStateChecker();
      }
      return robot;
}
```

Listing 9: Prüfung auf Bekanntheit eines Roboters; gegebenenfalls Erzeugung einer `Robot`-Instanz

Während der Instanzierung eines `Robot`-Objekts, erstellt dieses intern einen dedizierten Überwachungsthread der Klasse `RobotStateChecker`.

Nach erfolgter Instanzierung wird dieser Überwachungsthread gestartet, der für die Aktualisierung der graphischen Roboter-Liste zuständig ist.

```
private Robot(String name, UDPConnection conn, InetAddress addr, int port) {
      super();
      this.connection = conn;
      this.name = name;
      this.state = EState.OFFLINE;
      this.address = addr;
      this.port = port;
      this.connection.addIncomingMessageListener(this);
      Robot.addRobotListener(this);
      this.stateChecker = new RobotStateChecker(this);
}
```

Listing 10: Erzeugung eines `Robot`-Objekts und Erstellung einer `RobotStateChecker`-Instanz

Roboter können sich in den folgenden Zuständen befinden:

ONLINE	Die letzte empfangene Nachricht des Roboters ist max. 2 Sekunden alt und erscheint grün in der Roboter-Liste.	
UNREACHABLE	Der Roboter ist plötzlich, das heißt seit mindestens 2 aber maximal 10 Sekunden, nicht mehr erreichbar und wird rot in der Roboter-List dargestellt.	
OFFLINE	Seit mindestens 10 Sekunden wurde keine Nachricht mehr vom Roboter empfangen. Es ist davon auszugehen, dass es sich nicht um einen kurzzeitigen Ausfall handelt. Daher wird der Roboter graublau dargestellt.	

Um Zustandsänderungen zu bemerken, startet ein Roboter zunächst im Zustand `OFFLINE`. Zudem registriert der zugehörige `RobotStateChecker` einen `MessageListener` und setzt einen internen booleschen Wert mit der Bezeichnung `messageReceived` auf `false`.

In einer Endlosschleife versetzt sich der Überwachungsthread für maximal 2 Sekunden in den `WAIT`-Zustand. Trifft in dieser Zeit eine Nachricht vom überwachten Roboter ein, wird `messageReceived` auf `true` gesetzt und der Thread verlässt den `WAIT`-Zustand (siehe Listing 10).

```
synchronized public void messageReceived(Message msg) {
      Robot robot = Robot.getFromMessage(msg);
      if (RobotStateChecker.this.robot.equals(robot)) {
            this.messageReceived = true;
            this.notify();
      }
}
```

Listing 10: Warten auf eingehende Nachrichten vom überwachten Roboter

Wurde innerhalb der Wartezeit eine Nachricht empfangen, wird der Roboter in den ONLINE-Zustand versetzt. Anderenfalls wird die vergangene Zeit auf eventuell bereits verstrichene Zeit hinzuaddiert. Abhängig von der Summe wird der Roboter in den UNREACHABLE- oder den OFFLINE-Zustand versetzt.

```
synchronized public void run() {
      if(this.stop) return;
      this.messageReceived = false;
      try {
            this.wait(this.timeout * 1000);
      } catch (InterruptedException e) {}

      // Switch online state according to messageReceived
      if(PlatformUI.isWorkbenchRunning()) {
            // Saves the time passed between now and the last online state
            if(!this.messageReceived) consecutiveTimeout += this.timeout;
            else consecutiveTimeout = 0;

            EState newState = EState.UNREACHABLE;
            if(this.consecutiveTimeout >= this.maxConsecutiveTimeout) {
                  newState = EState.OFFLINE;
            } else if(messageReceived) {
                  newState = EState.ONLINE;
            }

            PlatformUI.getWorkbench().getDisplay().asyncExec(
                        new StateSetter(this.robot, newState));

            // Recheck
            new Thread(this, this.thread.getName()).start();
      }
}
```

Listing 11: Wechsel des Roboter-Zustandes

Mittels des `IRobotListener`-Interfaces, können Klassen über Zustandsänderungen des Roboters benachrichtigt werden. Kommt es zu einem Status-Wechsel, werden die Listener über die `robotChanged`-Methode, über die Änderung der Eigenschaft `STATE` in Kenntnis gesetzt.
Die graphische Roboterliste kann auf diese Weise die Darstellung des betroffenen Roboters ändern.

```
public void setState(EState state) {
      if(this.state.equals(state)) { return; }
      this.state = state;
      Robot.fireRobotChanged(this, PROPERTIES.STATE);
}
```

Listing 12: Status-Wechsel und Benachrichtigung registrierter Listener

Die übrigen Roboter-Eigenschaften werden direkt in den `Robot`-Instanzen verwaltet. In privaten Feldern werden unter anderem Name, IP, Port und Position gespeichert (siehe Listing 13).

```
private String name;
private HashMap<Integer, String> availableStrategies;
private HashMap<Integer, String> availableRoles;
private HashMap<Integer, String> availableBehaviors;
private int robotID;
private int strategyID;
private int roleID;
private int behaviorID;
private Point myPos;
private Point ballPos;
private Integer battery;
```

Listing 13: Von `Robot` verwaltete Roboter-Eigenschaften

Existierende `Robot`-Instanzen registrieren im Laufe ihrer Instanzierung einen eigenen `MessageListener` und prüfen eingehende Nachrichten darauf, ob sie von ihrem realen Pendant stammen. Sobald eine Nachricht eintrifft, werden abhängig vom Nachrichtentyp alle relevanten Informationen verwendet, um die internen Eigenschaften zu aktualisieren:

Status-Nachrichten werden Informationen zur Position des Roboters und der des Balls entnommen und GetRoles-Nachrichten enthalten eine Liste der auf dem Roboter implementierten Rollen.

Die so in Erfahrung gebrachten Änderungen der Roboter-Eigenschaftsfelder werden - analog zum Roboter-Status - an die registrierten `RobotListener` weitergeleitet.

```
public void messageReceived(Message msg) {
      // Filter Robots
      if(Robot.getFromMessage(msg) != this) return;
            // Only treat status messages
            if(msg.uses(EOperation.OP_STATUS)) {
                  final Status status_msg = new Status(msg);
                  […]
                  int strategyID = status_msg.getStrategyID();
                  // Has the current strategy ID changed?
                  if(this.strategyID != strategyID) {
                        this.strategyID = strategyID;
                        // Notify all listeners of the property changement
                        Robot.fireRobotChanged(this, PROPERTIES.STRATEGYID);
                  }
            }
      }
}
```

Listing 14: Behandlung einer eingehenden Nachricht durch eine `Robot`-Instanz und Benachrichtigung der `RobotListener` über Änderungen der Strategie-ID

Nachrichtenversand

Die UDPConnection-Klasse stellt eine Methode zum Versand von Objekten vom Typ MessageBody bereit. Die send-Methode benötigt über das MessageBody-Objekt hinaus die IP-Adresse und den Port des Empfängers.

```
public void send(InetAddress iaddr, int iport, MessageBody payload) {
     this.send(iaddr, iport, payload.toBytes());
}
```

Listing 15: Versandmethode für Objekte vom Typ MessageBody

Die generische send-Methode verwendet den geöffneten Socket zum Versand der Nachricht und informiert registrierte OutgoingMessageListener über die erfolgreiche Bearbeitung.

```
public void send(InetAddress iaddr, int iport, byte[] data) {
     DatagramPacket packageToSend = new DatagramPacket(data, data.length);
     packageToSend.setAddress(iaddr);
     packageToSend.setPort(iport);
     try {
          this.socket.send(packageToSend);
          this.fireMessageSent(Message.createFrom(this, packageToSend));
     } catch (IOException e) { } catch (InvalidMessageFormatException e) { }
}
```

Listing 16: Versand eines Byte-Arrays und Benachrichtigung registrierter Listener

Zur abstrakten Erzeugung von MessageBodys kann auf die MessageBodyFactory zurückgegriffen werden. Die *Factory* akzeptiert komplexe Datentypen als Argumente und wandelt diese in valide FUremote-Pakete um.
Die Plug-ins verwenden weder direkt die UDPConnection-Klasse, noch die MessageBodyFactory. Stattdessen generieren convenience-Methoden der Robot-Klasse die MessageBodys und versenden diese über die Schnittstelle, über die vom Roboter zuletzt Daten empfangen worden sind.

```
Robot lea = Robot.getFromName("Lea");
if(lea != null) myRobot.goToPosition(new Point(10, 20), 7)
```

Listing 18: Befehl an Roboter sich an eine bestimmte Position mit einem bestimmten Winkel zu begeben

```
private void sendMessage(MessageBody mb) {
     this.getConnection().send(this.getAddress(), this.getPort(), mb);
}

public void goToPosition(Point pos, int angle) {
     this.sendMessage(MessageBodyFactory.goToPosition(pos, angle));
}
```

Listing 19: Implementierung der `Robot`-Methoden `sendMessage` und `goToPosition`.

3. Ausblick

Im Rahmen dieser Bachelorarbeit wurden alle bisherigen Entwickler-Programme des Teams in eine stabile und erweiterbare Plattform migriert. In dieser Zeit konnten sowohl Erfahrungen mit der Eclipse Rich Client Platform als auch bzgl. der tatsächlichen Anforderungen gesammelt werden. In diesem Kapitel sollen mögliche Erweiterungen angerissen werden.

3.1. GameController

Ziel ist es bis spätestens zum RoboCup 2010 das GameController-Protokoll in FUremote zu implementieren. Dieses Protokoll wird zur Information über den aktuellen Spielstand, die verbleibende Zeit, usw. verwendet.
Die erfolgreiche Implementierung auf Roboterseite erhöht bereits deren Autonomie und würde FUremote die Spielsituation noch besser verfolgen lassen.

3.2. Automatische Farbkalibrierung

Roboter verwenden eine Kamera um ihre Umgebung visuell wahrzunehmen. Durch den enormen Miniaturisierungsgrad muss ein gewisser Teil der Kameralogik vom FUmanoid-Team gelöst werden. Dazu gehört die manuelle Kalibrierung bzgl. der Farben.

Auch wenn die Kalibrierung mittelfristig autonom auf den Robotern ausgeführt wird, muss es eine Möglichkeit geben, falsche Kalibrierungsergebnisse schnell zu erkennen und beheben zu können. Ein visuelle Integration des Kalibrierungsprozesses in FUremote würde eine hohe Qualität der Ergebnisse gewährleisten und in kritischen Situation hilfreich sein.

3.3. Netzwerk

Bisher ist die Netzwerkkomponente Bestandteil der FUmanoid Core-Plug-ins. Die Auslagerung in ein Netzwerk-Plug-in würde dem Plug-in-Entwurfsmuster entsprechen, den Grad der Modularisierung erhöhen und damit potentielle Änderungen auf noch weniger Plug-ins verringern.

Die Definition der verschiedenen Nachrichten-Typen findet bisher durch ihre Implementierung statt. Zukünftige Nachrichten-Formate deklarativ mit XML zu beschreiben, würde das Netzwerkmodul von ständigen Änderungen am Format verschonen und Protokolldetails besser vom Quelltext trennen.

3.4. Motion Editor

Der Motion Editor ist bereits eine enorme Verbesserung zu seinem Vorgänger. Dennoch sind Verbesserungen in Bezug auf die Benutzerfreundlichkeit möglich und notwendig. Zu den Optimierungsmöglichkeiten gehören die Möglichkeit, Aktionen rückgängig zu machen, mehr Shortcuts einzurichten und noch mehr Informationen zu den Robotern intuitiv darzustellen. Beispielsweise könnten die entsprechenden Eingabefelder einer sich gerade überhitzenden Motors rot hinterlegt werden.

3.5. Threads

Bisher sind Netzwerk, die Roboterüberwachung und die grafische Oberfläche in eigenständige Threads ausgelagert. FUremote leidet unter der strukturellen Schwäche, dass der Netzwerk-Thread zu lange Ausführungspfade je empfangener Nachricht zu bearbeiten hat. Erst bei Kontakt mit der GUI kommt es zu einer Übergabe der Daten zwischen Netzwerk- und GUI-Thread.

Die Trennung von Netzwerk in Controller und Netzwerk mit Hilfe von `BlockingQueues` würde Nachrichtenstaus unter Volllast vermeiden und FUremote flüssiger bedienbar machen.

4. Fazit

Die Anforderungen, im Rahmen dieser Bachelorarbeit Motion Editor und Serverprogramm in Java - unter Berücksichtigung der neuen Robotergeneration - zu implementieren, wurden erfüllt.
Gleichzeitig stellte die Verwendung der Eclipse Rich Client Platform eine hohe Einstiegshürde dar. Die Programmierung des Anwendungskerns benötigte viel Zeit - ohne dass unmittelbar einsatzbereite Software entstand.

Die Bedenken, Eclipse RCP könnte für die Zwecke des FUmanoid-Projekts zu komplex sein, haben sich dennoch nicht bestätigt: In der Vorbereitungsphase zum RoboCup 2009 und obwohl das Programm noch jung war, konnte FUremote bereits erfolgreich zur Weiterentwicklung der Roboter eingesetzt werden.

Nach der Fertigstellung des Motion Editors und der Integration der Serverprogramm-Funktionalitäten blieb genügend Zeit, um weitere Ergänzungen und Optimierungen zu machen.

FUremote befindet sich seit mehreren Monaten in einer stabilen Version im Einsatz. Wenn zukünftige Anwendungen direkt in FUremote integriert und die im dritten Kapitel angesprochenen Verbesserungen berücksichtigt werden, kann sich die Produktivität weiter erhöhen und einen noch souveräneren Umgang mit den Robotern während der Wettkämpfe ermöglichen.

5. Literaturverzeichnis

[1] http://www.fumanoids.de/about/, zugegriffen am 13.08.2009

[2] http://124.146.198.189/overview/21.html, zugegriffen am 24.08.2009

[3] http://www.robotis.com/zbxe/bioloid_en, zugegriffen am 08.09.2009

[4] Harrop, R., Machacek (2005): Pro Spring. Apress, Inc., 1. Auflage, New York, Seite 49-51

[5] http://wiki.eclipse.org/FAQ_Where_did_Eclipse_come_from%3F

[6] Holzner, S. (2004): Eclipse. O'Reilly Media, Inc., 1. Auflage, Sebastopol, Seite 1-3

[7] McAffer, J., Lemieux, J.-M. (2005): Eclipse Rich Client Platform: Designing, Coding, and Packaging Java™ Applications. Addison Wesley Professional., 1. Auflage, Saddle River

[8] Scarpino, M., Holder, S., Ng, S., Milhalkovic, L. (2005): SWT/JFace in Action: How to design graphical applications with Eclipse 3.0.Manning Publications, 1. Auflage, Greenwich

[9] http://dictionary.zdnet.com/definition/event+sink.html zugegriffen am 24.09.2009

[10] http://www.ibm.com/developerworks/library/j-dyn0603/ zugegriffen am 24.09.2009

[11] http://wiki.eclipse.org/Lazy_Start_Bundles zugegriffen am 25.09.2009

[12] http://www.eclipse.org/articles/Article-WorkbenchSelections/article.html zugegriffen am 03.10.2009

[13] http://en.wikipedia.org/wiki/Master-slave_(technology) zugegriffen am 03.10.2009

[14] http://tools.ietf.org/html/rfc768, zugegriffen am 04.10.2009

[15] http://docs.hp.com/en/B9106-90013/UDP.7P.html, zugegriffen am 04.10.2009

[16] http://wiki.eclipse.org/Command_Core_Expressions, zugegriffen am 19.10.2009

6. Anhang

Der Anhang dieser Bachelorarbeit soll Informationen bereitstellen, die beim Umgang mit der Software FUremote und bei der Entwicklung auf FUremote aufbauender Plug-ins hilfreich sein können.

Dazu ist dieses Kapitel in vier Abschnitte gegliedert:

1. Protokolldefinition
2. Einrichtung Eclipse
3. Erstellung eines Plug-ins
4. Erstellung einer Binary Version

6.1. Protokolldefinition

Das Protokoll zur Kommunikation zwischen FUmanoid und FUremote basiert auf UDP und ist wie folgt definiert (Stand 7. Oktober 2009):

Jedes Paket fängt mit einem Header an, der den Befehl und eventuelle Flags definiert.
Die restlichen Bytes des Pakets sind abhängig vom Befehl.

Alle Werte sind big-endian (network byte order) kodiert.

Header

Der Header besteht aus 4 Bytes:

```
uint16_t operationID
uint8_t flags
uint8_t unused          // set to 0
```

wobei für Flags momentan nur

```
FLAG_IS_ANSWER = 1
```

definiert ist.

Dieses Flag wird gesetzt wenn ein Paket eine Antwort auf eine Anfrage ist.

Log-Nachricht

Dieser Befehl schickt eine Log-Nachricht (Debug/Trace-Ausgabe) vom Client zum Server.

Befehls-ID: 0 (OP_LOG)
Flags:
Typ: Befehl von Client an Server

Parameter:

0 Log-Level
1 Subsystem
2 String beliebiger Länge
(nicht null-terminiert)

Log-Level kann die folgenden Werte annehmen:

0: Debug (wirklich alles ausgeben)
1: Info (informationelles)
2: Warnings (nur Warnungen)
3: Errors (nur Fehlermeldungen)

Subsystem die folgenden Werte:

0: Allgemein
1: Behavior
2: Comm
3: Motion
4: Motorbus
5: Role
6: Strategy
7: Vision

Status

Alle 500ms von jedem Client als Broadcast geschickt um seine aktuelle Situation der Umwelt mitzuteilen.

Befehls-ID: 1 (OP_STATUS)
Flags:
Typ: Broadcast von Client

Parameter:

0 robot ID

1 version ID (== 4)

2- 3 my orientation (degrees)

4- 7 strategy ID (32 bit, 0 = aus)

8-11 role ID (32 bit, 0 = aus)

12-15 behavior ID (32 bit, 0 = aus)

16-19 motion ID (32 bit, 0 = aus)

20-21 my Position X (absolute)

22-23 my Position Y (absolute)

24-25 ball position X (absolute)

26-27 ball position Y (absolute)

28 battery in percent
(255 = unknown) (since version 1)

29 competition mode
(0 = test mode, 1 = competition mode) (since version 2)

30 game state (0 = stopped, 1 = ready, 2 = set, 3 = playing) (since version 2)

31 ID of goalkeeper (255 = no goal keeper) (since version 2)

32 opponent goal color (blue = 0, yellow = 1) (since version 2) (since version 2)

33 kick off side (1 = we have kickoff, 0 = opponent has kickoff) (since version 2)

34 kick off mode (0 = none, 1 = regular, 2 = penalty, 3 = free) (since version 2)

35 team color (0 = Magenta, 1 = Cyan) (since version 3)

36 log level (since version 3)

37-40 time in seconds since 1.1.1970 0:00 (since version 4)

41-44 motion data (4 byte of status information) (since version 4)

45-48 behavior data (4 byte of status information) (since version 4)

49-50 role data (2 byte of status information) (since version 4)

51 strategy data (1 byte of status information) (since version 4)

52-53 number of seconds the game is in progress (since last GO signal)

54 unused

...

63 unused

Ein Wert von 0 bei den IDs bedeutet entweder "nicht vergeben" oder das diese Funktion noch nicht verfuegbar ist. Ungenutzte Werte sind mit 0 gefüllt.

Initialisierung

Vom Server an den Roboter geschickt um den Roboter zu initialisieren, wird meistens von einem OP_STARTGAME Befehl gefolgt sein. Entspricht den Kommandos "READY" und "SET" beim Spiel.

Befehls-ID: 2 (OP_READYSET)
Flags:
Typ: Befehl von Server an Client

Parameter:

0-3 strategyID (32 bit)

4-7 roleID (32 bit)

8 mode

9 opponent goal color (blue = 0, yellow = 1)

10 state (0 = ready, 1 = set)

11 ID of goalie

12 own team color (0 = Magenta, 1 = Cyan)

Ein Wert von 0 bei den IDs bedeutet "nicht gesetzt" bzw "nicht vergeben".

"Mode" kann folgende Werte annehmen:

0: my kick off

1: my free kick off

2: my penalty kick

3: enemy kick off

4: enemy free kick off

5: enemy penalty kick

Role aktivieren

Aktiviert eine Role.

Befehls-ID: 3 (OP_SETROLE)

Flags:

Typ: Befehl von Server an Client

Parameter:

0-3 ID (32 bit) of role to set (0 = keine Role)

Disable motors

Deaktiviert einen oder mehrere Motoren.

Befehls-ID: 4 (OP_DISABLEMOTORS)

Flags:

Typ: Befehl von Server an Client

Parameter:

Byte-Array mit den Motor IDs (wenn keine MotorIDs, dann alle Motoren)

Enable motors

Aktiviert einen oder mehrere Motoren

Befehls-ID: 5 (OP_ENABLEMOTORS)

Flags:

Typ: Befehl von Server an Client

Parameter:

Byte-Array mit den Motor IDs (wenn keine MotorIDs, dann alle Motoren)

Play motion

Spielt eine Bewegung ab. Kann auch genutzt werden (Y=1) um die Motoren auf eine bestimmte Position zu setzen.

Befehls-ID: 6 (OP_PLAYMOTION)

Flags:

Typ: Befehl von Server an Client

Parameter:

0 Anzahl N der motoren

1 Anzahl Y der Zeilen/Bewegungen

2+ int16-array der Groesse (N+1) * Y (N goal positions + duration)

Stimmt N nicht mit der tatsächlich vorhandenen Motorzahl überein, dann ist das Verhalten dieses Befehls undefiniert.

Get motor status

Auf diese Anfrage schickt der Client einige Informationen zu genau einem Motor.

Befehls-ID: 7 (OP_GETMOTORSTATUS)
Flags:
Typ: Anfrage von Server an Client

Parameter:
motorID

Befehls-ID: 7 (OP_GETMOTORSTATUS)
Flags: FLAG_IS_ANSWER
Typ: Antwort von Client auf Anfrage

Parameter:
0 motorID
1 voltage (Dezivolt, das heißt 100 => 10V)
2 temperature (°C)
3 torque (0 = motor aus, 1 = motor an)
4- 5 present position (0..1023)
6- 7 present speed (0..1023)
8- 9 present load (bits 0-9 load, bit 10 direction (1=cw, 0=ccw))
10 error code

Der Error Code setzt sich wie folgt zusammen:

Bit 0 gesetzt - input voltage error
Bit 1 gesetzt - angle limit error (sollte nie kommen)
Bit 2 gesetzt - overheating error
Bit 3 gesetzt - range error (sollte nie kommen)
Bit 4 gesetzt - checksum error (sollte nie kommen)
Bit 5 gesetzt - overload error
Bit 6 gesetzt - Instruction error (sollte nie kommen)

Relevant sind nur Bits 0, 2 und 5.

Get available strategies

Übermittelt die Liste der verfügbaren Strategien (ID und Name) an den Server. ID 0 ist stets "Keine Strategie" und kann optional vom Client mitgeschickt werden.

Befehls-ID: 8 (OP_GETSTRATEGIES)
Flags:
Typ: Anfrage von Server an Client

Parameter: none

Befehls-ID: 8 (OP_GETSTRATEGIES)
Flags: FLAG_IS_ANSWER
Typ: Antwort von Client auf Anfrage

Parameter: Sequence of
- uint32_t - id of strategy (32 bit)
- char[..] - name of strategy (null-terminated string)

Get available roles

Übermittelt die Liste der verfügbaren Roles (ID und Name) an den Server. ID 0 ist stets "Keine Role" und kann optional vom Client mitgeschickt werden.

Befehls-ID: 9 (OP_GETROLES)
Flags:
Typ: Anfrage von Server an Client

Parameter: none

Befehls-ID: 9 (OP_GETROLES)
Flags: FLAG_IS_ANSWER
Typ: Antwort von Client auf Anfrage

Parameter: Sequence of
- uint32_t - id of role (32 bit)
- char[..] - name of role (null-terminated string)

Abort

Deaktiviert sämtliche Verhalten (Strategien, Roles, etc) und führt den Roboter in die Idle-Position (aufrecht stehenbleiben), in welcher er bis zum Empfang neuer Befehle verbleibt.

Befehls-ID: 10 (OP_ABORT)
Flags:
Typ: Befehl von Server an Client

Parameter: none

Start game

Startet das Spiel (erfordert in der Regel einen vorhergehenden OP_READYSET Befehl).

Befehls-ID: 11 (OP_STARTGAME)
Flags:
Typ: Befehl von Server an Client

Parameter: none

Stop game

Stoppt das Spiel. Die Strategien etc bleiben weiter aktiv, so daß durch Senden von OP_STARTGAME das Spiel fortgesetzt werden kann. Dieser Befehl kann also für kurze Timeouts genutzt werden.

Befehls-ID: 12 (OP_STOPGAME)
Flags:
Typ: Befehl von Server an Client

Parameter: none

Set Walking Speed

Läßt den Roboter laufen.

Befehls-ID: 13 (OP_SETWALKINGSPEED)
Flags:
Typ: Befehl von Server an Client

Parameter:
0 Vorwärtsgeschwindigkeit
1 Seitwärtsgeschwindigkeit
2 Rotationsgeschwindigkeit

Wenn der Roboter noch nicht läuft, führt dieser Befehl zum Laufen.
Bei bereits laufendem Roboter werden die neuen Geschwindigkeiten
gesetzt. Um den Roboter auf der Stelle gehen zu lassen, alle Parameter auf 0 setzen.

Um das Laufen komplett zu stoppen kann OP_BECOMEIDLE genutzt werden.

Set Motor ID

Setzt die ID eines Motors.

Befehls-ID: 14 (OP_SETMOTORID)
Flags:
Typ: Befehl von Server an Client

Parameter:
0 ID des zu ändernden Motors (0..254)
1 zu setzende ID (1..253)

Ein Wert von 254 als ID des zu ändernden Motors bedeutet "Alle Motoren" und würde im Prinzip bedeuten das alle erreichbaren Motoren (unabhängig von ihrer ID) die neue ID erhalten. Das macht dann natürlich nur Sinn wenn nur ein einzelner Motor angeschalten ist.

Set motor

Setzt einen oder mehrere Motorwerte eines einzelnen Motors.

Befehls-ID: 15 (OP_SETMOTOR)

Flags:

Typ: Befehl von Server an Client

Parameter:

0 motorID (mandatory)

1 torque (0 = motor aus, 1 = motor an)

2- 3 goal position (0..1023)

Set log level

Setzt den Log-Level.

Befehls-ID: 16 (OP_SETLOGLEVEL)

Flags:

Typ: Befehl von Server an Client

Parameter:

log level

Dabei kann Log-Level die folgenden Werte annehmen:

0: Debug (wirklich alles ausgeben) (beinhaltet 1)

1: Info (informationelles) (behinhaltet 2)

2: Warnings (nur Warnungen) (beinhaltet 3)

3: Errors (nur Fehlermeldungen)

Bei Roboter-Programmstart wird das Log-Level immer auf 0 (DEBUG) gesetzt.

Image capture

Startet oder stoppt die Bildübertragung.

Befehls-ID: 17 (OP_IMAGECAPTURE)
Flags:
Typ: Befehl von Server an Client

Parameter:
0 Operation type (0 oder 1)
1 Image count
2-3 Interval in ms

Gültige Operationstypen:
0: Image Capture deaktivieren (restl. Param werden ignoriert)
1: Vorgegebene Anzahl von Bildern im vorgegebenen Interval senden, das erste Bild wird sofort geschickt (bei Anzahl = 0 werden bis auf Widerruf Bilder gesandt)

Befehls-ID: 17 (OP_IMAGECAPTURE)
Flags: FLAG_IS_ANSWER
Typ: Antwort von Client auf Anfrage

Parameter:
jpeg-Bild

Reboot System

Startet das gesamte System neu.

Befehls-ID: 18 (OP_REBOOT)
Flags:
Typ: Befehl von Server an Client

Parameter:
0 Muß auf 42 gesetzt sein

Go to position

Positioniert den Roboter an die gewünschte Stelle.

Befehls-ID: 19 (OP_GOTOPOSITION)
Flags:
Typ: Befehl von Server an Client

Parameter:
0-1 x-Koordinate
2-3 y-Koordinate
4-5 Winkel

Limit team

Limitiert die Team-Mitglieder zu bestimmten Robotern. Das ermöglicht, das wir zum Beispiel Testspiele 2vs1 spielen können ohne das sich die Roboter untereinander in die Quere kommen.

Befehls-ID: 20 (OP_LIMITTEAM)
Flags:
Typ: Befehl von Server an Client

Parameter:
0 ID des ersten Team-Mitglieds
1 ID des zweiten Team-Mitglieds
...

Die Zahl der Parameter ist variabel. Es muß aber mindestens ein Team-Mitglied spezifiziert werden, üblicherweise der Roboter, an den der Befehl geschickt wird.
Der empfangende Roboter nimmt sich im übrigen auch implizit in dieses Team auf.

Um auf ein "alle sind im Team" Verhalten zurückzukehren muß der Roboter neu gestartet werden.

Time Sync

Synchronisiert die Uhrzeit zwischen Roboter und Server. Der Client schickt diese Anfrage nur wenn seine Uhrzeit offensichtlich falsch ist (das heißt in 1970).

Befehls-ID: 21 (OP_TIMESYNC)
Flags:
Typ: Anfrage von Client an Server

Parameter:
0-3 'TIME' (Dummy-Text)

Nach Erhalt der Anfrage übermittelt der Server die aktuelle PC-Zeit an den Client mit folgendem Paket:

Befehls-ID: 21 (OP_TIMESYNC)
Flags: FLAG_IS_ANSWER
Typ: Antwort von Server auf Anfrage

Parameter:
0-3 Zeit in Sekunden seit 1.1.1970 0:00

Die Zeitangabe erfolgt dabei in Rechnerzeit, das heißt die Zeitzonenanpassung ist enthalten.

Get available behaviors

Übermittelt die Liste der verfügbaren Behaviors (ID und Name) an den Server. ID 0 ist stets "Kein Behavior" und kann optional vom Client mitgeschickt werden.

Befehls-ID: 22 (OP_GETBEHAVIORS)
Flags:
Typ: Anfrage von Server an Client

Parameter: none

Befehls-ID: 22 (OP_GETBEHAVIORS)
Flags: FLAG_IS_ANSWER
Typ: Antwort von Client auf Anfrage

Parameter: Sequence of
- uint32_t - id of behavior (32 bit)
- char[..] - name of behavior (null-terminated string)

Get motor offsets

Übermittelt die aktuellen Offset-Werte der Motoren an den Server.

Befehls-ID: 23 (OP_GETMOTOROFFSETS)
Flags:
Typ: Anfrage von Server an Client

Parameter: none

Befehls-ID: 23 (OP_GETMOTOROFFSETS)
Flags: FLAG_IS_ANSWER
Typ: Antwort von Client auf Anfrage

Parameter: Sequence of
- uint8_t - ID of motor
- int16_t - current offset (-1023..+1023)

Set motor offsets

Setzt Motor-Offsets.

Befehls-ID: 24 (OP_SETMOTOROFFSETS)

Flags:

Typ: Befehl von Server an Client

Parameter: none

Parameter: Sequence of

uint8_t - ID of motor

int16_t - offset for motor

6.2. Einrichtung Eclipse

FUremote basiert auf der Eclipse Rich Client Platform und basiert daher auf der Entwicklung mit der Eclipse-Umgebung.

Ursprünglich auf Version 3.4 basierend, wurde FUremote am 04.08.2009 auf Version 3.5 der Eclipse-Plattform aktualisiert.

Zweck der Einrichtung:

1. Anpassungen des Cores und existierender Plug-ins
2. Entwicklung neuer Plug-ins
3. Erstellung einer Binary-Version

Die Einrichtung besteht auf den folgenden Teilen:

1. Installation Eclipse
   Ermöglicht die grundsätzliche Entwicklung mit FUremote.
2. Importieren der FUremote-Plug-ins
3. Einrichtung Target Platform & Installation SWT DeltaPack
   Verhindert unnötige Plug-in-Abhängigkeiten
4. Einrichtung Run Configuration
   Erlaubt die Wahl der mitzustartenden Plug-ins

Installation Eclipse

Die Eclipse-Entwicklungsumgebung steht in mehreren Versionen zum Download. Dabei handelt es sich immer um eine Grundversion und zusätzlichen Plug-ins, die eine bestimmte Funktionalität bereitstellen.

Die für diesen Zweck minimale Version heißt «Eclipse for RCP/Plug-in Developers» und kann unter http://www.eclipse.org/downloads/ bezogen werden.

Das Archiv muss entpackt werden und sollte eine ausführbare Datei im Stammverzeichnis des entpackten Verzeichnisses besitzen.

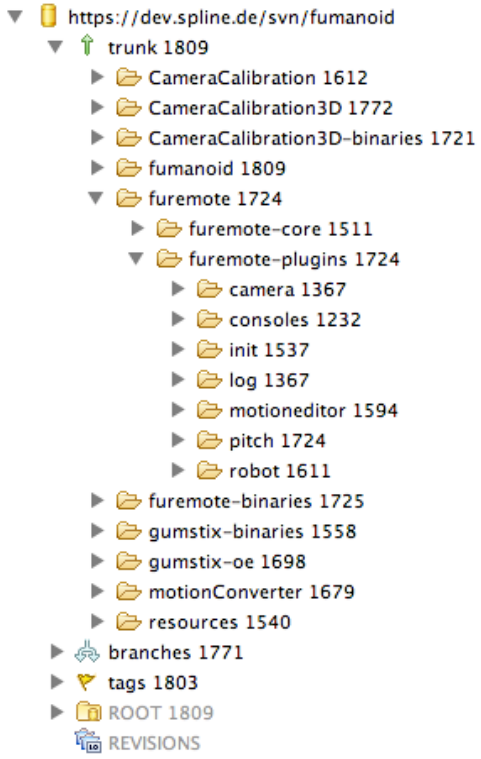
Importieren der FUremote-Plug-ins

Zum Importieren der FUremote-Plug-ins muss zunächst ein Subversion-Plug-in installiert werden.
Unter http://benjchristensen.com/2009/06/24/eclipse-galileo-3-5-and-subversion/ ist ein knappes Howto zur Installation des Subclipse-Plug-ins zu finden.

Das hinzuzufügende SVN Repository lautet: https://dev.spline.de/svn/fumanoid.

Auszuchecken sind:

- trunk/furemote/furemote-core
- trunk/furemote/furemote-Plug-ins/*
  Hinweis: Es können alle Plug-ins auf einmal heruntergeladen werden, indem sie markiert werden und im Kontextmenü «Check Out» gewählt wird.



FUremote sollten an dieser Stelle bereits lauffähig sein. Um dies zu testen, muss beim «de.fumanoid.furemote»-Projekt im Kontextmenü «Run as», «Eclipse Application» gewählt werden.

Es handelt sich hierbei um das Starten des FUremote Cores mit der Standardkonfiguration, das heißt es sind keine weiteren Plug-ins geladen worden.

Der nächste Abschnitt beschäftigt sich mit der Einrichtung der Target Platform. Erst im Anschluss sollte FUremote mit allen Plug-ins gestartet werden.

Einrichtung Target Platform & Installation SWT DeltaPack

Standardmäßig wird ein Eclipse Plug-in-Projekt gegen die aktuell verwendete Eclipse-Installation kompiliert.

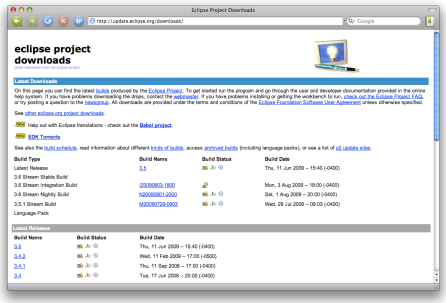
Durch die Verwendung einer separat eingerichteten Target Platform werden für die Kompilation nur die Plug-ins verwendet, die auch tatsächlich benötigt werden. Desweiteren werden durch diese Praxis Probleme vermieden, die durch die Verwendung unterschiedlicher Eclipse-Versionen auftreten würden.

Es ist daher von großer Wichtigkeit, dass alle FUmanoid-Mitglieder ihre Target Platform identisch einrichten.

Installation

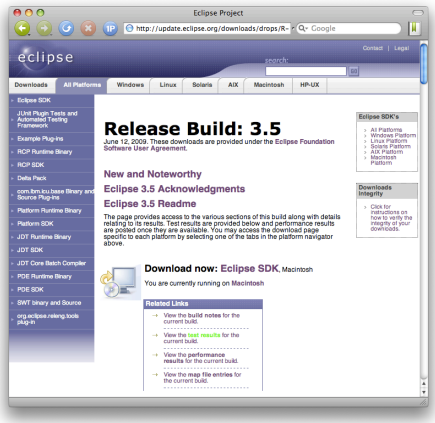
Seit Version 3.5 muss das «RCP SDK» nicht mehr separat heruntergeladen werden. Es enthält die notwenigsten Plug-ins zur Entwicklung einer eigenen RCP-Anwendung.

Allerdings fehlt noch das «DeltaPack», welches das plattformabhängige SDK in ein plattformunabhängiges umwandelt. In diesem Schritt wird das SDK u.a. um die fehlenden SWT-Implementierungen der gängigen Betriebssysteme ergänzt.



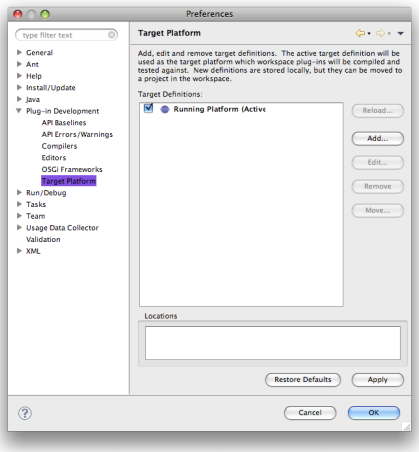
Das DeltaPack kann unter http://update.eclipse.org/downloads/ bezogen werden.

Nach Auswahl der gewünschten Version kann auf der Folgeseite im linken Navigationsbereich der Link «DeltaPack» ausgewählt werden.



Konfiguration

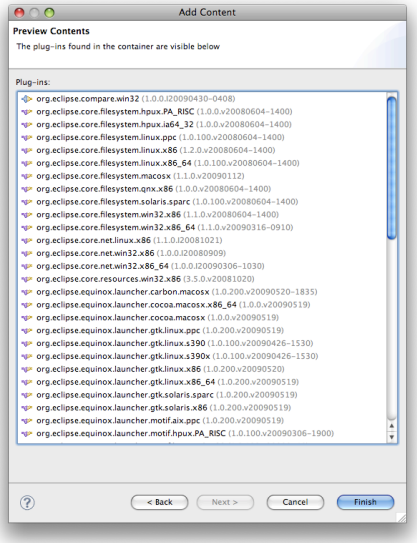
Im gestarteten Eclipse muss der Menüpunkt «Preferences» aufgerufen werden.
Unter «Plug-in Development», «Target Platform», «Add...» öffnet sich ein Assistent zur Einrichtung einer neuen Target Platform.

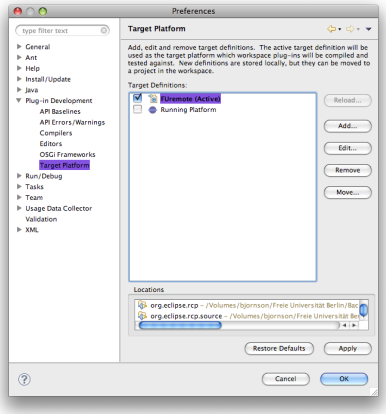


Im Assistenten ist das Template «Base RCP (with Source)»[vii] zu wählen.

Es fehlt noch das DeltaPack, das über «Add...», «Directory» hinzugefügt werden muss.

Der Assistent muss nun eine Liste gefundene Plug-ins - wie auf der Abbildung zu sehen - anzeigen.





Abschließend muss als Name «FUremote» angegeben, der Assistent geschlossen und die neue Target Platform aktiviert werden.

Evtl. fehlende Plug-ins können nun über diesen Weg der Target Platform hinzugefügt werden, ohne das es zu einer Vermischung von eigener Entwicklungsplattform und der Codebasis von FUremote kommt.

[vii] Der Unterschied zwischen den Optionen mit und ohne Quellen besteht darin, dass Eclipse während der Arbeit die in den Quellen eingebetteten Javadoc-Kommentare verwendet, um hilfreiche Toolstipps einzublenden. Weiterhin wird durch die Verwenden der Option mit Quellen der Plug-in-Deskriptor-Editor sehr mächtig. Von der Wahl der Option ohne Quellen wird dringend abgeraten.

Einrichtung Run Configuration

Basis der Run Configuration ist die im 2. Schritt erzeugte Run Configuration mit seinen Standardeinstellungen.

Im Kontextmenü des «de.fumanoid.furemote»-Projekts unter «Run As», «Run Configurations...» sollte bereits eine «Eclipse Application» mit dem Namen «de.fumanoid.furemote.product» existieren.

Diese Konfiguration sollte noch wie folgt abgeändert werden:

1. Aktivierung der Option «Clear», «workspace»
   Diese Option bewirkt, dass alle Einstellungen zur Fensteranordnung, Anwendungskonfiguration, etc. vor jedem Start gelöscht werden sollen.
   Diese Option ist äußerst wichtig, da sonst u.a. neue Views nicht zu sehen sind.

2. Im Reiter «Plug-ins» muss das Projekt «de.fumanoid.furemote» und schließlich der Button «Add Required Plug-ins» geklickt werden.
   Damit werden alle notwendigen Plug-ins der Run Configuration hinzugefügt.

3. Für die Entwicklung an FUremote sollten ebenfalls sämtliche Plug-ins aktiviert werden. Mit «Add Required Plug-ins» und «Validate Plug-ins» kann sichergestellt werden, dass keine Plug-ins fehlen.
   Sollten Plug-ins fehlen, müssen diese der Target Platform hinzugefügt und die Run Configuration erneut getestet werden.

6.3. Erstellung eines Plug-ins

Die Erstellung eines Plug-ins für die Eclipse Rich Client Platform ist ein fehlerträchtiger Prozess.

Dieser Abschnitt beschäftigt sich mit der beispielhaften Erstellung eines Plug-ins, welches die folgenden Eigenschaften aufweisen wird:

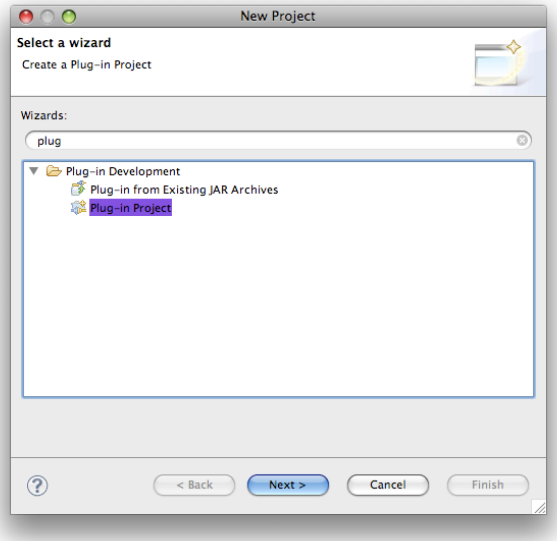
- Deklaration und Implementierung eines SayHello-Commands mit entsprechendem Handler
- Deklaration und Implementierung eines Views
- Deklaration und Implementierung einer Perspective
- Einbindung des Commands, des Views und der Perspective in die Menüstruktur von FUmanoid unter Einhaltung des Designs

Die folgende Erklärung setzt voraus, dass eine mit Anhang A kompatible Installation und Konfiguration von Eclipse und FUremote vorliegt.

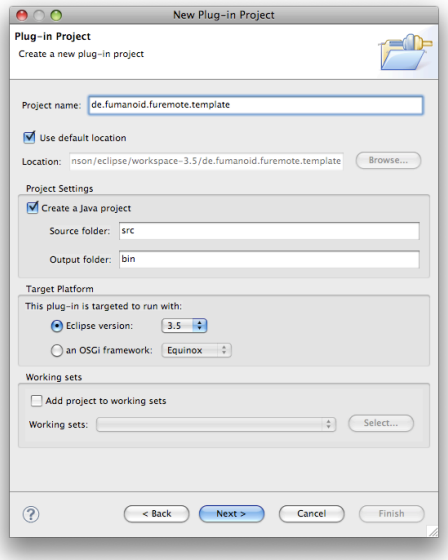
Die Erstellung des Plug-ins wird nur lückenhaft skizziert und soll das Verständnis zur Arbeitsweise des Eclipse RCP und von FUremote erleichtern. Das fertige Plug-in befindet sich im SVN unter dem Projektnamen «de.fumanoid.furemote.template».

Erstellung

Ein Plug-in wird mithilfe des «New Project Wizard» und dem Projekttyp «Plug-in Project» erstellt.

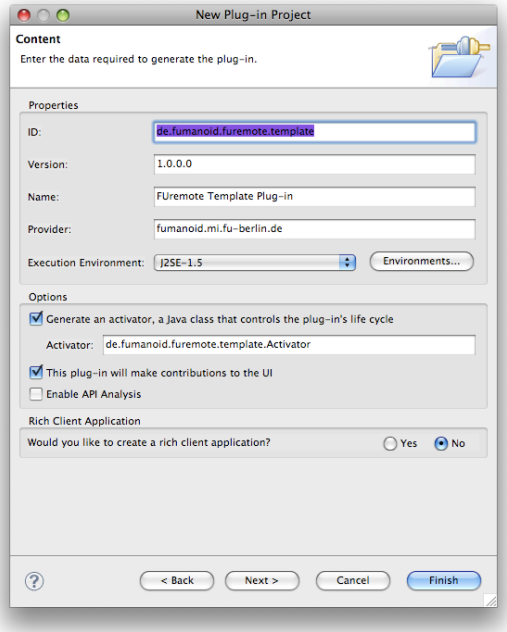


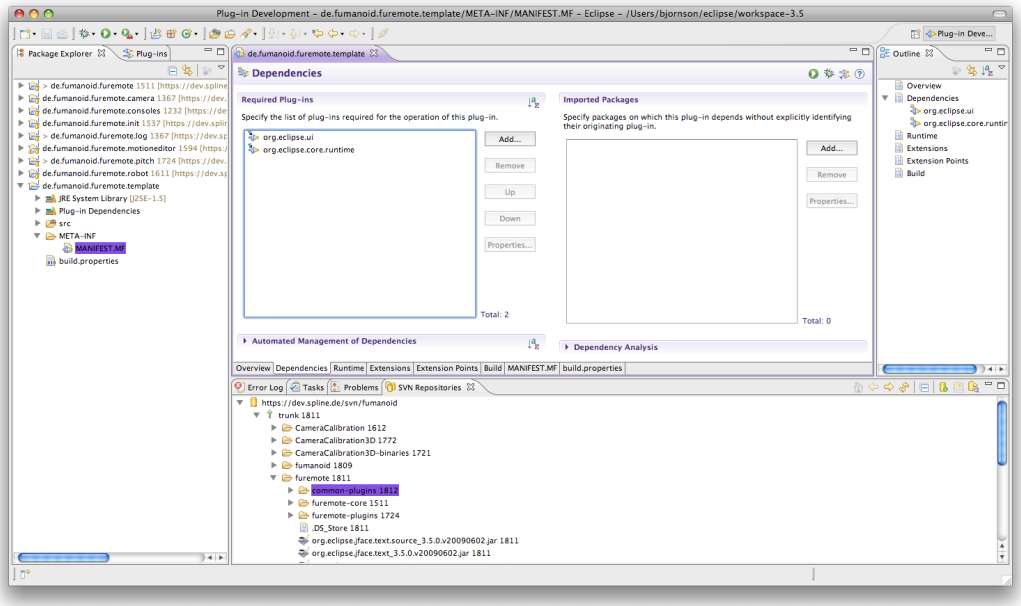
Die Plug-in-ID folgen dem Benennungsschema **de.fumanoid.furemote.NAME** und sollte auch als Projektname verwendet werden. Die übrigen Einstellungen können auf dieser Seite unverändert übernommen werden.



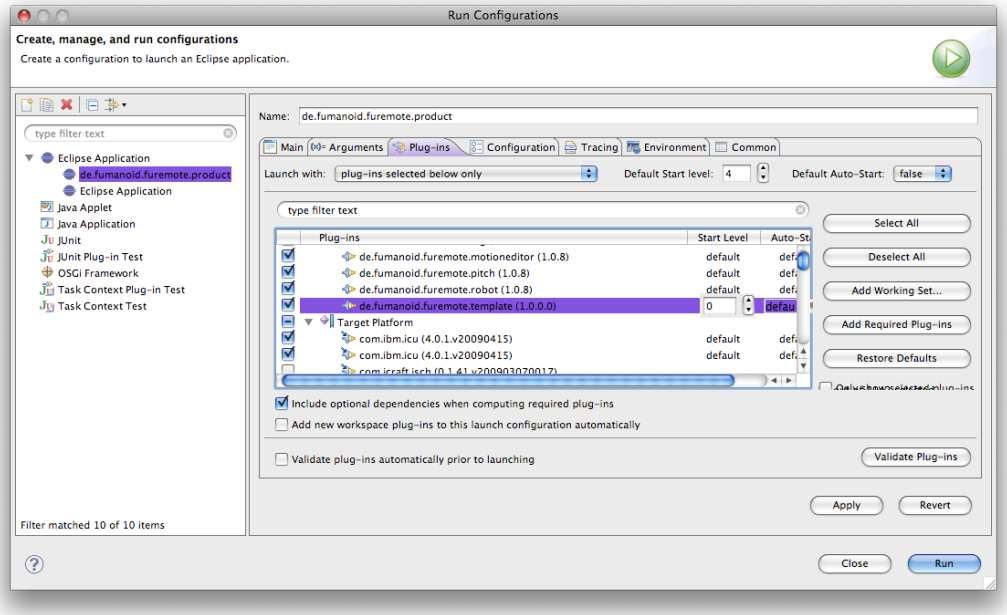
Auf der Folgeseite muss die mit dem Projektnamen identische ID engetragen werden. Der Plug-in-Name hat das Schema **FUremote NAME Plug-in**. Unter Provider ist der DNS-Name der Projektwebsite einzutragen.

Nachdem «Finish» betätit wurde, erstellt Eclipse ein minimales Plug-in, das praktisch keine Funktionalität besitzt. Damit das Plug-in mit FUremote inkorperieren, muss es Abhängigen zu FUremote deklarieren.





Wir werden den neuen Plug-in die Plug-ins «de.fumanoid.furemote» und «de.fumanoid.furemote.robot» hinzufügen.
Dazu muss in Project Explorer die MANIFEST.MF geöffnet und im Reiter «Dependencies» die beiden Plug-ins hinzugefügt werden.



Nun kann die Run Configuration von FUremote um das neu erstellte Plug-in ergänzt werden; allerdings ohne sichtbaren Effekt.

Damit der Anwender das Plug-in verwenden kann, muss es in das Menü eingebunden werden.
Zu diesem Zweck muss in den Reiter «Extension» gewechselt und als neue Extension «org.eclipse.ui.menus» gewählt werden.

Diese Extension erlaubt es, das Menü der RCP-Anwendung um eigene Menüs zu erweitern.

Im Kontextmenü der eben erstellten Extension muss «menuContribution» gewählt werden.

Als «locationURI» wird der Locator «menu:file? after=FileOperationsStart» verwendet, der sich wiefolgt erklärt:
«menu» ist der Zugriff auf die Menüleiste, wobei «toolbar» beispielsweise zur Erweiterung des Kontextmenüs geeignet ist. «file» ist der Bezeichner des in praktisch jeder Anwendung existierenden Dateimenüs. «after» gibt an, ob der neu hinzuzufügende Menüknoten vor oder nach das Element «FileOperationsStart» eingefügt werden soll. «FileOperationsStart» ist eine im Plug-in «de.fumanoid.furemote» definierter Menüeintrag.

Unterhalb der menuContribution muss nun ein command-Knoten mit der Bezeichnung **de.fumanoid.furemote.NAME.commands.SayHello** erstellt werden.[viii]

Das fehlende Command und der dazugehörige Handler werden analog definiert.

Durch Wechsel in den Reiter «Plug-in.xml» kann man sehen, welchen Code der grafische XML-Editor generiert.

Die Datei Plug-in.xml sollte nun wie folgt aussehen.

[viii] Die Erläuterung der Funktion von Commands, Handlern, usw. sind nicht Gegenstand dieses Abschnitts und würden den Umfang sprengen.

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.4"?>
<plugin>
      <extension point="org.eclipse.ui.menus">
      <menuContribution
      locationURI="menu:file?after=FileOperationsStart">
      <command commandId=
      "de.fumanoid.furemote.template.commands.SayHello"
      style="push">
      </command>
      </menuContribution>
      </extension>
      <extension point="org.eclipse.ui.commands">
      <command
      id="de.fumanoid.furemote.template.commands.SayHello"
      name="Say Hello">
      </command>
      </extension>
      <extension point="org.eclipse.ui.handlers">
      <handler
      class="de.fumanoid.furemote.template.handlers.SayHello"
      commandId="de.fumanoid.furemote.template.commands.SayHello">
      </handler>
      </extension>
</plugin>
```

Listing Plug-in.xml

Der Handler des Commands SayHello öffnet lediglich eine MessageBox:

```
package de.fumanoid.furemote.template.handlers;
```

```
import org.eclipse.core.commands.AbstractHandler;
import org.eclipse.core.commands.ExecutionEvent;
import org.eclipse.core.commands.ExecutionException;
import org.eclipse.swt.SWT;
import org.eclipse.swt.widgets.MessageBox;
import org.eclipse.swt.widgets.Shell;
import org.eclipse.ui.handlers.HandlerUtil;

public class SayHello extends AbstractHandler {
     public Object execute(ExecutionEvent event)
          throws ExecutionException {
          Shell shell = HandlerUtil
               .getActiveWorkbenchWindow(event).getShell();

          MessageBox messageBox = new MessageBox(shell,
               SWT.OK | SWT.ICON_INFORMATION);
          messageBox.setText("Hello Saying");
          messageBox.setMessage("Hope this wasn't that complicated...");
          messageBox.open();
          return null;
     }
}
```

SayHello.java

Für ein minimales Plug-in im praxistauglichen Umfang fehlen noch:

1. Deklaration und Implementierung eines Views
2. Optional Definition einer Perspective
3. Kenntnisse über die wichtigsten Schnittstellen zu FUremote (RobotListener, NetworkListener, GameControl)

Das fertige Beispiel-Plug-in befindet in der aktuellen Version im SVN unter https://dev.spline.de/svn/fumanoid/trunk/furemote/furemote-plugins/template.

6.4. Erstellung einer Binary Version

Dieser Abschnitt setzt voraus, dass die Eclipse Entwicklungsumgebung entsprechend der Beschreibung 6.2. Einrichtung Eclipse eingerichtet wurde. Insbesondere ist es notwendig, dass das RCP Delta Pack installiert wurde, da sonst nur die zum aktuell verwendeten Betriebssystem passende Version generiert werden kann.

Was ist eine Binary Version?

Java generiert grundsätzlich plattformunabhängigen Bytecode, der auf jeder Architektur und jedem Betriebssystem läuft, für welches eine Java Virtual Machine existiert.

FUremote verwendet für die visuelle Darstellung SWT, das auf native Funktionalitäten des jeweiligen Betriebssystems zugreift. Das erzeugt zum einen ein vertrautes Erscheinungsbild, wie man es als Anwender erwarten würde, zum anderen aber entsteht daraus eine weitere Systemvoraussetzung für FUremote, denn nun wird neben der JavaVM auch eine SWT-Implementierung benötigt.

Diese Einschränkung ist glücklicherweise eher theoretischer Natur, da es für alle großen Betriebssysteme Implementierung gibt, die im Delta Pack zusammengefasst sind.

Damit FUremote nicht das ganze Delta Pack enthalten muss, wird ein so genanntes *Product* in Eclipse angelegt. Dessen Export-Assistent generiert für jede gewünschte Zielplattform ein Verzeichnis und kopiert jeweils nur die tatsächlich benötigten Plug-ins. Außerdem wird die exportierte Anwendung um einen *Launcher* ergänzt, der ebenfalls abhängig von der Zielplattform ist und es erlaubt, FUremote - so wie jedes andere Programm auch - zu starten.

Die exportierte Anwendung ist auf das jeweilige Betriebssystem zugeschnitten und verhält wie eine typische Anwendung. Als Anwender erfährt man an keiner Stelle, dass FUremote auf Java basiert, sondern es sich um eine native Applikation handeln muss. Aus diesem Grund wird eine solches Produkt als Binary Version bezeichnet.

Der Export-Vorgang ist geradlinig: Zunächst muss die bereits konfigurierte Produkt-Definition `FUremote.product` im Plug-in `de.fumanoid.furemote` mit Eclipse geöffnet werden. Im standardmäßig geöffneten Reiter "Overview" muss der Link "Eclipse Product export wizard" gewählt werden.

Im darauf folgenden Export-Assistenten ist sicherzustellen, dass die Option "Synchronize before exporting" deaktiviert ist. Sie führt dazu, dass in der `plugin.xml` gemachten Anpassungen nicht überschrieben werden.

Außerdem muss die Option "Export for multiple platforms" aktiviert sein. Nach Bestätigung durch "Finish" ist sie es, die das Fenster "Cross-platform export" öffnet. In diesem Fenster sollten die Betriebssysteme der Wahl getroffen werden.

Üblicherweise werden Versionen für die folgenden Plattformen bereitgestellt:

- Linux (gtk/x86)
- Mac OS X (cocoa/x86)
- Windows (win32/x86)

Eine aktuelle Version ist nach ihrer Prüfung auf Funktionstüchtigkeit im SVN unter https://dev.spline.de/fumanoid/trunk/furemote-binaries als ZIP-Archiv bereitzustellen.