Programmierung mit LEGO MINDSTORMS NXT

III. Objektorientierte Programmierung mit NXT

Steffen Hemer
Arbeitsgruppe Robotersysteme
Fachbereich Informatik
Technische Universität Kaiserslautern

30.09.2010

RLAB
THE ROBOTICS RESEARCH LAB

Inhalt

- Motivation
- Objektorientierte Programmierung
  - Java
  - LeJOS
- Modellierung und Dokumentation
- Anwendungsbeispiel

Motivation

- Anwendungsbeispiel:
  Navigation in unbekannter Umgebung

- Teilaufgaben
  - Roboterkinematik
  - Wandfolgen und Hindernisvermeidung
  - Lokalisierung
  - Kartenerstellung
  - Navigation mit Karten

Motivation

- Warum nicht alles mit NXT-G programmieren?
  - Roboterkinematik:
    → in NXT-G machbar
  - Hindernisausweichen:
    → in NXT-G machbar
  - Lokalisation:
    → benötigt trigonometrische Funktionen
  - Kartierung:
    → benötigt komplexe Datentypen (Felder)
  - Navigation:
    → benötigt mathematische Funktionen
    und komplexe Datentypen

Motivation

- Vorteile textueller Programmiersprachen gegenüber NXT-G
  - Volle Mächtigkeit
  - Intuitive arithmetische Operationen
  - Voller mathematischer Funktionsumfang
  - Echte parallele Programmstrukturen möglich
  - Handhabung von Variablen einfach
  - Objektorientierte Programmierung
  - Rekursionen sind möglich
  - Komplexe Datentypen
  - Programme skalieren besser
  - Standardisierte Dokumentation
  - Aussagekräftige Compilermeldungen

Objektorientierte Programmierung

- Wichtige Eigenschaften
  - Klassen als Bauplan für Objekte
  - Vererbung
  - Kapselung von Informationen
  - reale Vorbilder
  - komplexe Datenstrukturen (Felder, Listen, etc)
- Beispiele: Java, C++, C#
  (nahezu alle modernen Programmiersprachen)
- Vereinfacht die Umsetzung von der Modellierung zum Programm
- Standardisierte Modellierung mit Unified Modelling Language (UML)

Java

- Objektorientierte Sprache
- 1995 erschienen
- Plattformunabhängig
- Frei verfügbar
- Stand der Forschung, daher optimale Vorbereitung auf Studium oder Ausbildung
- Funktionsweise:
  1. Erstellung von Quelltext in Java
  2. Übersetzung in plattformunabhängigen Bytecode
  3. Interpretierung durch plattformabhängige Java Virtual Machine

LeJOS

- Lego-Java-Operating-System
- Java-Virtual-Machine-Portierung für LEGO MINDSTORMS
- Programmierung am PC mit normaler Java-Syntax
- Ausführen des Bytecodes auf dem LEGO-Baustein
- Zwei Versionen
  - LeJOS NXJ (für LEGO MINDSTORMS NXT)
  - LeJOS RCX (für die Vorgängerversion RCX)

LeJOS

- NXT-Firmware
  - Java-Virtual-Machine (JVM)
  - Verwaltungsmenü auf dem NXT
- Sammlung an Java-Bibliotheken
  - Zugriff auf Motoren und Sensoren
  - Höhere Ebenen und Zusatzfunktionen
- Diverse Hilfsprogramme
  - Kompilieren
  - Hochladen
  - Firmwareersetzung
  - Debugkonsole
  - Fernsteuerung

Inhalt Java

- Klassen → Wiederverwendbarkeit, Kapselung
- Variablen
  - Datentypen → Starke Typisierung
  - Felder → Komplexe Datentypen
  - Operatoren
- Methoden → Kapselung
- Kontrollstrukturen
  - Schleifen
  - Verzweigungen → Bessere Handhabung
- Bibliotheken → Mathematische Funktionen

Klassen in Java

- Definiton Klassenkopf
  - Modifikator (`public, abstract, final`)
  - Schlüsselwort `class`
  - Klassenname
  - ggf. weitere Schlüsselwörter `implements, extends` (Vererbung)
- Klassenrumpf
  - innerhalb { } Definition aller Methoden und Eigenschaften

```
1 // Kopf der Klasse
2 public class MyClass {
3   // Klassenrumpf
4   ...
5   // Konstruktor
6   public MyClass() { ... }
7 }
```


Klassen in Java

- Konstruktor
  - erzeugt Objekt der Klasse in Verbindung mit `new`
  - Name wie Klasse
  - impliziter Rückgabewert (Objekt der Klasse)
  - ggf. Parameter
- Klasse ist kleinste ablauffähige Einheit in Java
  - Vorhandensein der Methode

```
public static void main(String[] args){...}
```

  - Nur genau eine Klasse mit dieser Methode
  - Entspricht „Einstiegspunkt“ in NXT-G

Variablen

- Ebenfalls Modifikator für Sichtbarkeit
- Schlüsselwort `static` für Klassenvariablen
- Referenzierung mit Klassenname.Variablenname (nicht klassenintern, hier ggf. mit `this`.Variablenname)
- Global: Definition in Klassenrumpf
- Lokal: Definition innerhalb Methodenrumpf

```
1 //oeffentliche Integer-Klassenvariable
2 public static int aNumber;
3 //nicht-oeffentliche Gleitkommavariable
4 private float anotherNumber;
5 //Zeichenkettenvariable ohne Angabe der Sichtbarkeit
6 String aWord;
```


Datentypen

- Klasse = komplexer Datentyp
- Einfache Datentypen:

Typ	Größe	Wertebereich	Standardwert
`boolean`	1 bit	true, false	`false`
`char`	16 bit	Unicode-Zeichen 'a'...'z','A'...'Z', usw.	`u0000`
`byte`	8 bit	-128 ... 127	0
`short`	16 bit	-32.768 ... 32.767	0
`int`	32 bit	-2.147.483.648 ... 2.147.483.647	0
`long`	64 bit	-9.223.372.036.854.775.808 ... 9.223.372.036.854.775.807	0
`float`	32 bit	(Fließkommazahl)	`0.0`
`double`	64 bit	(Fließkommazahl)	`0.0`

- Ausnahme: String = Klasse, allerdings vereinfachte Zuweisung `String str = "Test";`
- Wrapper-Klassen bieten z.B. Umwandlungfunktionalität

Felder

- Zusammenfassung mehrerer Daten des selben Typs
- Elemente beginnend bei 0 durchnummeriert
- Deklaration: Datentyp gefolgt von eckigen Klammern
- Feste Längenangabe bei Konstruktion
- Zugriff über Angabe der Position in eckigen Klammern

```
1  // Deklaration
2  int[] intArray;
1  // Anlegen eines Feldes fuer neun Integerwerte
2  intArray = new int[9];
1  // Zuweisung eines Zahlenwertes
2  intArray[3] = 123;
3  int x = intArray[5];
```

- Dynamische Alternative z.B. ArrayList

Operatoren

- Einfache Operationen auf elementaren Datentypen
- Grundrechenarten: +, -, *, /, %(Modulo)
- Infixoperatoren
  - `v=2+3; v=v*5;` v hat den Wert 25
  - `color="brown"; color=color+"-white"` Stringkonkatenation
- Präfixoperatoren
  - `v=-v;` Negation
  - `++v` bzw `--v` erhöht bzw verringert den Wert von v um 1
- Postfixoperatoren
  - `v++` bzw `v--` erhöht bzw verringert den Wert von v um 1, allerdings bleibt Wert zunächst erhalten (bei Zuweisung)

Operatoren

- Vergleichsoperatoren
  - ==, !=, <, >, <=, >=
  - Ergebnis boolescher Wert
- Logische Operatoren
  - & (AND), | (OR), ^ (XOR), ! (Negation)
  - Kombination mit Vergleichsoperatoren
  - Doppelte Operatoren: Veränderung der Auswertungsreihenfolge

→ Vorteile gegenüber NXT-G
  - Einfache, natürliche Schreibweise (mathematische Formel)
  - Bessere Übersicht durch Kombination mehrere Operatoren

Methoden

- Aufbau vergleichbar mit Variablendeklaration
- `void` signalisiert Methode ohne Rückgabewert
- Beliebige Parameter (in Klammern)
- Mehrfache Definition mit unterschiedlichen Parametern möglich (Überladen)
- Methodenrumpf enthält Funktionalität
- Rückgabewert durch `return`

```
1  //Methode ohne Rueckgabewert mit 2 Parametern
2  public void bark (int times, float volume){
3    //Methodenrumpf
4  }
5  //Methode mit Gleitkomma-Rueckgabewert ohne Parameter
6  public float eat (){
7    //Methodenrumpf
8  }
```

Kontrollstrukturen (Schleifen)

- for-Schleife
  - Zählschleife
  - Zählvariable nutzbar innerhalb Schleifenrumpf

```
for (int i=0 ; i <= 10; ++i){
  //zu wiederholende Anweisungen
}
```

- while-Schleife
  - Beliebige Bedingung
  - Bedingung am Anfang geprüft

```
int i=0;
while (i <= 10){
  //zu wiederholende Anweisungen
  ++i;
}
```

- do-while-Schleife
  - Bedingung am Ende geprüft
  - entspricht Schleife in NXT-G

```
int i=0;
do{
  //zu wiederholende Anweisungen
  ++i;
} while (i <= 10)
```


Kontrollstrukturen (Verzweigungen)

- if-else
  - prüft Bedingung
  - verzweigt in if-Zweig, falls wahr
  - verzw. in else-Zweig andernfalls
  - auch ohne else nutzbar

```
if (aDog.weight >= 10.0){
    str = "grosser Hund";
}else{
    str = "kleiner Hund";
}
```

- switch-case
  - Prüft Variable in mehreren Gleichbedingungen
  - Nur `int` und `char`
  - falls kein `break`, Anweisung aus nächstem Fall
  - falls keine erfüllt, default-Fall

```
switch (aDog.age){
  case 0:
  case 1:  str = "Welpe"; break;
  case 2:
  case 3:
  case 4:  str = "Junghund"; break;

  default: str = "alter Hund";  break;
}
```

Schnittstellen und Vererbung

- Schnittstelle (`interface`)
  - Klasse, deren Methodendeklarationen lediglich aus Methodenkopf besteht
  - Klasse kann mehrere Schnittstellen implementieren (`implements`)
  - keine Objekte einer Schnittstelle direkt instanziierbar
- Vererbung
  - Klasse kann nur von genau einer Oberklasse erben (`extends`)
  - Alle Klasse erben automatisch von Klasse `Object`
  - Methoden und Eigenschaften referenzierbar mit `super`

Bibliotheken

- Java (ebenfalls in LeJOS vorhanden)
  - java.lang.Math - enthält diverse mathematische Funktionen wie Wurzel, Sinus, min/max, Betrag, u.ä.
  - java.lang.System - enthält Systemfunktionen wie `System.exit(0), System.out.println(data), System.in.read(byte[])`
- LeJOS
  - Paket lejos.nxt – enthält Klassen zur Ansteuerung der Basiskomponenten (Motoren, Sensoren, Buttons, Anzeige, etc.)
  - Paket lejos.nxt.addon – Klassen für Zusatzsensoren wie Kompass etc.
  - Weitere Pakete mit Klassen für höhere Ebenen

Bibliotheken

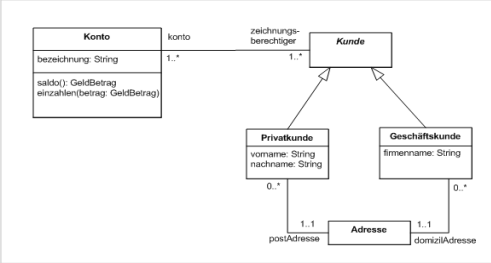
- Beispiel: Motor (`lejos.nxt.Motor`)
  - Einzelmotoransteuerung
  - 3 Instanzen vorhanden (`Motor.A, Motor.B, Motor.C`)
  - `void setSpeed(int speed)` setzt Geschwindigkeit in Grad/Sekunde, max 900, es folgt keine Bewegung!
  - `void forward()` Vorwärtsbewegung, ebenso Methoden für Rückwärts, Richtungsumkehr, Stoppen und Auslaufen
  - `int getTachoCount()` gibt gedrehten Winkel zurück, zurückzusetzen auf 0 mit `void resetTachoCount()`
  - `void rotate(int angle)` dreht um angegebenen Winkel, auch mit Parameter `boolean immediateReturn`: entspricht „Warten auf Abschluss“

Bibliotheken

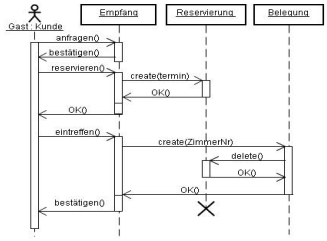
- Beispiel: Kompasssensor (lejos.nxt.addon.CompassSensor)
  - Konstruktor hat lediglich Parameter für Sensorport
  - Auch Sensorports sind bereits vier mal instanziiert (z.B. `SensorPort.S1`)
  - Auslesen der Richtung in Grad mit `float getDegrees()`, 0 entspricht Norden, Drehrichtung Uhrzeigersinn, Auflösung 0.1
  - `float getCartesianZero()`: kartesische Kompassorientierung, unabhängig von Norden auf 0 setzbar
  - Kalibrierung mit `void startCalibration()`, Drehung, stoppen mit `void startCalibration()`

Modellierung

- Unified Modelling Language
  - verschiedene Diagrammtypen zur Modellierung unterschiedlicher Problemsichten
  - jeweils 7 Typen von Struktur- und Verhaltensdiagramme



Bsp. Klassendiagramm



Bsp. Sequenzdiagramm

Möglichkeit zur Dokumentation - JavaDoc

- Automatische Erstellung von html-Dokumentationsseiten direkt aus Java-Quelltextdateien
- Spezielle Syntax erlaubt das Dokumentieren im Quelltext
- Dokumentation der Eigenschaften und Funktionsweisen von Klassen und deren Methoden
- Voranstellung eines /** ...*/ Blocks vor Klassen-/Methodendeklaration
- Spezielle JavaDoc-Tags (beginnend mit @) enthalten Metadaten mit dokumentativem Charakter (für Klasse z.B. @author name, für Methoden z.B. @param name beschreibung oder auch @return beschreibung)
- Wahl von aussagekräftigen Variablen- und Methodennamen

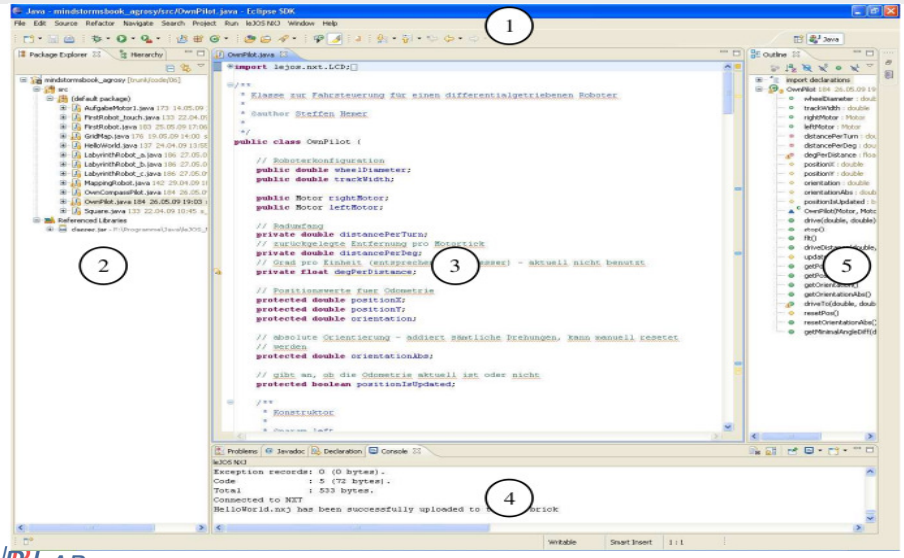
Entwicklungsumgebung Eclipse

- Verwendung von Eclipse, da Plugin für LeJOS vorhanden (mittlerweile auch für Netbeans)
- quelloffen
- 2001 aus IBM-Vorgänger entstanden
- ursprünglich v.a. für Java
- selbst in Java entwickelt
- erweiterbar mit Plugins
- aktuelle Version 3.6.0 (Helios)

Vorteile einer IDE

- IDE = Integrierte Entwicklungsumgebung
- Nutzung einer IDE bringt viele Vorteile im Vergleich zu Programmierung mit einfachem Editor und Kommandozeile
- Bietet Quelltexteditor mit Hervorhebung, Code-Formatierung, Einblendung von Informationen wie JavaDocs, Vorschläge zur Fehlerbehebung, Autovervollständigung zu Methoden und Eigenschaften v. Objekten
- Übersicht über die Struktur eines Projektes
- Integration von Compiler, Linker und vereinfachter Aufruf dieser durch graphische Symbole
- Integration weiterer Hilfsprogramme (Versionsverwaltung, speziell für NXT: Programmupload)

Aufbau Eclipse



Aufbau Eclipse

1) Werkzeugleiste: Erstellung von neuen Klassen etc, Kompilieren
2) Package Explorer: Übersicht über die Klassen, die Hierarchie und verwendete Bibliotheken eines bzw mehrere Projekte
3) Quelltexteditor: Bearbeitung des Quelltextes mit diversen Hilfestellungen wie Syntaxhervorhebung, Vorschläge zur Autovervollständigung, Einblendung v. Möglichkeiten zur Fehlerbehebung und JavaDocs, Quelltextformattierung
4) Konsole: Anzeige der Programmausgabe, Anzeige von erkannten Problemen im Quelltext, Anzeigen von JavaDoc-Ausgabe
5) Outline: kompakte Übersicht über die Eigenschaften der aktuell bearbeitete Klasse

Anwendungsbeispiel Navigation

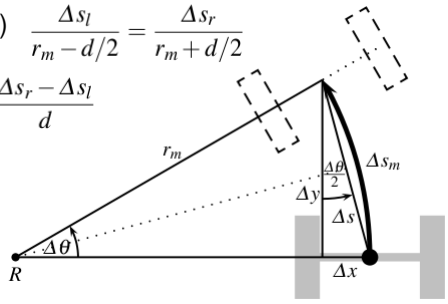
- Teilaufgabe Lokalisierung (Odometrie)
  - von griech. hodós, „Weg“ und métron, „Maß“
  - Relative Positionsmessung über Integration der Wegstrecke
  - Festlegung einer Startpose (x_0,y_0,θ_0) (meist = 0,0,0)
  - Fortlaufende Addition der Änderungen
  - Genauigkeit je kürzer die Zeitschritte
  - Ungenauigkeit durch Schlupf, unterschiedliche Reibung nicht vorhersagbar

Systemanalyse

- Mathematische Analyse des vereinfachtes Modell
  - Wegstreckenänderung eines Rades errechnet aus Radumfang und Raddrehung $\Delta s_l = T_l \cdot U$ $\Delta s_r = T_r \cdot U$
  - Wegstrecke kinematisches Zentrum $\Delta s_m = (\Delta s_l + \Delta s_r)/2$
  - Strahlensatz (nach r_m auflösen) $\frac{\Delta s_l}{r_m - d/2} = \frac{\Delta s_r}{r_m + d/2}$
  - Winkeländerung $\Delta\theta = \frac{\Delta s_m}{r_m} = \frac{\Delta s_r - \Delta s_l}{d}$
  - Positionsänderungen

$$\Delta x = \Delta s \cdot cos\left(\frac{\Delta\theta}{2} + \theta_{i-1}\right)$$

$$\Delta y = \Delta s \cdot sin\left(\frac{\Delta\theta}{2} + \theta_{i-1}\right)$$



Systementwurf

- Motorencoder auslesen
- Motorencoder zurücksetzen für nächsten Schritt
- Abweichung in aktuellem Schritt berechnen
- Pose aktualisieren

- Einbettung in Fahrbefehle

Codierung

```
217   /**
218    * Odometrieberechnung
219    */
220   protected void updatePos() {
221     // Strecke rechtes Rad (delta_s_r)
222     double _distanceRightWheel = rightMotor.getTachoCount()
223         * distancePerDeg;
224     // Strecke linkes Rad (delta_s_l)
225     double _distanceLeftWheel = leftMotor.getTachoCount() *
            distancePerDeg;
226
227     // Zuruecksetzen der Motordrehzahlen
228     rightMotor.resetTachoCount();
229     leftMotor.resetTachoCount();
230
231     // Strecke Robotermittelpunkt (delta_s_m)
232     double _distanceMiddle = (_distanceRightWheel +
            _distanceLeftWheel) / 2;
233
234     double _radius = 0;
235     double _deltaTheta = 0;
236     double _deltaX = 0;
237     double _deltaY = 0;
238     double _deltaS = 0;
239
240     if (_distanceRightWheel != _distanceLeftWheel) {
241       _radius = (_distanceMiddle * trackWidth)
242           / (_distanceRightWheel - _distanceLeftWheel);
243       if (_radius != 0) {
244         // 1. Fall: Kurvenfahrt
245         _deltaTheta = (_distanceMiddle / _radius); // rad
246         _deltaS = 2 * _radius * Math.sin(_deltaTheta / 2);
247         _deltaX = _deltaS * Math.cos((_deltaTheta / 2) + orientation
              );
248         _deltaY = _deltaS * Math.sin((_deltaTheta / 2) + orientation
              );
```

Testen

- Funktioniert das Implementierte ?
- Wie genau ist die Lokalisation ?
- Ausreichend für Navigation ?
  - Nein!

Verbesserung

- Verwendung von Kompasssensor zusätzlich zu Odometrie
  - Anpassung der Orientierung
  - Erweitern der bisherigen Ergebnisse (Vererbung!)
  - Mittelwertbildung um Ungenauigkeiten des Kompasssensors zu reduzieren
  - Sinnvoll: dynamische Datenstruktur Warteschlange (Queue)
  - ggf. Umrechnung von Sensorkoordinaten in Roboterkoordinaten

Verweise

- Eclipse
  `http://www.eclipse.org/`
- LeJOS Projekt: Homepage
  `http://lejos.sourceforge.net`