Lejos - Einfach gemacht

Dieses Buch enthält Grundlangen zur richtigen Benutzung von LejosKit

Inhaltsangabe

Inhaltsangabe	2
Vorwort	3
Adressanten	3
Quellcode und Javadocs	3
Geschichte von LejosKit	3
Vorbereitung	4
Download	4
Installation	4
Classpath hinzufügen unter nxjlink	*4*
Installieren unter Eclipse.	*4*
Erstes Programm	6
Hallo Welt Programm ohne LejosKit	6
Hallo Welt in Parts mit LejosKit	7
Hallo Welt mit Dialogen	9
Programme	11
Program - die Mutterklasse	11
MoveControllerProgram	11
Allgemein	*11*
Eigenen MoveController verwenden	*12*

Vorwort

Adressanten

Ich habe das Buch als Nachschlagewerk für LejosKit geschrieben. Das erste Kapitel enthält einen kurze Anleitung, wie man LejosKit benutzen sollte. Anschließend werde ich mehr Details zu dein einzelnen Klassen geben, wie man sich richtig implementiert. Das schließt Grundkenntnisse in Java sowie Polymorphie ein. Ohne die lassen sich die Kernteile von LejosKit nicht nachvollziehen, da LejosKit auf Javas wunderbare Polymorphie angewiesen ist.
Es werden außerdem Grundkenntnisse im Englisch vorausgesetzt, da die Javadocs komplett in englisch sind.

Quellcode und Javadocs

Der Quellcode von LejosKit wird auf Bitbucket.org gehostet - ein kostenloser Git Hoster für bis zu 5 Teammitgliedern. Die Git Repository ist hier. Dort lassen sich aktuelle Downloads finden (wie z.B. das LejosKit.jar) sowie ein ZIP-Datei mit den Javadocs. Außerdem finden können Sie dort den kompletten Quellcode sehen und die Versionsgeschichte von LejosKit.
Die Javadocs lassen sind hier auch noch mal zum Online nachlesen. Es ist zu empfehlen, die parallel zu lesen, da diese ergänzende Informationen enthalten.

Geschichte von LejosKit

LejosKit wurde ursprünglich entwickelt, um es einfacher zu machen, ein Lejos Roboter zu entwickelt. Ich habe angefangen mit Klassen. Dabei wurde die Klassen für Lejos immer größer, so das es notwendig wurde, die Projekt zu trennen - und so entstand LejosKit. Zur Zeiten lassen noch Teile im Quellcode darauf schließen. Diese werden allerdings in den nächsten Versionen vollkommen gelöscht sein.

Vorbereitung

Download

Lejos kann von der Offiziellen Website heruntergeladen werden: http://www.lejos.org.
Bitte laden Sie Version 1.9 oder spät herunter für ihr Betriebssystem runter.
LejosKit ist separat und kann unter (LEJOS_KIT_DOWNLOAD_LINK; TODO) heruntergeladen werden.

Installation

Bitte entnehmen Sie Hinweise zur Installation von Lejos von den Installationsdokumenten von Webseite.
Nachdem Sie LejosKit heruntergeladen haben, fügen sie bitte die Jar-Datei (LejosKit.jar) in den Classpath der von nxjlink hinzu.

Classpath hinzufügen unter nxjlink

Wenn Sie das Ihr Programm mit nxjc kompilieren benutzen sie bitte folgende Befehl:

```
$ nxjc -classpath lib/LejosKit.jar src/MyRunner.java -d bin
$ nxjlink -o MyProgram.nxj MyRunner -cp "bin/:lib/LejosKit.jar"
```

Der erste Befehl kompiliert das *MyRunner.java*, welches in *src/* ist in den Ordner bin. Gehen sie sicher, das dieser existiert. -*classpath* weißt darauf hin, das sich die Jar des LejosKit in lib befindet und LejosKit.jar heißt.
Der 2. Befehl erstellt die NXJ Datei mit dem Namen MyProgram.nxj. Sie benutzt MyRunner als Hauptklasse (wo die main Methode drinnen ist). Als classpath muss bin/:lib/LejosKit.jar verwendet werden. bin ist das Verzeichnis, wo alle kompilierten Java Dateien drinnen sind und der 2. Pfad ist wieder die LejosKit Jar-Datei.

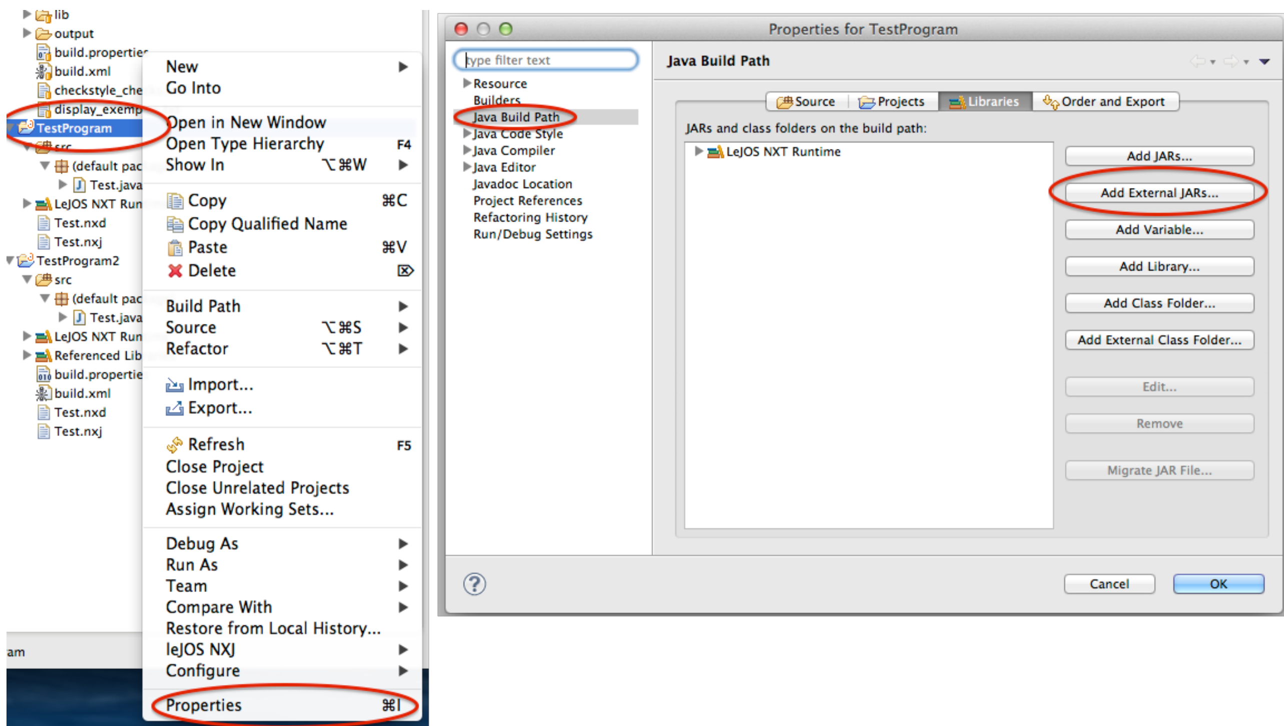
Installieren unter Eclipse.

Wenn Sie Eclipse benutzt, erstellen sie ein neues Lejos Projekt. Bitte gehen stellen Sie sicher, das Sie das neuste Eclipse Plugin für Lejos verwenden. Mehr Informationen finden Sie hier: http://www.lejos.org/nxt/nxj/tutorial/Preliminaries/UsingEclipse.htm.
Fügen Sie sich bitte die heruntergeladene Datei zu den classpath von Eclipse. Hier ist eine Anleitung wie:

1. Klicken Sie mit rechts auf ihr Programm und wählen Sie Properties/Eigenschaften aus.
2. Gehe Sie auf "Java Build Path" und wählen Sie Libraries/Bibliotheken aus.
3. Klicken Sie auf Add External Jar/Entfernte Jar hinzufügen und wählen Sie die Datei auf Ihr System aus.

Wichtig: Die Datei darf nicht gelöscht werden, sonst müssen Sie die Schritte wiederholen. Stellen Sie also sicher, dass die Datei nicht gelöscht oder verschoben wird.

4. Klicken Sie auf Ok.



Erstes Programm

In diesem Kapitel werden wir die Grundlagen des Kits kennen lernen. Wir werden uns befassen, wie ein Lejos Projekt mit LejosKit aussehen soll. Wir werden nicht in Details mit Klassen gehen, sondern wesentlich die Grundlegende Struktur lernen.
Um Sachen einfach zu machen, werden wir erstmal mit einen Programm anfangen, das nicht LejosKit benutzt, werden dies aber dann erweitern.

Hallo Welt Programm ohne LejosKit

Lass uns sagen, unser Programm soll 3 Sachen können.

1. "Hallo Welt, dies ist ein Textprogramm" auf das Display aufgeben und es nach einen Klick auf Ok wieder löschen.
2. Den NXT zu einen Gewissen Punk bewegen.
3. Wieder eine Nachricht ausgeben, dass alles erfolgreich war.

Nun, ohne LejosKit wäre es eine Möglichkeit alles in eine Methode zu stecken.

```
public static void main(String[] args) {
    // Zeige Nachricht
    LCD.drawString("Hallo Welt ich", 0, 0);
    LCD.drawString("bin ein", 0, 1);
    LCD.drawString("Textprogramm", 0, 2);
    Button.ENTER.waitForPressAndRelease();
    LCD.clear();

    DifferentialPilot pilot = new DifferentialPilot(
            DifferentialPilot.WHEEL_SIZE_NXT1, 5, Motor.A, Motor.B
    );
    pilot.setTravelSpeed(100);

    // Fahre Weg
    Navigator navigator = new Navigator(pilot);
    navigator.addWaypoint(new Waypoint(50, 50));
    navigator.addWaypoint(new Waypoint(50, 100));
    navigator.followPath();
    navigator.waitForStop();

    // Zeige Nachricht
    LCD.drawString("Der Roboter ist", 0, 0);
    LCD.drawString("nun fertig.", 0, 1);
    LCD.drawString("Bitte Druecken", 0, 2);
    LCD.drawString("sie die rote", 0, 3);
    LCD.drawString("Taste zum", 0, 4);
    LCD.drawString("Beenden.", 0, 5);

    Button.ESCAPE.waitForPressAndRelease();
}
```

Im Moment kann man den Code zwar noch gut lesen allerdings wird sich das ändern, wenn das Programm komplexer wird. Wird es zu einer Herausforderung werden, so ein Code aufrecht zu erhalten. Man kann jetzt jeden einzelnen Schritt in eine eigene Methode packen, was zur folge hat, das die Hauptklasse noch immer zu komplex wird. Java empfiehlt es außerdem Dateien immer kürzer als 2000 Codezeilen zu halten[1]. Das kann schnell überschritten werden, wenn Sie versuchen ihr ganzes Programm in nur eine Klasse zu packen. Die nächste Alternative ist es, Klassen für jeden Schritt zu erstellen. Diese zu laden ist dennoch wieder uneinheitlich und mühsam.
Das nächste Problem ist, wie die Nachricht auf das Display gezeichnet wird. Man muss die Nachricht in einzelne Teile unterteilen, um den anzeigen zu lassen. Ist das nicht nervig? Wäre es nicht schön, wenn das automatisch gemacht wird? Für die beiden Probleme hat LejosKit ausgezeichnete Lösungen. Wie sie diese effektive nutzen lernen sie während den nächsten 2 Kapiteln.

Hallo Welt in Parts mit LejosKit

In LejosKit jeder Schritt der der Robot macht ist ein Programm. Folglich gibt es in LejosKit eine fundamentale Klasse: Program. *Program* ist abstrakt. Wir müssen es also erweitern, wenn wir ein neuen Schritt machen wollen. Fangen wir also an, unseren Code von oben in einzelne Teile zu zerkleinern. Erstmal müssen wir uns Gedanken über die Klassennamen machen. Da der erste Schritt den Benutzer begrüßt, können wir ihn also SagHalloProgram nennen. Beim 2. wird's schwieriger, da das Programm kein wirkliches Ziel hat. Lass uns als annehmen, es fährt zu einer Tür. Also nennen wir das FahrZurTuerProgram. Das letzte heißt GoodbyeProgram, da es den Benutzer verabschiedet. Hier ist der Code dazu:

SagHalloProgram.java

```
public class SagHalloProgram extends Program {
    @Override
    public void go(Runner runInstance) {
        LCD.drawString("Hallo Welt ich", 0, 0);
        LCD.drawString("bin ein", 0, 1);
        LCD.drawString("Textprogramm", 0, 2);
        Button.ENTER.waitForPressAndRelease();
        LCD.clear();
    }
}
```

FahrZurTuerProgram.java

```
public class FahrZurTuerProgram extends Program {
    @Override
    public void go(Runner runInstance) {
        DifferentialPilot pilot = new DifferentialPilot(
```

[1] Java Code Guidelines: http://java.sun.com/docs/codeconv/html/CodeConventions.doc2.html (tot. Kopie von archive.org: https://web.archive.org/web/20050205041406/http://java.sun.com/docs/codeconv/html/CodeConventions.doc2.html).

```
                DifferentialPilot.WHEEL_SIZE_NXT1, 5, Motor.A, Motor.B
        );

        Navigator navigator = new Navigator(pilot);
        navigator.addWaypoint(new Waypoint(50, 50));
        navigator.addWaypoint(new Waypoint(50, 100));
        navigator.followPath();
        navigator.waitForStop();
    }

}
```

GoodbyeProgram.java

```
public class GoodbyeProgram extends Program {
    @Override
    public void go(Runner runInstance) {
        LCD.drawString("Der Roboter ist", 0, 0);
        LCD.drawString("nun fertig.", 0, 1);
        LCD.drawString("Bitte Druecken", 0, 2);
        LCD.drawString("sie die rote", 0, 3);
        LCD.drawString("Taste zum", 0, 4);
        LCD.drawString("Beenden.", 0, 5);

        Button.ESCAPE.waitForPressAndRelease();
    }
}
```

So sieht der Code geordneter aus, allerdings weiß der Robot jetzt weder was Programme sind (er kann nicht das anhand des Dateinamens bestimmen) und er weiß auch nicht die Reihenfolge. Als brauchen wir eine Klasse, die alle Programme ausführt, einen *Runner*. Der *Runner* hat 2 Methoden, die wichtig für uns sind: *addProgram* und *start*. Der erste Befehl ist zum Hinzufügung eines Programms, und der 2. zum Starten aller Programme in der Reihenfolge, in der sie hinzugefügt wurden. Unter der Haube macht *start* allerdings noch einiges mehr (mehr dazu im Verlauf des Buches).
Hier wäre eine Möglichkeit, wie man den Runner benutzen kann:

```
public static void main(String[] args) {
    Runner myRunner = new Runner();
    myRunner.addProgram(new SagHalloProgram());
    myRunner.addProgram(new FahrZurTuerProgram());
    myRunner.addProgram(new GoodbyeProgram());

    myRunner.start();
}
```

Programm nennt ist egal, dennoch empfehle ich folgende Regeln. Der Name sollte mit einen großen Buchstaben starten und CamleCase benutzen, wie es bei Java üblich ist. Der Name sollte immer mit *Program* enden (vorzugsweise die amerikanische Schreibweise), um zu erkennen das es ein Programm ist. Der Name sollte kurz sagen was das Programm macht. Wenn das Programm für einen Namen zu komplex ist, sollten Sie versuchen, das

Programm in mehrere kleine zu zerteilen. Der Name sollte auch möglichst kurz sein. Es ist aber wichtiger, dass der Name genau sagt, was das Programm macht. Wenn der Name als lang wird, ist das nicht schlimm. Solange es nicht abstrakt ist. Ein Programm sollte nicht recycelt werden. Obwohl unsere zwei unserer Programm (*SagHalloProgram* und *GoodbyeProgram*) die selbe Aufgabe machen (einen Text anzeigen), und daher theoretisch eine Klasse mit nur einer Variable mit dem jeweiligen Text nur nötig wäre, sollte das nicht gemacht werden. Hier ist ein Beispiel:

```
public static void main(String[] args) {
    Runner myRunner = new Runner();
    String[] sagHalloText = new String[] {
        "Hallo welt...",
        "..."
    };

    myRunner.addProgram(new ZeigeText(sagHalloText));
    myRunner.addProgram(new FahrZurTuerProgram());

    String[] goodbyeText = new String[] {
        "Das war's"
    };
    myRunner.addProgram(new ZeigeText(goodbyeText));

    myRunner.start();
}
```

Wenn sich die Funktionen von 2 Programmen so knapp unterscheiden, sollte man ein weiteres, abstraktes, Programm machen. Dieses könnten wir dann *ZeigeText* nennen und es könnte so aussehen:

```
public abstract class ZeigeText extends Program {
    public abstract String[] getMessages();

    @Override
    public void go(Runner runInstance) {
        // Code der die Nachrichten aus getMessages() anzeigt.
    }
}
```

SagHalloProgram und *GoodbyeProgram* würden dann nur noch getMessages erweitern, das die jeweiligen Nachrichten ausgibt.
Für das Ausgeben von Text bietet LejosKit eine bessere Lösung, die komplexer, flexibler und einfacher ist…

Hallo Welt mit Dialogen

LejosKit bietet sogenannte Dialog. Ein Dialog dient um Inhalte auf dem Bildschirm anzuzeigen. Ein Dialog kann sehr komplexe Dialog darstellen. In diesem Abschnitt kümmern wir uns aber um einen normalen Textdialog. Den Textdialog, den wir brauchen ist ein *MessageDialog*. Dieser Zeigt eine Nachricht auf dem ganzen Bildschirm. MessageDialog erwartet einen String, der die komplette Nachricht enthält in den Konstruktor. MessageDialog zerteilt die Nachricht so, dass sie auf den Bildschirm passt. Sollte die Nachricht zu lang sein, wird eine *MessageTooLongException* geworfen. Wenn Sie einen statischen Text verwenden (einen Text, der sich nicht verändert kann) und der nicht zu lang ist, werden Sie keine Probleme bekommen und können die Exception einfach ignorieren. Wenn der Text variiert (die länge dynamisch verändern, durch Werte die außerhalb kommen), ist es wichtig, diese Exception nicht zu ignorieren.
Das zweite, was wir brauchen ist ein *SimpleDialogProgram*. Das besondere an einem *SimpleDialogProgram* ist, das es keine Rolle spielt, welche Art von Dialog angezeigt werden soll, solle er von *SimpleDialog* abstammt. Für einen normalen Dialog verwenden Sie bitte *DialogProgram*. Weite Informationen zu den Unterscheiden finden sie in den nächsten Kapiteln.
Hier ist also unser Code für *SagHalloProgram* und *GoodbyeProgram*:

```
public class SagHalloProgram extends SimpleDialogProgram {
    @Override
    public SimpleDialog getSimpleDialog() {
        try {
            return new MessageDialog("Hallo Welt, dies ist ein 
Textprogramm");
        } catch (MessageTooLongException e) {
            throw new RuntimeException(e);
        }
    }
}

public class GoodbyeProgram extends SimpleDialogProgram {
    @Override
    public SimpleDialog getSimpleDialog() {
        try {
            return new MessageDialog("Der Roboter ist nun fertig."
                    + "Bitte Druecken sie die rote Taste zum Beenden");
        } catch (MessageTooLongException e) {
            throw new RuntimeException(e);
        }
    }
}
```

Dank *MessageDialog* und *SimpleDialogProgram* ist kaum noch Code notwendig, um Text auf dem Bildschirm dazustellen.

Wie man auch noch das *FahrZurTuerProgram* optimieren kann werde ich euch in den nächsten Kapiteln zeigen.

Programme

Ein Programm ist in LejosKit der Grundbaustein jedes Roboters. Wie man Programme allerdings richtig verwendet, und welche es gibt werde ich in diesem Kapitel zeigen.

Program - die Mutterklasse

Program ist die Mutterklasse aller Programme. Sie enthält 3 Methoden: *go(Runner)*, *clearUp(Runner)*, und *onCompletion(CompletionEvent)*.
go(Runner) ist Methode, die benutzt wird wenn das Programm gestartet wird und etwas gemacht soll. Die Method soll erst dann beendet werden, wenn auch das Programm fertig ist. Sobald die Method einmal zu ende ist, wird das Programm nicht mehr weiterarbeiten können. Um das zu erreichen sollte man am besten eine Schleife in die Funktion schreiben, die erst dann fertig ist, wenn auch das Program fertig ist.
clearUp(Runner) wird aufgerufen, das Programm ein einen Zustand gehen kann, so das es keine Spuren mehr hinterlässt. Das heißt es muss ggf. Listeners entfernen sowie den Bildschirm wieder löschen, falls es ihn benutzt hat. *clearUp(Runner)* kann theoretisch auch aufgerufen werden, wenn ein Programm pausiert wird. Dieses Verhalten ist allerdings noch nicht implementiert.
onCompletion(CompletionEvent) wird aufgerufen wenn das Programm fertig ist. Es wird oft verwenden um ein programmspezifisches Verhalten aus zuführen, wenn *go()* in der Superklasse schon erweitert wurde. Außerdem hat *onCompletion(CompletionEvent)* die Fähigkeit, den Runner zu stoppen (siehe *CompletionEvent*). *CompletionEvent* naher Zukunft weiteres Verhalten bekommen, das ähnlich ist. Man könnte die Method verwenden, um beispielsweise werte im Config zu verändern, die vorher ausgewählt wurden.

MoveControllerProgram

Allgemein

Das MoveControllerProgram ist dient dazu, den Roboter fahren zu lassen. Erstmal ist es wichtig, die richtigen *Config* Werte zu setzen. Im verlaufe des Buches erfahren sie, wie man Config richtig benutzt, allerdings reichen erstmal die Configwerte, die von MoveControllerProgram erwartet wird: *robot.wheeldiameter*, *robot.trackwidth*, *motor.left* und *motor.right*. *robot.wheeldiameter* gibt an, wie weit breit das Rad ist. *robot.trackwidth* gibt an, wie weit die Räder von einandern entfernt sind. Mehr Informationen yu den beiden Werten, gibt es auf der Seite von lejos.org. Da ein Config nur *Strings* oder *Storables* annimmt, gehen sie bitte sicher das die Zahlen *Doubles* als *Strings* sind. *motor.left* und *motor.right* welches der link und rechter Motor des Roboters sind (benutzen Sie bitte *MotorPortName* für den Namen des Motors. Sie können dabei die ID, Name oder

“ShortName” austauschbar verwenden, da der Motor durch *getPortByName* ermittelt wird).
Der Code für die Config sieht also so aus:
```
public static void main(String[] args) {
    Config c = Config.getInstance();
    c.setProperty("motor.left", MotorPortName.PORT_B.getValue());
    c.setProperty("motor.right", MotorPortName.PORT_C.getValue());
    c.setProperty("robot.wheeldiameter",
            String.valueOf(MoveController.WHEEL_SIZE_NXT2));
    c.setProperty("robot.wheeldiameter", "10");

    Runner runner = new Runner();
    // Lass die Programme laufen...
}
```

Eigenen MoveController verwenden

Nun, kann es aber losgehen. Erstellt eine neues Program von der Subklasse *MoveControllerProgram*. In die *move* Methode können Sie Ihren Code platzieren, um den Robot zu bewegen. Beachten Sie das die Methode erst fertig am Ende ankommt, wenn ihr Roboter fertig ist. Sonst geht LejosKit davon aus, dass das Programm fertig ist. Hier wäre ein Beispielcode, der den Roboter um 10cm bewegt mit der Geschwindigkeit von 20cm/s.

```
public class MyMoveController extends MoveControllerProgram {
    @Override
    public void move(RotateMoveController moveController) {
        moveController.setTravelSpeed(20);
        moveController.travel(10);
    }
}
```

Da *RotateMoveController*.travel wartet, bis die Bewegung fertig ist, müssen wir nicht am Ende überprüfen ob moveController fertig ist.
Wenn Sie einen eigenen *MoveController* verwenden wollen, können Sie getMoveController überschreiben. Dort können Sie entweder die Einstellungen des *DifferentialPilot* (der Standard *MoveController*) ändern oder einen neuen *MoveController* verwenden. Wenn ihr *MoveController* eigene Methoden hat, die Sie in *move* brauchen, müssen sie wie folgt vorgehen:

1. Überschreiben Sie *getMoveController()* in Ihrer Klasse, wo Sie Ihren MoveController erstellen. Er muss *RotateMoveController* erweitern.
2. Passen Sie den Returntype von *getMoveController()* an.
3. Sie können nun den Type von parameter *moveController* Konvertieren (type casting). Es ist nicht empfohlen, den noch *getMoveController()* zu benutzen, da dieser unnötig Speicher und Laufzeit verbraucht (Lejos 0.9 hat noch kein *Garbage Collector* (englisch)). Sie können nicht wissen, ob ihre Klasse eine Superklasse ist und ihre Subklasse lange braucht um den MoveController zu erstellen.
   Sie können sich sicher sein, dass der Parameter immer eine *Instance* von ihren Typ sein

wird (oder einen Subtyp), außer ihre Superklasse erweitern und implementiert *go* falsch, wenn sie den Returntyp von *getMoveController()* angepasst haben. Dann ist es allerdings nicht ihr Problem sondern ein Designfehler in der Subklasse.

Hier ist ein bisschen Beispielcode:

```
public class ErweiteresMCProgramme extends MoveControllerProgram {
    @SuppressWarnings("unused")
    @Override
    public void move(RotateMoveController moveController) {
        MyCustomMoveController myController =
                (MyCustomMoveController)moveController;
        // mach was mit myController
    }

    @Override
    public MyCustomMoveController getMoveController(RegulatedMotor
            leftMotor, RegulatedMotor rightMotor) {
        MyCustomMoveController myController = new
MyCustomMoveController();
        // passe myController an.
        return myController;
    }
}
```

sdf