

Problemorientiertes Programmieren

RoboCode

1. Termin – Teil II: Den Roboter schreiben



Christian Riess, Eva Eibenberger

Lehrstuhl für Mustererkennung (Inf. 5)

Friedrich-Alexander-Universität Erlangen-Nürnberg



Roboter yay!

- Auf zu einem praktischen Problem – den Robotern
- Mehr Java kann man unterwegs lernen
- Grundsätzlich kann man den Roboter mit **irgendeinem Editor** schreiben – genauso wie **irgendein anderes Programm**
- Ist die Wahl des Editors also egal?
Subjektiv nicht: Nimm die Umgebung, in der Du am produktivsten bist.

(Beispiele: für C++ bevorzuge ich **vi**, für Java **eclipse**. Christoph Egger (FSI Inf.) leistet am meisten im **emacs**. Für QT-Oberflächen benutzt Johannes Jordan (Inf5) den **QT creator**, ansonsten **scite**)
- **Best tool for the job!**



Also – welche Umgebung?

- Editoren müsst ihr euch privat aneignen.
- Wir starten mit einem einfachen, für vieles nützlichen Editor, **scite**.
Außerdem wird ein Editor mit RoboCode mitgeliefert, und wer will kann jederzeit auf **eclipse** umsteigen.
- Ans Werk, und gutes Gelingen!
 1. RoboCode-Oberfläche herunterladen (.oO google: robocode download)
 2. Im Home liegt eine Datei robocode-<versionsnummer>.jar
 3. “.jar” steht für „Java Archive“. Führe es im Terminal aus:
`java -jar robocode-<versionsnummer>.jar`
Klick Dich durch die Installation.
 4. Starte die Umgebung mit „`cd robocode`“ und „`./robocode.sh`“



Der erste Roboter...

- Wir haben ein Skript geschrieben, das eine Roboterschablone kopiert
- Starte eine Konsole, und führe `~sichries/createRobot.sh` aus...

```
faii00j [~]> ~/sichries/createRobot.sh
Usage: ./createRobot TYPE PACKAGE ROBOTNAME , where

TYPE=
0: empty robot
1: template 1
2: template 2
3: template 3

PACKAGE is the package name (typically lowercase, no special characters, e.g. hans)
ROBOTNAME is the robot name, typically starting uppercase, no special characters (e.g. Dampf)
complete example: ./createRobot 1 hans Dampf

faii00j [~]> ~/sichries/createRobot.sh 0 rock Hard
created robot /home/cip/2001/sichries/robocode/robots/rock/Hard.java -> happy editing!
faii00j [~]> cd ~/robocode/robots/rock/
faii00j [~/robocode/robots/rock]> scite Hard.java
```

Package-Name:

- Kleinbuchstaben
- Keine Sonderzeichen
- Keine Umlaute

Robotername:

- Großer Anfangsbuchstabe
- Keine Sonderzeichen
- Keine Umlaute

Speicherort:

- In dem Home, unter robocode/robots
- In einem Unterverzeichnis, das wie das Package heißt
- In einer Datei, die wie die Klasse heißt

- ...und öffne den Robotercode in `scite`.

(createRobot kopiert in `~/robocode/robots/<package>/` den Anfangscode; wir könnten das manuell machen, aber so ist's bequemer).



Scite: Programmieren und Übersetzen

```

1 package rock;
2
3
4 import robocode.*; // for most robocode-classes
5 import robocode.util.*; // for further robocode-classes
6 import java.awt.*; // for colors
7
8 /**
9  * Look at the line indentations below!
10 * - When a new code block opens, the subsequent code is shifted one more tab
11 *   to the right.
12 * - When a code block closes, the subsequent code is shifted one tab to the
13 *   left.
14 */
15 - public class Hard extends AdvancedRobot {
16
17     // example for a class variable: the content of this variable remains valid
18     // after a method is finished.
19     boolean nothingToDo;
20
21     long lastScanTime;
22
23 - public void run() {
24     // paint the robot in distinct colors
25     setBodyColor(Color.green);
26     setGunColor(Color.yellow);
27     setRadarColor(Color.black);
28
29     // we did not see anybody in the beginning
30     lastScanTime = -10;
31     nothingToDo = true;
32
33     // repeat forever (i.e. execute the content of the curly braces while
34     // the condition in brackets is true - but "true" is always true.
35     // Thus, we repeat forever...)
36 - while(true) {
37     System.out.println("round number = " + getTime() + " my health: " + getEnergy());

```

Übersetzt den
Roboter (F7)

Die Arena starten:
„Tools“->„Go“ (F5)

Codefenster



Programmierfehler finden:

■ Syntaktische Fehler zeigt der Compiler auf:

```

File Edit Search View Tools Options Language Buffers Help
[Icons]
1 package rock;
2
3
4 import robocode.*; // for most robocode-classes
5 import robocode.util.*; // for further robocode-classes
6 import java.awt.*; // for colors
7
8 /**
9  * Look at the line indentations below!
10  * - When a new code block opens, the subsequent code is shifted one more tab
11  *   to the right.
12  * - When a code block closes, the subsequent code is shifted one tab to the
13  *   left.
14  */
15 -public class Hard extends AdvancedRobot {
16
17     // example for a class variable: the content of this variable remains valid
18     // after a method is finished.
19     boolean nothingToDo;
20
21     long lastScanTime;
22
23 - public void run() {
24     // paint the robot in distinct colors
25     setBodyColor(Color.green);
26     setGunColor(Color.yellow);
27     setRadarColor(Color.black);
28
29     // we did not see anybody in the beginning
30     lastScanTime = -10;
31     nothingToDo = true;
32
33     // repeat forever (i.e. execute the content of the curly braces while
34     // the condition in brackets is true - but "true" is always true.
35     // Thus, we repeat forever...)
36 - while(true) {
37     System.out.println("round number = " + getTime() + ", my health: " +
38     getEnergy());

```

```

>make
javac -classpath /usr/lib/jvm/java-6-openjdk/bin/javac.jar Hard.java
Hard.java:19: cannot find symbol
symbol : class boolean
location: class rock.Hard
    boolean nothingToDo;
    ^
1 error
make: *** [all] Error 1
>Exit code: 2

```

Dateiname,
Zeilennummer,
Problembeschreibung:
„Symbol ist unbekannt“

Welches Symbol?
„boolean“

Wo genau in der Zeile?
Dort, wohin „^“ zeigt



Fehlerfreie Übersetzung

- Der Übersetzer gibt „0“ zurück, wenn kein Fehler aufgetreten ist
- Wichtig: Nur ein fehlerfrei übersetzter Roboter kann in der Arena benutzt werden!

```

File Edit Search View Tools Options Language Buffers Help
1 package rock;
2
3
4 import robocode.*; // for most robocode-classes
5 import robocode.util.*; // for further robocode-classes
6 import java.awt.*; // for colors
7
8 /**
9  * Look at the line indentations below!
10  * - When a new code block opens, the subsequent code ↵
11     ↳ is shifted one more tab
12     ↳ to the right.
13  * - When a code block closes, the subsequent code is ↵
14     ↳ shifted one tab to the
15     ↳ left.
16  */
17 - public class Hard extends AdvancedRobot {
18
19     // example for a class variable: the content of ↵
20     ↳ this variable remains valid
21     // after a method is finished.
22     boolean nothingToDo;
23
24     long lastScanTime;
25
26 - public void run() {
27     // paint the robot in distinct colors
28     setBodyColor(Color.green);
29     setGunColor(Color.yellow);
30     setRadarColor(Color.black);
31
32     // we did not see anybody in the beginning
33     lastScanTime = -10;
34     nothingToDo = true;
35
36     // repeat forever (i.e. execute the content of ↵
37     ↳ the curly braces while
38
39 }

```

```

>make
javac -classpath /usr/lib/jvm/java-.../lib/robo.jar rock.java
>Exit code: 0

```

li=1 co=1 INS (LF)



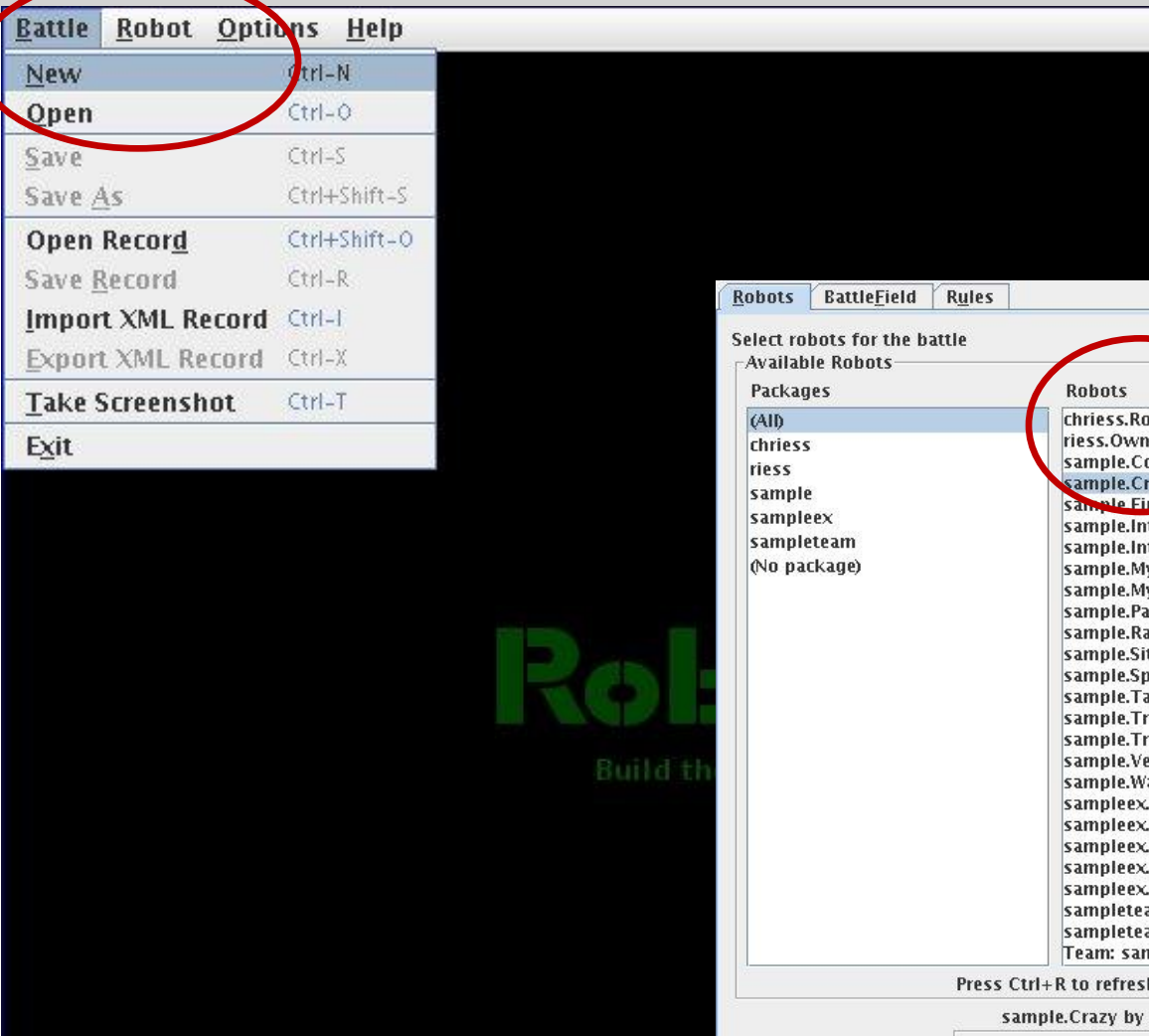
```

1 package rock;
2 import robocode.*; // for most robocode-classes
3 import robocode.util.*; // for further robocode-classes
4 import java.awt.*; // for colors
5
6 - public class Hard extends AdvancedRobot {
7
8     // example for a class variable: the content of this variable remains valid
9     // after a method is finished.
10    boolean nothingToDo;
11    long lastScanTime;
12
13    - public void run() {
14        // paint the robot in distinct colors
15        setBodyColor(Color.green);
16        setGunColor(Color.yellow);
17        setRadarColor(Color.black);
18
19        // we did not see anybody in the beginning
20        lastScanTime = -10;
21        nothingToDo = true;
22
23        // repeat forever (i.e. execute the content of the curly braces while
24        // the condition in brackets is true - but "true" is always true.
25        // Thus, we repeat forever...)
26    - while(true) {
27        System.out.println("round number = " + getTime() + ", my health: " + getHealth());
28
29        - if (nothingToDo) {
30            setTurnRadarRight(360);
31            setTurnRight(20);
32            setAhead(30);
33            waitFor(new TurnCompleteCondition(this)); // executes the command
34            // above, and remains here until the 20 degrees turn is finished.
35        } else {
36            // check if we did not see the enemy for a while
37    - if (getTime() - lastScanTime > 10) {
38                nothingToDo = true;
39            }
40            execute(); // keep executing the selected commands
41        }
42    }
43    }
44
45    // example for an event: The execution of the code jumps here when we see a robot in the radar
46    - public void onScannedRobot(ScannedRobotEvent e) {
47        setTurnRight(e.getBearing());
48        waitFor(new TurnCompleteCondition(this));
49        setFire(1.0);
50        nothingToDo = false;
51    }

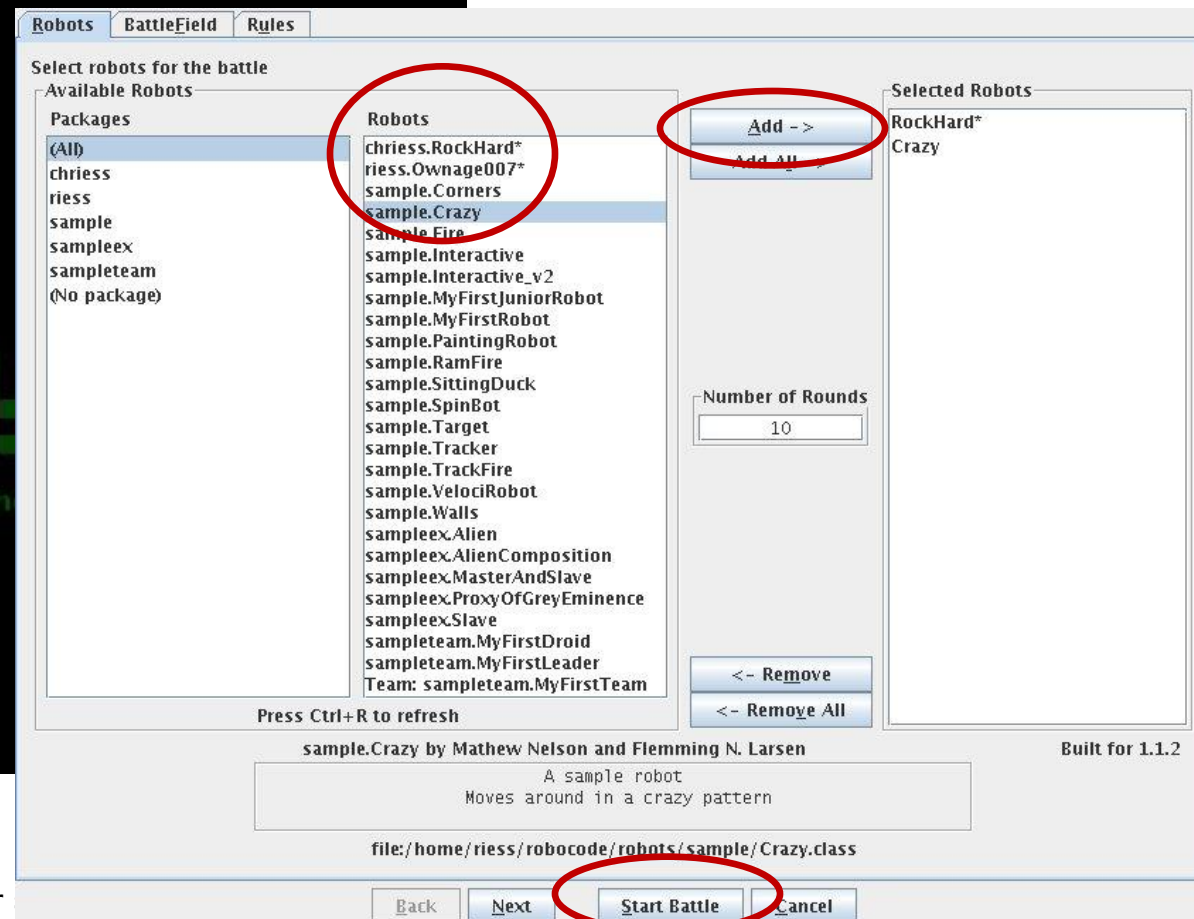
```

- In dem mitgelieferten Editor wird der minimal notwendige Code für einen Roboter bereitgestellt.
- Wir lesen den Code gleich; erst wollen wir jedoch testen, ob alles funktioniert:
 - compilieren,
 - in die Arena einbinden.

Auf in die Arena!



- Der neue Roboter sollte automatisch in der Auswahl erscheinen.



Christian Riess, Eva Eibenberger

15.08.2011 1. Termin – Teil II: Den Roboter



...nach dem Kampf...

- Ein paar Aspekte an dem Prototypen sind... suboptimal gelöst sind.
- Wir haben **beobachtet**, dass der Prototyp sein Potential nicht voll nutzt.
- Sehen reicht nicht: wir müssen die Aktionen im Code **analysieren**,
- dann Verbesserungsmöglichkeiten in der Arena **recherchieren**,
- eine Idee theoretisch **ausknobeln**,
- und **implementieren**.

- Und danach?
(Ab in die Arena,
hoffen dass die Arbeit sich gelohnt hat,
irgendwas wird nicht funktionieren
(kann nicht; Dinge funktionieren nie einfach so),
also wieder zum Anfang (beobachten, analysieren...))

„import robocode.*“ heißt „wir benutzen alles aus dem package robocode“ („*“ ist ein „Wildcard“, also „alles“)

„extends“ kann man lesen wie „ist ein“, oder „hat die Funktionen von“, in diesem Fall die Funktionen von „Robot“ (siehe Dokumentation, ein paar Folien später)

„run“ ist die Methode, die von der Arena aufgerufen wird, um den Roboter zu starten; er läuft, bis das Match zu Ende ist (siehe Architektur der Arena, ein paar Folien später)

„while(true)“ heißt „für alle Ewigkeit...“

setAhead, setTurnRight, setTurnRadarRight liest sich fast wie ein englischer Text.

Offenbar wird hier unterschieden, ob man den Gegner vor kurzem gesehen hat oder nicht (nothingToDo). Wenn er nicht gesehen wurde, sieht sich der Roboter um und fährt dabei ein wenig vorwärts und nach rechts.

„on...“ Methoden, die ein „...Event“ übergeben bekommen, werden von der Arena aufgerufen, wenn etwas passiert. Die Anweisungen in „run“ werden so lange unterbrochen.

z.B. die Methode **onScannedRobot** wird aufgerufen, wenn in unserem Scanner ein anderer Roboter auftaucht. Was der Scanner sieht, kann man aus „ScannedRobotEvent“, hier „e“ genannt, auslesen.

Hier drehen wir uns dorthin, wo der Gegner gesehen wurde, warten bis die Drehung komplett ist, und schießen.

```
1 package robocode;
2 import robocode.*; // for most robocode-classes
3 import robocode.util.*; // for further robocode-classes
4 import java.awt.*; // for colors
5
6 - public class Hard extends AdvancedRobot {
7
8     // example for a class variable: the content of this variable remains
9     // after a method is finished.
10    boolean nothingToDo;
11    long lastScanTime;
12
13 - public void run() {
14    // paint the robot in distinct colors
15    setBodyColor(Color.green);
16    setGunColor(Color.yellow);
17    setRadarColor(Color.black);
18
19    // we did not see anybody in the beginning
20    lastScanTime = -10;
21    nothingToDo = true;
22
23    // repeat forever (i.e. execute the content of the curly braces wh
24    // the condition in brackets is true - but "true" is always true.
25    // Thus, we repeat forever...)
26 - while(true) {
27    System.out.println("round number = " + getTime() + ", my health: " + getEnergy());
28
29    if (nothingToDo) {
30        setTurnRadarRight(360);
31        setTurnRight(20);
32        setAhead(30);
33        waitFor(new TurnCompleteCondition(this)); // executes
34        // above, and remains here until the 20 degrees turn is f
35    } else {
36        // check if we did not see the enemy for a while
37        if (getTime() - lastScanTime > 10) {
38            nothingToDo = true;
39        }
40        execute(); // keep executing the selected commands
41    }
42 }
43 }
44
45 // example for an event: The execution of the code jumps here when we
46 - public void onScannedRobot(ScannedRobotEvent e) {
47    setTurnRight(e.getBearing());
48    waitFor(new TurnCompleteCondition(this));
49    setFire(1.0);
50    nothingToDo = false;
51    lastScanTime = getTime();
52 }
53 }
```



Exkurs: Interne Verarbeitung der Anweisungen

- Ein Roboter wird in einem „Thread“ gestartet („pseudo-separates Program“).
- Die Arena lässt jeden Thread abwechselnd ein paar Millisekunden rechnen. Spielanweisungen, z.B. „`setAhead()`“, werden gespeichert, bis jeder Thread gerechnet hat.
- Nach `execute()` oder `waitFor()` führt die Arena Roboteranweisungen aus. z.B. „`setAhead()`“: Roboter ein kleines Stück vorbewegen und speichern, wie viel Wegstrecke noch verbleibt
- Die Arena prüft, ob ein „Ereignis“ eingetreten ist. Falls ja, wird der Thread/Roboter nicht innerhalb von „run“ fortgesetzt, sondern in der Ereignis-Methode. z.B. wenn ein Roboter im Scannerbereich liegt, werden dessen Daten in `ScannedRobotEvent` gespeichert. In der nächsten Rechenphase wird `onScannedRobotEvent()` mit diesen Daten aufgerufen.
- ...und wieder von vorne
(jeder Roboter darf rechnen; entweder erst in einem Ereignis, oder dort, wo er zuletzt unterbrochen wurde).

Christian Riess, Eva Eibenberger



Anweisungen erteilen

- Anweisungen beginnen mit set: `setAhead()`, `setBack()`, usw.
- Die Anweisungen werden von der Arena gesammelt. Sobald `execute()` oder `waitFor()` aufgerufen wird (siehe zwei Folien früher!), werden diese Anweisungen gleichzeitig ausgeführt.
- Was wenn zwischenzeitlich zweimal die selbe Anweisung, oder widersprüchliche Anweisungen erteilt wurden?
- Dann gilt nur die letzte!

Beispiel: Wir wollen Rechtskurven fahren, mit `setAhead(100)` und `setTurnRight(20)`. Ein Scanner-Ereignis tritt ein, wir haben den Gegner gesehen – und wollen lieber nach vorne links auf ihn zufahren mit `setAhead(300)`, `setTurnLeft(10)`. Dann gelten nur diese letzten beiden Kommandos.



Anweisungen erteilen

- Warum gibt es `execute()` und `waitFor()`?
- Wenn alles „seinen normalen Gang geht“ benutzen wir `execute()`. Semantisch heißt das „mach einfach, was ich gesagt habe“.
- Manchmal gibt es jedoch inhaltliche Abhängigkeiten: Z.B. wollen wir **erst** die Kanone zum Gegner drehen, und **dann** schießen.
- Mit `execute()` lässt sich solch eine Abhängigkeit nur schwierig implementieren. Die Lösung bietet `waitFor()`:
- `waitFor(new TurnCompleteCondition(this))` funktioniert wie `execute()`, setzt aber im Code nicht fort. Wenn also `setFire()` erst danach kommt, wissen wir, dass der Roboter richtig orientiert ist.



In der Dokumentation recherchieren

- Ein paar Aspekte an dem Prototypen sind... suboptimal gelöst sind.
 - Bisher haben wir **beobachtet**, dass der Prototyp Schwächen hat.
 - Sehen reicht nicht: wir müssen die Aktionen im Code **analysieren**,
 - dann Verbesserungsmöglichkeiten in der Arena **recherchieren**,
 - eine Idee theoretisch **ausknobeln**,
 - und **implementieren**.
-
- Und danach?
 - (.oO Ab in die Arena,
beten dass die Arbeit sich gelohnt hat,
irgendwas funktioniert nicht
(kann nicht; Dinge funktionieren nie einfach so),
also wieder zum Anfang (beobachten, analysieren...)

RoboCode- und Java-Dokumentation lesen



- JavaDoc: Referenzdokumentation in HTML
- Im Installationsverzeichnis unter javadoc/index.html

(.oO HTML: am besten
im Browser öffnen)

Links unten ist
eine Liste der
verfügbaren
Klassen: am
Anfang ist die
Klasse Robot
wichtig

Robocode 1.7.3.0-Beta API

Packages	
robocode	Robot API used for writing robots for Robocode.
robocode.annotation	Contains annotations that can be used with Robocode.
robocode.control	The Robocode Control API is used for controlling the Robocode application from another external application.
robocode.control.events	Battle events that occurs during a game, and which are used for the robocode.control.IBattleListener class.
robocode.control.snapshot	Snapshots of the battle turns, robots, bullets, scores etc.
robocode.robotinterfaces	Robot Interfaces used for creating new robot types, e.g. with other programming languages.
robocode.robotinterfaces.peer	Robot peers available for implementing new robot types based on the Robot Interfaces.
robocode.util	Utility classes that can be used when writing robots.

All Classes

- [AdvancedRobot](#)
- [BattleAdaptor](#)
- [BattleCompletedEvent](#)
- [BattleEndedEvent](#)
- [BattleErrorEvent](#)
- [BattleEvent](#)
- [BattlefieldSpecification](#)
- [BattleFinishedEvent](#)
- [BattleMessageEvent](#)
- [BattlePausedEvent](#)
- [BattleResults](#)
- [BattleResumedEvent](#)
- [BattleRules](#)
- [BattleSpecification](#)
- [BattleStartedEvent](#)
- [Bullet](#)
- [BulletHitBulletEvent](#)
- [BulletHitEvent](#)
- [BulletMissedEvent](#)
- [BulletState](#)
- [Condition](#)
- [CustomEvent](#)
- [DeathEvent](#)
- [Droid](#)
- [Event](#)
- [GameTurnCompleteCondition](#)
- [HitByBulletEvent](#)
- [HitByBulletEvent](#)

Copyright © 2011 [Robocode](#). All Rights Reserved.

RoboCode- und Java-Dokumentation lesen



File Edit View History Bookmarks Tools Help

file:///home/riess/robocode/javadoc/index.html

Overview Package **Class** Tree Deprecated Index Help

PREV CLASS NEXT CLASS FRAMES NO FRAMES
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

robocode

Class Robot

[java.lang.Object](#)
└ [robocode._RobotBase](#)
 └ [robocode._Robot](#)
 └ [robocode.Robot](#)

All Implemented Interfaces:
[Runnable](#), [IBasicEvents](#), [IBasicEvents2](#), [IBasicEvents3](#), [IBasicRobot](#), [IInteractiveEvents](#), [IInteractiveRobot](#), [IPaintEvents](#), [IPaintRobot](#)

Direct Known Subclasses:
[AdvancedRobot](#)

```
public class Robot
  extends _RobotBase
  implements IInteractiveRobot, IPaintRobot, IBasicEvents3, IInteractiveEvents, IPaintEvents
```

The basic robot class that you will extend to create your own robots.

Please note the following standards will be used:
heading - absolute angle in degrees with 0 facing up the screen, positive clockwise. $0 \leq \text{heading} < 360$.
bearing - relative angle to some object from your robot's heading, positive clockwise. $-180 < \text{bearing} \leq 180$
All coordinates are expressed as (x,y).
All coordinates are positive.
The origin (0,0) is at the bottom left of the screen.
Positive x is right.
Positive y is up.

Author:
Matthew A. Nelson (original), Flemming N. Larsen (contributor), Matthew Reeder (contributor), Stefan Westen (contributor), Pavel Savara (contributor)

See Also:
[robocode.sourceforge.net](#), [Building your first robot](#), [JuniorRobot](#), [AdvancedRobot](#), [TeamRobot](#), [Droid](#)

Field Summary

Fields inherited from class [robocode._RobotBase](#)

[out](#)

Constructor Summary

[Robot\(\)](#)
Constructs a new robot.

file:///home/riess/robocode/javadoc/robocode/Robot.html

Allgemein
nützliche
Informationen
stehen am
Anfang der
Seite

Klasse
AdvancedRobot
auswählen; oder
für elementarere
Befehle Robot

RoboCode- und Java-Dokumentation lesen



Danach folgt eine Liste der Methoden, die ein Robot anbietet, zusammen mit einer einzeiligen Beschreibung.

z.B.:

„double getX()
Returns the X position of the robot. (0, 0) is at the bottom left of the battlefield.“

Die Methode heißt `getX()`, und wenn man sie aufruft, bekommt man den X-Wert als `double`.

The screenshot shows a web browser window displaying the RoboCode Java API documentation. The browser address bar shows the file path: `file:///home/riess/robocode/javadoc/index.html`. The page content is a table of methods for the `Robot` class. The `getX()` method is circled in red. The table lists various methods with their return types and descriptions.

Return Type	Method Name	Description
	<code>getInteractiveEventListener()</code>	This method is called by the game to notify this robot about interactive events, i.e. keyboard and mouse events.
String	<code>getName()</code>	Returns the robot's name.
int	<code>getNumRounds()</code>	Returns the number of rounds in the current battle.
int	<code>getOthers()</code>	Returns how many opponents that are left in the current round.
	<code>getPaintEventListener()</code>	This method is called by the game to notify this robot about painting events.
double	<code>getRadarHeading()</code>	Returns the direction that the robot's radar is facing, in degrees.
Runnable	<code>getRobotRunnable()</code>	This method is called by the game to invoke the <code>run()</code> method of your robot, where the program of your robot is implemented.
int	<code>getRoundNum()</code>	Returns the current round number (0 to <code>getNumRounds() - 1</code>) of the battle.
long	<code>getTime()</code>	Returns the game time of the current round, where the time is equal to the current turn in the round.
double	<code>getVelocity()</code>	Returns the velocity of the robot measured in pixels/turn.
double	<code>getWidth()</code>	Returns the width of the robot measured in pixels.
double	<code>getX()</code>	Returns the X position of the robot. (0,0) is at the bottom left of the battlefield.
double	<code>getY()</code>	Returns the Y position of the robot. (0,0) is at the bottom left of the battlefield.
void	<code>onBattleEnded(BattleEndedEvent event)</code>	This method is called after the end of the battle, even when the battle is aborted.
void	<code>onBulletHit(BulletHitEvent event)</code>	This method is called when one of your bullets hits another robot.
void	<code>onBulletHitBullet(BulletHitBulletEvent event)</code>	This method is called when one of your bullets hits another bullet.
void	<code>onBulletMissed(BulletMissedEvent event)</code>	This method is called when one of your bullets misses, i.e. hits a wall.
void	<code>onDeath(DeathEvent event)</code>	This method is called if your robot dies.
void	<code>onHitByBullet(HitByBulletEvent event)</code>	This method is called when your robot is hit by a bullet.
void	<code>onHitRobot(HitRobotEvent event)</code>	This method is called when your robot collides with another robot.
void	<code>onHitWall(HitWallEvent event)</code>	This method is called when your robot collides with a wall.

RoboCode- und Java-Dokumentation lesen



Weiter unten auf der Seite sind die einzelnen Methoden noch genauer beschrieben.

Interessant sind auch die „See Also“-Einträge: z.B. in der Methode `getHeading()` Methode wird auf `getGunHeading()` und `getRadarHeading()` verwiesen.

The screenshot shows a web browser window displaying the Java documentation for the RoboCode API. The browser's address bar shows the file path: `file:///home/riess/robocode/javadoc/index.html`. The page content is organized into sections for different methods. A red circle highlights the `getHeading()` method section. The `getHeading()` section includes the following text:

```
public double getHeading()

Returns the direction that the robot's body is facing, in degrees. The value returned will be between 0 and 360 (is excluded).

Note that the heading in Robocode is like a compass, where 0 means North, 90 means East, 180 means South, and 270 means West.

Returns:
    the direction that the robot's body is facing, in degrees.

See Also:
    getGunHeading\(\), getRadarHeading\(\)
```

Below the `getHeading()` section, other methods are listed, including `getHeight()`, `getWidth()`, and `getName()`. The `getHeight()` section includes:

```
public double getHeight()

Returns the height of the robot measured in pixels.

Returns:
    the height of the robot measured in pixels.

See Also:
    getWidth\(\)
```

The `getWidth()` section includes:

```
public double getWidth()

Returns the width of the robot measured in pixels.

Returns:
    the width of the robot measured in pixels.

See Also:
    getHeight\(\)
```

The `getName()` section includes:

```
public String getName()
```

On the left side of the browser window, there is a navigation pane with a list of classes and packages. The `getHeading()` method is highlighted in the main content area.



Andere Roboter angucken!

- Zum Anfang kann man auch schauen, was andere Roboter machen:
 - Im Installationsverzeichnis unter `robots/sample` stöbern
 - Im RoboCode-Wiki im Internet
- Beachte: Wir schreiben einen `AdvancedRobot`
(-> im Code steht `Hard extends AdvancedRobot`).
Einige Beispielroboter sind jedoch nur `Robots` und benutzen leicht unterschiedliche Befehle.
(-> z.B. `Walls extends Robot`)



In der Dokumentation recherchieren

- Ein paar Aspekte an dem Prototypen sind... suboptimal gelöst sind.
- Bisher haben wir **beobachtet**, dass der Prototyp Schwächen hat.
- Sehen reicht nicht: wir müssen die Aktionen im Code **analysieren**,
- dann Verbesserungsmöglichkeiten in der Arena **recherchieren**,
- eine Idee theoretisch **ausknobeln**,
- und **implementieren**.

- Und danach?
(.oO Ab in die Arena,
beten dass die Arbeit sich gelohnt hat,
irgendwas funktioniert nicht
(kann nicht; Dinge funktionieren nie einfach so),
also wieder zum Anfang (beobachten, analysieren...))

Zeichnungen helfen

```
49  /**
50   * onHitWall: What to do when you hit a wall
51   */
52  public void onHitWall(HitWallEvent e) {
53      // Replace the next line with any behavior you would like
54      back(20);
55  }
56 }
```

- Beispiel: Angenommen, wir bauen eine Kollisionsbehandlung für Wände ein:
Wenn der Roboter eine Wand trifft, fährt er ein Stück zurück
- Sehr oft hat man während dem Programmieren nur einen Teil der möglichen Fälle im Kopf

[.oO und wenn er die Wand im Rückwärtsgang rammt?]



Au Backe, nicht auszudenken.

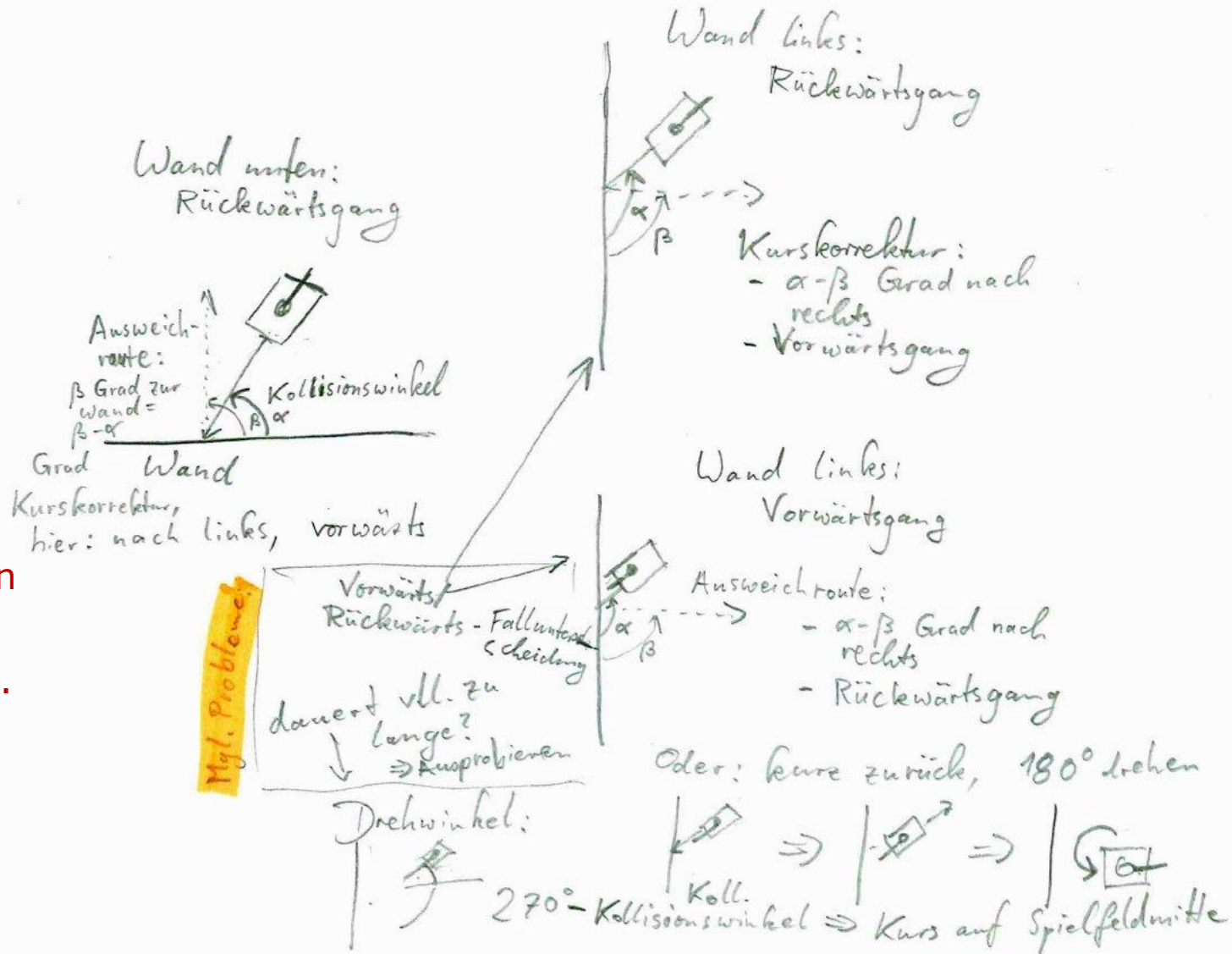
Christian Riess, Eva Eibenberger



Zeichnungen helfen

Auf dem Papier findet man eine Lösung oft einfacher als am Rechner.

Außerdem fühlt man sich dabei ein bisschen wie Daniel Düsentrieb.





In der Dokumentation recherchieren

- Ein paar Aspekte an dem Prototypen sind... suboptimal gelöst sind.
 - Bisher haben wir **beobachtet**, dass der Prototyp Schwächen hat.
 - Sehen reicht nicht: wir müssen die Aktionen im Code **analysieren**,
 - dann Verbesserungsmöglichkeiten in der Arena **recherchieren**,
 - eine Idee theoretisch **ausknobeln**,
 - und **implementieren**.
-
- Und danach?
 - (.oO Ab in die Arena,
beten dass die Arbeit sich gelohnt hat,
irgendwas funktioniert nicht
(kann nicht; Dinge funktionieren nie einfach so),
also wieder zum Anfang (beobachten, analysieren...)



Implementieren: Beispiel in run()

- Hier sind beispielsweise zwei „Strategien“ in der Fallunterscheidung:

1. Wenn „`nothingToDo`“: neu orientieren mit der Radardrehung
2. Ansonsten (d.h. wenn man den Gegner gesehen hat) nur überprüfen wie lange das her ist. Zu lange? Dann `nothingToDo` auf `true` setzen.

```

23 // repeat forever (i.e. execute the content of the curly braces while
24 // the condition in brackets is true - but "true" is always true,
25 // Thus, we repeat forever...)
26 while(true) {
27     System.out.println("round number = " + getTime() + ", my health: " + getEnergy());
28
29     if (nothingToDo) {
30         setTurnRadarRight(360);
31         setTurnRight(20);
32         setAhead(30);
33         waitFor(new TurnCompleteCondition(this)); // executes the commands
34         // above, and remains here until the 20 degrees turn is finished.
35     } else {
36         // check if we did not see the enemy for a while
37         if (getTime() - lastScanTime > 10) {
38             nothingToDo = true;
39         }
40         execute(); // keep executing the selected commands
41     }
42 }

```

An dieser Stelle könnte man einbauen, dass man auf den Gegner zufährt – schließlich hat man ihn ja kürzlich gesehen!



Implementieren: Beispiel Ereignis

- Zur Ereignisbehandlung muss man sich oft den aktuellen Zustand merken:

```
public class Hard extends AdvancedRobot {
    int fahreVorwaerts; // im Ereignis muessen wir den Zustand kennen, in dem wir
                        // unterbrochen wurden

    public void run() {
        while(true) {
            if (getDistanceRemaining() < 1) {
                if (fahreVorwaerts == 1) {
                    setBack(100);
                } else {
                    setAhead(100);
                }
                fahreVorwaerts = Math.abs(fahreVorwaerts-1); // Zustand umdrehen: 1->0, 0->1
            }
            execute();
        }
    }

    public void onHitWall(HitWallEvent e) {
        if (fahreVorwaerts == 1) { // wenn wir hier hineinspringen, haben wir jetzt die
                                // Information, was wir zuletzt gemacht haben.

            setBack(150);
        } else {
            setAhead(150);
        }
        execute();
    }
}
```



Roboter in das EST einstellen

- Roboter = „Übungsabgabe“
- Die „Übung“ ist das ganze Semester offen
- Für EST-Uploads müssen alle Einreichungen den selben Namen haben; daher: Roboter in ein .zip-File verpacken:

```
sichries@fau00j> zip my_robot.zip robocode/robots/chriess/RockHard.java
```

- Jeder Roboter im EST muss einen eindeutigen Namen haben!
- M.a.W.: Wer ihr zu zweit oder dritt arbeitet, und jeder seine Variante hochladen will, muss der Roboter individuell benannt sein, z.B. RockHard2 oder RockHardest.
- Wenn man seinen eigenen Roboter im EST erneut einreicht, kann man den Namen beibehalten



Turniere spielen

- Ab Montag werden die Roboter jede Nacht aus dem EST heruntergeladen und compiliert.
- Dann wird ein Turnier gespielt.
- Die Turnierergebnisse werden tagesaktuell unter <http://www5.cs.fau.de/fileadmin/lectures/SS11/robocode> veröffentlicht (zugreifbar aus dem Uninetz)

Good to go!



- Viel Erfolg, und viel Spaß!



Bild von 3D Realms

Let's rock!