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…

```
faui00j [~]> ~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

faui00j [~]> ~sichries/createRobot.sh 0 rock Hard
created robot /home/cip/2001/sichries/robocode/robots/rock/Hard.java -> happy editing!
faui00j [~]> cd ~/robocode/robots/rock/
faui00j [~/robocode/robots/rock]> scite Hard.java
```

Package-Name:
- Kleinbuchstaben
- Keine Sonderzeichen
- Keine Umlaute

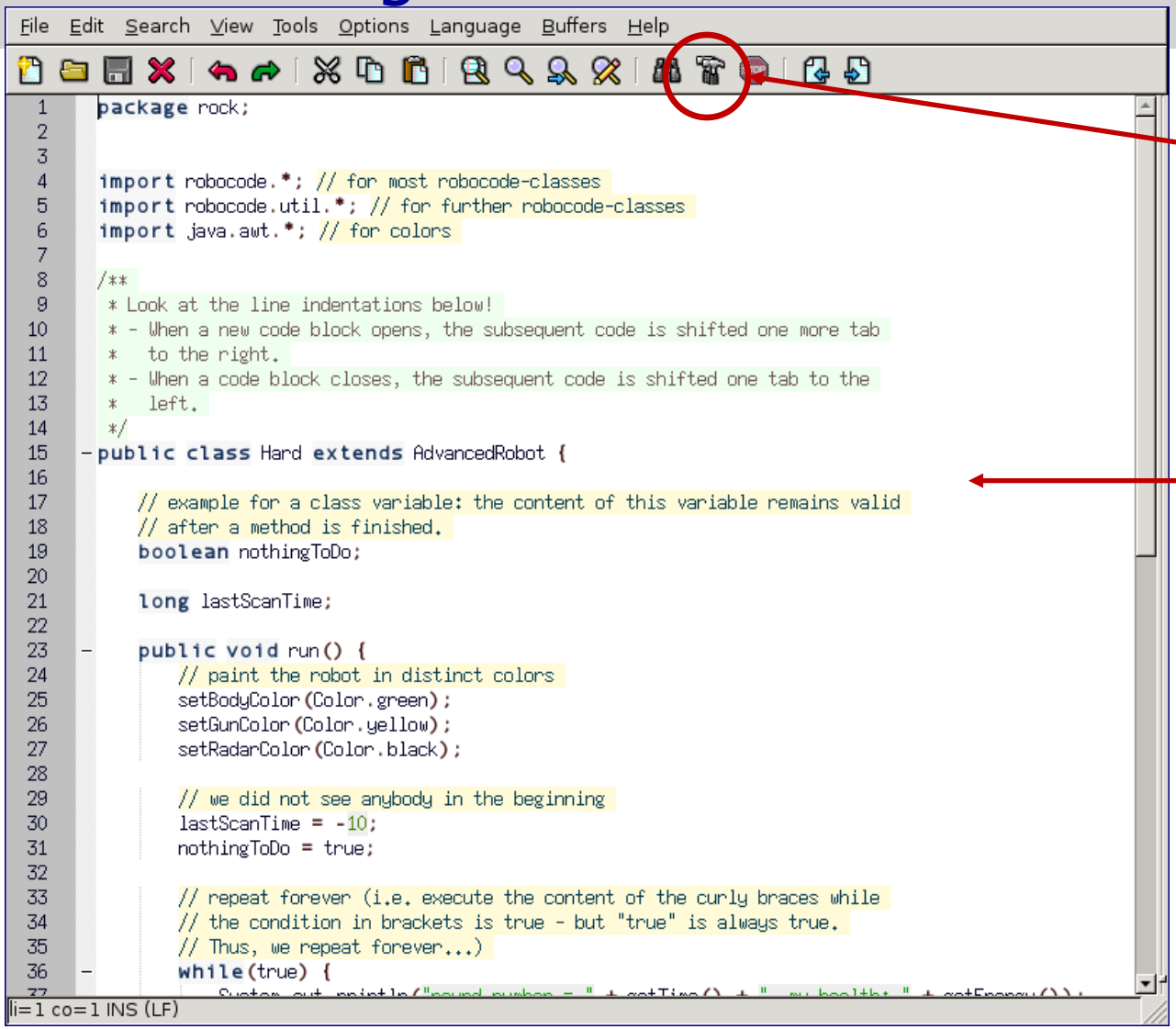
Robotername:
- Großer Anfangsbuchstabe
- Keine Sonderzeichen
- Keine Umlaute

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



Übersetzt den Roboter (F7)

Die Arena starten: „Tools“->”Go” (F5)

Codefenster

Christian Riess, Eva Eibenberger

Programmierfehler finden:

- Syntaktische Fehler zeigt der Compiler auf:

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 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     booleam 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());
38
```

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

li=1 co=1 INS (LF)

Dateiname, Zeilennummer, Problembeschreibung: „Symbol ist unbekannt“

Welches Symbol? „booleam“

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
      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
```

```
>make
javac -classpath /usr/lib/jvm/java-
>Exit code: 0
```

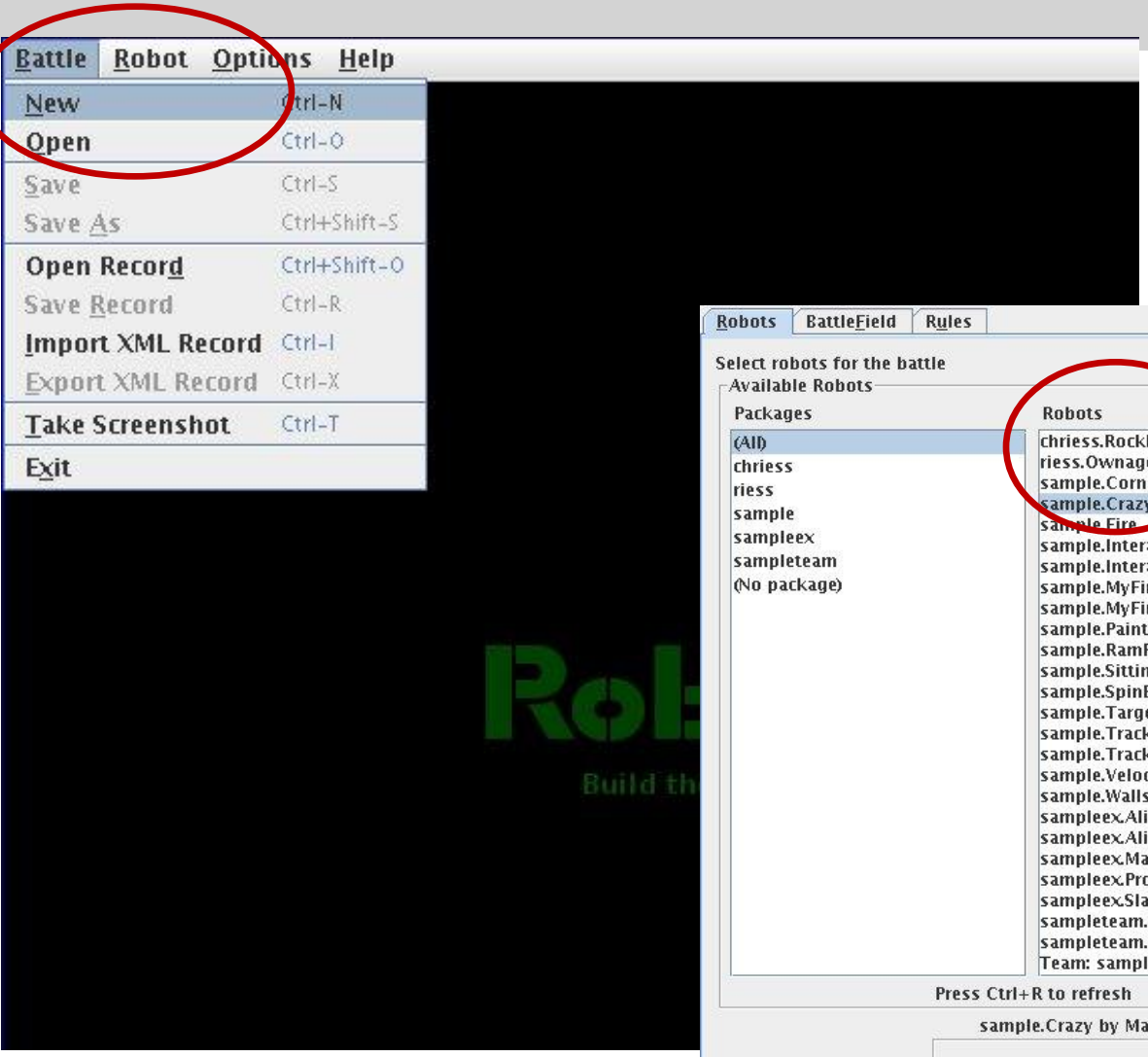
li=1 co=1 INS (LF)

Christian Riess, Eva Eibenberger

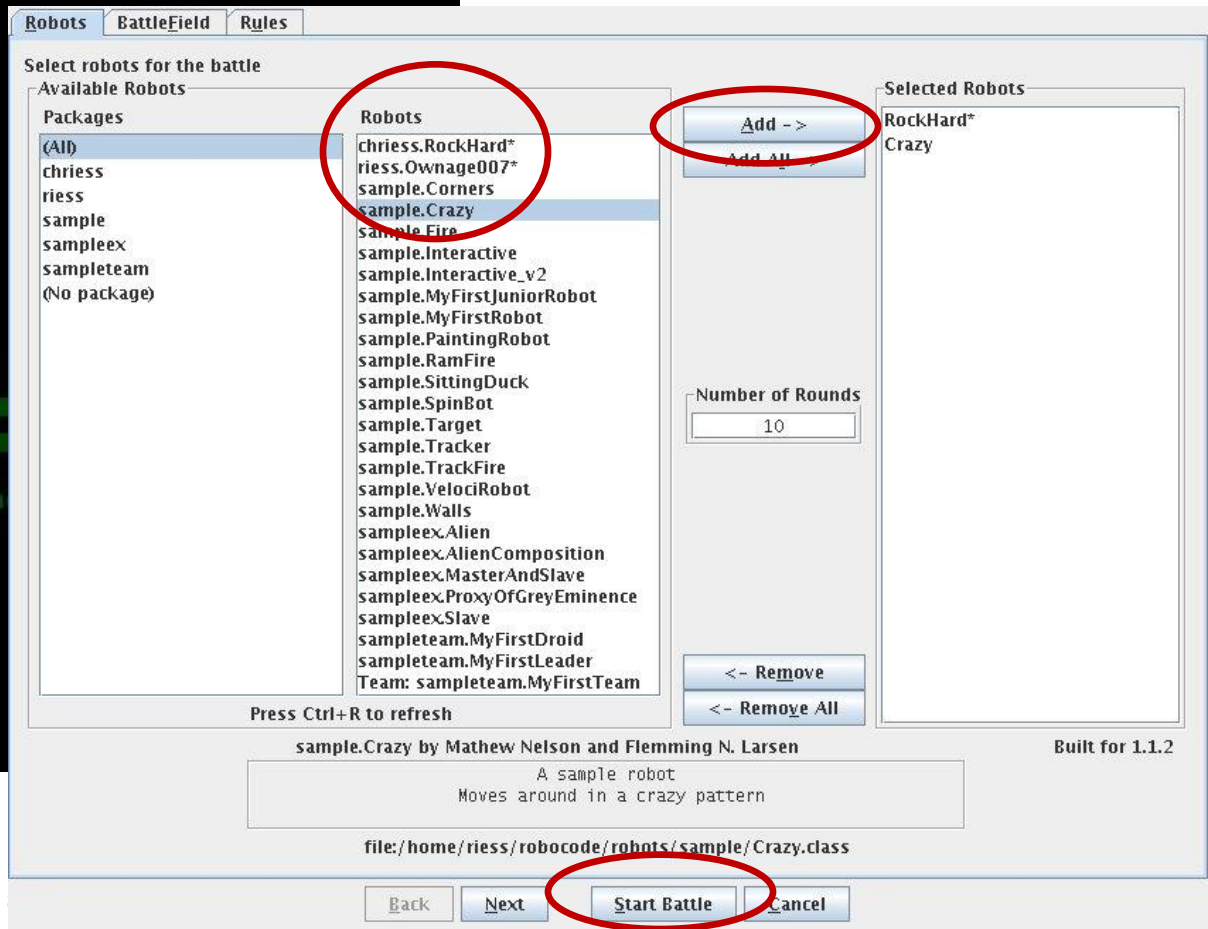
```java
package rock;
import robocode.*; // for most robocode-classes
import robocode.util.*; // for further robocode-classes
import java.awt.*; // for colors

public class Hard extends AdvancedRobot {

    // example for a class variable: the content of this variable remains valid
    // after a method is finished.
    boolean nothingToDo;
    long lastScanTime;

    public void run() {
        // paint the robot in distinct colors
        setBodyColor(Color.green);
        setGunColor(Color.yellow);
        setRadarColor(Color.black);

        // we did not see anybody in the beginning
        lastScanTime = -10;
        nothingToDo = true;

        // repeat forever (i.e. execute the content of the curly braces while
        // the condition in brackets is true - but "true" is always true.
        // Thus, we repeat forever...)
        while(true) {
            System.out.println("round number = " + getTime() + ", my health: " + g

            if (nothingToDo) {
                setTurnRadarRight(360);
                setTurnRight(20);
                setAhead(30);
                waitFor(new TurnCompleteCondition(this)); // executes the command
                 // above, and remains here until the 20 degrees turn is finished.
            } else {
                // check if we did not see the enemy for a while
                if (getTime() - lastScanTime > 10) {
                    nothingToDo = true;
                }
                execute(); // keep executing the selected commands
            }
        }
    }

    // example for an event: The execution of the code jumps here when we see a robot in the radar
    public void onScannedRobot(ScannedRobotEvent e) {
        setTurnRight(e.getBearing());
        waitFor(new TurnCompleteCondition(this));
        setFire(1.0);
        nothingToDo = false;
```

- 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.



…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…)

```
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
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 }
```

"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.

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).

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?

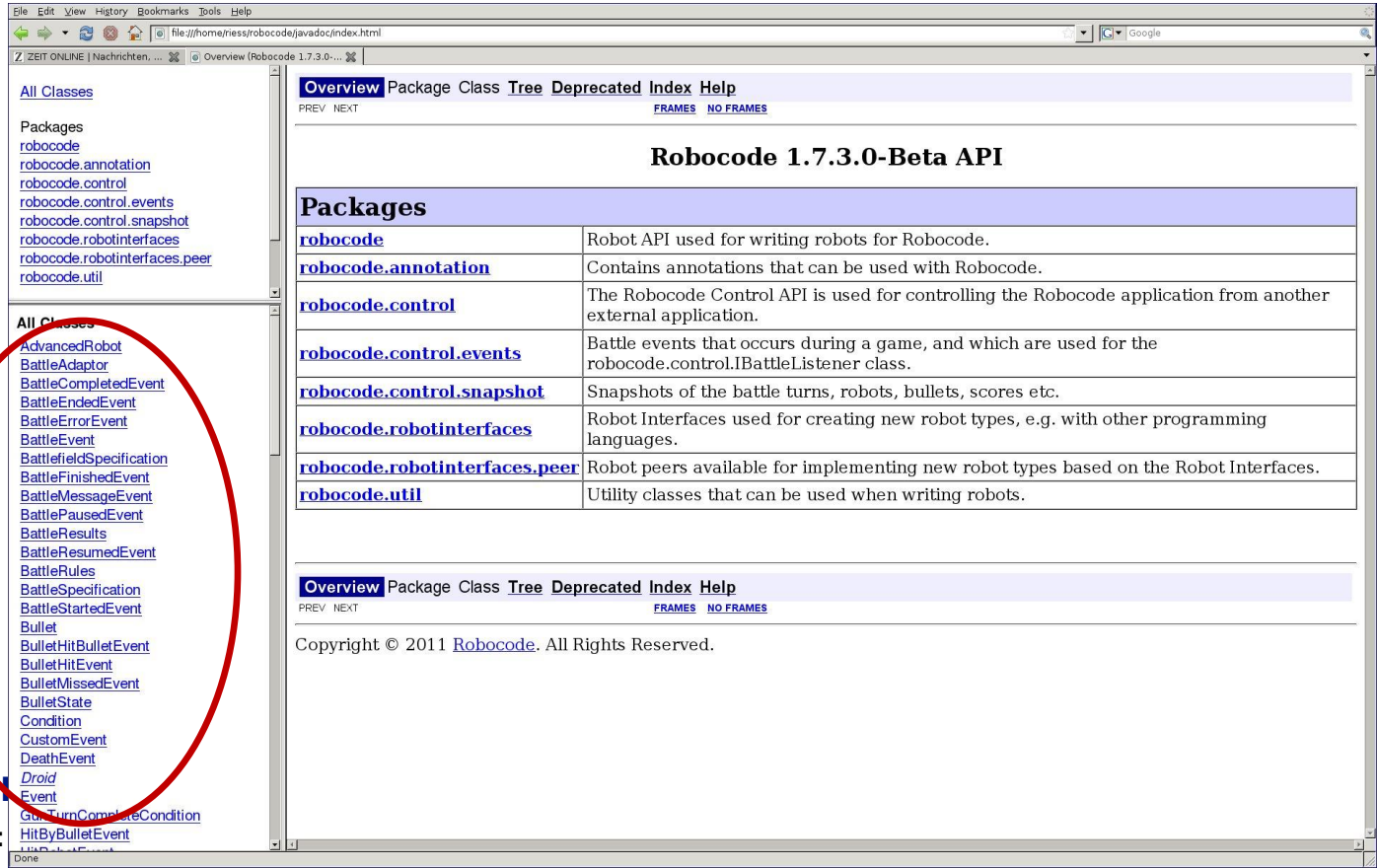
  (.oO Ab in die Arena,
  beten dass die Arbeit sich gelohnt hat,
  irgendwas funktioniert nicht
  (kann nicht; Dinge funktionieren nie einfach so),
  also wieder zum Anfang (beobachten, analysieren…)

RoboCode- und Java-Dokumentation lesen

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

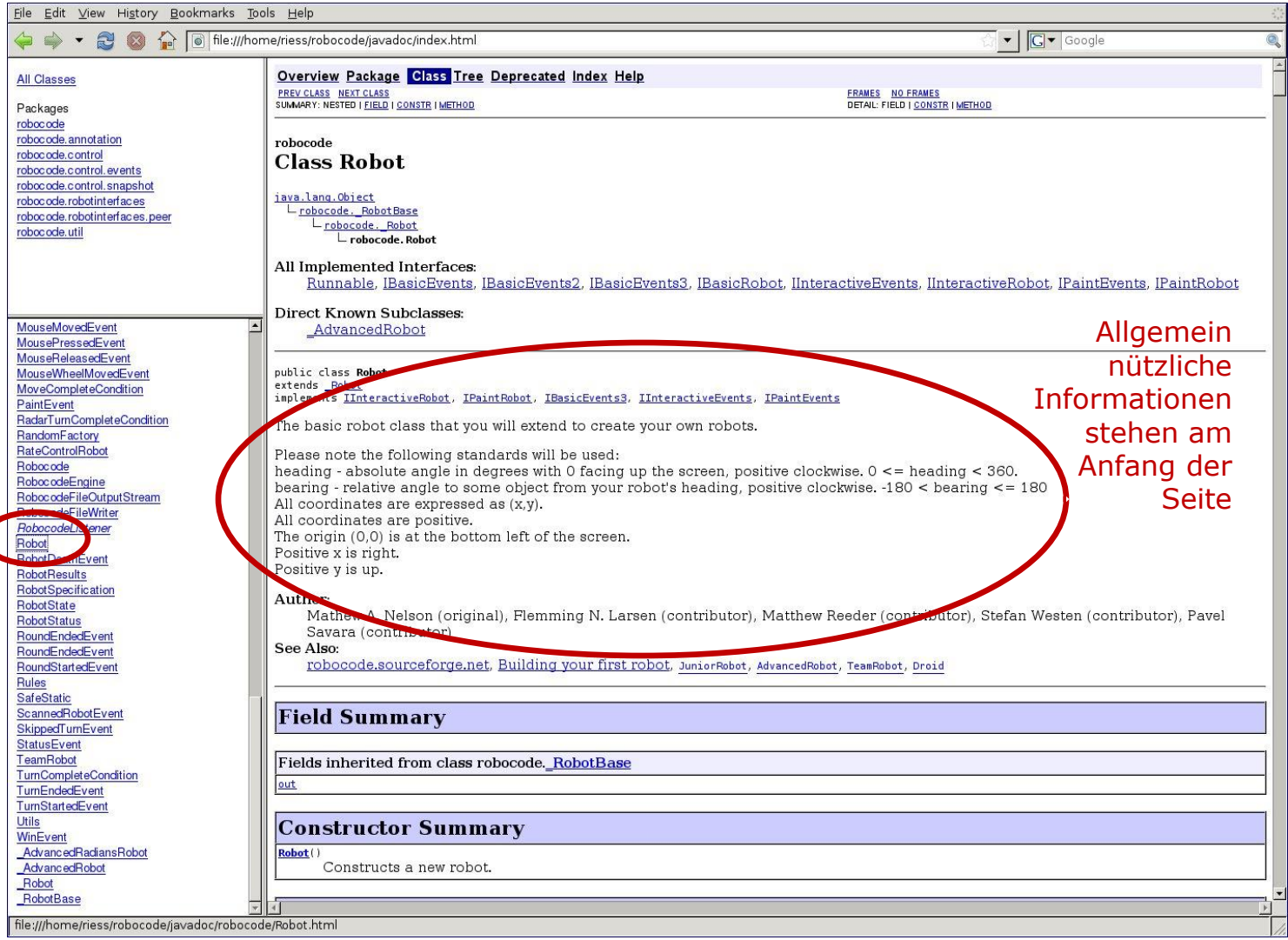
(.oO HTML: am besten im Browser öffnen)

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



RoboCode- und Java-Dokumentation lesen

Klasse `AdvancedRobot` auswählen; oder für elementarere Befehle `Robot`



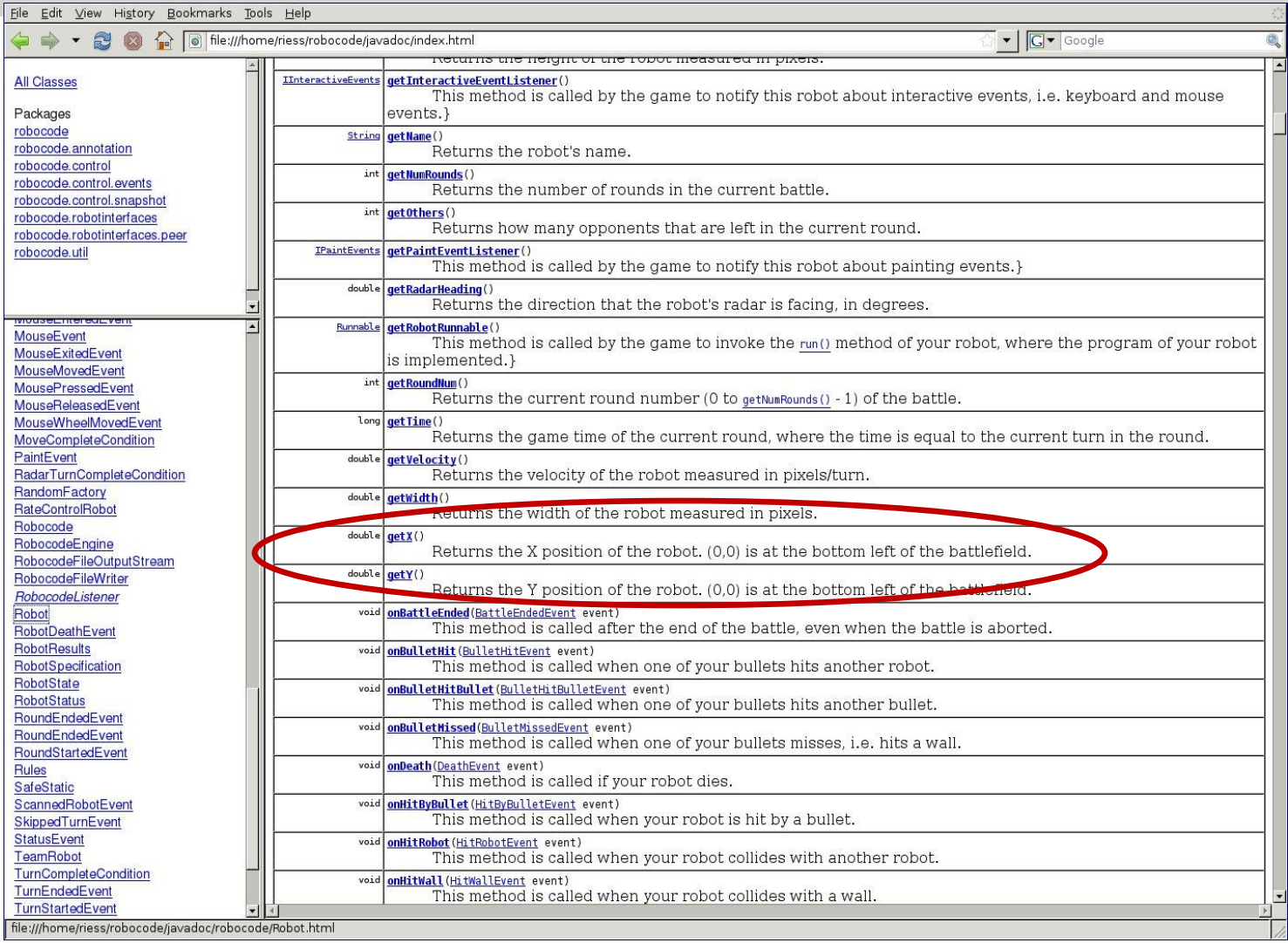
Allgemein nützliche Informationen stehen am Anfang der Seite

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."

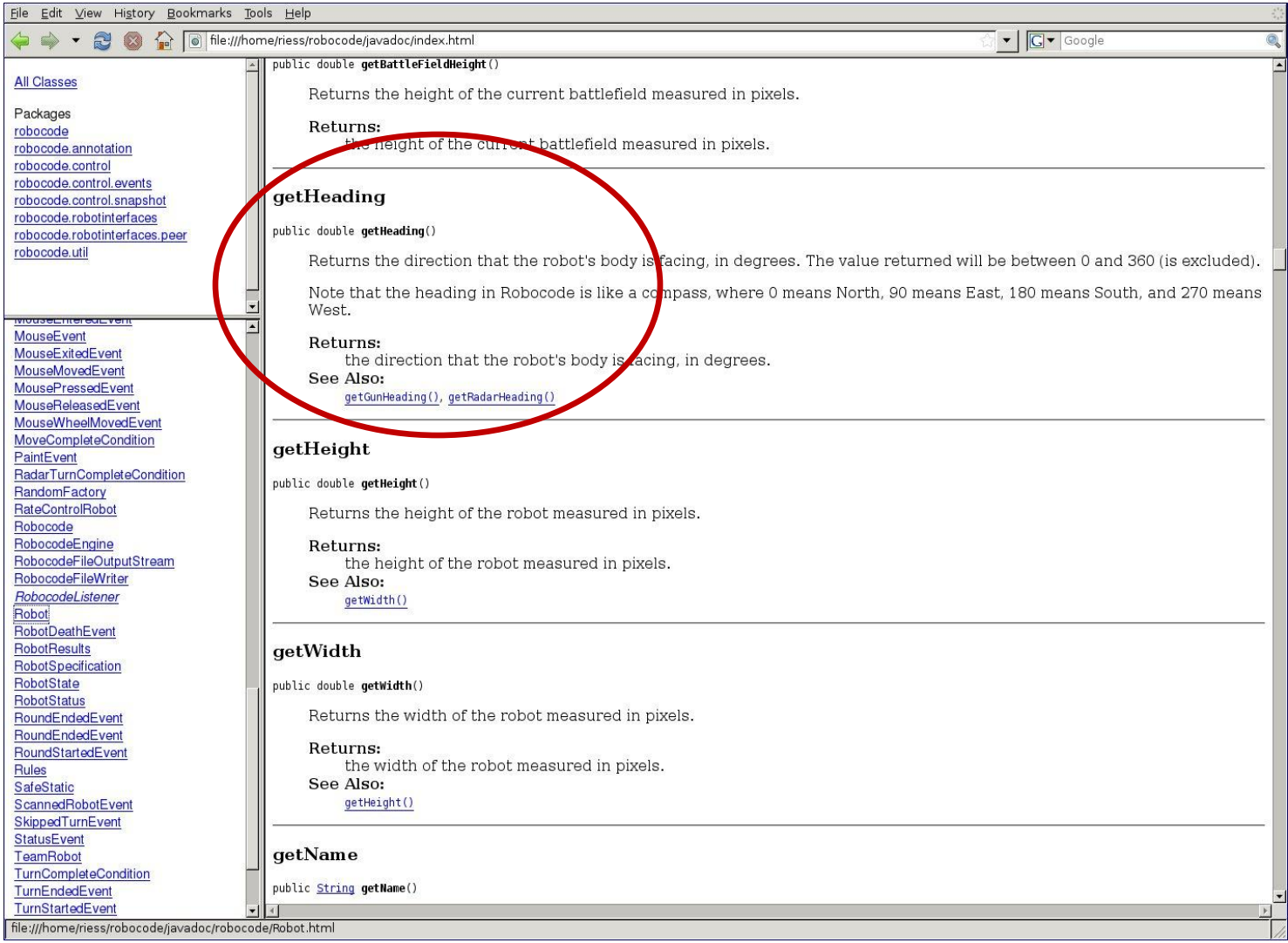
Die Methode heißt `getX()`, und wenn man sie aufruft, bekommt man den X-Wert als `double`.



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.



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

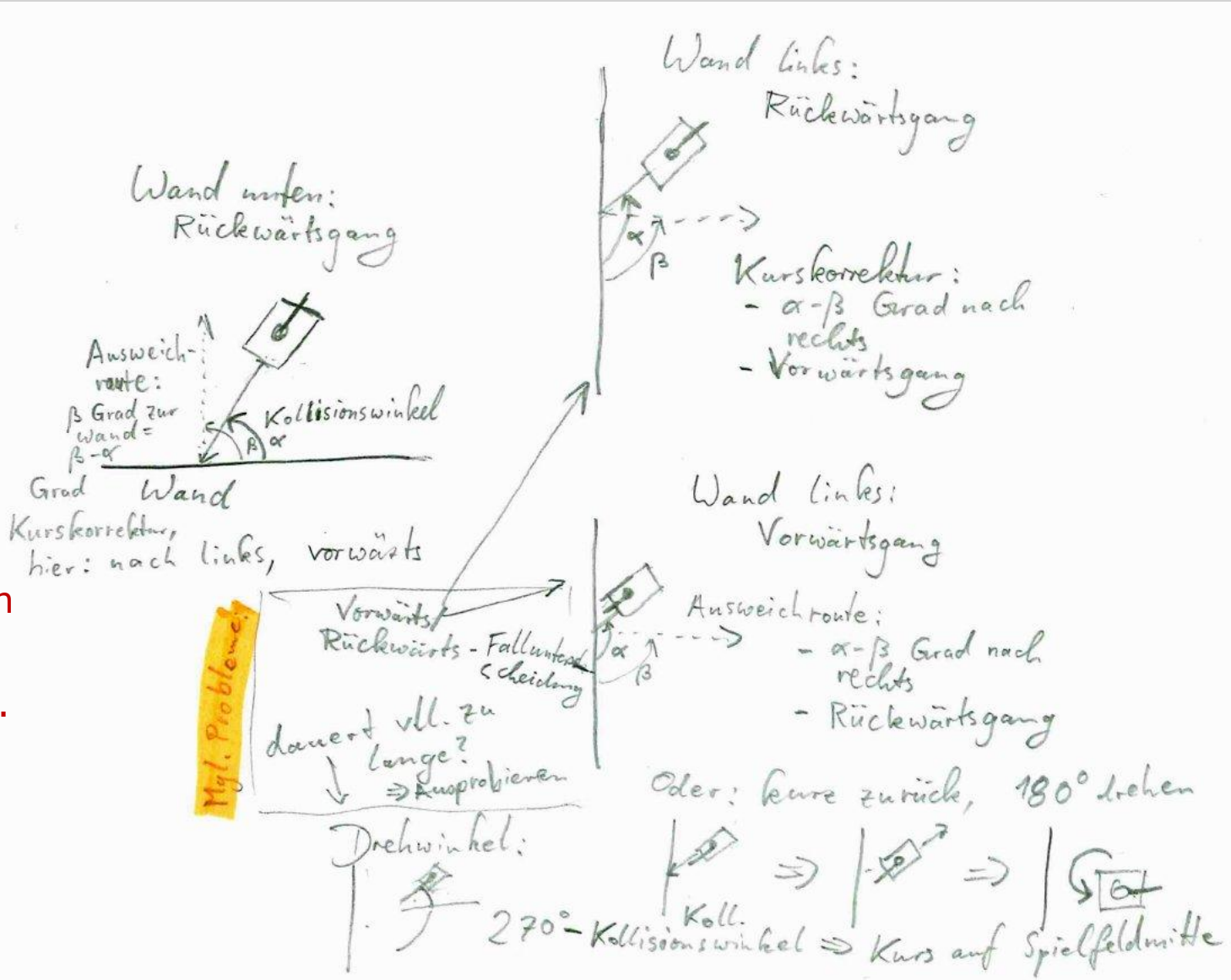
[.oO und wenn er die Wand im Rückwärtsgang rammt?]

Au Backe, nicht auszudenken.

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:

```
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          }
```

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.

   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@faui00j> 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!