

Informatik I: Einführung in die Programmierung

21. OOP: RoboRally als Beispiel

Albert-Ludwigs-Universität Freiburg



Bernhard Nebel

12. Dezember 2014

1 Motivation



Motivation

- Die Spielregeln
- Eine OOP-Analyse
- Exkurs: Mehrfachvererbung
- Programm-entwurf
- Ein kleiner Test
- Zusammenfassung

12. Dezember 2014

B. Nebel – Info I

3 / 109

Motivation



Motivation

- Die Spielregeln
- Eine OOP-Analyse
- Exkurs: Mehrfachvererbung
- Programm-entwurf
- Ein kleiner Test
- Zusammenfassung

- OOP kann man an kleinen Beispielen erklären.
- Interessant wird es aber eher bei größeren Beispielen.
- Da sieht man dann etwas vom OOP-Entwurf.
- Multi-Agenten-Systeme (aus der KI) kann man gut nutzen, da sie ja inhärent aus selbständig agierenden Einheiten bestehen
- Einfacher ist vielleicht ein Spiel, bei dem es kleine Roboter gibt
- Außerdem müssen wir ja auch noch ein Weihnachtsgeschenk basteln ...

12. Dezember 2014

B. Nebel – Info I

4 / 109

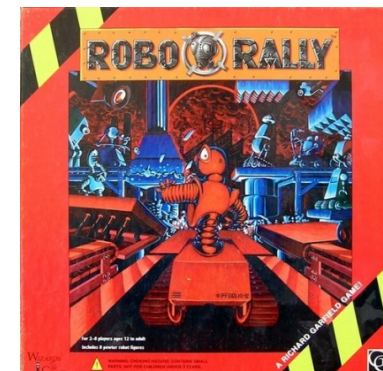
RoboRally



Motivation

- Die Spielregeln
- Eine OOP-Analyse
- Exkurs: Mehrfachvererbung
- Programm-entwurf
- Ein kleiner Test
- Zusammenfassung

- RoboRally ist ein Brettspiel für 2-8 Personen entworfen von Robert Garfield, herausgegeben von *Wizards of the Coast*, 1994.
- Auszeichnung als bestes Science Fiction/Fantasy Spiel 1994



12. Dezember 2014

B. Nebel – Info I

5 / 109

Die Story

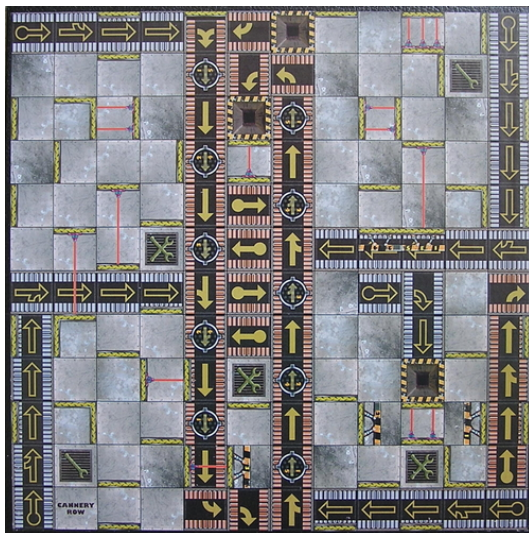
- Als einer von vielen Supercomputern in einer vollautomatischen Fabrik haben Sie es geschafft. Sie sind brillant, leistungsstark, hochentwickelt und... gelangweilt.
- Also machen Sie sich Freude auf Kosten der Fabrik.
- Mit den anderen Computern programmieren Sie **Fabrikroboter** und lassen sie gegeneinander antreten in wilden, zerstörerischen **Rennen** über die Fabrikflure. Seien Sie der erste, der die **Checkpoints** in richtiger Reihenfolge anfährt und **gewinnen** Sie alles: die Ehre, den Ruhm und Neid Ihrer mitstreitenden Computer.
- Aber zuerst muss Ihr Roboter an **Hindernissen** wie Industrielaser und Fließbändern vorbei und natürlich an den gegnerischen Robotern.
- Aber Vorsicht: Einmal **programmiert**, lässt sich so ein Roboter nicht mehr stoppen...

Motivation
Die Spielregeln
Eine OOP-Analyse
Exkurs: Mehrfachvererbung
Programm-entwurf
Ein kleiner Test
Zusammenfassung

2 Die Spielregeln

Motivation
Die Spielregeln
Eine OOP-Analyse
Exkurs: Mehrfachvererbung
Programm-entwurf
Ein kleiner Test
Zusammenfassung

Ein Spielbrettbeispiel



Motivation
Die Spielregeln
Eine OOP-Analyse
Exkurs: Mehrfachvererbung
Programm-entwurf
Ein kleiner Test
Zusammenfassung

Die Bestandteile des Spiels

- 8 verschiedene **Spielsteine** – die Roboter
- 6 verschiedene **Spielbretter**, die auch zusammen gelegt werden können
- 84 verschiedene **Programmierungskarten**, die Befehle wie *1 Feld vorwärts*, *2 Felder vorwärts*, *Linksrotation* usw. sowie **Prioritäten** enthalten
- 26 **Optionskarten**, und
- zusätzliche **Markierungen** und **Zähler**, um die Ziele festzulegen und den Zustand der Roboter abzubilden

Motivation
Die Spielregeln
Eine OOP-Analyse
Exkurs: Mehrfachvererbung
Programm-entwurf
Ein kleiner Test
Zusammenfassung

- Es wird das Spielbrett ausgewählt, die nummerierten **Checkpoints** gesetzt (die nacheinander zu besuchen sind) und die Roboter auf die Startfelder gesetzt.
- Jetzt wird in jeder Runde folgendes gemacht:
 - 1 Jeder Spieler erhält verdeckt 9 Programmierkarten (außer der Roboter ist **abgeschaltet**).
 - 2 Davon wählt er fünf zur **Programmierung** aus, die er verdeckt in einer Reihe „in die **Register** 1–5“ hinlegt.
 - 3 Jetzt muss man ggfs. eine **Abschaltung** ankündigen.
 - 4 Dann werden die fünf so genannten **Registerphasen** 1–5 ausgeführt, in denen die Roboter bewegt und durch die Fabrikelemente und andere Roboter herum geschubst werden.
 - 5 Steht der Roboter am Ende einer Runde auf einem Reparaturfeld, werden Schäden **repariert**.
- Man hat **gewonnen**, wenn man am Ende einer Registerphase den **letzten Checkpoint** erreicht hat.






Motivation
Die Spielregeln
Eine OOP-Analyse
Exkurs: Mehrfachvererbung
Programm-entwurf
Ein kleiner Test
Zusammenfassung

- 1 Es werden die Karten aller Spieler eines Registers umgedreht.
- 2 Die Karten werden absteigend nach ihrer **Priorität** geordnet.
- 3 Beginnend mit der höchsten Priorität, werden die Roboter entsprechend ihrer Programmierkarte **bewegt**.
- 4 Danach wirken jeweils die **Fabrikelemente** auf die Roboter ein (inkl. Laser) und die Roboter schießen mit ihrem **Laser** auf andere Roboter.
- 5 Steht ein Roboter jetzt auf einem Checkpoint oder Reparaturfeld, darf er das Feld markieren (mit dem **Archivkopie**-Marker) bzw. hat **gewonnen**, wenn er das Zielfeld erreicht hat.

Motivation
Die Spielregeln
Eine OOP-Analyse
Exkurs: Mehrfachvererbung
Programm-entwurf
Ein kleiner Test
Zusammenfassung

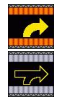
- Es gibt folgende Karten:
 - 1, 2, oder 3 Felder vorwärts,
 - 1 Feld rückwärts,
 - links oder rechts 90° Drehung,
 - 180° Drehung.
- Der Roboter bewegt sich schrittweise auf dem Spielfeld.
- Fällt er dabei in eine **Grube**, ist er zerstört (er hat allerdings 3 Leben!).
- Fährt er gegen eine **Wand**, bleibt er stehen.
- Fährt er gegen einen **anderen Roboter**, wird dieser auf das Nachbarfeld geschubst. Steht der andere Roboter allerdings vor einer Wand, bleiben beide Roboter stehen. Das gilt auch für Schlangen von Robotern.
- Üben wir das mal: http://www.wizards.com/avalonhill/robo_demo/robodemo.asp

Motivation
Die Spielregeln
Eine OOP-Analyse
Exkurs: Mehrfachvererbung
Programm-entwurf
Ein kleiner Test
Zusammenfassung

	Offener Bereich: Hier kann sich der Roboter frei bewegen.
	Wand: Hier wird der Roboter (und der Laser) gestoppt.
	Fallgrube (Pit): Kommt der Roboter auf dieses Feld, fällt er in die Grube und ist zerstört. Dies gilt auch, wenn der Roboter das Spielfeld verlässt.
	Förderband (Conveyor belt): Hier wird der Roboter ein Feld in die angezeigte Richtung transportiert.
	Express-Förderband: Der Roboter wird 2 Felder transportiert.

Motivation
Die Spielregeln
Eine OOP-Analyse
Exkurs: Mehrfachvererbung
Programm-entwurf
Ein kleiner Test
Zusammenfassung

Fabrikelemente (2)



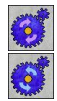
Drehendes (Express-)Förderband: Der Roboter wird in die angegebene Richtung gedreht, wenn er von einem anderen Förderbandfeld kommt.

Motivation
Die Spielregeln



Schieber (Pusher): Schiebt den Roboter auf das Nachbarfeld, falls *aktiv* (während der angegebenen Registerphasen).

Eine OOP-Analyse



Drehscheibe (Gear): Der Roboter wird um 90° in die angegebene Richtung gedreht.

Exkurs: Mehrfachvererbung
Programm-entwurf



Schrottpresse (Crusher): Falls die Presse *aktiv* ist (während der angegebenen Registerphasen), wird der Roboter, der auf diesem Feld steht, zerstört.

Ein kleiner Test
Zusammenfassung

Fabrikelemente (3)



Es wird ein Laserstrahl abgeschossen, der alle Roboter auf dem Weg beschädigt, falls sie nicht hinter einer Wand oder einem anderen Roboter stehen.

Motivation
Die Spielregeln



Reparaturfeld: Hier wird am Ende jeder Registerphase eine Archivkopie abgelegt. Am Ende einer Runde wird entsprechend der Anzahl der Schraubenschlüssel Schadenspunkte reduziert.

Eine OOP-Analyse

Exkurs: Mehrfachvererbung

Programm-entwurf

Ein kleiner Test

Zusammenfassung



Checkpoints: Diese müssen in der nummerierten Reihenfolge angelaufen werden. Auch hier wird am Ende einer Registerphase eine Archivkopie abgelegt.

Fabrikablauf (Schritte)



- 1 Expressförderbänder bewegen sich um ein Feld.
- 2 Expressförderbänder und Förderbänder bewegen sich um ein Feld. Kommt es dabei zu Kollisionen zwischen Robotern, werden diese nicht bewegt.
- 3 Schieber werden aktiv.
- 4 Drehscheiben drehen sich.
- 5 Schrottpressen werden aktiv.
- 6 Die Standlaser und die Robotlaser (zielen nach vorne) werden aktiviert.
- 7 Danach werden die Checkpoints und Reparaturfelder bearbeitet.

Motivation
Die Spielregeln

Eine OOP-Analyse

Exkurs: Mehrfachvererbung

Programm-entwurf

Ein kleiner Test

Zusammenfassung

Gleich mal ausprobieren mit Passwort GEARHEAD: http://www.wizards.com/avalonhill/robo_demo/robodemo.asp

Beschädigungen



- Bei jedem **Lasertreffer** gibt es einen Schadenspunkt und bei jeder **Wiederbelebung** zwei.
- Bei 10 Schadenspunkten wird der Roboter **zerstört**.
- Für jeden Schadenspunkt gibt es **eine Programmierkarte weniger**.
- Bei mehr als 5 Schadenspunkten werden die Register absteigend von Register 5 **gesperrt**, d.h. die dort liegende Karte bleibt liegen und wird in jeder Runde ausgeführt.
- Schadenspunkte werden auf **Reparaturfeldern** reduziert.
- Abschaltung reduziert die Schadenspunkte auf Null.

Motivation
Die Spielregeln

Eine OOP-Analyse

Exkurs: Mehrfachvererbung

Programm-entwurf

Ein kleiner Test

Zusammenfassung

- Ein Roboter wird zerstört, wenn er
 - 1 in eine Grube fährt,
 - 2 über den Spielfeldrand hinaus fährt,
 - 3 durch eine Schrottpresse zerkleinert wird, oder
 - 4 zu viele Schadenspunkte (10) angesammelt hat.
- Der Roboter wird dann sofort aus dem Spiel genommen.
- In der nächsten Runde darf er dann an der Stelle weitermachen, an der die letzte **Archivkopie** liegt (unter Abzug von zwei Schadenspunkten und einem Lebenspunkt).
- Beginnen zwei Roboter ihren Zug gleichzeitig auf einem Feld, so starten sie **virtuell**. D.h. sie interagieren mit allen Fabrikelementen, aber nicht mit anderen Robotern und deren Lasern. Sie **materialisieren** sich vollständig, wenn sie am Ende einer Runde alleine auf einem Feld stehen.

Motivation

Die Spielregeln

Eine OOP-Analyse

Exkurs: Mehrfachvererbung

Programm-entwurf

Ein kleiner Test

Zusammenfassung

- Außerdem gibt es noch **Optionskarten**, die man statt zwei Reparaturpunkten aufnehmen kann.
- Dieses sind z.B. Waffenmodifikation, zusätzliche Waffen, Neuprogrammierung, Modifikation der Aktion usw.
- Wir wollen diese aber im weiteren erst einmal ignorieren.

Motivation

Die Spielregeln

Eine OOP-Analyse

Exkurs: Mehrfachvererbung

Programm-entwurf

Ein kleiner Test

Zusammenfassung

Motivation

Die Spielregeln

Eine OOP-Analyse

Exkurs: Mehrfachvererbung

Programm-entwurf

Ein kleiner Test

Zusammenfassung

- Wir wollen nicht das gesamte Spiel modellieren (zumindest nicht heute).
- Speziell sollen nur folgende Dinge modelliert werden:
 - die *Ausführung einer Programmierkarte*,
 - *freie Plätze, Gruben, Wände, Drehscheiben, Schieber, Laser, Förderbänder*.
- Rudimentäre Benutzerschnittstelle:
 - Einfache Eingabe der Instruktion als Funktionsaufrufe/Methoden
 - Ausgabe: Ein Trace der Operationen und u.U. das resultierende Spielfeld.
- Allerdings soll die Programmierung so flexibel erfolgen, dass das Programm einfach erweitert werden kann, um das Spiel letztendlich vollständig abzudecken und eine GUI zu integrieren.

Motivation

Die Spielregeln

Eine OOP-Analyse

Exkurs: Mehrfachvererbung

Programm-entwurf

Ein kleiner Test

Zusammenfassung

- Man beginnt damit, die verschiedenen Arten von Objekten zu skizzieren,
- eine Vererbungs- und Enthaltenssein-Struktur zu bestimmen,
- Attribute festzulegen,
- und die Operationen/Methoden festzulegen.
- Dafür gibt es eine Menge von formalen Werkzeugen, z.B. UML, ER-Modelle, ...
- Diese formalen Werkzeuge wollen wir hier aber ignorieren (→ Software-Engineering & Software-Praktikum).
- **Wichtig:** Es soll kein **prozedurales Design** sein, bei dem eine zentrale Instanz sequentiell mit riesigen Fallunterscheidungen das Problem löst, sondern die Objekte sollen **selbständig ihre Aufgaben lösen**, ggfs. durch Delegation!

Motivation
Die Spielregeln
Eine OOP-Analyse
Exkurs: Mehrfachvererbung
Programm-entwurf
Ein kleiner Test
Zusammenfassung

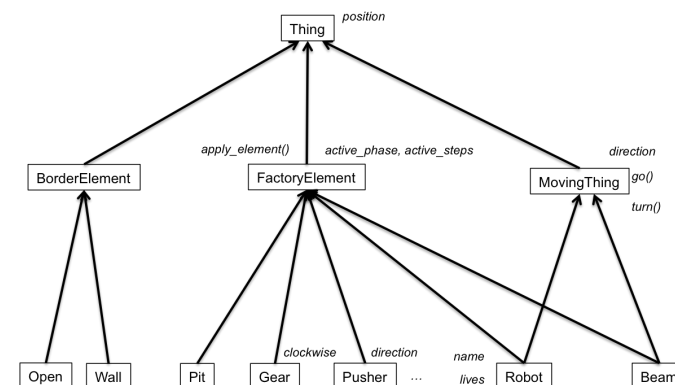
- Wenn sich ein Roboter auf das Nachbarfeld bewegen will, könnte er
 - **überprüfen**, ob eine Wand seinen Weg blockiert und stehen bleiben (prozedurales Design);
 - eine **Methode eines Grenzobjekts** aufrufen, in der dann die entsprechende Bewegung durchgeführt, und der Erfolg dann zurückgegeben wird (OO-Design).
- Im zweiten Fall könnten Erweiterungen, wie halbdurchlässige Wände, Türen, die nur in bestimmten Registerphasen offen sind, usw. als zusätzliche Klassen realisiert werden, ohne dass man an der Robot-Klasse etwas ändern muss.
- Objekteigenschaften in den Klassen lokalisieren, zu denen sie gehören!

Motivation
Die Spielregeln
Eine OOP-Analyse
Exkurs: Mehrfachvererbung
Programm-entwurf
Ein kleiner Test
Zusammenfassung

- **Bewegliche Elemente:** Roboter, Laserstrahlen (als Objekte die fliegen)
- **Fabrikelemente:** freie Plätze, Förderbänder, Drehscheiben, Gruben, Laser, Roboter (da sie Laserstrahlen schießen), Laserstrahlen (da sie auf Roboter einwirken)
- **Grenzelemente:** Freie Übergänge, Spielrandbegrenzung, Wände, ...
- die **Fabrik**,
 - allerdings nur eine,
 - soll sie nach außen hin Services (=Methoden) anbieten?

Motivation
Die Spielregeln
Eine OOP-Analyse
Exkurs: Mehrfachvererbung
Programm-entwurf
Ein kleiner Test
Zusammenfassung

Wir können alle Klassen, die auf dem Spielplan eine Rolle spielen, in einer Hierarchie anordnen und ihre Methoden und Attribute vorläufig festlegen.



Motivation
Die Spielregeln
Eine OOP-Analyse
Exkurs: Mehrfachvererbung
Programm-entwurf
Ein kleiner Test
Zusammenfassung

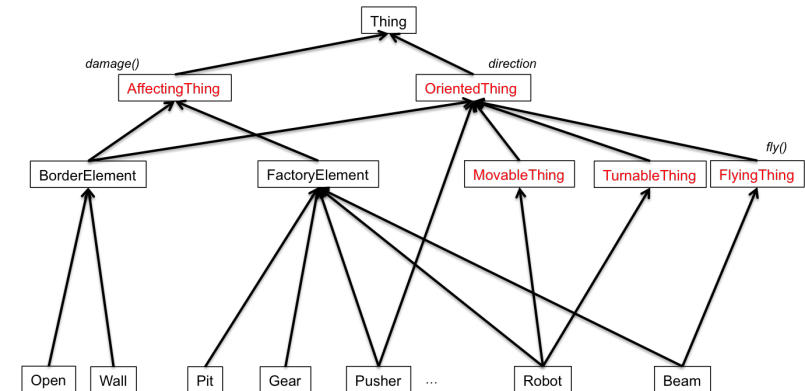
Einige Beobachtungen

- Wir haben Mehrfachvererbung (da Roboter sich bewegende Dinge und Fabrikelemente sind)
- Das geht tatsächlich in Python!
- Wir könnten tatsächlich die Klassenstruktur noch etwas verfeinern:
 - Man kann eine Unterscheidung zwischen orientierten und nicht-orientierten Dingen vornehmen (**OrientedThing**).
 - Es gibt Dinge, die sich drehen können (**TurnableThing**).
 - Manche Dinge können auf Roboter einwirken, indem sie in beschädigen (**AffectingThing**).
 - Wir wollen auch einen Unterschied machen zwischen beweglichen Dingen und fliegenden Dingen (**MovableThing** und **FlyingThing**).

Motivation
Die Spielregeln
Eine OOP-Analyse
Exkurs: Mehrfachvererbung
Programm-entwurf
Ein kleiner Test
Zusammenfassung

Neue Struktur

Neue Klassen sind rot markiert:



Motivation
Die Spielregeln
Eine OOP-Analyse
Exkurs: Mehrfachvererbung
Programm-entwurf
Ein kleiner Test
Zusammenfassung

Die Factory-Klasse

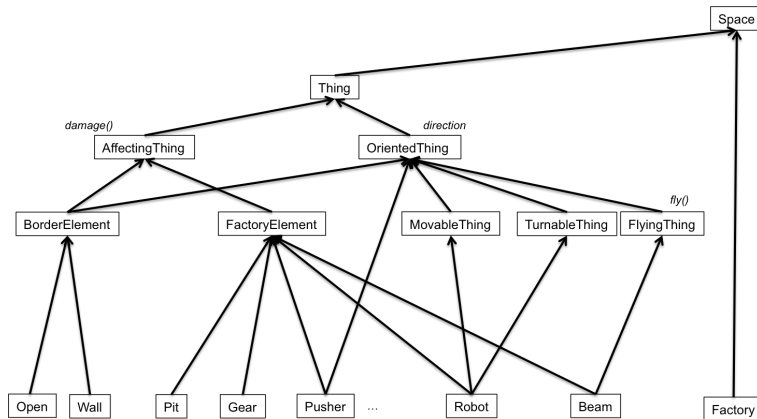
- Neben den Objekten auf dem Spielfeld benötigt man auch noch ein Objekt, das alle anderen Objekte zusammen fasst, um z.B. die **Kommunikation** zwischen den Objekten zu ermöglichen.
- Außerdem muss der Ablauf der verschiedenen Phasen und Schritte kontrolliert werden.
- Dafür gibt es die **Factory**-Klasse. Diese enthält das Spielfeld mit all seinen Elementen.
- Der interessanteste Punkt ist die Zusammenarbeit zwischen der Factory und den Robotern.
- Das **Factory**-Objekt anthält alle anderen Objekte. Damit diese Objekte untereinander kommunizieren können (z.B. Grenzobjektmethode aufrufen), benötigen sie den Zugriff auf das Factory-Objekt.
- Wenn ein Objekt in der Factory **installiert** wird, trägt die Factory einen Verweis auf sich in das Objekt ein.

Motivation
Die Spielregeln
Eine OOP-Analyse
Exkurs: Mehrfachvererbung
Programm-entwurf
Ein kleiner Test
Zusammenfassung

Die Space-Klasse

- Die Änderung der **Orientierung** und **Bewegung** anhand der Orientierung eines Objekts spielen eine zentrale Rolle.
- Erst dachte ich, dass man das in der Klasse **OrientedThing** lokalisieren könnte, aber das scheint vernünftigerweise nicht möglich zu sein. Factory benötigt auch die Operationen.
- Dies ist nun Teil der Space-Klasse, die Wurzelklasse ist – unterhalb von **object**, die implizit Python-Superklasse aller Klassen ist.
- Dort gibt es einige Methoden um Richtungen und Nachbarfelder zu bestimmen.

Motivation
Die Spielregeln
Eine OOP-Analyse
Exkurs: Mehrfachvererbung
Programm-entwurf
Ein kleiner Test
Zusammenfassung



Motivation
Die Spielregeln
Eine OOP-Analyse
Exkurs: Mehrfachvererbung
Programm-entwurf
Ein kleiner Test
Zusammenfassung

Welche Klasse ist wohl konzeptuell am anspruchvollsten?

Motivation
Die Spielregeln
Eine OOP-Analyse
Exkurs: Mehrfachvererbung
Programm-entwurf
Ein kleiner Test
Zusammenfassung

■ Mehrfachvererbung ist in Python möglich. Dabei ist das Folgende zu beachten:

- 1 Bei der Suche nach dem zu ererbenden Attribut oder zu ererbenden Methode wird die **Method-Resolution-Order (MRO)** angewandt, bei der alle **Unterklassen vor Oberklassen** und ansonsten **links vor rechts** gesucht wird.
- 2 Links und rechts ergibt sich durch die Nennung der Klassen in der Liste der Superklassen einer neuen Klasse.
- 3 Annahme: Wir haben eine Methode *A* in *FactoryElement* und in *OrientedThing*. Welche wird in *Pusher* ererbt?
- 4 Allerdings sollte man im Normalfall solche Konflikte nicht haben, da man ja gerade Klassen kombinieren möchte, die keine Gemeinsamkeiten (außer ihrer Superklasse) haben.

Motivation
Die Spielregeln
Eine OOP-Analyse
Exkurs: Mehrfachvererbung
Programm-entwurf
Ein kleiner Test
Zusammenfassung

■ super() ist problematisch:

- 1 Bei **Erweiterungen** von Methoden mit Hilfe von super() muss man mit einbeziehen, dass die Signatur (die Parameterstruktur) u.U. unbekannt ist: Man verwende eine **kooperative** Weise der Bearbeitung der Parameter mit Hilfe von positionalen und Schlüsselwortlisten (*list, **kwlist)
- 2 Dies betrifft in den meisten Fällen die **__init__**-Methode.
- 3 Es muss immer eine oberste Klasse geben, die den Schluss der super()-Aufrufe bildet (bei **__init__** ist das implizit **object**).
- 4 **Achtung:** Wegen der MRO ist es möglich, dass mit super() nicht eine Superklasse sondern eine Geschwisterklasse als nächstes aufgerufen wird (z.B. bei **__init__** mit super() in allen Klassen im Beispiel: *TurnableThing* nach *MovableThing*).

Motivation
Die Spielregeln
Eine OOP-Analyse
Exkurs: Mehrfachvererbung
Programm-entwurf
Ein kleiner Test
Zusammenfassung

MRO - einfach gemacht

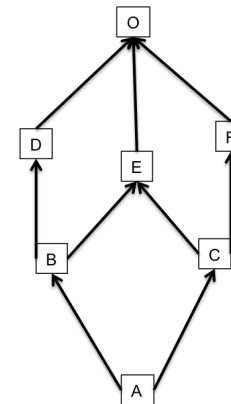
- Im Normalfall kann man die MRO mit den Regeln *links vor rechts*, wobei immer *Unterklasse vor Oberklasse* kommen muss, einfach selbst bestimmen.
- Im Beispiel: Robot, FactoryElement, AffectingElement, MoveableThing, TurnableThing, OrientedThing, Thing, Space, **object**.
- Die Standard-Klassenmethode `mro()` gibt die Liste der Oberklassen entsprechend der MRO aus.

Motivation
Die Spielregeln
Eine OOP-Analyse
Exkurs: Mehrfachvererbung
Programm-entwurf
Ein kleiner Test
Zusammenfassung

Python-Interpreter

```
>>> Robot.mro()
[<class '__main__.Robot'>, <class '__main__.FactoryElement'>, <class '__main__.AffectingElement'>, <class '__main__.MoveableThing'>, <class '__main__.TurnableThing'>, <class '__main__.OrientedThing'>, <class '__main__.Thing'>, <class '__main__.Space'>, <class 'object'>]
```

MRO - komplizierter Fall



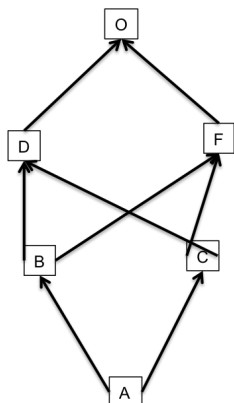
MRO-complex.py

```
class D: pass
class E: pass
class F: pass
class B(D, E): pass
class C(E, F): pass
class A(B, C): pass
```

MRO: A, B, D, C, E, F, O

Motivation
Die Spielregeln
Eine OOP-Analyse
Exkurs: Mehrfachvererbung
Programm-entwurf
Ein kleiner Test
Zusammenfassung

MRO - hoffnungsloser Fall



MRO-fail.py

```
class D: pass
class F: pass
class B(D, F): pass
class C(F, D): pass
class A(B, C): pass
```

MRO: A, B, C, ? →
Python-Fehler

Motivation
Die Spielregeln
Eine OOP-Analyse
Exkurs: Mehrfachvererbung
Programm-entwurf
Ein kleiner Test
Zusammenfassung

Der C3-Algorithmus

- Wie kann man die Vererbungsstrategie formal beschreiben?
- Die **Linearisierung** einer Klasse C mit den (geordneten) Superklassen S_1, \dots, S_n , symbolisch $L(C)$, ist eine Liste von Klassen, die rekursiv wie folgt gebildet wird:

$$L(C) = [C] + \text{merge}(L(S_1), \dots, L(S_n), [S_1, \dots, S_n])$$

- Die Funktion *merge* selektiert dabei nacheinander Elemente aus den Listen und fügt diese der Linearisierung hinzu.

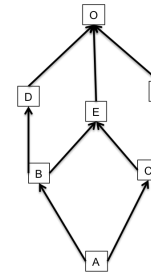
Motivation
Die Spielregeln
Eine OOP-Analyse
Exkurs: Mehrfachvererbung
Programm-entwurf
Ein kleiner Test
Zusammenfassung

Die merge-Funktion

- 1 Es wird das erste Element (der **head**) der ersten Liste betrachtet.
- 2 Taucht dieses nicht als zweites oder späteres Element in einer der späteren Listen auf (im **tail**), dann wird es zur Linearisierung hinzugenommen und aus allen Listen gestrichen.
- 3 Ansonsten lässt man die erste Liste so und probiert das erste Elemente der nächsten Liste usw.
- 4 Nachdem ein Element entfernt wurde, fängt man wieder mit der ersten Liste an.
- 5 Können so alle Listen geleert werden, ist das Ergebnis die Linearisierung von C.
- 6 Ansonsten gibt es keine Linearisierung!

Motivation
Die Spielregeln
Eine OOP-Analyse
Exkurs: Mehrfachvererbung
Programm-entwurf
Ein kleiner Test
Zusammenfassung

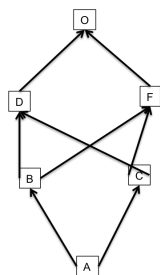
Beispiel: Der komplexe Fall



- 1 $L(O) = [O]$
- 2 $L(D) = [D] + \text{merge}(L(O), [O]) = [D, O]$
- 3 $L(E) = [E, O]$
- 4 $L(F) = [F, O]$
- 5 $L(B) = [B] + \text{merge}(L(D), L(E), [D, E])$
- 6 $L(B) = [B] + \text{merge}([D, O], [E, O], [D, E])$
- 7 $L(B) = [B, D, E, O]$
- 8 $L(C) = [C, E, F, O]$ analog
- 9 $L(A) = [A] + \text{merge}(L(B), L(C), [B, C])$
- 10 $L(A) = [A] + \text{merge}([B, D, E, O], [C, E, F, O], [B, C])$
- 11 $L(A) = [A, B, D, C, E, F, O]$

Motivation
Die Spielregeln
Eine OOP-Analyse
Exkurs: Mehrfachvererbung
Programm-entwurf
Ein kleiner Test
Zusammenfassung

Beispiel: Der hoffungslose Fall



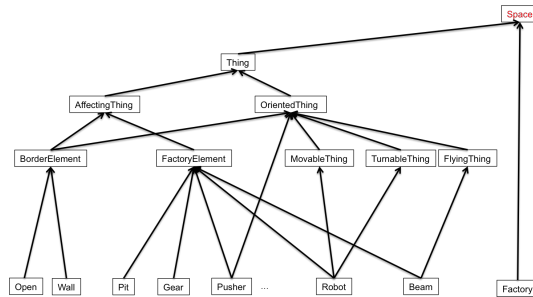
- 1 $L(O) = [O]$
- 2 $L(D) = [D] + \text{merge}(L(O), [O]) = [D, O]$
- 3 $L(F) = [F, O]$
- 4 $L(B) = [B] + \text{merge}(L(D), L(F), [D, F])$
- 5 $L(B) = [B] + \text{merge}([D, O], [F, O], [D, F])$
- 6 $L(B) = [B, D, F, O]$
- 7 $L(C) = [C] + \text{merge}(L(F), L(D), [F, D])$
- 8 $L(C) = [C] + \text{merge}([F, O], [D, O], [F, D])$
- 9 $L(C) = [C, F, D, O]$
- 10 $L(A) = [A] + \text{merge}(L(B), L(C), [B, C])$
- 11 $L(A) = [A] + \text{merge}([B, D, F, O], [C, F, D, O], [B, C])$
- 12 $L(A) = ?$

Motivation
Die Spielregeln
Eine OOP-Analyse
Exkurs: Mehrfachvererbung
Programm-entwurf
Ein kleiner Test
Zusammenfassung

5 Programm-entwurf

Motivation
Die Spielregeln
Eine OOP-Analyse
Exkurs: Mehrfachvererbung
Programm-entwurf
Ein kleiner Test
Zusammenfassung

Klassenhierarchie: Die Space-Klasse



Motivation
Die Spielregeln
Eine OOP-Analyse
Exkurs: Mehrfachvererbung
Programm-entwurf
Ein kleiner Test
Zusammenfassung

Die Space-Klasse (1)

- Es wird das normale **kartesische Koordinatensystem** angenommen.
- Die **Himmelsrichtungen** dienen zur Beschreibung der Orientierung.
- Wir müssen die Himmelsrichtungen **transformieren** können.
- Wir wollen das **Nachbarfeld** eines gegebenen Feldes bei gegebener Himmelsrichtung bestimmen. D.h. bei 'Nord' wird auf die y-Komponente eins addiert.

Motivation
Die Spielregeln
Eine OOP-Analyse
Exkurs: Mehrfachvererbung
Programm-entwurf
Ein kleiner Test
Zusammenfassung

Die Space-Klasse (2)

roborally.py

```
class Space:
    left_trans = dict(N="W", E="N", S="E", W="S")
    move_xy = dict(N=(0,1), E=(1,0), S=(0,-1), W=(-1,0))
    def to_left(self, dir):
        return self.left_trans[dir]

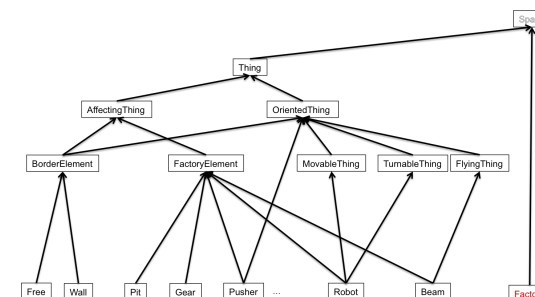
    def to_back(self, dir):
        return self.left_trans[self.left_trans[dir]]

    def to_right(self, dir):
        return self.left_trans[self.left_trans[
            self.left_trans[dir]]]

    def neighbour(self, pos, dir):
        return(pos[0]+self.move_xy[dir][0],
            pos[1]+self.move_xy[dir][1])
```

Motivation
Die Spielregeln
Eine OOP-Analyse
Exkurs: Mehrfachvererbung
Programm-entwurf
Ein kleiner Test
Zusammenfassung

Klassenhierarchie: Die Factory-Klasse



Motivation
Die Spielregeln
Eine OOP-Analyse
Exkurs: Mehrfachvererbung
Programm-entwurf
Ein kleiner Test
Zusammenfassung

Factory-Klasse (1): Initialisierung



roborally.py

```
class Factory(Space):
    def __init__(self, cols=5, rows=5, installs=None):
        self.rows = rows
        self.cols = cols
        self.step = 0
        self.reg_phase = 0
        self.agents = [] # all moveable objects
        self.beams = [] # all temp beams
        self.floor = dict() # floor with coords
        self.init_floor(cols, rows, installs)

    def init_floor(self, cols, rows, installs):
        # Insert borders, free elements and all
        # objects to be installed
        ...
```

Motivation
Die Spielregeln
Eine OOP-Analyse
Exkurs: Mehrfachvererbung
Programm-entwurf
Ein kleiner Test
Zusammenfassung

Factory-Klasse (2): floor-Struktur, Agenten und Strahlen



- Das self.floor-Dict enthält für jede Bodenzelle einen Eintrag, indiziert durch Tupel der Form (x,y).
- Jeder Eintrag ist wiederum ein Dict, indiziert mit den Himmelsrichtungen N, S, E, W und P (für Place).
- Für jede Himmelsrichtung wird die Art der Begrenzung als entsprechendes Objekt eingetragen: OpenBorder, Wall, ggfs. andere.
- Bei OpenBorder wird bei der Initialisierung festgehalten, ob es sich um eine Spielfeldbegrenzung handelt.
- Unter P wird das jeweilige FactoryElement eingetragen.
- Alle MoveableThings werden in der Liste self.agents eingetragen
- Alle temporären Strahlen (Beams) werden in der Liste self.beams eingetragen (und nach jeder Registerphase wieder gelöscht).

Motivation
Die Spielregeln
Eine OOP-Analyse
Exkurs: Mehrfachvererbung
Programm-entwurf
Ein kleiner Test
Zusammenfassung

Factory-Klasse (3): Methoden



roborally.py

```
class Factory(Space):
    ...
    def install(self, obj):
        # Install an object in the right slot and
        # insert a pointer back to factory in each object!
        ...
    def occupied(self, pos, virtual=False):
        # Checks for agents in this field and returns them
        ...
    def collision(self, agent):
        # Checks whether there is something else at pos
        ...
    def push_conflict(self, pusher):
        # Checks whether there is another pusher
        # affecting the same cells.
        # If so, we set the conflict flag in both pushers.
```

Motivation
Die Spielregeln
Eine OOP-Analyse
Exkurs: Mehrfachvererbung
Programm-entwurf
Ein kleiner Test
Zusammenfassung

Factory-Klasse (4): Weitere Methoden

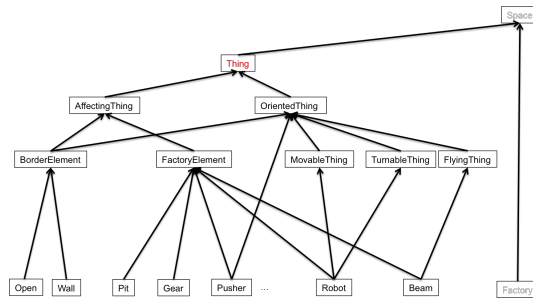


roborally.py

```
class Factory(Space):
    ...
    def exec_reg_phase(self, reg_phase, cmdlist):
        # Execute one register phase
        ...
    def apply(self):
        # Apply all elements to all agents at their pos
        ...
    def resolve_conflicts(self):
        # Resolve all conflicts after one step
        ...
```

Motivation
Die Spielregeln
Eine OOP-Analyse
Exkurs: Mehrfachvererbung
Programm-entwurf
Ein kleiner Test
Zusammenfassung

Klassenhierarchie: Die Thing-Klasse



Motivation
Die Spielregeln
Eine OOP-Analyse
Exkurs: Mehrfachvererbung
Programm-entwurf
Ein kleiner Test
Zusammenfassung

Thing-Klasse

- Alle Dinge (innerhalb der Fabrik) haben eine **Position** pos, die sich natürlich bei beweglichen Dingen ändern kann!
- Wenn die **Fabrik** angegeben wird, wird sie eingetragen.
- Manche Dinge haben einen **Namen**. Bei denen die keinen haben, nutzen wir den Klassennamen.

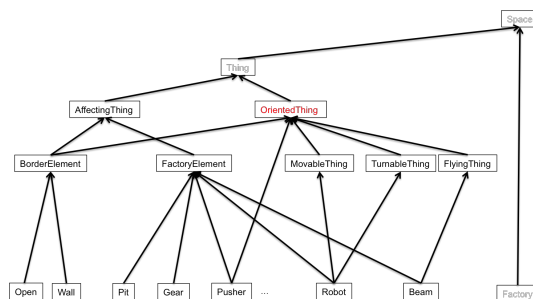
roborally.py

```
class Thing(Space):
    def __init__(self, x, y, factory=None, **kw):
        self.factory = factory
        self.pos = (x, y)

    def __str__(self):
        try:
            return self.name.upper()
        except AttributeError:
            return self.__class__.__name__.upper()
```

Motivation
Die Spielregeln
Eine OOP-Analyse
Exkurs: Mehrfachvererbung
Programm-entwurf
Ein kleiner Test
Zusammenfassung

Klassenhierarchie: Die OrientedThing-Klasse



Motivation
Die Spielregeln
Eine OOP-Analyse
Exkurs: Mehrfachvererbung
Programm-entwurf
Ein kleiner Test
Zusammenfassung

OrientedThing-Klasse

- Alle Dinge, die man orientieren kann, haben eine **Richtung** dir, die sich bei drehbaren Objekten ändern kann.
- Die Position ist als 2-Tupel (x, y) repräsentiert (bekommen wir von der Thing-Klasse.)

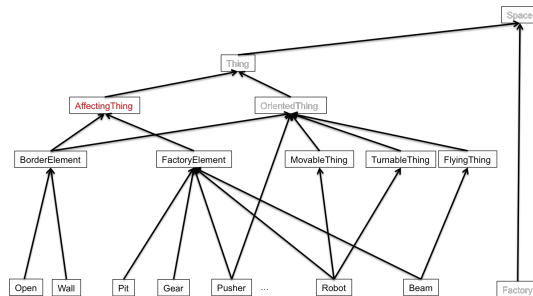
roborally.py

```
class OrientedThing(Thing):
    """Anything oriented using cardinal directions
    (N, E, S, W)
    """

    def __init__(self, x, y, dir="N", **kw):
        super().__init__(x, y, **kw)
        self.dir = dir
```

Motivation
Die Spielregeln
Eine OOP-Analyse
Exkurs: Mehrfachvererbung
Programm-entwurf
Ein kleiner Test
Zusammenfassung

Klassenhierarchie: Die AffectingThing-Klasse



Motivation
Die Spielregeln
Eine OOP-Analyse
Exkurs: Mehrfachvererbung
Programm-entwurf
Ein kleiner Test
Zusammenfassung

AffectingThing-Klasse

- Dinge, die andere Objekte beeinflussen können, indem sie ihnen Lebensenergie entziehen oder im schlimmsten Fall töten.

roborally.py

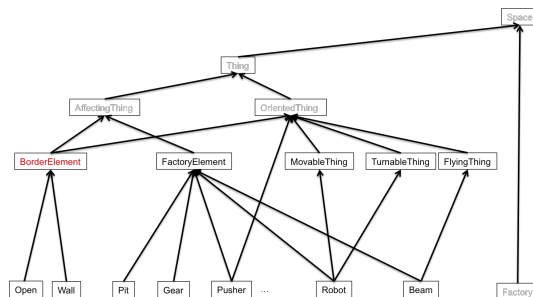
```
class AffectingThing(Thing):
```

```
    def kill(self, obj):
        obj.pos = None
        obj.killed = True
```

```
    def damage(self, agent):
        agent.damage += 1
        if agent.damage >= 10:
            self.kill(agent)
```

Motivation
Die Spielregeln
Eine OOP-Analyse
Exkurs: Mehrfachvererbung
Programm-entwurf
Ein kleiner Test
Zusammenfassung

Klassenhierarchie: Die BorderElement-Klasse



Motivation
Die Spielregeln
Eine OOP-Analyse
Exkurs: Mehrfachvererbung
Programm-entwurf
Ein kleiner Test
Zusammenfassung

BorderElement-Klasse

- BorderElement ist eine **abstrakte Klasse**, die nur vorgibt, welches Interface vorhanden sein muss.

roborally.py

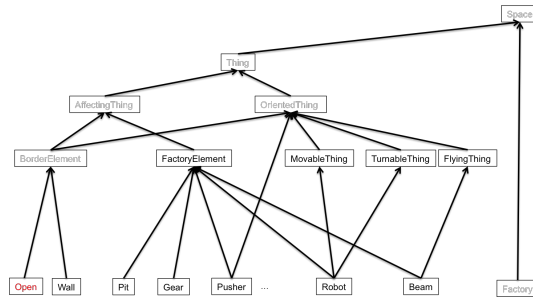
```
class BorderElement(OrientedThing, AffectingElement):
```

```
    def leavecell(self, thing, onlycheck=False):
        #Try to leave cell through this border (with
        #orientation). Change thing.pos (if not onlycheck)
        #and return True if successful.
        raise NotImplementedError("leavecell undefined")
```

```
    def entercell(self, thing, onlycheck=False):
        #Try to enter cell through this border.
        raise NotImplementedError("entercell undefined")
```

Motivation
Die Spielregeln
Eine OOP-Analyse
Exkurs: Mehrfachvererbung
Programm-entwurf
Ein kleiner Test
Zusammenfassung

Klassenhierarchie: Die OpenBorder-Klasse



Motivation
Die Spielregeln
Eine OOP-Analyse
Exkurs: Mehrfachvererbung
Programm-entwurf
Ein kleiner Test
Zusammenfassung

OpenBorder-Klasse (1)



- OpenBorder behandelt Verlassen und Eintritt. Eintreten ist immer möglich!

roborally.py

```
class OpenBorder(BorderElement):
```

```
    def __init__(self, x, y, factoryexit=False, **kw):
        super().__init__(x, y, **kw)
        self.exit = factoryexit
```

```
    def entercell(self, thing, onlycheck=False):
        if not onlycheck:
            thing.pos = self.pos
        return True
```

Motivation
Die Spielregeln
Eine OOP-Analyse
Exkurs: Mehrfachvererbung
Programm-entwurf
Ein kleiner Test
Zusammenfassung

OpenBorder-Klasse (2)



- Wenn die Grenze Fabrikgrenze ist, droht der Tod!

roborally.py

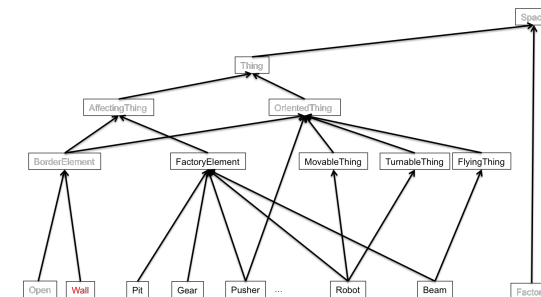
```
class OpenBorder(BorderElement):
```

```
    ...
```

```
    def leavecell(self, thing, onlycheck=False):
        if self.exit:
            if not onlycheck: self.kill(thing)
            return True
        else:
            return self.factory.floor[self.neighbour(
                self.pos, self.dir)][self.to_back(
                self.dir)].entercell(thing, onlycheck)
```

Motivation
Die Spielregeln
Eine OOP-Analyse
Exkurs: Mehrfachvererbung
Programm-entwurf
Ein kleiner Test
Zusammenfassung

Klassenhierarchie: Die Wall-Klasse



Motivation
Die Spielregeln
Eine OOP-Analyse
Exkurs: Mehrfachvererbung
Programm-entwurf
Ein kleiner Test
Zusammenfassung

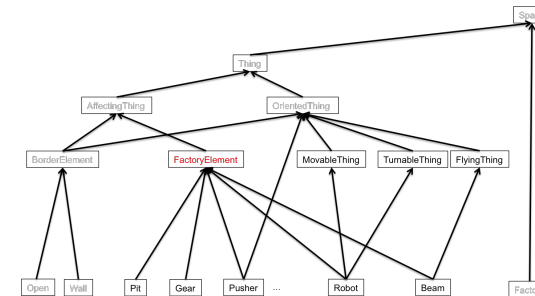
- Bei einer Wall kommen wir weder rein noch raus!

roborally.py

```
class Wall(BorderElement):

    def leavecell(self, thing, onlycheck=False):
        return False

    def entercell(self, thing, onlycheck=False):
        return False
```



- Ein FactoryElement ist nur in bestimmten Schritten (self.active_steps) aktiv. Außerdem sind Kollisionen nicht immer relevant (z.B. sind sie in Pits irrelevant).

roborally.py

```
class FactoryElement(AffectingElement):
    nocollisions = False # only True in pits
    active_steps = { }

    def __init__(self, x, y, reg_phases=None, **kw):
        if (reg_phases):
            self.active_reg_phases = reg_phases
        else:
            self.active_reg_phases = {1, 2, 3, 4, 5}
        super().__init__(x, y, **kw)
```

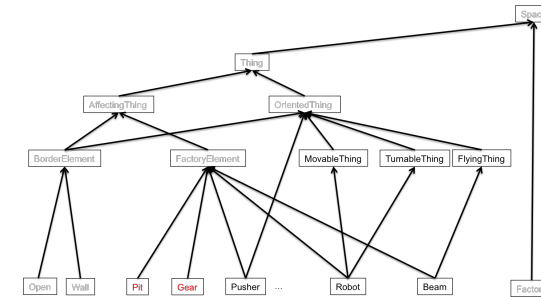
roborally.py

```
class FactoryElement(AffectingElement):
    ...
    def apply_element(self):
        if (self.factory.step in self.active_steps and
            self.factory.reg_phase in self.active_reg_phases):
            for agent in self.factory.occupied(self.pos,
                virtual=True): self.acton(agent)
            self.act()

    def act(self): pass
    # act in isolation
    def acton(self, agent): pass
    # act on agent
    def on_arrival(self, agent, dir): pass
    # called on curved conveyor belts
```

- Bisher hatten wir als Kombinationsmechanismen für Methoden kennen gelernt:
 - Von Superklasse **erben** und unmodifiziert nutzen.
 - Die Superklassenmethode durch eigene Methode **überschreiben**.
 - Die Superklassenmethode **erweitern**, durch Aufruf von `super()`.
- Hier haben wir den Fall, dass die Superklasse die Erledigung der Aufgabe an eine **Subklasse delegiert**. Die Subklassen sollten die `acton`- und `act`-Methode implementieren. Sonst passiert nichts!
- Sinnvoll, da die **Vorbedingungen** für alle Subklassen gleich sind, die einzelnen `act` and `acton`-Methoden aber speziell sind.

Motivation
Die Spielregeln
Eine OOP-Analyse
Exkurs: Mehrfachvererbung
Programm-entwurf
Ein kleiner Test
Zusammenfassung



Motivation
Die Spielregeln
Eine OOP-Analyse
Exkurs: Mehrfachvererbung
Programm-entwurf
Ein kleiner Test
Zusammenfassung

- Pits sind immer tödlich! Kollisionen innerhalb von Pits sind aber irrelevant.

`roborally.py`

```

class Pit(FactoryElement):
    # A pit kills the agent (unless it is retracted)

    nocollisions = True
    active_steps = {0, 1, 2, 3, 4, 5, 6, 7, 8}

    def acton(self, agent):
        self.kill(agent)
    
```

Motivation
Die Spielregeln
Eine OOP-Analyse
Exkurs: Mehrfachvererbung
Programm-entwurf
Ein kleiner Test
Zusammenfassung

- Das Rotieren wird an die Agenten delegiert.

`roborally.py`

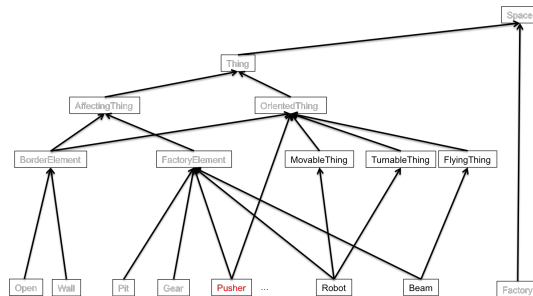
```

class Gear(FactoryElement):
    active_steps = {4}

    def __init__(self, x, y, clockwise=True, **kw):
        super().__init__(x, y, **kw)
        self.clockwise = clockwise

    def acton(self, agent, factory):
        if self.clockwise:
            agent.rotate_right()
        else:
            agent.rotate_left()
    
```

Motivation
Die Spielregeln
Eine OOP-Analyse
Exkurs: Mehrfachvererbung
Programm-entwurf
Ein kleiner Test
Zusammenfassung



Motivation
Die Spielregeln
Eine OOP-Analyse
Exkurs: Mehrfachvererbung
Programm-entwurf
Ein kleiner Test
Zusammenfassung

- Markiere berührte Felder, teste auf Konflikte mit anderen Pushern und delegiere die Bewegung an die Agenten.

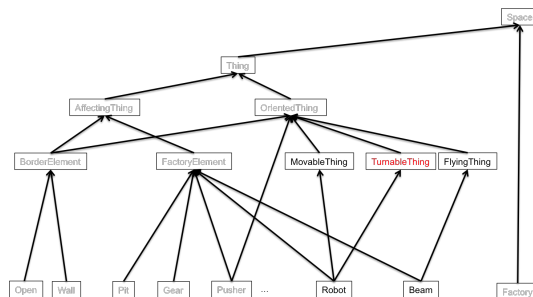
roborally.py

```
class Pusher(FactoryElement, OrientedThing):
    active_steps = {3}

    def __init__(self, x, y, **kw):
        super().__init__(x, y, **kw)
        self.marked = set() # affected positions
        self.conflict = False # conflict detected

    def acton(self, agent):
        # mark all affected positions
        self.marked = agent.mark(self.dir, self)
        if not self.factory.push_conflict(self):
            agent.move(self.dir)
```

Motivation
Die Spielregeln
Eine OOP-Analyse
Exkurs: Mehrfachvererbung
Programm-entwurf
Ein kleiner Test
Zusammenfassung



Motivation
Die Spielregeln
Eine OOP-Analyse
Exkurs: Mehrfachvererbung
Programm-entwurf
Ein kleiner Test
Zusammenfassung

- Alle Dinge, die man drehen kann (eigentlich nur Roboter), können ihre Orientierung ändern.

roborally.py

```
class TurnableThing(OrientedThing):

    def rotate_left(self, *rest):
        self.dir = self.to_left(self.dir)

    def u_turn(self, *rest):
        self.dir = self.to_back(self.dir)

    def rotate_right(self, *rest):
        self.dir = self.to_right(self.dir)
```

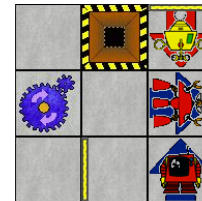
Motivation
Die Spielregeln
Eine OOP-Analyse
Exkurs: Mehrfachvererbung
Programm-entwurf
Ein kleiner Test
Zusammenfassung

MoveableThing-Klasse (1): Prinzipien

- Bewegbare Objekte (eigentlich nur Roboter), können auf drei verschiedene Arten bewegt werden:
 - 1 Die Bewegung ist durch einen Agenten **initiiert** und im Ablauf **priorisiert** (Programmkarte). Der Agent bewegt sich in die durch seine Orientierung und die Spielkarte vorgegebene Richtung. Dabei kann er vor ihm stehende Roboter schubsen, wenn nicht der erste in der Schlange von einer Wand gebremst wird.
 - 2 Alle Agenten bewegen sich **parallel** auf den **Förderbändern**. Enden mehrere Roboter auf dem gleichen Feld, werden sie zurückgesetzt.
 - 3 Alle Roboter(-schlangen) werden **parallel** durch die **Pusher** geschubst. Bei Konflikten sagen die Spielregeln nichts—aber **Konflikte** sollten so wie bei Förderbändern behandelt werden.
- Generelle Idee: Bewegung erst einmal ausführen. Im Konfliktfall **zurücknehmen**.

Motivation
Die Spielregeln
Eine OOP-Analyse
Exkurs: Mehrfachvererbung
Programm-entwurf
Ein kleiner Test
Zusammenfassung

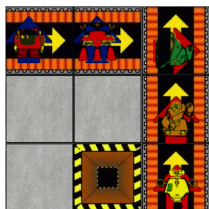
MoveableThing-Klasse (2): Beispiel für Eigenbewegung



- 1 Twonky soll 1 Feld geradeaus gehen.
- 2 Twonky geht und trifft HulkX90.
- 3 HulkX90 wird geschubst und trifft Spinbot.
- 4 Spinbot wird geschubst.
- 5 Spinbot kann aber nicht weiter.
- 6 Kollisionsauflösung: HulkX90 muss zurück!
- 7 Das führt zur Kollision mit Twonky: Muss auch zurück.

Motivation
Die Spielregeln
Eine OOP-Analyse
Exkurs: Mehrfachvererbung
Programm-entwurf
Ein kleiner Test
Zusammenfassung

MoveableThing-Klasse (3): Beispiel Förderband



- 1 Alle Roboter werden sollen parallel um ein Feld bewegt werden.
- 2 Bewegung wird durchgeführt und Trundelbot fällt vom Spielfeldrand. Zoombot (von unten) und HulkX90 (von links) haben einen Konflikt.
- 3 Beide werden zurückgesetzt und haben dann jeweils einen Konflikt mit Spinbot (unten) bzw. Twonky (links).
- 4 Diese werden auch noch zurückgesetzt, womit alle Konflikte aufgelöst wären.

Motivation
Die Spielregeln
Eine OOP-Analyse
Exkurs: Mehrfachvererbung
Programm-entwurf
Ein kleiner Test
Zusammenfassung

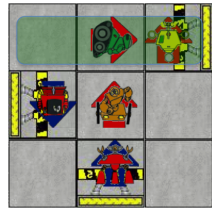
MoveableThing-Klasse (4): Beispiel Pusher



- Die parallele Betätigung von Pushern kann zu Konflikten führen.
- Kann nicht wie bei Förderbändern behandelt werden, da Schlangen in ihrer Gesamtheit bewegt werden müssen und diese sich **überkreuzen** können.
- Idee: Wenn ein **Paar von Pushern** sich beeinflussen könnte, werden beide nicht bewegt. **Markiere beeinflusste Felder** und bilde Mengenschnitt.
- Machte alles in **einem Durchlauf** durch alle Pusher!

Motivation
Die Spielregeln
Eine OOP-Analyse
Exkurs: Mehrfachvererbung
Programm-entwurf
Ein kleiner Test
Zusammenfassung

MoveableThing-Klasse (5): Beispiel Pusher ausführlich



- 1 Markiere ersten Pusher.
- 2 Da kein Konflikt: Schubse!
- 3 Markiere Felder für zweiten Pusher.
- 4 Kein Konflikt bisher: Schubse!
- 5 Markiere Felder für dritten Pusher.
- 6 Konflikte mit den anderen beiden, deshalb keine Bewegung.
- 7 Alle Roboter, die von an Konflikten beteiligten Pushern bewegt wurden, müssen zurückgesetzt werden (inkl. in Pits gestoßene Roboter!)

Motivation
Die Spielregeln
Eine OOP-Analyse
Exkurs: Mehrfachvererbung
Programm-entwurf
Ein kleiner Test
Zusammenfassung

MoveableThing-Klasse (6): Zusätzliche Attribute



- Die letzte Konfiguration (Position und Ausrichtung) wird vor einer Bewegung in `lastconf` gespeichert, damit man weiß, wo man beim Rücksetzen hin muss.
- `pushmarker` enthält den letzten Pusher. Wichtig um nach einem Spielzug einen Push-Konflikt festzustellen.

`roborally.py`

```
class MoveableThing(OrientedThing):
```

```
    def __init__(self, x, y, dir="N", **kw):
        super().__init__(x, y, dir, **kw)
        self.lastconf = None # last configuration
        self.pushmarker = None # last pusher
```

Motivation
Die Spielregeln
Eine OOP-Analyse
Exkurs: Mehrfachvererbung
Programm-entwurf
Ein kleiner Test
Zusammenfassung

MoveableThing-Klasse (7): Eigene oder geschubste Bewegung



- Aktive oder passive Bewegung initiiert durch einen **programmierten Roboterschritt** oder durch Pusher

`roborally.py`

```
def move(self, dir):
    self.lastconf = (self.pos, self.dir)
    # try to move into direction dir
    if not self.factory.floor[self.pos][dir].\
        leavecell(self):
        #could not leave cell
        self.lastconf = None
        return
    colliders = self.factory.collision(self)
    for collider in colliders:
        # if collision with another robot, push
        collider.move(dir)
```

Motivation
Die Spielregeln
Eine OOP-Analyse
Exkurs: Mehrfachvererbung
Programm-entwurf
Ein kleiner Test
Zusammenfassung

MoveableThing-Klasse (8): Parallele passive Bewegung



- Alle Agenten werden gleichzeitig bewegt

`roborally.py`

```
def transport(self, dir):
    if self.lastconf: # has already been moved
        return False
    self.lastconf = (self.pos, self.dir)
    if self.factory.floor[self.pos][dir].\
        leavecell(self):
        return True
    return False
```

Motivation
Die Spielregeln
Eine OOP-Analyse
Exkurs: Mehrfachvererbung
Programm-entwurf
Ein kleiner Test
Zusammenfassung

MoveableThing-Klasse (8): Markiere Felder



- Markiere beteiligte Felder und erinnere Pusher (für spätere Konfliktauflösung)

roborally.py

```
def mark(self, dir, pusher):
    self.pushmarker = pusher
    marked = { self.pos }
    if self.factory.floor[self.pos][dir].leavecell(
        self, onlycheck=True):
        neighbour_cell = self.neighbour(self.pos, dir)
        if (neighbour_cell in self.factory.floor and
            not self.factory.floor[neighbour_cell][
                'P'].nocollisions):
            marked |= { neighbour_cell }
            for a in self.factory.occupied(
                neighbour_cell):
                marked |= a.mark(dir, pusher)
    return marked
```

Motivation
Die Spielregeln
Eine OOP-Analyse
Exkurs: Mehrfachvererbung
Programm-entwurf
Ein kleiner Test
Zusammenfassung

MoveableThing-Klasse (9): Konfliktauflösung



- resolve wird am Ende jedes Schritts für jeden Agenten aufgerufen.

roborally.py

```
def resolve(self):
    collider = self.factory.collision(self)
    if collider:
        for a in collider + [self]:
            a.retract()
    if self.pushmarker and self.pushmarker.conflict:
        self.retract()
def retract(self):
    if self.lastconf:
        (self.pos, self.dir) = self.lastconf
        self.lastconf = None
        self.killed = False
        for a in self.factory.collision(self):
            a.retract()
```

Motivation
Die Spielregeln
Eine OOP-Analyse
Exkurs: Mehrfachvererbung
Programm-entwurf
Ein kleiner Test
Zusammenfassung

MoveableThing-Klasse (10): Abschluss



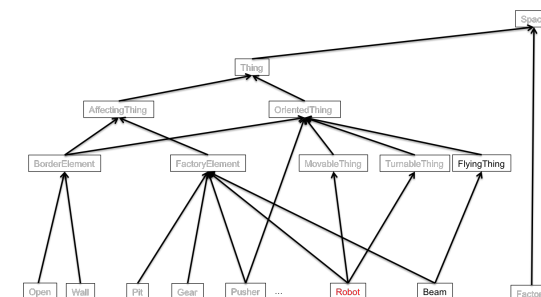
- Zum Schluss werden alle temporären Variablen wieder zurück gesetzt.

roborally.py

```
def reset(self):
    self.lastconf = None
    if self.pushmarker:
        self.pushmarker.conflict = False
        self.pushmarker.marked = set()
        self.pushmarker = None
```

Motivation
Die Spielregeln
Eine OOP-Analyse
Exkurs: Mehrfachvererbung
Programm-entwurf
Ein kleiner Test
Zusammenfassung

Die Klassenhierarchie: Roboter-Klasse



Motivation
Die Spielregeln
Eine OOP-Analyse
Exkurs: Mehrfachvererbung
Programm-entwurf
Ein kleiner Test
Zusammenfassung

Robot-Klasse (1)

- Roboter haben zusätzliche Zustandsattribute und können Befehle ausführen.

roborally.py

```
class Robot(MoveableThing, TurnableThing, FactoryElement):
```

```
    active_steps = { 6 }
```

```
    def __init__(self, x, y, dir="N",
                  name="Anonymous", **kw):
        super().__init__(x, y, dir, **kw)
        self.name = name
        self.damage = 0
        self.lives = 3
        self.alive = True
        self.virtual = False
        self.killed = False
```

Motivation

Die Spielregeln

Eine OOP-Analyse

Exkurs: Mehrfachvererbung

Programm-entwurf

Ein kleiner Test

Zusammenfassung

Robot-Klasse (2)

- Roboter können (am Ende eines Schrittes) sterben und selber Strahlen schießen.

roborally.py

```
def die(self):
    self.alive = False
    self.lives -= 1
    self.damage = 0

def act(self):
    if self.alive and not self.virtual:
        Beam(self.pos[0], self.pos[1], shootby=self,
            factory=self.factory, dir=self.dir)
```

Motivation

Die Spielregeln

Eine OOP-Analyse

Exkurs: Mehrfachvererbung

Programm-entwurf

Ein kleiner Test

Zusammenfassung

Robot-Klasse (3): Bewegung

roborally.py

```
def go(self):
    self.onestep(True)
```

```
def backup(self):
    self.onestep(False)
```

```
def onestep(self, forward):
    """active execution of one step"""
    if not self.alive:
        return
    if forward:
        self.move(self.dir)
    else:
        self.move(self.to_back(self.dir))
```

Motivation

Die Spielregeln

Eine OOP-Analyse

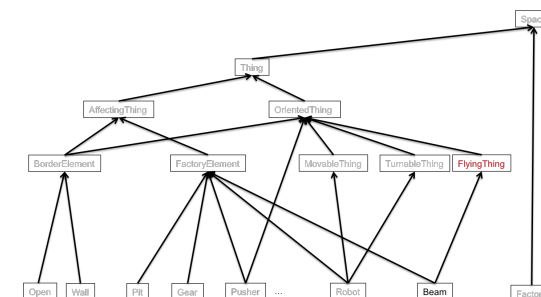
Exkurs: Mehrfachvererbung

Programm-entwurf

Ein kleiner Test

Zusammenfassung

Die Klassenhierarchie: FlyingThing-Klasse



Motivation

Die Spielregeln

Eine OOP-Analyse

Exkurs: Mehrfachvererbung

Programm-entwurf

Ein kleiner Test

Zusammenfassung

FlyingThing-Klasse: Fliegen



- Ein FlyingThing fliegt solange, bis es auf ein Hindernis trifft (oder die Fabrik verlässt).
- Der Initiator selbst ist kein Hindernis!

roborally.py

```
class FlyingThing(OrientedThing):
```

```
    def fly(self, shootby=None):
        if not self.pos: return # left factory!
        occ = self.factory.occupied(self.pos, virtual=True)
        if (occ and shootby not in occ):
            pass # shooter is not its own target
        elif not self.factory.floor[self.pos][self.dir].\
            leavecell(self): pass #could not leave cell
        else:
            self.fly()
```

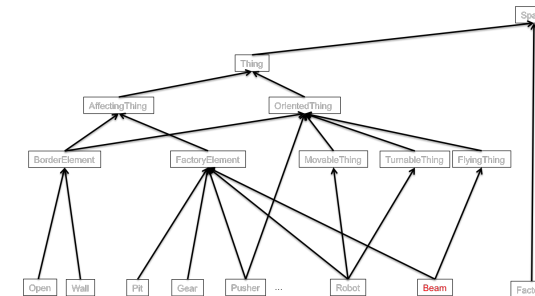
Motivation
Die Spielregeln
Eine OOP-Analyse
Exkurs: Mehrfachvererbung
Programm-entwurf
Ein kleiner Test
Zusammenfassung

12. Dezember 2014

B. Nebel – Info I

94 / 109

Die Klassenhierarchie: Beam-Klasse



Motivation
Die Spielregeln
Eine OOP-Analyse
Exkurs: Mehrfachvererbung
Programm-entwurf
Ein kleiner Test
Zusammenfassung

12. Dezember 2014

B. Nebel – Info I

95 / 109

Beam-Klasse: Strahlen schießen und einwirken lassen



roborally.py

```
class Beam(FlyingThing, FactoryElement):
```

```
    active_steps = { 6 }
```

```
    def __init__(self, x, y, shootby=None, **kw):
        super().__init__(x, y, **kw)
        self.shootby = shootby
        self.factory.beams.append(self)
        self.fly(shootby=shootby)
```

```
    def acton(self, agent):
        if not self.pos or self.shootby is agent:
            return
        self.damage(agent)
```

Motivation
Die Spielregeln
Eine OOP-Analyse
Exkurs: Mehrfachvererbung
Programm-entwurf
Ein kleiner Test
Zusammenfassung

12. Dezember 2014

B. Nebel – Info I

96 / 109

Zurück zur Factory-Klasse: apply und resolve



roborally.py

```
class Factory(Space):
```

```
    def apply(self):
        for pos in self.floor:
            self.floor[pos]['P'].apply_element()
        for a in self.agents: # robots as factory el.
            a.apply_element()
        for b in self.beams: # consider all shot beams
            b.apply_element()
        self.beams = [] # remove all beams
```

```
    def resolve_conflicts(self):
        for a in self.agents: a.resolve()
        for a in self.agents: a.reset()
        for a in self.agents:
            if a.killed: a.die()
```

Motivation
Die Spielregeln
Eine OOP-Analyse
Exkurs: Mehrfachvererbung
Programm-entwurf
Ein kleiner Test
Zusammenfassung

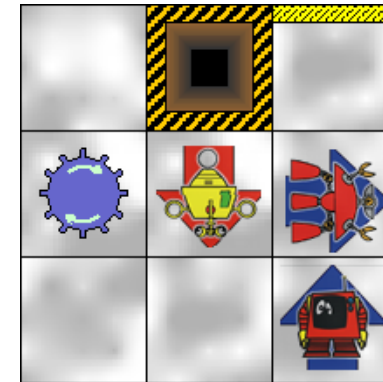
12. Dezember 2014

B. Nebel – Info I

97 / 109

6 Ein kleiner Test

Test-Szenario



- Twonky (rechte untere Ecke bei (3, 1)) soll die anderen rumschubsen, sie in den Abgrund stürzen, und ein bisschen Karussell fahren.

Der Test

```
roborally.py

if __name__ == "__main__":
    t = Robot(3, 1, dir="N", name="Twonky")
    s = Robot(2, 2, dir="S", name="Spinbot")
    h = Robot(3, 2, dir="E", name="HulkX90")
    fac = Factory(3, 3, installs=(Wall(3, 3, dir="N"),
                                     Pit(2, 3),
                                     Gear(1, 2,
                                           clockwise=True),
                                     t, s, h))

    fac.exec_reg_phase(1, [(2, t.go)])
    fac.exec_reg_phase(2, [(1, t.rotate_left),
                           (1, h.rotate_right)])
    fac.exec_reg_phase(3, [(2, t.go), (1, h.go),
                           (2, s.go)])
```

Der Test: Ein Trace (1)

Python-Interpreter

```
init: TWONKY starts at (3, 1) with orientation N
init: SPINBOT starts at (2, 2) with orientation S
init: HULKX90 starts at (3, 2) with orientation E
*** Starting register phase 1
GO command: TWONKY
onestep: TWONKY wants to make 1 step forward (dir=N)
move: move of TWONKY from (3, 1) in direction N
move: move of HULKX90 from (3, 2) in direction N
GO command: TWONKY
onestep: TWONKY wants to make 1 step forward (dir=N)
move: move of TWONKY from (3, 2) in direction N
move: move of HULKX90 from (3, 3) in direction N
HULKX90 bumped into a wall and does not leave (3, 3)
move: HULKX90 cannot move because of a wall
retract: send TWONKY back to (3, 2)
```

Der Test: Ein Trace (2)



Python-Interpreter

```
BEAM shot by TWONKY at (3, 2), direction N
fly: BEAM wants to fly from (3, 2) in direction N
fly: BEAM wants to fly from (3, 3) in direction N
fly: BEAM cannot move because of an obstacle
BEAM shot by SPINBOT at (2, 2), direction S
fly: BEAM wants to fly from (2, 2) in direction S
fly: BEAM wants to fly from (2, 1) in direction S
BEAM was killed leaving factory at (2, 1)
BEAM shot by HULKX90 at (3, 3), direction E
fly: BEAM wants to fly from (3, 3) in direction E
BEAM was killed leaving factory at (3, 3)
HULKX90 got damaged by laserbeam at (3, 3)
```

Motivation
Die Spielregeln
Eine OOP-Analyse
Exkurs: Mehrfachvererbung
Programm-entwurf
Ein kleiner Test
Zusammenfassung

Der Test: Ein Trace (3)



Python-Interpreter

```
*** Starting register phase 2
rotate_left: TWONKY facing now W
rotate_right: HULKX90 facing now S
BEAM shot by TWONKY at (3, 2), direction W
fly: BEAM wants to fly from (3, 2) in direction W
fly: BEAM wants to fly from (2, 2) in direction W
fly: BEAM cannot move because of an obstacle
BEAM shot by SPINBOT at (2, 2), direction S
fly: BEAM wants to fly from (2, 2) in direction S
fly: BEAM wants to fly from (2, 1) in direction S
BEAM was killed leaving factory at (2, 1)
BEAM shot by HULKX90 at (3, 3), direction S
fly: BEAM wants to fly from (3, 3) in direction S
fly: BEAM wants to fly from (3, 2) in direction S
fly: BEAM cannot move because of an obstacle
SPINBOT got damaged by laserbeam at (2, 2)
```

Motivation
Die Spielregeln
Eine OOP-Analyse
Exkurs: Mehrfachvererbung
Programm-entwurf
Ein kleiner Test
Zusammenfassung

Der Test: Ein Trace (4)



Python-Interpreter

```
TWONKY got damaged by laserbeam at (3, 2)
*** Starting register phase 3
GO command: TWONKY
onestep: TWONKY wants to make 1 step forward (dir=W)
move: move of TWONKY from (3, 2) in direction W
move: move of SPINBOT from (2, 2) in direction W
GO command: TWONKY
onestep: TWONKY wants to make 1 step forward (dir=W)
move: move of TWONKY from (2, 2) in direction W
move: move of SPINBOT from (1, 2) in direction W
SPINBOT was killed leaving factory at (1, 2)
SPINBOT dies
GO command: HULKX90
onestep: HULKX90 wants to make 1 step forward (dir=S)
move: move of HULKX90 from (3, 3) in direction S
```

Motivation
Die Spielregeln
Eine OOP-Analyse
Exkurs: Mehrfachvererbung
Programm-entwurf
Ein kleiner Test
Zusammenfassung

Der Test: Ein Trace (5)



Python-Interpreter

```
GO command: SPINBOT
onestep: SPINBOT wants to make 1 step forward (dir=S)
onestep: SPINBOT is dead and cannot move
GO command: SPINBOT
onestep: SPINBOT wants to make 1 step forward (dir=S)
onestep: SPINBOT is dead and cannot move
GEAR: turn TWONKY clockwise at (1, 2)
rotate_right: TWONKY facing now N
BEAM shot by TWONKY at (1, 2), direction N
fly: BEAM wants to fly from (1, 2) in direction N
fly: BEAM wants to fly from (1, 3) in direction N
BEAM was killed leaving factory at (1, 3)
BEAM shot by HULKX90 at (3, 2), direction S
fly: BEAM wants to fly from (3, 2) in direction S
fly: BEAM wants to fly from (3, 1) in direction S
BEAM was killed leaving factory at (3, 1)
```

Motivation
Die Spielregeln
Eine OOP-Analyse
Exkurs: Mehrfachvererbung
Programm-entwurf
Ein kleiner Test
Zusammenfassung

- Der Anspruch war gewesen, das Design an der **Erweiterbarkeit** auszurichten. Ist das gelungen?
- Elemente mit **Doppelrollen** funktionieren problemlos (**Mehrfachvererbung** hilft hier)!
- Viele **Fabrikelemente** lassen sich leicht integrieren (Portale, temporäre Türen oder fallgruben, halbdurchlässige Wände, Öllachen)
- **Zusatzwaffen** sind auch nicht zu schwierig
- Interessant wäre eine Ergänzung um eine **GUI** ...
- **Achtung**: Ich habe die Umsetzung als ein Softwareprojekt im fortgeschrittenen Semester gefunden.
- **Idee**: Die Berechnung einer **optimalen Strategie** wäre natürlich das, was wirklich interessant wäre – KI