

Informatik I: Einführung in die Programmierung

18. OOP: RoboRally als Beispiel

Albert-Ludwigs-Universität Freiburg



**UNI
FREIBURG**

Bernhard Nebel

12. Dezember 2014

1 Motivation



Motivation

Die
Spielregeln

Eine OOP-
Analyse

Exkurs:
Mehrfachver-
erbung

Programm-
entwurf

Ein kleiner
Test

Erweiterung:
Visualisie-
rung

- 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 ...

Motivation

Die
Spielregeln

Eine OOP-
Analyse

Exkurs:
Mehrfachver-
erbung

Programm-
entwurf

Ein kleiner
Test

Erweiterung:
Visualisie-
rung

- 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



Motivation

Die
Spielregeln

Eine OOP-
Analyse

Exkurs:
Mehrfachver-
erbung

Programm-
entwurf

Ein kleiner
Test

Erweiterung:
Visualisie-
rung



- 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

Erweiterung: Visualisierung

2 Die Spielregeln



Motivation

**Die
Spielregeln**

Eine OOP-
Analyse

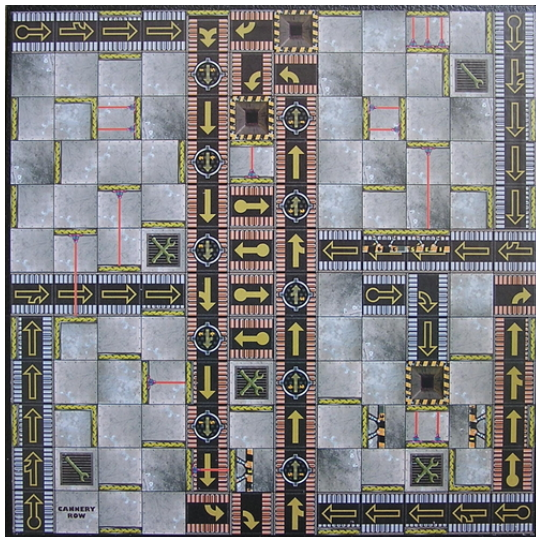
Exkurs:
Mehrfachver-
erbung

Programm-
entwurf

Ein kleiner
Test

Erweiterung:
Visualisie-
rung

Ein Spielbrettbeispiel



Motivation

Die
Spielregeln

Eine OOP-
Analyse

Exkurs:
Mehrfachver-
erbung

Programm-
entwurf

Ein kleiner
Test

Erweiterung:
Visualisie-
rung

- 8 verschiedene **Spielsteine** – die Roboter
- 6 verschiedene **Spielbretter**, die auch zusammen gelegt werden können
- 84 verschiedene **Programmierkarten**, die Befehle wie *1 Feld vorwärts*, *2 Felder vorwärts*, *Linksdrehung* 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:
Mehrfachver-
erbung

Programm-
entwurf

Ein kleiner
Test

Erweiterung:
Visualisie-
rung



- 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:
Mehrfachver-
erbung

Programm-
entwurf

Ein kleiner
Test

Erweiterung:
Visualisie-
rung



- 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:
Mehrfachver-
erbung

Programm-
entwurf

Ein kleiner
Test

Erweiterung:
Visualisie-
rung

- 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.
- Üben wir das mal: http://www.wizards.com/avalonhill/robo_demo/robodemo.asp

Motivation

Die
Spielregeln







Eine OOP-
Analyse

Exkurs:
Mehrfachver-
erbung

Programm-
entwurf

Ein kleiner
Test

Erweiterung:
Visualisie-
rung

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

Motivation

Die
Spielregeln

Eine OOP-
Analyse

Exkurs:
Mehrfachver-
erbung

Programm-
entwurf

Ein kleiner
Test

Erweiterung:
Visualisie-
rung



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



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



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



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



Checkpunkte und Reparaturfelder: Hier wird am Ende jeder Registerphase eine Archivkopie abgelegt. Am Ende einer Runde wird entsprechend der Anzahl der Schrau-

Motivation

Die
Spielregeln

Eine OOP-
Analyse

Exkurs:
Mehrfachver-
erbung

Programm-
entwurf

Ein kleiner
Test

Erweiterung:
Visualisie-
rung

- 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:
Mehrfachver-
erbung

Programm-
entwurf

Ein kleiner
Test

Erweiterung:
Visualisie-
rung

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

- Bei jedem **Lasertreffer** gibt es einen Schadenspunkt und bei jede **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:
Mehrfachver-
erbung

Programm-
entwurf

Ein kleiner
Test

Erweiterung:
Visualisie-
rung

- 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:
Mehrfachver-
erbung

Programm-
entwurf

Ein kleiner
Test

Erweiterung:
Visualisie-
rung



- Außerdem gibt es noch **Optionskarten**, die man statt zwei Reparaturpunkten aufnehmen kann.
- Diese 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:
Mehrfachver-
erbung

Programm-
entwurf

Ein kleiner
Test

Erweiterung:
Visualisie-
rung

3 Eine OOP-Analyse



Motivation

Die
Spielregeln

**Eine OOP-
Analyse**

Exkurs:
Mehrfachver-
erbung

Programm-
entwurf

Ein kleiner
Test

Erweiterung:
Visualisie-
rung



- 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*.
- Die Benutzerschnittstelle soll nur sehr rudimentär bleiben.
 - Einfache Eingabe der Instruktion
 - 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:
Mehrfachver-
erbung

Programm-
entwurf

Ein kleiner
Test

Erweiterung:
Visualisie-
rung

- 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!**

Motivation

Die
Spielregeln

Eine OOP-
Analyse

Exkurs:
Mehrfachver-
erbung

Programm-
entwurf

Ein kleiner
Test

Erweiterung:
Visualisie-
rung

- Roboter,
- Aktive Fabrikelemente: Förderbänder, Drehscheiben, . . .
- Passive Fabrikelemente: Plätze, Wände, Gruben
- Spielbrett (?),
 - allerdings nur eines
 - soll dies aktiv sein?
 - soll es nach außen hin Services (=Methoden) anbieten?
- Die Spielkontrolle, die den gesamten Spielablauf koordiniert (lassen wir erst einmal weg)

Motivation

Die
Spielregeln

Eine OOP-
Analyse

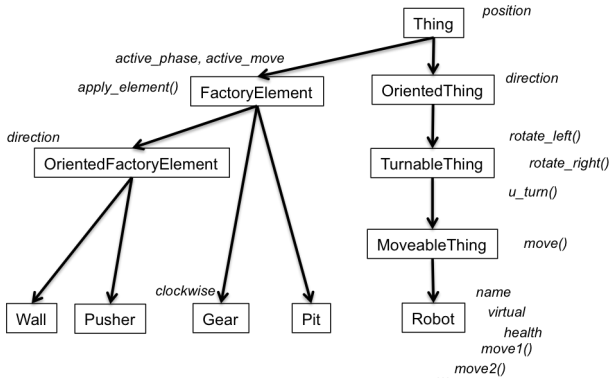
Exkurs:
Mehrfachver-
erbung

Programm-
entwurf

Ein kleiner
Test

Erweiterung:
Visualisie-
rung

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



Motivation

Die
Spielregeln

Eine OOP-
Analyse

Exkurs:
Mehrfachver-
erbung

Programm-
entwurf

Ein kleiner
Test

Erweiterung:
Visualisie-
rung



- 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.
- **Vereinfachungen:**
 - 1 Das Spielfeld wird am Anfang so initialisiert, das es durch **Gruben** (Pits) umgeben ist, so dass das Verlassen des Felds automatisch zum Tod führt.
 - 2 Für jede Wand auf einem Feld wird die **entsprechende Wand** im angrenzenden Feld eingeführt.
- Der interessanteste Punkt ist die Zusammenarbeit zwischen der Factory und den Robotern.

Motivation

Die
Spielregeln

Eine OOP-
Analyse

Exkurs:
Mehrfachver-
erbung

Programm-
entwurf

Ein kleiner
Test

Erweiterung:
Visualisie-
rung

- 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 statische Methoden.

Motivation

Die
Spielregeln

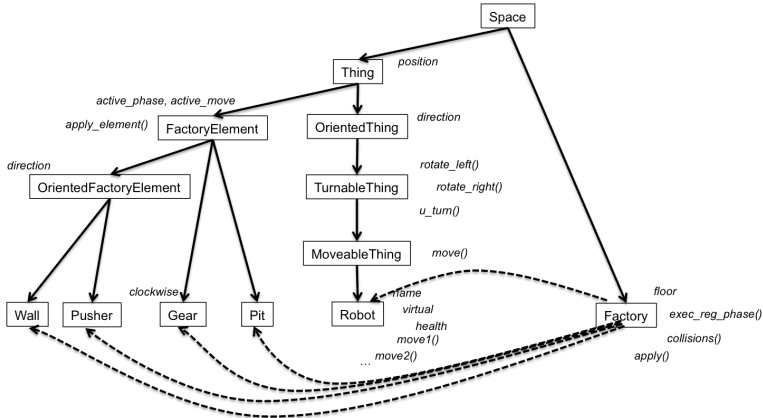
Eine OOP-
Analyse

Exkurs:
Mehrfachver-
erbung

Programm-
entwurf

Ein kleiner
Test

Erweiterung:
Visualisie-
rung



Motivation

Die
Spielregeln

Eine OOP-
Analyse

Exkurs:
Mehrfachver-
erbung

Programm-
entwurf

Ein kleiner
Test

Erweiterung:
Visualisie-
rung

4 Exkurs: Mehrfachvererbung



Motivation

Die
Spielregeln

Eine OOP-
Analyse

**Exkurs:
Mehrfachver-
erbung**

Programm-
entwurf

Ein kleiner
Test

Erweiterung:
Visualisie-
rung

- In dem Klassendiagramm ist auffallend, dass `OrientedThing` und `OrientedFactoryElement` vorkommen, die zweite Klasse aber **keine Unterklasse** von der ersteren ist.
- Tatsächlich ist die Orientierbarkeit eines Objekts **orthogonal** zu seiner übrigen Natur (ob Robot oder Fabrikelement).
- Und muss jedes `MoveableThing` auch drehbar sein? Muss es eine Orientierung haben?
- Wäre es da nicht besser, statt einem Vererbungsbaum einen Vererbungsgraphen zu haben?

Motivation

Die
Spielregeln

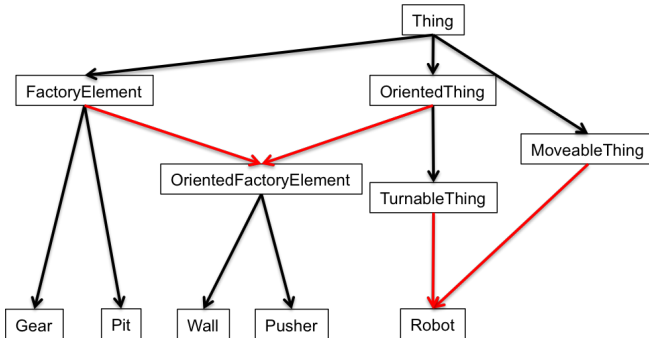
Eine OOP-
Analyse

Exkurs:
Mehrfachver-
erbung

Programm-
entwurf

Ein kleiner
Test

Erweiterung:
Visualisie-
rung



Motivation

Die
Spielregeln

Eine OOP-
Analyse

Exkurs:
Mehrfachver-
erbung

Programm-
entwurf

Ein kleiner
Test

Erweiterung:
Visualisie-
rung

Sowohl `OrientedFactoryElement` als auch `Robot` haben
zwei Superklassen!

- 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:
Mehrfachver-
erbung

Programm-
entwurf

Ein kleiner
Test

Erweiterung:
Visualisie-
rung



- `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: `MoveableThing` nach `OrientedThing`).

Motivation

Die
Spielregeln

Eine OOP-
Analyse

Exkurs:
Mehrfachver-
erbung

Programm-
entwurf

Ein kleiner
Test

Erweiterung:
Visualisie-
rung



- Im Normalfall kann man die MRO mit den Regeln *links vor rechts* und *Unterklasse vor Oberklasse*, wobei das *erste vor dem zweiten* angewandt wird, einfach selbst bestimmen.
- Im Beispiel: Robot, TurnableThing, OrientedThing, MoveableThing, Thing, (object).
- Die Standard-Klassenmethode `mro()` gibt die Liste der Oberklassen entsprechend der MRO aus.

Python-Interpreter

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

Motivation

Die
Spielregeln

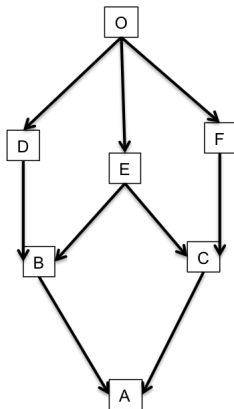
Eine OOP-
Analyse

Exkurs:
Mehrfachver-
erbung

Programm-
entwurf

Ein kleiner
Test

Erweiterung:
Visualisie-
rung



`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

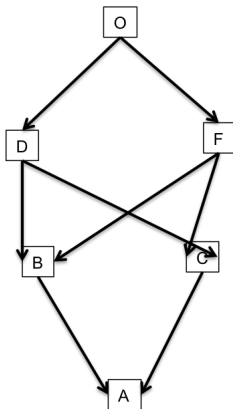
Eine OOP-
Analyse

Exkurs:
Mehrfachver-
erbung

Programm-
entwurf

Ein kleiner
Test

Erweiterung:
Visualisie-
rung



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:
Mehrfachver-
erbung

Programm-
entwurf

Ein kleiner
Test

Erweiterung:
Visualisie-
rung

- 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:
Mehrfachver-
erbung

Programm-
entwurf

Ein kleiner
Test

Erweiterung:
Visualisie-
rung



- 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 Element 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

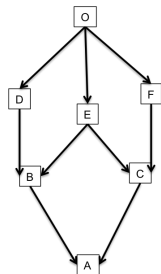
Exkurs:
Mehrfachver-
erbung

Programm-
entwurf

Ein kleiner
Test

Erweiterung:
Visualisie-
rung

Beispiel: Die komplexe Struktur



- 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:
Mehrfachver-
erbung

Programm-
entwurf

Ein kleiner
Test

Erweiterung:
Visualisie-
rung

5 Programmentwurf



Motivation

Die
Spielregeln

Eine OOP-
Analyse

Exkurs:
Mehrfachver-
erbung

**Programm-
entwurf**

Ein kleiner
Test

Erweiterung:
Visualisie-
rung

- 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:
Mehrfachver-
erbung

Program-
entwurf

Ein kleiner
Test

Erweiterung:
Visualisie-
rung

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(dir):
        return Space.left_trans[dir]
    def to_back(dir):
        return Space.left_trans[Space.left_trans[dir]]
    def to_right(dir):
        return Space.left_trans[Space.left_trans[
            Space.left_trans[dir]]]
    def neighbour(pos, dir):
        return(pos[0]+Space.move_xy[dir][0],
            pos[1]+Space.move_xy[dir][1])
    neighbour=staticmethod(neighbour)
```

Motivation

Die
Spielregeln

Eine OOP-
Analyse

Exkurs:
Mehrfachver-
erbung

Programm-
entwurf

Ein kleiner
Test

Erweiterung:
Visualisie-
rung



- Alle Dinge (innerhalb der Fabrik) haben eine **Position** pos, die sich natürlich bei beweglichen Dingen ändern kann!

roborally.py

```
class Thing(Space):  
    """Each thing has a position on the floor"""  
    def __init__(self, x, y):  
        self.pos = (x, y)
```

Motivation

Die
Spielregeln

Eine OOP-
Analyse

Exkurs:
Mehrfachver-
erbung

Programm-
entwurf

Ein kleiner
Test

Erweiterung:
Visualisie-
rung



- 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.

`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:
Mehrfachver-
erbung

Program-
entwurf

Ein kleiner
Test

Erweiterung:
Visualisie-
rung

- Alle Dinge, die man drehen kann, können ihre Orientierung ändern.

roborally.py

```
class TurnableThing(OrientedThing):  
  
    def rotate_left(self, *rest):  
        self.dir = Space.to_left(self.dir)  
  
    def u_turn(self, *rest):  
        self.dir = Space.to_back(self.dir)  
  
    def rotate_right(self, *rest):  
        self.dir = Space.to_right(self.dir)
```

Motivation

Die
Spielregeln

Eine OOP-
Analyse

Exkurs:
Mehrfachver-
erbung

Program-
entwurf

Ein kleiner
Test

Erweiterung:
Visualisie-
rung

- Alle Dinge, die man bewegen kann, haben eine letzte Konfiguration `lastconf` (für das Rücksetzen fehlgeschlagener Operationen).

`roborally.py`

```
class MoveableThing(TurnableThing):
```

```
    def __init__(self, x, y, dir="N", **kw):  
        super().__init__(x, y, dir, **kw)  
        self.lastconf = None # last conf., i.e. pos and dir
```

Motivation

Die
Spielregeln

Eine OOP-
Analyse

Exkurs:
Mehrfachver-
erbung

Programm-
entwurf

Ein kleiner
Test

Erweiterung:
Visualisie-
rung



- Entweder ist die Bewegung durch einen Agenten **initiiert** und im Ablauf priorisiert (Programmkarte) oder sie erfolgt **parallel** für alle Agenten (Förderband).
- Ausgelöst wird die Bewegung durch ein **Methodenaufruf** aus der Factory, wobei das Factory-Objekt als Parameter übergeben wird.
- Jeder Agent bewegt sich anhand der Regeln um ein Feld, und schiebt u.U. andere Agenten.
- Steht er vor einer **Wand**, bewegt er sich nicht (das erfährt er von der Factory!)
- Darauf aufbauend werden **Kollisionen** detektiert und die Bewegungen „**rückabgewickelt**“
- Diese Methode funktioniert sowohl für die Befehls-Bewegungen als auch die parallel ausführbaren Aktionen.

Motivation

Die Spielregeln

Eine OOP-Analyse

Exkurs: Mehrfachvererbung

Programm-entwurf

Ein kleiner Test

Erweiterung: Visualisierung



- 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.

Programm-
entwurf

Ein kleiner
Test

Erweiterung:
Visualisie-
rung

- Die Programmierkarten können eine Bewegung starten. Nachdem alle implizierten Bewegungen ausgeführt wurden (**pushes**), ist die Factory dafür verantwortlich, **Kollisionen** aufzulösen und die **Pit-Behandlung** zu starten.

`roborally.py`

```
def startmove(self, dir, factory):  
    self.move(dir, factory)  
    factory.resolve_conflicts()  
    factory.apply('last') # check for pits!
```

Motivation

Die
Spielregeln

Eine OOP-
Analyse

Exkurs:
Mehrfachver-
erbung

Programm-
entwurf

Ein kleiner
Test

Erweiterung:
Visualisie-
rung

MoveableThing-Klasse (5): Allgemeine Bewegung



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

`roborally.py`

```
def move(self, dir, factory):
    oldpos = self.pos
    self.lastconf = (self.pos, self.dir)
    self.pos = Space.neighbour(self.pos, dir)
    factory.apply('after') # check for walls
    if oldpos == self.pos:
        # return if stopped by wall
        self.lastconf = None
        return
    for collider in factory.collision(self):
        # if collision with another robot, push
        collider.move(dir, factory)
```

Motivation

Die
Spielregeln

Eine OOP-
Analyse

Exkurs:
Mehrfachver-
erbung

Programm-
entwurf

Ein kleiner
Test

Erweiterung:
Visualisie-
rung

- Alle Agenten werden gleichzeitig bewegt

`roborally.py`

```
def transport(self, dir, factory):  
    self.lastconf = (self.pos, self.dir)  
    self.pos = Space.neighbour(self.pos, dir)  
    factory.apply('after') # check for walls!
```

Motivation

Die
Spielregeln

Eine OOP-
Analyse

Exkurs:
Mehrfachver-
erbung

Programm-
entwurf

Ein kleiner
Test

Erweiterung:
Visualisie-
rung

MoveableThing-Klasse (7): Konfliktauflösung



- Bei Kollisionen wird die Bewegung zurückgenommen

robolly.py

```
def resolve(self, factory):
    "Is called after every robot has been moved"
    collider = factory.collision(self)
    if collider:
        for a in collider + [self]:
            a.retract(factory)
    self.lastconf = None
def retract(self, factory):
    if self.lastconf:
        self.pos = self.lastconf[0]
        self.dir = self.lastconf[1]
        self.lastconf = None
    for a in factory.collision(self):
        a.retract(factory)
```

Motivation

Die
Spielregeln

Eine OOP-
Analyse

Exkurs:
Mehrfachver-
erbung

Programm-
entwurf

Ein kleiner
Test

Erweiterung:
Visualisie-
rung

- Roboter haben zusätzliche Zustandsattribute und können Befehle ausführen.
- Ihre Druckdarstellung ist ihr Name.

roborally.py

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

    def __str__(self):
        return self.name.upper()
```

Motivation

Die
Spielregeln

Eine OOP-
Analyse

Exkurs:
Mehrfachver-
erbung

Programm-
entwurf

Ein kleiner
Test

Erweiterung:
Visualisie-
rung

- Ein Roboter kann *aktiv* einen Schritt vorwärts oder rückwärts fahren.

`roborally.py`

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

Motivation

Die
Spielregeln

Eine OOP-
Analyse

Exkurs:
Mehrfachver-
erbung

Programm-
entwurf

Ein kleiner
Test

Erweiterung:
Visualisie-
rung

roborally.py

```
class Robot(MoveableThing):
    ...
    def move1(self, factory):
        self.onestep(True, factory)
    def move2(self, factory):
        self.onestep(True, factory)
        self.onestep(True, factory)
    def move3(self, factory):
        self.onestep(True, factory)
        self.onestep(True, factory)
        self.onestep(True, factory)
    def backup(self, factory):
        self.onestep(False, factory)
# rotate cmds are implemented in TurnableThing
```

Motivation

Die
Spielregeln

Eine OOP-
Analyse

Exkurs:
Mehrfachver-
erbung

Programm-
entwurf

Ein kleiner
Test

Erweiterung:
Visualisie-
rung

roborally.py

```
class FactoryElement(Thing):
    active_reg_phase = {1, 2, 3, 4, 5}
    active_steps = { }

    def apply_element(self, agent, factory):
        if (factory.step in
            self.active_steps and
            factory.reg_phase in
            self.active_reg_phases and
            agent.alive):
            self.acton(agent, factory)
    def acton(self, agent, factory):
        raise NotImplementedError("acton undef")
```

Motivation

Die
Spielregeln

Eine OOP-
Analyse

Exkurs:
Mehrfachver-
erbung

Program-
entwurf

Ein kleiner
Test

Erweiterung:
Visualisie-
rung



- Bisher hatten wir als Kombinationsmechanismen für Methoden kennen gelernt:
 - 1 Von Superklasse **ererbten** und unmodifiziert nutzen.
 - 2 Die Superklassenmethode durch eigene Methode **überschreiben**.
 - 3 Die Superklassenmethode **erweitern**, durch Aufruf von `super()`.
- Hier haben wir den Fall, dass die Superklasse die Erledigung der Aufgabe an eine **Subklasse delegiert**. Alle Subklassen müssen die `action`-Methode implementieren. Sonst sind sie nicht lauffähig.
- Man spricht auch von **abstrakten Klassen**, die keine Instanzen haben können.

Motivation

Die
Spielregeln

Eine OOP-
Analyse

Exkurs:
Mehrfachver-
erbung

Programm-
entwurf

Ein kleiner
Test

Erweiterung:
Visualisie-
rung

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:
Mehrfachver-
erbung

Programm-
entwurf

Ein kleiner
Test

Erweiterung:
Visualisie-
rung

Ein Mauer schickt den Agenten **zurück**, falls die Mauer „durchschritten“ wurde.

`robolly.py`

```
class Wall(OrientedFactoryElement):
    active_steps = {0, 1, 2, 3, 4, 5, 6}
    # We have always the corresponding walls
    def acton(self, agent, factory):
        if agent.lastconf:
            if (agent.lastconf[0] ==
                Space.neighbour(self.pos, self.dir)):
                agent.pos = agent.lastconf[0]
                agent.dir = agent.lastconf[1]
                agent.lastconf = None
```

Motivation

Die
Spielregeln

Eine OOP-
Analyse

Exkurs:
Mehrfachver-
erbung

Programm-
entwurf

Ein kleiner
Test

Erweiterung:
Visualisie-
rung



roborally.py

```
class Factory(Space):
    def __init__(self, cols=5, rows=5, installs=None):
        self.agents = []
        self.rows = rows
        self.cols = cols
        self.step = 0
        self.reg_phase = 0
        self.floor = dict()
        # ordinary elements
        self.floor['before'] = dict()
        # after move, i.e., walls
        self.floor['after'] = dict()
        # after move completed, i.e., pits
        self.floor['last'] = dict()
        self.init_floor(cols, rows, installs)
```

Motivation

Die
Spielregeln

Eine OOP-
Analyse

Exkurs:
Mehrfachver-
erbung

Programm-
entwurf

Ein kleiner
Test

Erweiterung:
Visualisie-
rung

Finally: Factory-Klasse (2): Methoden



roborally.py

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

Motivation

Die
Spielregeln

Eine OOP-
Analyse

Exkurs:
Mehrfachver-
erbung

Programm-
entwurf

Ein kleiner
Test

Erweiterung:
Visualisie-
rung

6 Ein kleiner Test



Motivation

Die
Spielregeln

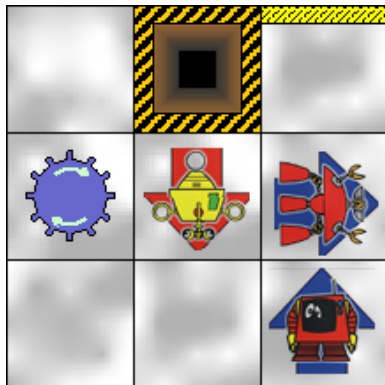
Eine OOP-
Analyse

Exkurs:
Mehrfachver-
erbung

Programm-
entwurf

**Ein kleiner
Test**

Erweiterung:
Visualisie-
rung



Motivation

Die
Spielregeln

Eine OOP-
Analyse

Exkurs:
Mehrfachver-
erbung

Program-
entwurf

Ein kleiner
Test

Erweiterung:
Visualisie-
rung

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

roborally.py

```
if __name__ == "__main__":
    t = Robot(3, 1, "N", "Twonky")
    s = Robot(2, 2, "S", "Spinbot")
    h = Robot(3, 2, "E", "HulkX90")
    fac = Factory(3, 3,
                  installs=(Wall(3, 3, "N"),
                             Pit(2, 3),
                             Gear(1, 2, True),
                             t, s, h))
    fac.exec_reg_phase(1, (t.move2,))
    fac.exec_reg_phase(2, (t.rotate_left,
                           h.rotate_right))
    fac.exec_reg_phase(3, (t.move2,
                           h.move1, s.u_turn))
```

Motivation

Die
Spielregeln

Eine OOP-
Analyse

Exkurs:
Mehrfachver-
erbung

Programm-
entwurf

Ein kleiner
Test

Erweiterung:
Visualisie-
rung

Python-Interpreter

```
*** Starting register phase 1
MOVE2 command: TWONKY
onestep: TWONKY wants to make 1 step forw. (dir=N)
startmove: TWONKY wants to go from (3, 1) to (3, 2)
move: try move of TWONKY from (3, 1) to (3, 2)
move: try move of HULKX90 from (3, 2) to (3, 3)
onestep: TWONKY wants to make 1 step forw. (dir=N)
startmove: TWONKY wants to go from (3, 2) to (3, 3)
move: try move of TWONKY from (3, 2) to (3, 3)
move: try move of HULKX90 from (3, 3) to (3, 4)
HULKX90 bumped into a wall and is back at (3, 3)
move: HULKX90 cannot move because of an obstacle
retract: send TWONKY back to (3, 2)
```

Motivation

Die
Spielregeln

Eine OOP-
Analyse

Exkurs:
Mehrfachver-
erbung

Programm-
entwurf

Ein kleiner
Test

Erweiterung:
Visualisie-
rung

Python-Interpreter

```
*** Starting register phase 2
rotate_left: TWONKY facing now W
rotate_right: HULKX90 facing now S
*** Starting register phase 3
MOVE2 command: TWONKY
onestep: TWONKY wants to make 1 step forw. (dir=W)
startmove: TWONKY wants to go from (3, 2) to (2, 2)
move: try move of TWONKY from (3, 2) to (2, 2)
move: try move of SPINBOT from (2, 2) to (1, 2)
onestep: TWONKY wants to make 1 step forw, (dir=W)
startmove: TWONKY wants to go from (2, 2) to (1, 2)
move: try move of TWONKY from (2, 2) to (1, 2)
move: try move of SPINBOT from (1, 2) to (0, 2)
SPINBOT fell into a pit at (0, 2)
MOVE1 command: HULKX90 . . .
```

Motivation

Die
Spielregeln

Eine OOP-
Analyse

Exkurs:
Mehrfachver-
erbung

Programm-
entwurf

Ein kleiner
Test

Erweiterung:
Visualisie-
rung

7 Erweiterung: Visualisierung



Motivation

Die
Spielregeln

Eine OOP-
Analyse

Exkurs:
Mehrfachver-
erbung

Programm-
entwurf

Ein kleiner
Test

**Erweiterung:
Visualisie-
rung**



- Könnten wir nicht **einfach** eine Visualisierung mit tkinter einbinden?
- Kleine Bilder aus dem Netz als GIFs
- Wir brauchen das Modul **PIL** (Python Image Library) um z.B. Rotation von Bildern durchführen zu können. Mit pip3 das Paket `Pillow` nachladen.
- Den Quelltext des vorhanden Codes **ändern**?
- Nein: Die Programmlogik ist ja OK. Erweiterung durch neue **Subklassen**, die die **Darstellung** behandeln!
- Jede dieser neuen Klasse erbt von der **Roborally-Klasse** (für die Programmlogik) und von der nächsthöheren **Darstellungsklasse**.

Motivation

Die
Spielregeln

Eine OOP-
Analyse

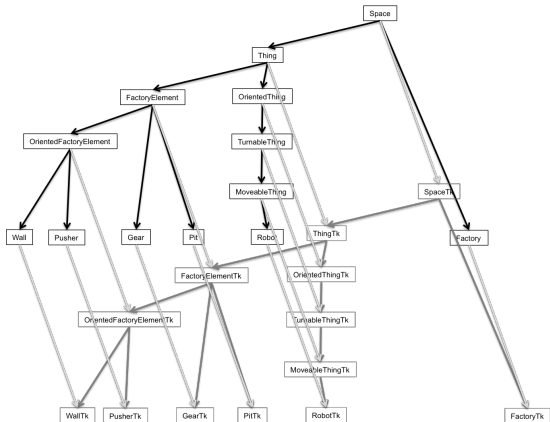
Exkurs:
Mehrfachver-
erbung

Programm-
entwurf

Ein kleiner
Test

Erweiterung:
Visualisie-
rung

Neues finales Klassendiagramm



Motivation

Die
Spielregeln

Eine OOP-
Analyse

Exkurs:
Mehrfachver-
erbung

Programm-
entwurf

Ein kleiner
Test

Erweiterung:
Visualisie-
rung

- Sieht komplizierter aus, als es ist. Einfach Struktur kopieren und dann mechanisch Oberklassen eintragen.
- Massive Mehrfachverbundung (in Rauten)
- Jede Menge Importe (inklusive der roborally) notwendig

roborallyTk.py

```
from roborally import *  
import tkinter as tk  
from PIL import Image, ImageTk  
import time
```

Motivation

Die
Spielregeln

Eine OOP-
Analyse

Exkurs:
Mehrfachver-
erbung

Programm-
entwurf

Ein kleiner
Test

Erweiterung:
Visualisie-
rung

SpaceTk

```
class SpaceTk(Space):  
  
    width = 192  
    height= 192  
    angle_value = dict(N=0, E=270, S=180, W=90)  
  
    def angle(dir):  
        return SpaceTk.angle_value[dir]  
    angle = staticmethod(angle)  
  
    # load image from GIF file  
    def make_image(filename):  
        return Image.open("gifs/" + filename)  
    make_image = staticmethod(make_image)
```

Motivation

Die
Spielregeln

Eine OOP-
Analyse

Exkurs:
Mehrfachver-
erbung

Programm-
entwurf

Ein kleiner
Test

Erweiterung:
Visualisie-
rung

ThingTk

```
class ThingTk(Thing, SpaceTk):
    image = SpaceTk.make_image("thing.gif")

    def __init__(self, x, y, **kw):
        super().__init__(x, y, **kw)
        self.photo = ImageTk.PhotoImage(self.image)
        self.pic_index = 0; self.redraw()

    def redraw(self):
        if self.pic_index:
            cv.delete(self.pic_index); self.pic_index = 0
        if self.pos:
            self.pic_index = cv.create_image(self.pos[0]*64-32+1,
                SpaceTk.height-(self.pos[1]*64-32),
                image=self.photo)

        cv.update()
```

Motivation

Die
Spielregeln

Eine OOP-
Analyse

Exkurs:
Mehrfachver-
erbung

Programm-
entwurf

Ein kleiner
Test

Erweiterung:
Visualisie-
rung

- Jedes Objekt besitzt ein (PIL) `image`; defaultmäßig `"thing.gif"`.
- `tkinter` kann mit dem PIL-Bild nichts anfangen, deshalb muss ein `PhotoImage` erzeugt werden.
- Dieses `PhotoImage` kann nicht einfach als Ergebnis an `create_image` übergeben, sondern **muss** an eine Variable gebunden werden, damit es nicht verschwindet!
- Die `redraw`-Methode löscht das alte Bild, wenn vorhanden, und zeichnet es dann neu an der neuen Stelle.

Motivation

Die
Spielregeln

Eine OOP-
Analyse

Exkurs:
Mehrfachver-
erbung

Programm-
entwurf

Ein kleiner
Test

Erweiterung:
Visualisie-
rung

OrientedThingTk

```
class OrientedThingTk(OrientedThing, ThingTk):  
  
    def rotate(self, degree):  
        self.image = self.image.rotate(degree)  
        self.photo = ImageTk.PhotoImage(self.image)  
  
    def __init__(self, x, y, dir='N', **kw):  
        self.rotate(SpaceTk.angle(dir))  
        super().__init__(x, y, dir=dir, **kw)
```

- Orientierbare Dinge können in die richtige Richtung gedreht werden!
- Die selbe Methode kann zum dynamischen Reorientieren benutzt werden.

Motivation

Die
Spielregeln

Eine OOP-
Analyse

Exkurs:
Mehrfachver-
erbung

Programm-
entwurf

Ein kleiner
Test

Erweiterung:
Visualisie-
rung



TurnableThingTk

```
class TurnableThingTk(TurnableThing, OrientedThingTk):
    def redraw_rotated(self, degree):
        self.rotate(degree); self.redraw()

    def rotate_left(self, *rest):
        super().rotate_left(*rest)
        self.redraw_rotated(90)

    def u_turn(self, *rest): ...

    def rotate_right(self, *rest): ...
```

- Ergänzt die `redraw`-Methoden um das Neuzeichnen nach einer Rotation.

Motivation

Die
Spielregeln

Eine OOP-
Analyse

Exkurs:
Mehrfachver-
erbung

Programm-
entwurf

Ein kleiner
Test

Erweiterung:
Visualisie-
rung

MoveableThingTk

```
class MoveableThingTk(MoveableThing, TurnableThingTk):
    # visualize moves
    def move(self, *l, **kw):
        super().move(*l, **kw)
        self.redraw()

    def transport(self, *l, **kw):
        super().transport(*l, **kw)
        self.redraw()

    def retract(self, *l, **kw):
        super().retract(*l, **kw)
        self.redraw()
```

Motivation

Die
Spielregeln

Eine OOP-
Analyse

Exkurs:
Mehrfachver-
erbung

Programm-
entwurf

Ein kleiner
Test

Erweiterung:
Visualisie-
rung

- Der Rest der Klassen ist einfach umzusetzen.
- In der Initialisierung das richtige Bild wählen, ggfs. das Objekt drehen, u.U. bei der Darstellung anpassen.

PitTk

```
class PitTk(Pit, FactoryElementTk):  
  
    image = SpaceTk.make_image("pit.gif")  
  
    def acton(self, agent, factory):  
        super().acton(agent, factory)  
        cv.delete(self.pic_index)  
        self.pic_index = 0
```

Motivation

Die
Spielregeln

Eine OOP-
Analyse

Exkurs:
Mehrfachver-
erbung

Programm-
entwurf

Ein kleiner
Test

Erweiterung:
Visualisie-
rung



FactoryTk

```
class FactoryTk(Factory, SpaceTk):
    image = SpaceTk.make_image("openfloor.gif")

    def __init__(self, cols=5, rows=5, **kw):
        self.photo = ImageTk.PhotoImage(self.image)
        for x in range(cols):
            for y in range(rows):
                cv.create_image(x*64+32+1,
                               SpaceTk.height-(y*64+32),
                               image=self.photo, tags="floor")
        cv.tag_lower("floor")
        cv.update()
        super().__init__(cols=cols, rows=rows, **kw)
```

Motivation

Die
Spielregeln

Eine OOP-
Analyse

Exkurs:
Mehrfachver-
erbung

Programm-
entwurf

Ein kleiner
Test

Erweiterung:
Visualisie-
rung

- Beim Erzeugen mit `create_image` haben wir den Bildern eine Markierung (**Tag**) mitgegeben, unter dem wir alle diese Bilder wieder finden können.
- Am Schluss werden alle Bilder mit dem *Tag* "floor" ganz nach **unten** gelegt. D.h. alle anderen Bilder liegen **über** diesen Bildern, wenn sie an der gleichen Stelle sichtbar gemacht werden.
- Jetzt müssen wir bloß noch unser Testprogramm anpassen ...

Motivation

Die Spielregeln

Eine OOP-Analyse

Exkurs: Mehrfachvererbung

Programm-entwurf

Ein kleiner Test

Erweiterung: Visualisierung

- Der Anspruch war gewesen, das Design an der Erweiterbarkeit auszurichten.
- Ist das gelungen?
- Viele Fabrikelemente lassen sich leicht integrieren (z.B. Crusher, Portal, Temp-Door)
- Einige brauchen zusätzlichen Aufwand (z.B. Laser)
- 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

Motivation

Die
Spielregeln

Eine OOP-
Analyse

Exkurs:
Mehrfachver-
erbung

Programm-
entwurf

Ein kleiner
Test

Erweiterung:
Visualisie-
rung