12. Vom Entwurf zur Implementation

Java-Beispiel:
- Maze.java
- Mouse.java
- MazeTest.java
- MouseMaze.java
- Easel.java
- SoftFrame.java

Schwerpunkte

- Planung der Implementationsschritte ausgehend von der Softwarearchitektur
- Implementation von Labyrinth und Maus
- Separate Implementation und separater Test der Komponenten

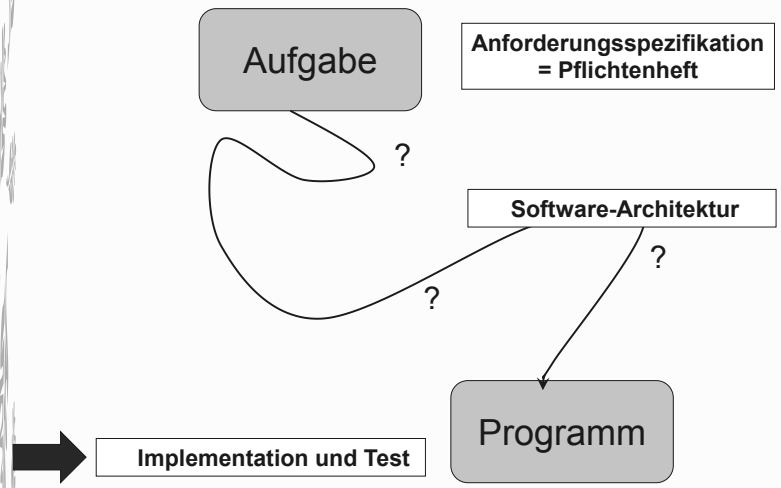
Entwicklungsprozess des Programms 'Maus im Labyrinth'

Anforderungsanalyse

Design

→ Implementation und Test

Vom Problem zum Programm: Maus im Labyrinth



Java Quellen

Klassen	LOC	
Mouse	61	**Maus**
Maze	55	**Labyrinth**
MouseMaze	64	**Algorithmus**
Easel	23	**textuelle Ausgabe des Labyrinths**
SoftFrame	135	
MazeTest	40	**Test des Labyrinths**
Summe:	378	

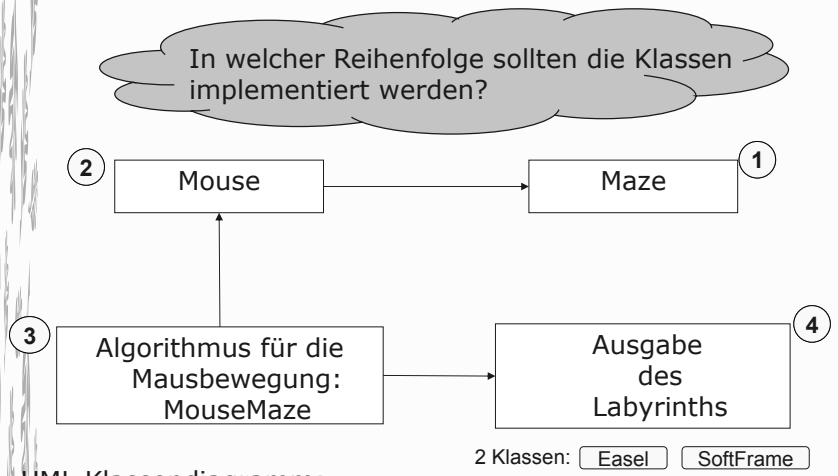
→ Vorlesung: oft nur Grundidee erläutert
→ Selbst: viele (z. T. trickreiche) Details erschließen

```
turnleft()     {
   direction = (direction + 3) % 4;
}
```


Labyrinth: textuelle Ausgabeform

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx          x
x                                             x
x                                             x
x                                             x
x                                             x
x                                             x
x                                             x
x                                             x
x                                             x
x           x           x           xxxxxxxxxxx
            x           x                     x
            x           x                     x
            x           x                     x
            x           x                     x
            x           x                     x
            x           x                     x
            x           x                     x
            x           x                     x
x           x           x           x         x
x           x                       x         x
x           x                       x         x
x           x                       x         x
x           x                       x         x
x           x                       x         x
x           x                       x         x
x           x                       x         x
x           x                       x         x
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

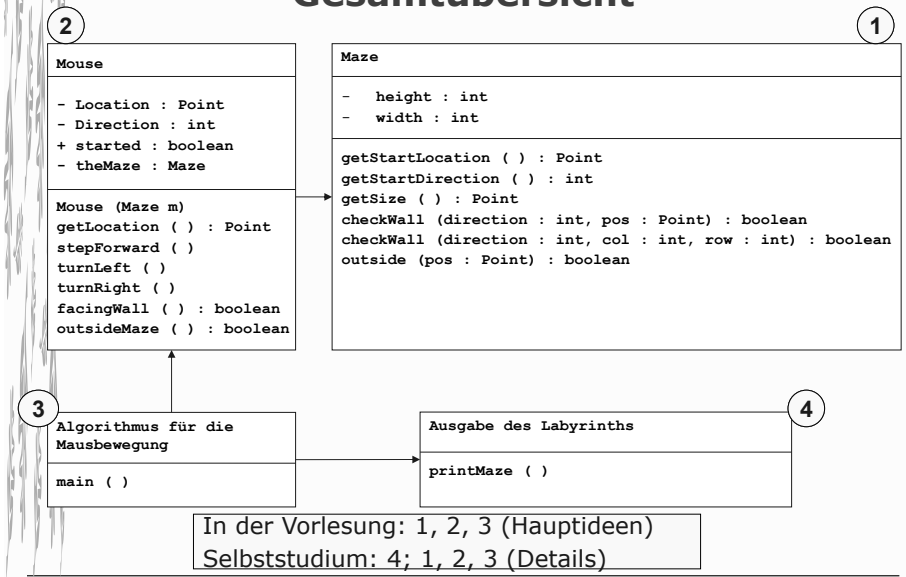

Planung der Implementationsschritte



2 Klassen: Easel SoftFrame

- UML Klassendiagramm:
  - Abhängigkeiten → Implementationsreihenfolge
  - ABER: durch Interface beliebige Reihenfolge möglich (Impl., nicht Test)

Softwarearchitektur: Gesamtübersicht

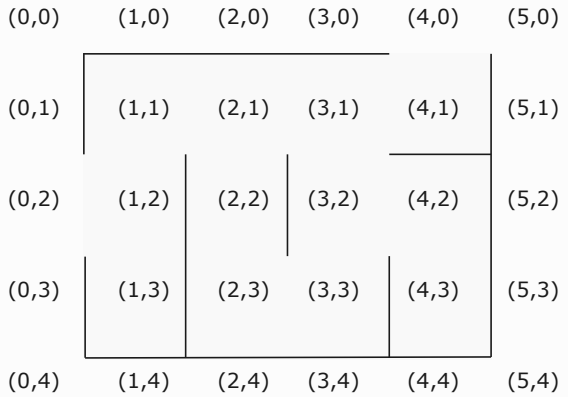


In der Vorlesung: 1, 2, 3 (Hauptideen)
Selbststudium: 4; 1, 2, 3 (Details)

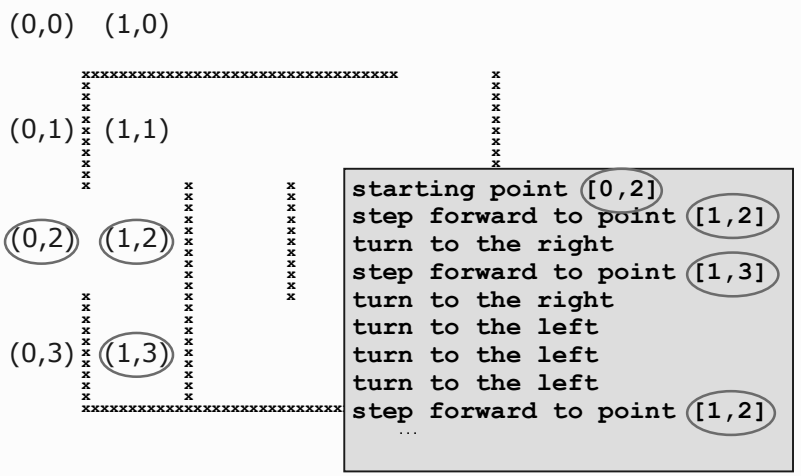
Implementation des Labyrinths

Maze.java

Labyrinth: Koordinaten festlegen



Ausgabe der Lösung: vollständiger Bewegungspfad mit Koordinaten



Interface der Komponenten: durch Java–Klasse implementieren

- Beginnend beim Klassendiagramm

Maze

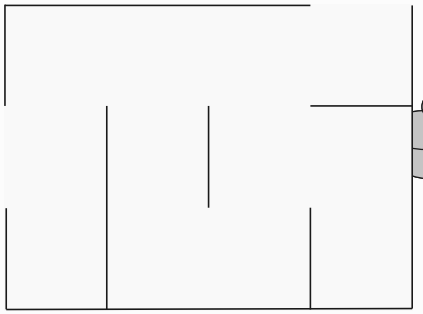
- **height : int**
- **width : int**

Zustand besteht aus weiteren Daten

```
getStartLocation(): Point
getStartDirection(): int
getSize(): Point
checkWall(direction: int, pos: Point): boolean
checkWall(direction: int, col: int, row: int): boolean
outside(pos: Point): boolean
```

Grundproblem für das Labyrinth:
Wie durch Datenstrukturen repräsentieren ?

Implementation des Labyrinths



Wie stellt man das Labyrinth mittels Java-Datenstrukturen dar ?

Diskussion von Varianten: Vor- und Nachteile

Beispiel: 2-dimensionales Array:
- Jedes Element entspricht einem Raum
- Je Raum: 4 Angaben zu Wand/Öffnung

Probleme:
Redundanz und Widersprüchlichkeit

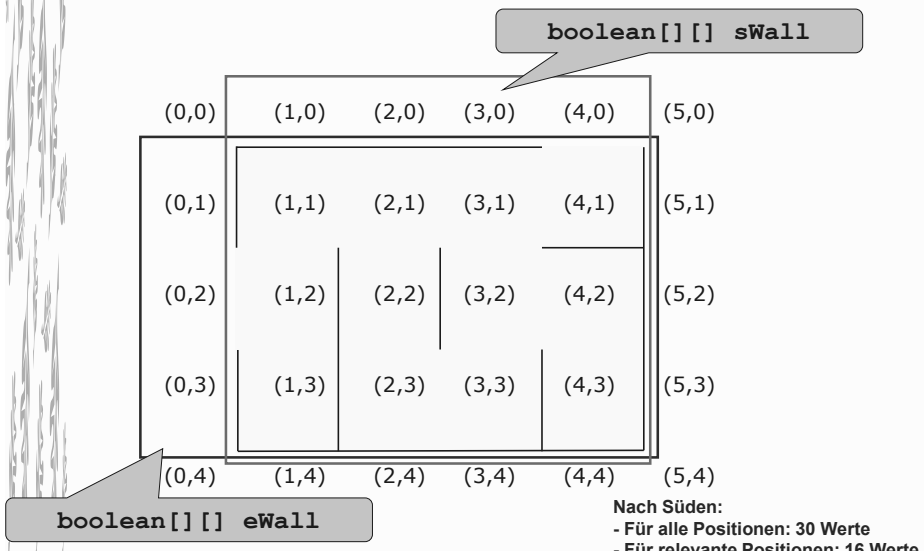
Repräsentation des Labyrinths: Wände abspeichern

Maze.java

Für jede *relevante* Position:

- Befindet sich südlich eine Wand?
  → `boolean[][] sWall`
- Befindet sich östlich eine Wand?
  → `boolean[][] eWall`

Repräsentation des Labyrinths: Wände für relevante Positionen abspeichern

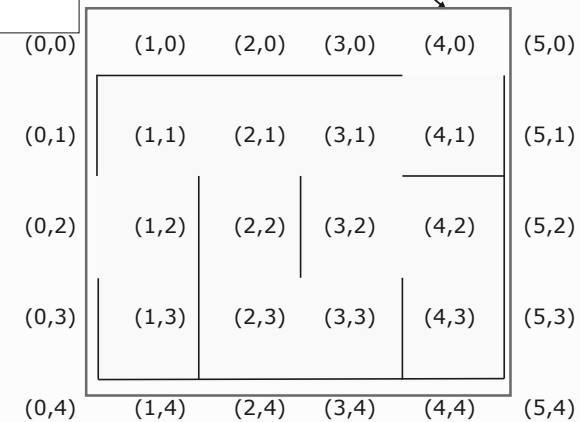


Nach Süden:
- Für alle Positionen: 30 Werte
- Für relevante Positionen: 16 Werte

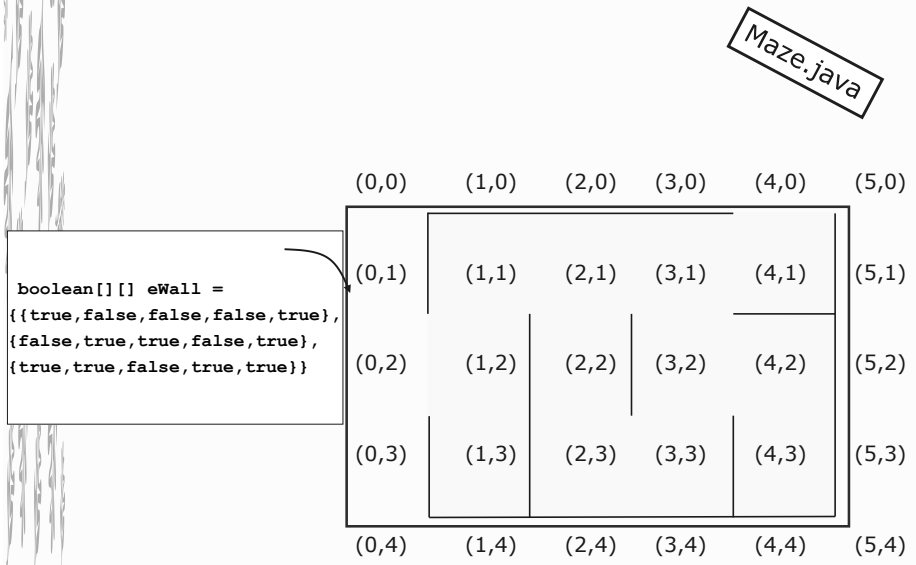
Wände des Labyrinths (Süd): Beispiel

```
boolean[][] sWall =
{{true, true, true, false},
 {false, false, false, true},
 {false, false, false, false},
 {true, true, true, true}}
```

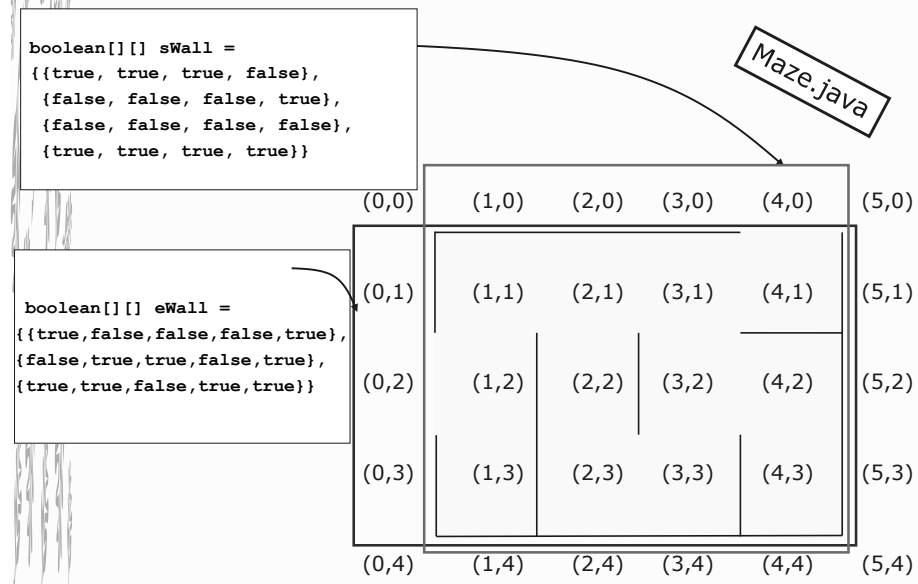
Maze.java



Wände des Labyrinths (Ost): Beispiel



Wände des Labyrinths: Beispiel



Nächster Schritt: Implementation der Methoden

Grundlage: Repräsentation der Daten des Labyrinths

```
boolean[][] sWall =
{{true, true, true, false},
 {false, false, false, true},
 {false, false, false, false},
 {true, true, true, true}}
```

```
boolean[][] eWall =
{{true,false,false,false,true},
{false,true,true,false,true},
{true,true,false,true,true}}
```

```
getStartLocation(): Point
getStartDirection(): int
getSize(): Point
checkWall(direction: int, pos: Point): boolean
checkWall(direction: int, col: int, row: int): boolean
outside(pos: Point): boolean
```


Die Methode 'outside'

pos = (x, y), z.B. (0, 2), (1, 3)

Maze.java

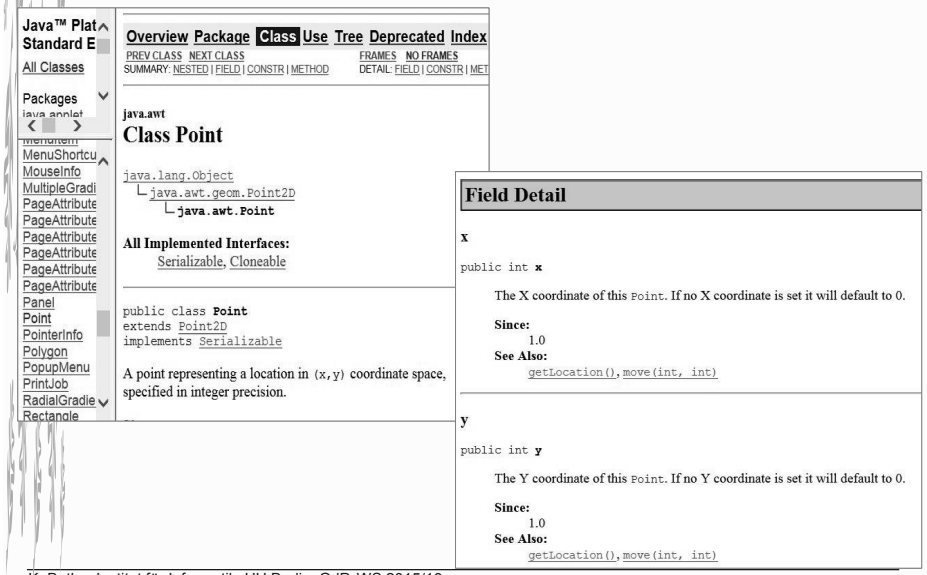
```
public boolean outside(Point pos){
  return ((pos.x < 1)          // links ...
  ||      (pos.x > width)      // rechts ...
  ||      (pos.y < 1)          // oberhalb ...
  ||      (pos.y > height)     // unterhalb ...
  );                           // des Labyrinths
}
```

Java-API:

Point pos → repräsentiert Punkt (x , y)

Point: eine der wenigen API-Klassen mit sichtbaren Variablen
→ Komponentenart: Datensammlung

API-Klasse Point



Java™ Plat
Standard E
All Classes
Packages
MenuShortcu
MouseInfo
MultipleGradi
PageAttribute
PageAttribute
PageAttribute
PageAttribute
PageAttribute
PageAttribute
Panel
Point
PointerInfo
Polygon
PopupMenu
PrintJob
RadialGradie
Rectangle

Overview Package Class Use Tree Deprecated Index
PREV CLASS NEXT CLASS FRAMES NO FRAMES
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | MET

java.awt
Class Point

java.lang.Object
└ java.awt.geom.Point2D
└ java.awt.Point

All Implemented Interfaces:
Serializable, Cloneable

public class Point
extends Point2D
implements Serializable

A point representing a location in (x,y) coordinate space, specified in integer precision.

Field Detail

x

public int x

The X coordinate of this Point. If no X coordinate is set it will default to 0.

Since:
1.0
See Also:
getLocation(), move(int, int)

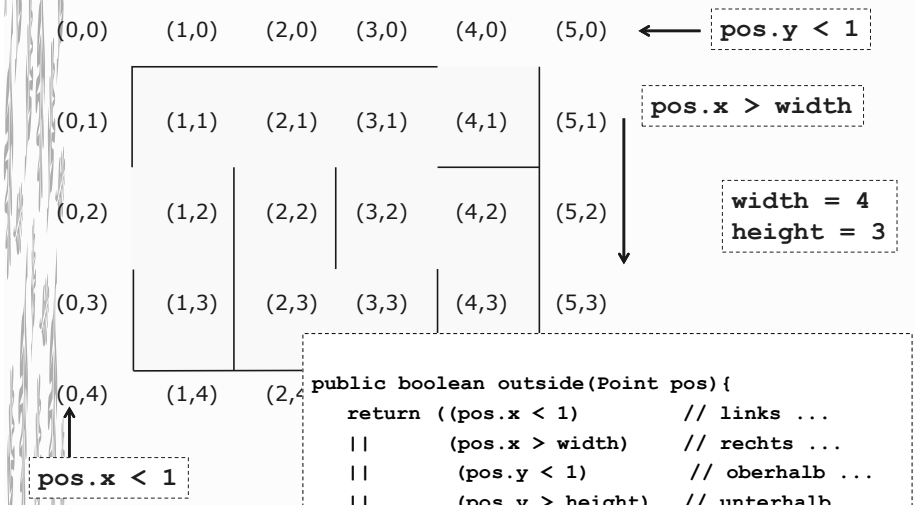
y

public int y

The Y coordinate of this Point. If no Y coordinate is set it will default to 0.

Since:
1.0
See Also:
getLocation(), move(int, int)

Die Methode 'outside'




```
public boolean outside(Point pos){
   return ((pos.x < 1)          // links ...
   ||      (pos.x > width)      // rechts ...
   ||      (pos.y < 1)          // oberhalb ...
   ||      (pos.y > height)     // unterhalb ...
   );                           // des Labyrinths
}
```


Richtungen

Maze.java

```
final static int
  NORTH = 0, EAST = 1, SOUTH = 2, WEST = 3;
```

Wird benötigt für den Wandtest: Punkt und Richtung gegeben

Alternative: Aufzählungstyp

Methode 'checkWall'

Maze.java

```
// Is there a wall to the 'dir' direction
// of location (col, row)?

public boolean checkWall(int dir, int col, int row) {
  switch (dir) {
    case NORTH: return sWall[row-1][col-1];
    case SOUTH: return sWall[row][col-1];
    case EAST:  return eWall[row-1][col];
    default:    return eWall[row-1][col-1];
  }
}
```

Warum gerade diese Indizes?

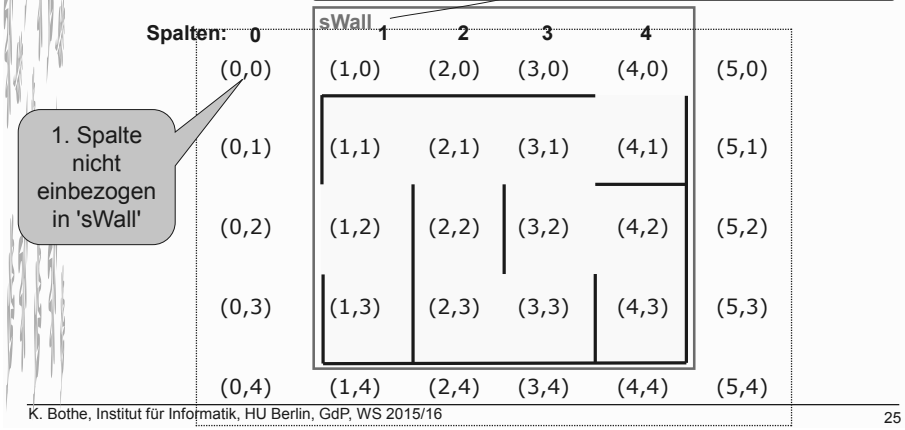
Positionen im Labyrinth-Gebiet
<> Indizes in sWall und eWall
(nur relevante Positionen gespeichert)

Methode 'checkWall': technisches Detail

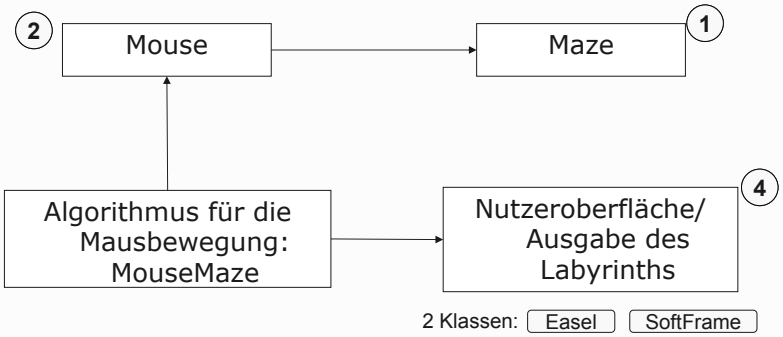
```
boolean checkWall (int dir, int col, int row) {
    switch (dir){
          case NORTH: return sWall[row-1][col-1];
          case SOUTH: return sWall[row][col-1];
          ...
```

Maze.java

Nur relevante Positionen → Spalte in sWall beginnt mit Index 0



Test des Programms: als Ganzes oder in Teilen?



Was heißt Testen? (Teilaktivitäten)

Komponenten: separate Implementation & separater Test

Test in mehreren Ebenen:

- **Komponententest:**
  - Test der Komponente unabhängig vom Rest des Systems.
- **Integrationstest:**
  - Später: Test der Zusammenarbeit mit anderen Komponenten des Systems.

Test: Aufruf des Programms (bzw. von Komponenten) mit Eingabedaten und Überprüfung der Ergebnisse

Komponententest des Labyrinths

MazeTest.java

```
class Maze {…
}
```

zu testende Komponente

```
class MazeTest {…
}
```

Testrahmen für die Klasse Maze

Wie kann man ein Labyrinth testen?

Klasse MazeTest: Testrahmen für die Klasse Maze

MazeTest.java

```
public class MazeTest {
  . . .
  public static void main (String[] args) {
     // create a maze
    Maze theMaze = new Maze();
    Point mazeSize = theMaze.getSize();

     // erwartete Resultate :  "+" tatsaechliche Resultate
    System.out.println("start location is (0,2):"
                       +   theMaze.getStartLocation());
    System.out.println("start direction is EAST = 1: "
                       + theMaze.getStartDirection());
    System.out.println("outside true: "
                       +   theMaze.outside (new Point(0,2)));
    System.out.println("not outside -> false:"
                       + theMaze.outside (new Point(4,3)));
    System.out.println("not outside -> false: "
                       + theMaze.outside(new Point(2,2)));
    System.out.println("wall -> true: "
                       + theMaze.checkWall(NORTH, 1, 1));
    System.out.println("no wall -> false: "
                       + theMaze.checkWall(SOUTH, 1, 1));
    System.out.println("wall -> true: "
                       + theMaze.checkWall(EAST,4,1));
 }
}
```

Das Interface des Labyrinths testen: Jede Methode wird getestet.

Testrahmen: Ausgabe der Testergebnisse

Ausgabe: Resultate eines Testlaufes

```
% java MazeTest0
Testergebnisse:
[x=0,y=2]
1
true
false
false
true
false
true
```

unzureichend

Testrahmen: Prinzip

MazeTest.java

```
 // erwartete Resultate :  "+" tatsaechliche Resultate
System.out.println("start location is (0,2):"
                   +   theMaze.getStartLocation());
System.out.println("start direction is EAST = 1: "
                   + theMaze.getStartDirection());
System.out.println("outside true: "
                   +   theMaze.outside (new Point(0,2)));
System.out.println("not outside -> false:"
                   + theMaze.outside (new Point(4,3)));
System.out.println(„no wall -> false: "
                   + theMaze.checkWall(SOUTH, 1, 1));
System.out.println("wall -> true: "
                   + theMaze.checkWall(EAST, 4, 1));
```

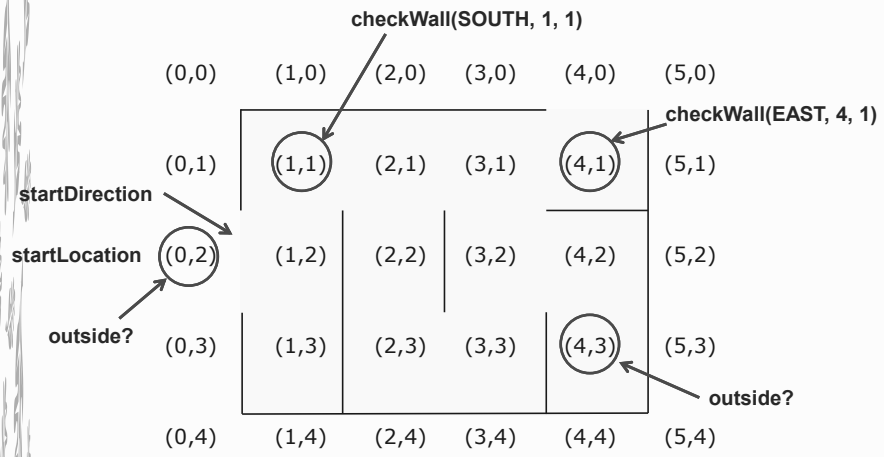
vorher: Semantische Interpretation

erwarteter Wert

tatsächlicher Wert

Eingabedaten

Labyrinth: getestete Situationen



Testrahmen: Ausgabe der Testergebnisse

Schnelle Überschaubarkeit korrekter bzw. falscher Resultate

```
% java MazeTest0
Testergebnisse:
[x=0,y=2]
1
true
false
false
true
false
tr
```

unzureichend

vorher: Semantische Interpretation

```
% java MazeTest
Start location is (0,2): [x=0,y=2]
Start direction is EAST = 1: 1
Outside true : true
Not outside -> false : false
Not outside -> false : false
Wall -> true : true
No wall -> false : false
Wall -> true : true
```

erwarteter Wert

tatsächlicher Wert

Testrahmen: Ausgabe der Testergebnisse

Schnelle Überschaubarkeit korrekter bzw. falscher Resultate

Nächster Schritt:
Automatisierter Vergleich, d.h. tatsächlicher Wert = erwarteter Wert?

```
% java MazeTest
Start location is (0,2): [x=0,y=2]
Start direction is EAST = 1: 1
Outside true : true
Not outside -> false : false
Not outside -> false : false
Wall -> true : true
No wall -> false : false
Wall -> true : true
```

vorher: Semantische Interpretation

erwarteter Wert

tatsächlicher Wert

Implementation der Maus

Implementation der Maus

- Ausgangspunkt: Interface / Architektur

Mouse.java

```
Mouse
- Location : Point
- Direction : int
+ started : boolean
- theMaze : Maze
Mouse (Maze m)
getLocation ( ) : Point
stepForward ( )
turnLeft ( )
turnRight ( )
facingWall ( ) : boolean
outsideMaze ( ) : boolean
```

Das sind schon alle Daten (Attribute) der Maus

Aktuelle Position: Location

Position bzw. Richtung ändern (Labyrinth spielt Keine Rolle)

Implementation: Anfrage an das Labyrinth → **Botschaft** an das Labyrinth senden

Kommunikation zwischen den Objekten: Senden einer Botschaft

Mouse.java

Die Maus sendet eine Botschaft an das Labyrinth: 'Falls ich eine bestimmte Position und Richtung im Labyrinth habe, stehe ich dann vor einer Wand?'

Operation der Maus

```
public boolean facingWall() {
   return
       theMaze.checkWall (direction, location);
}
```

Operation des Labyrinths

Sender → Empfänger

Versenden einer **Botschaft** an ein anderes Objekt (Labyrinth)
= Aufruf einer Methode dieses Objektes

→ Reaktion des Empfängerobjekts:
z. B. verändert sich oder gibt Wert zurück
(hier: Wert 'true' oder 'false')

Methoden: 'turnLeft' und 'turnRight'

Änderung der Richtung (int direction)

Mouse.java

```
public void turnLeft() {
   printoutMove("turn to the left");
   direction = (direction + 3) % 4;
}
```

Nach links:
Richtung + 3
(aber "Modulo")

```
public void turnRight() {
   printoutMove("turn to the right");
   direction = (direction + 1) % 4;
}
```

Nach rechts:
Richtung + 1
(aber "Modulo")

```
final static int
   NORTH = 0, EAST = 1, SOUTH = 2, WEST = 3;
```


Methode: 'stepForward'

Änderung der Position (Point location)

Mouse.java

```
public void stepForward() {
   switch (direction) {
       case NORTH:  location.y--; break;
       case EAST:   location.x++; break;
       case SOUTH:  location.y++; break;
       case WEST:   location.x--; break;
   }
   printoutMove("step forward");
}
```

Protokollierung der Bewegung

Konstruktor

Mouse.java

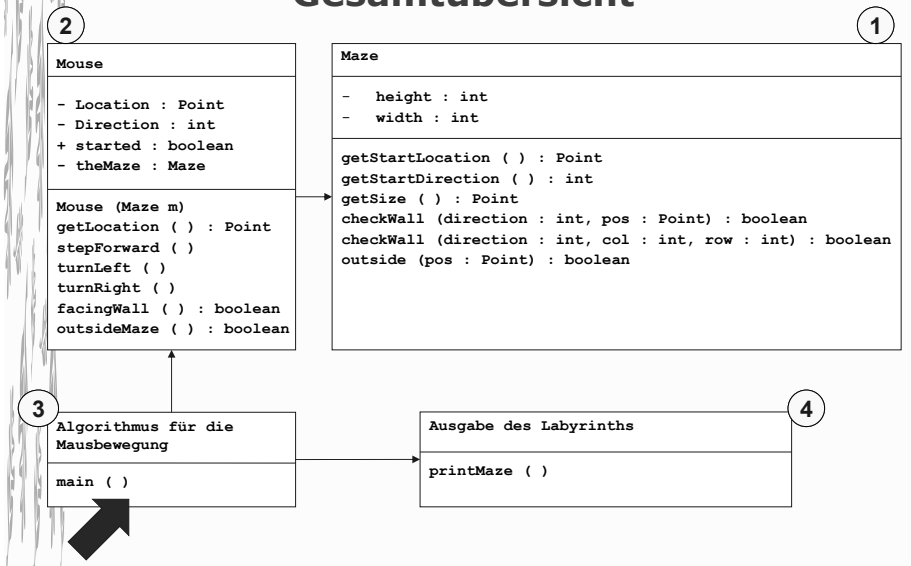
```
public Mouse(Maze m) {
   // Where do I start?
   location = m.getStartLocation();
   printoutMove("starting Point " +
   "[" + location.x + "," + location.y + "]");
   // In what direction do I face initially?
   direction = m.getStartDirection();
   theMaze = m; // my maze!
}
```

Operation des Labyrinths

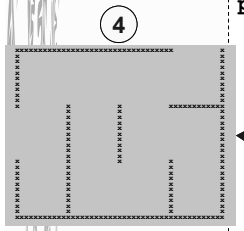
Protokollierung der Ausgangssituation

Implementation des Suchalgorithmus

Softwarearchitektur: Gesamtübersicht



Suchalgorithmus: Klasse 'MouseMaze' (1)



MouseMaze.java

```
public static void main ( ... ) {
   Maze theMaze = new Maze();
   Mouse littleMouse = new Mouse(theMaze);

   printMaze(theMaze);

   //move the mouse step by step
   do{
       makeStep(littleMouse);
   }
   while (!littleMouse.outsideMaze());
}
```


Suchalgorithmus: Klasse 'MouseMaze' (2)

MouseMaze.java

Bewegt die Maus einen Schritt, falls nötig inkl. Drehungen

```
public static void main ( ... ) {
   Maze theMaze = new Maze();
   Mouse littleMouse = new Mouse(theMaze);

   printMaze(theMaze);

   //move the mouse step by step
   do{
       makeStep(littleMouse);
   }
   while (!littleMouse.outsideMaze());
}
```

Suchalgorithmus: Klasse 'MouseMaze' (3)

```
private static void makeStep(Mouse m){
   if (m.started){
        if (!m.outsideMaze()){
                m.turnRight();
                while (m.facingWall()){
                        m.turnLeft();
                }
                m.stepForward();
        }
   } else {
        m.stepForward();
        m.started = true;
   }
}
```

sichtbares Attribut

Bewegt die Maus einen Schritt, falls nötig inkl. Drehungen

vgl. Pseudocodelösung: → dasselbe ?

Strenger Pseudocode: gesamter Algorithmus

```
step forward; // vom Eingang
WHILE(NOT outside the maze?)
  BEGIN // do next step
      turn right;
      WHILE (facing a wall?) DO
            turn left;
      ENDWHILE
      step forward;
  END
ENDWHILE
```


Kritikpunkte der Implementation (1)

- **Konstanten `NORTH = 0,..` mehrfach definiert in 3 Klassen:**
  - → Fehlerquelle
  - → besser: als Interface-Komponente nur einmal
- **Variable '`started`' von außen sichtbar!**
  - → stattdessen:
    - private + extra zwei Zugriffsoperationen
    - oder: Algorithmus ändern (!)

Suchalgorithmus: Algorithmus ändern → ohne 'started'

MouseMaze.java

```
public static void main ( ... ) {
   Maze theMaze = new Maze();
   Mouse littleMouse = new Mouse(theMaze);

   printMaze(theMaze);

   littleMouse.stepForward();      // NEU

   //move the mouse step by step
   do{
        makeStep(littleMouse);     // keine Test „started“ nötig
   }
   while (!littleMouse.outsideMaze());
}
```

Bewegt die Maus einen Schritt, falls nötig inkl. Drehungen

Suchalgorithmus: Algorithmus ändern → ohne 'started'

```
private static void makeStep(Mouse m){
   if (m.started){
        if (!m.outsideMaze()){
                m.turnRight();
                while (m.facingWall()){
                        m.turnLeft();
                }
                m.stepForward();
        }
   } else {
        m.stepForward();
        m.started = true;
   }
}
```

sichtbares Attribut

Bewegt die Maus einen Schritt, falls nötig inkl. Drehungen

Kritikpunkte der Implementation (2)

- **Datendarstellung des Labyrinths fehleranfällig**
  - true/false - Folgen korrekt?
    → graphisches Interface zur Eingabe des Labyrinths
  - Variablen **`height`**, **`width`** können im Widerspruch zu **`eWall, sWall`** stehen
  - Anfangsposition der Maus kann im Widerspruch zu **`eWall, sWall`** stehen
    → besser berechnen lassen