TECHNISCHE UNIVERSITÄT DARMSTADT

# Kapitel 01g:
# Interfaces mit FopBot

**Karsten Weihe**

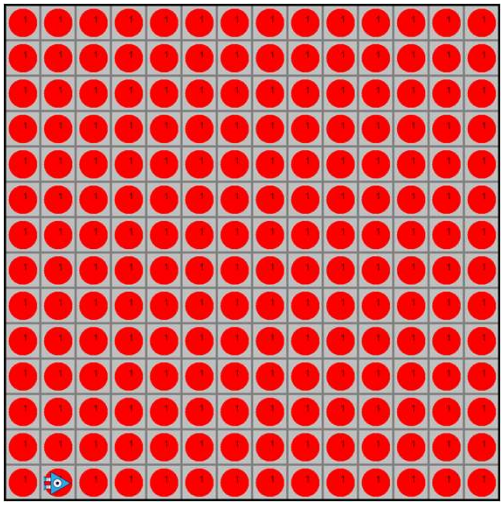
TECHNISCHE UNIVERSITÄT DARMSTADT

# Vierte eigene Roboterklasse: erster einfacher Pacman-Roboter

**Wir schauen uns jetzt mehrere Roboterklassen nacheinander an, deren Roboter nach verschiedenen Strategien durch die Gegend laufen und jede Münze aufnehmen, auf die sie treffen. Diese Roboterklassen benennen wir nach dem bekannten alten Spiel, in dem es so ähnliche Roboter gab.**

# Erster einfacher PacmanRobot

TECHNISCHE UNIVERSITÄT DARMSTADT

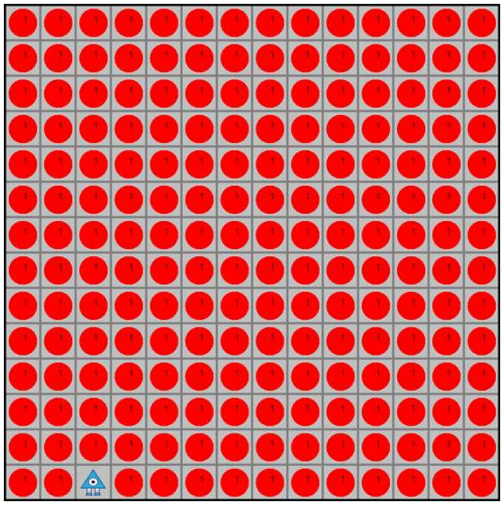


**Der erste einfache Pacman-Roboter soll immer im Zickzack gehen, das heißt, er soll nicht nur vorwärtsgehen, sondern auch seine Richtung ändern, immer abwechselnd nach links und rechts.**

**Unten links haben wir einen Roboter von dieser Art platziert.**

## Erster einfacher PacmanRobot

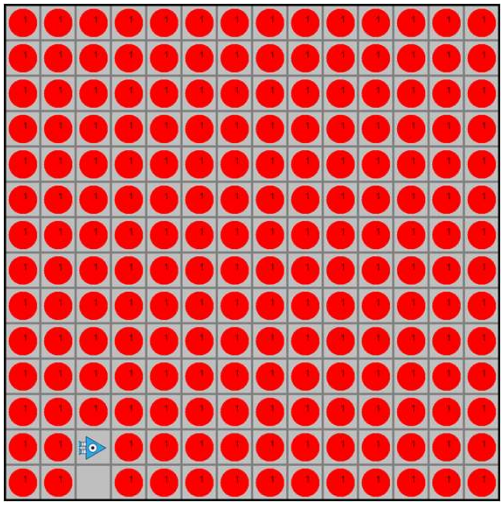
TECHNISCHE UNIVERSITÄT DARMSTADT



**Der Roboter soll neben move noch eine Methode bekommen, die wie move einen Schritt vorwärtsgeht. Aber sie macht auf dem neuen Feld noch zwei Dinge: Erstens nimmt sie alle Münzen auf, die auf dem neuen Feld sind, zweitens dreht sie den Roboter um 90 Grad. Diese Drehung soll immer abwechselnd sein: War die Drehung bei einem Aufruf der neu zu erstellenden Methode nach links, soll sie beim nächsten Aufruf nach rechts sein und umgekehrt.**

## Erster einfacher PacmanRobot

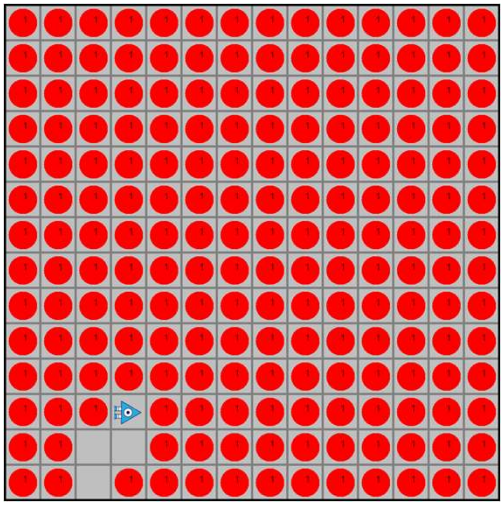
TECHNISCHE UNIVERSITÄT DARMSTADT



**Nach einem weiteren Aufruf sehen wir gut, dass auf dem vorhergehenden Feld keine Münzen mehr sind. Der Roboter hat sich diesmal nach rechts gewandt.**

## Erster einfacher PacmanRobot

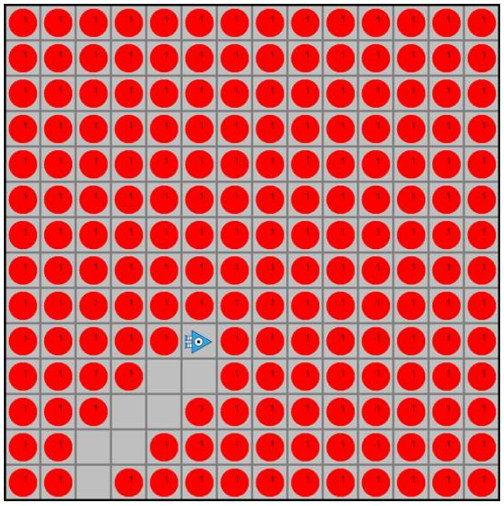
TECHNISCHE UNIVERSITÄT DARMSTADT



**Nach zwei weiteren Schritten sehen wir schon andeutungsweise, welche Spur der Roboter durch das Münzenfeld zieht.**

## Erster einfacher PacmanRobot

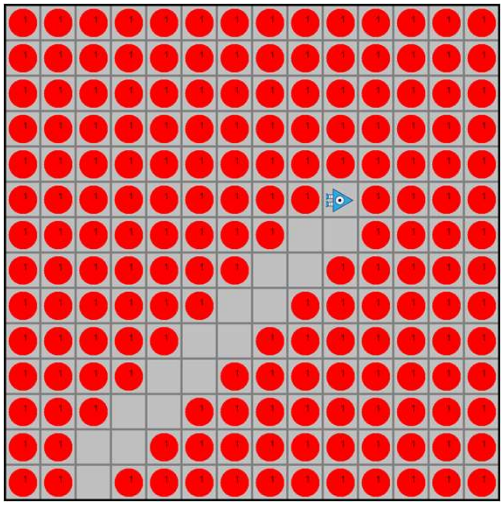
TECHNISCHE UNIVERSITÄT DARMSTADT



**Nach nochmals vier weiteren Schritten wird das Muster vollends deutlich. Der Roboter frisst sich praktisch auf Treppenstufen vorwärts.**

## Erster einfacher PacmanRobot

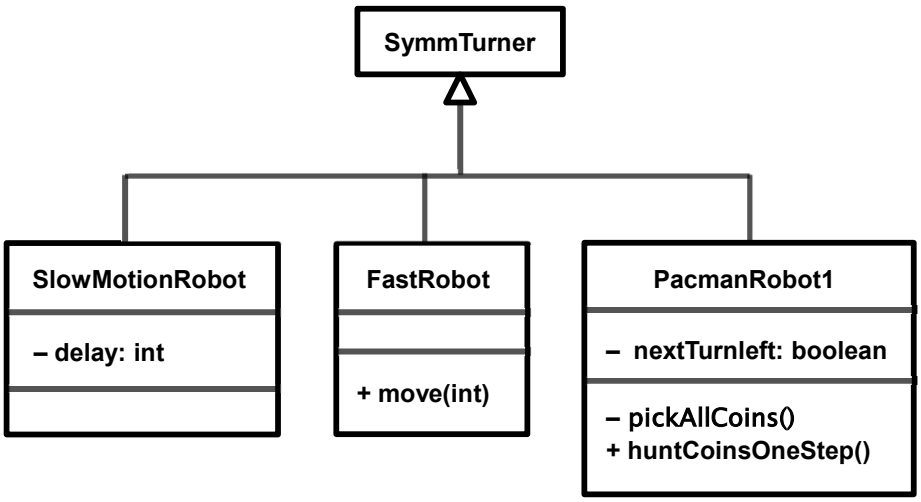
TECHNISCHE UNIVERSITÄT DARMSTADT



**Nach weiteren acht Schritten hat sich das Muster natürlich nicht verändert, nur verlängert.**

# Erster einfacher PacmanRobot

TECHNISCHE UNIVERSITÄT DARMSTADT




**So fügt sich die neue Klasse PacmanRobot1 in die in Kapitel 01f realisierte Hierarchie von Klassen ein.**

## Erster einfacher PacmanRobot

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class PacmanRobot1 extends SymmTurner {

    private boolean nextTurnLeft;

    public PacmanRobot1 ( int x, int y,
                                  Direction direction ) {
        super ( x, y, direction, 0 );
        nextTurnLeft = true;
    }

    ..........
}
```

**Die Klasse PacmanRobot1 für den ersten einfachen Pacman-Roboter ist also von Klasse SymmTurner aus Kapitel 01f abgeleitet. Daher hat PacmanRobot1 nicht nur von Robot eine Methode zum Linksdrehen geerbt, sondern auch von SymmTurner die Methode zum Rechtsdrehen, natürlich auch alle weiteren Methoden von Robot.**

## Erster einfacher PacmanRobot

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class PacmanRobot1 extends SymmTurner {

    private boolean nextTurnLeft;

    public PacmanRobot1 ( int x, int y,
                                  Direction direction ) {
        super ( x, y, direction, 0 );
        nextTurnLeft = true;
    }

    ..........
}
```

**In diesem Fall geben wir aber schon beim Konstruktor der Basisklasse vor, dass ein PacmanRobot1 zu Beginn noch keine Münzen haben soll. Daher gibt es keine Anzahl Münzen in der Parameterliste des Konstruktors von PacmanRobot1, und der entsprechende Wert beim Aufruf des Konstruktors der Basisklasse wird fest auf 0 gesetzt.**

## Erster einfacher PacmanRobot

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class PacmanRobot1 extends SymmTurner {

   private boolean nextTurnLeft;

   public PacmanRobot1 ( int x, int y,
                         Direction direction ) {
      super ( x, y, direction, 0 );
      nextTurnLeft = true;
   }

   ..........
}
```

**Wie schon analog beim Füllen eines Zimmers in Kapitel 01c, wird dieses boolesche Attribut dem Roboter bei jedem move sagen, ob er dann noch nach links oder nach rechts drehen soll. Das Attribut ist private, weil es nur eine interne Hilfsfunktion innerhalb der Klasse PacmanRobot1 hat. Die erste Drehung des Roboters im einführenden Beispiel war nach links; das korrespondiert damit, dass nextTurnLeft mit true initialisiert wird.**

# Erster einfacher PacmanRobot

TECHNISCHE UNIVERSITÄT DARMSTADT

```
private void pickAllCoins () {
    while ( isNextToACoin() )
        pickCoin();
}

public void huntCoinsOneStep() {
    move();
    pickAllCoins();
    !!! ausfuellen !!!
}
```

**Wir brauchen eine Methode, mit der ein Pacman-Roboter alle Münzen auf dem Feld aufheben kann, auf dem er gerade steht.**

## Erster einfacher PacmanRobot

TECHNISCHE UNIVERSITÄT DARMSTADT

```
private void pickAllCoins () {
    while ( isNextToACoin() )
        pickCoin();
}

public void huntCoinsOneStep() {
    move();
    pickAllCoins();
    !!! ausfuellen !!!
}
```

**Diese Methode ist private, weil sie nur eine interne Hilfe ist und nicht von außerhalb verwendet werden soll. Durch das private kann die Methode bekanntlich nur von anderen Methoden der Klasse PacmanRobot1 aufgerufen werden, aber nicht von außerhalb der Klasse PacmanRobot1.**

## Erster einfacher PacmanRobot

TECHNISCHE UNIVERSITÄT DARMSTADT

```
private void pickAllCoins () {
    while ( isNextToACoin() )
        pickCoin();
}

public void huntCoinsOneStep() {
    move();
    pickAllCoins();
    !!! ausfuellen !!!
}
```

**Die Methode isNextToACoin kennen wir schon vom Zimmerfüllen. Sie liefert einen booleschen Wert zurück, also true oder false, und kann daher so wie hier gezeigt in der Fortsetzungsbedingung der while-Schleife aufgerufen werden.**

## Erster einfacher PacmanRobot

TECHNISCHE UNIVERSITÄT DARMSTADT

```
private void pickAllCoins () {
    while ( isNextToACoin() )
        pickCoin();
}

public void huntCoinsOneStep() {
    move();
    pickAllCoins();
    !!! ausfuellen !!!
}
```

**Solange es Münzen auf dem Feld gibt, auf dem der Roboter gerade steht, soll also immer eine aufgehoben werden. Mit anderen Worten: Es werden alle Münzen auf diesem Feld aufgehoben, und sobald es keine Münzen auf diesem Feld mehr gibt, ist die while-Schleife und damit die ganze Methode zu Ende.**

## Erster einfacher PacmanRobot

TECHNISCHE UNIVERSITÄT DARMSTADT

```
private void pickAllCoins () {
    while ( isNextToACoin() )
        pickCoin();
}

public void huntCoinsOneStep() {
    move();
    pickAllCoins();
    !!! ausfuellen !!!
}
```

**Schlussendlich müssen wir die Methode definieren, mit der der Roboter einen Schritt auf seiner Jagd nach Münzen macht.**

## Erster einfacher PacmanRobot

TECHNISCHE UNIVERSITÄT DARMSTADT

```
private void pickAllCoins () {
    while ( isNextToACoin() )
        pickCoin();
}

public void huntCoinsOneStep() {
    move();
    pickAllCoins();
    !!! ausfuellen !!!
}
```

**Wie üblich wird der Vorwärtsschritt an die Methode move der Basisklasse delegiert. Da die Methode move von Robot weder in SymmTurner noch in PacmanRobot1 überschrieben wird, brauchen wir kein super davor, denn die Implementation aus der indirekten Basisklasse Robot wird ja über die direkte Basisklasse SymmTurner einfach an PacmanRobot1 vererbt.**

## Erster einfacher PacmanRobot

TECHNISCHE UNIVERSITÄT DARMSTADT

```
private void pickAllCoins () {
    while ( isNextToACoin() )
        pickCoin();
}

public void huntCoinsOneStep() {
    move();
    pickAllCoins();
    !!! ausfuellen !!!
}
```

**Und danach hebt der Roboter auf dem neuen Feld alle Münzen auf.**

## Erster einfacher PacmanRobot

TECHNISCHE UNIVERSITÄT DARMSTADT

```
private void pickAllCoins () {
    while ( isNextToACoin() )
        pickCoin();
}

public void huntCoinsOneStep() {
     move();
     pickAllCoins();
     !!! ausfuellen !!!
}
```

**Jetzt müssen wir noch implementieren, wie er danach seine Richtung selbstständig ändert.**

# Die if-Abfrage

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public void huntCoinsOneStep() {
    ..........
    if ( nextTurnLeft == true ) {
        turnLeft();
        nextTurnLeft = false;
    }
    else {
        turnRight();
        nextTurnLeft = true;
    }
}
```

**Wir müssen eine Fallunterscheidung machen, je nachdem, ob der Roboter sich nach links oder nach rechts drehen soll. Wenn nextTurnLeft gleich true ist, dreht er sich nach links, wie der Name schon sagt.**

# Die if-Abfrage

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public void huntCoinsOneStep() {
    ..........
    if ( nextTurnLeft == true ) {
       turnLeft();
       nextTurnLeft = false;
    }
    else {
       turnRight();
       nextTurnLeft = true;
    }
}
```

**Diese Linksdrehung passiert als erstes.**

# Die if-Abfrage

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public void huntCoinsOneStep() {
    ..........
    if ( nextTurnLeft == true ) {
        turnLeft();
        nextTurnLeft = false;
    }
    else {
        turnRight();
        nextTurnLeft = true;
    }
}
```

**Dann muss der Roboter sich im Attribut nextTurnLeft noch merken, dass er als nächstes nicht nach links, sondern nach rechts drehen soll.**

## Die if-Abfrage

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public void huntCoinsOneStep() {
    ..........
    if ( nextTurnLeft == true ) {
        turnLeft();
        nextTurnLeft = false;
    }
    else {
        turnRight();
        nextTurnLeft = true;
    }
}
```

**Für die Behandlung des Falles, dass die if-Abfrage false liefert, haben wir schon das Schlüsselwort else kennen gelernt.**

# Die if-Abfrage

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public void huntCoinsOneStep() {
    ..........
    if ( nextTurnLeft == true ) {
        turnLeft();
        nextTurnLeft = false;
    }
    else {
        turnRight();
        nextTurnLeft = true;
    }
}
```

**Es wird also genau einer dieser beiden Blöcke von Anweisungen ausgeführt: der obere, falls nextTurnLeft gleich true ist, der untere, falls nextTurnLeft gleich false ist.**

**Beachten Sie, dass die beiden Blöcke spiegelsymmetrisch zueinander dasselbe tun, nur dass links und rechts ausgetauscht sind.**

## Die if-Abfrage

TECHNISCHE UNIVERSITÄT DARMSTADT

```
if ( nextTurnLeft == true ) {
   turnLeft();
   nextTurnLeft = false;
 }
else {
   turnRight();
   nextTurnLeft = true;
}
```

```
if ( nextTurnLeft == false ) {
   turnRight();
   nextTurnLeft = true;
}
else {
   turnLeft();
   nextTurnLeft = false;
}
```

**Ein kleiner Exkurs: Es gibt verschiedene Möglichkeiten, wie diese if-Abfrage logisch äquivalent hätte formuliert werden können. Links ist noch einmal die Formulierung, die wir auf der letzten Folie gesehen hatten.**

## Die if-Abfrage

TECHNISCHE UNIVERSITÄT DARMSTADT

```
if ( nextTurnLeft == true ) {
   turnLeft();
   nextTurnLeft = false;
 }
else {
   turnRight();
   nextTurnLeft = true;
}
```

```
if ( nextTurnLeft == false ) {
   turnRight();
   nextTurnLeft = true;
}
else {
   turnLeft();
   nextTurnLeft = false;
}
```

**Statt auf nextTurnLeft gleich true hätte auch auf nextTurnLeft gleich false abgefragt werden können. Dann vertauschen sich die beiden Fälle, das heißt, die beiden Blöcke für die beiden Fälle müssen vertauscht werden.**

# Die if-Abfrage

TECHNISCHE UNIVERSITÄT DARMSTADT

```
if ( nextTurnLeft == true ) {
   turnLeft();
   nextTurnLeft = false;
 }
else {
   turnRight();
   nextTurnLeft = true;
}
```

```
if ( nextTurnLeft )
   turnLeft();
else
   turnRight();
nextTurnLeft
     = ! nextTurnLeft;
```

**Statt explizit auf nextTurnLeft gleich true zu testen, hätten wir natürlich auch einfach nur nextTurnLeft in die Bedingung hineinschreiben können. Das hatten wir früher auch schon gemacht.**

# Die if-Abfrage

TECHNISCHE UNIVERSITÄT DARMSTADT

```
if ( nextTurnLeft == true ) {
   turnLeft();
   nextTurnLeft = false;
 }
else {
   turnRight();
   nextTurnLeft = true;
}
```

```
if ( nextTurnLeft )
   turnLeft();
else
   turnRight();
nextTurnLeft
     = ! nextTurnLeft;
```

**Und noch etwas hätten wir unabhängig davon anders machen können: die Neusetzung von nextTurnLeft aus der if-Abfrage ganz herausziehen und mit dem schon gesehenen Ausrufezeichen als Umkehrung des Wertes in nextTurnLeft ohne Fallunterscheidung verwirklichen.**

# Die if-Abfrage

TECHNISCHE UNIVERSITÄT DARMSTADT

```
if ( nextTurnLeft == false ) {
   turnRight();
   nextTurnLeft = true;
 }
else {
   turnLeft();
   nextTurnLeft = false;
}
```

```
if ( ! nextTurnLeft )
   turnRight();
else
   turnLeft();
nextTurnLeft
    = ! nextTurnLeft;
```

**Oder auch so: Statt explizit auf nextTurnLeft gleich false zu testen, kann man auch hier wieder das Ausrufezeichen verwenden.**

# Array von PacmanRobot1

TECHNISCHE UNIVERSITÄT DARMSTADT

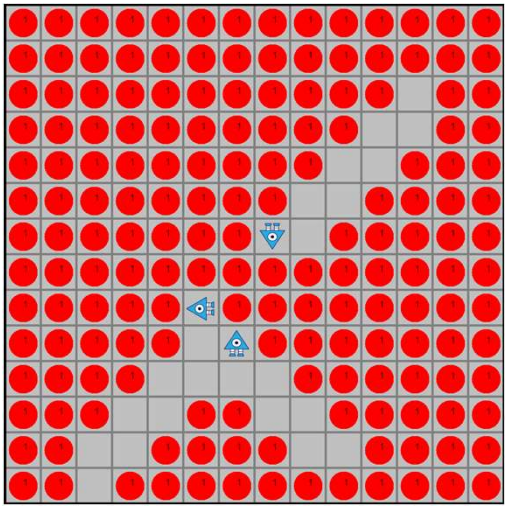
```
PacmanRobot1 paccy1 = new PacmanRobot1 ( 1, 0, RIGHT );
PacmanRobot1 paccy2 = new PacmanRobot1 ( 9, 0, UP );
PacmanRobot1 paccy3 = new PacmanRobot1 ( 12, 11, LEFT );

for ( ... 9 ... ) {
   paccy1.huntCoinsOneStep();
   paccy2.huntCoinsOneStep();
   paccy3.huntCoinsOneStep();
}
```

**Zurück zu Pacman-Robotern. Die Definition der Klasse PacmanRobot1 hatten wir schon fertiggestellt, jetzt lassen wir die Roboter laufen. Die Datei heißt PacmanRobot1Example.java.**

**Konkret lassen wir drei Roboter der Klasse PacmanRobot1 parallel jeweils neun Vorwärtsschritte tun.**

## Array von PacmanRobot1

TECHNISCHE UNIVERSITÄT DARMSTADT



**Und hier sehen wir das Ergebnis. Jeder der drei Pacman-Roboter hat seine Spur durch die FopBot-World gezogen, jeweils abhängig vom Startfeld und der Startrichtung.**

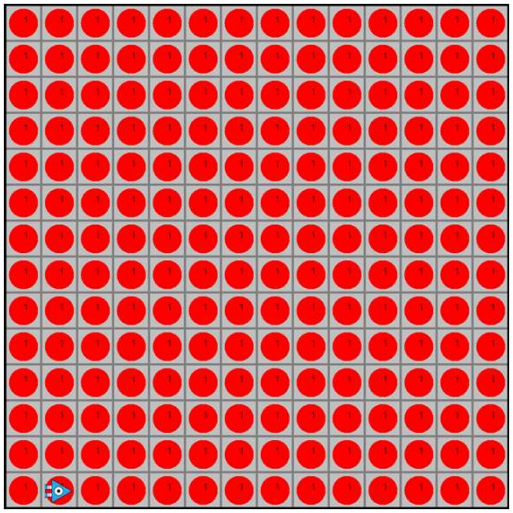
TECHNISCHE UNIVERSITÄT DARMSTADT

# Fünfte eigene Roboterklasse: zweiter einfacher Pacman-Roboter

**Wir wollen jetzt eine zweite Pacman-Roboterklasse definieren mit Namen PacmanRobot2, die nicht bei *jedem* Vorwärtssschritt die Richtung wechselt, sondern nur bei jedem *zweiten*.**

## Zweiter einfacher PacmanRobot

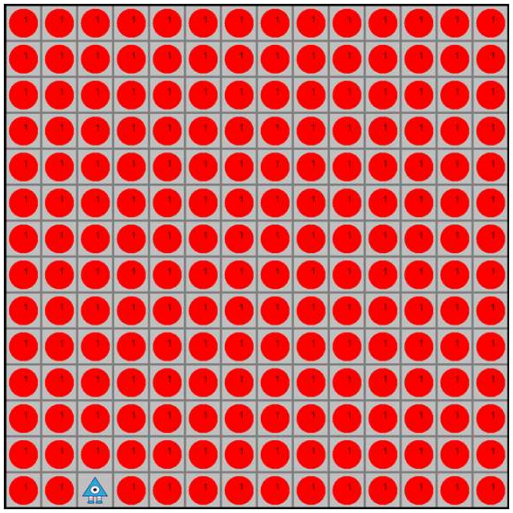
TECHNISCHE UNIVERSITÄT DARMSTADT



**Daher hinterlassen Roboter dieser Klasse solche eigentümlichen Spuren, wie wir sie gleich sehen. Der Roboter ist wieder auf derselben Feld platziert wie der Roboter im einführenden Beispiel für PacmanRobot1.**

# Zweiter einfacher PacmanRobot

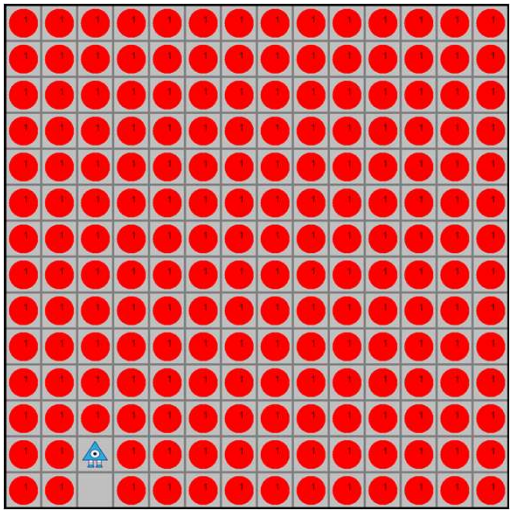
TECHNISCHE UNIVERSITÄT DARMSTADT



**Nach dem ersten Schritt unterscheiden sich PacmanRobot1 und PacmanRobot2 noch nicht.**

## Zweiter einfacher PacmanRobot

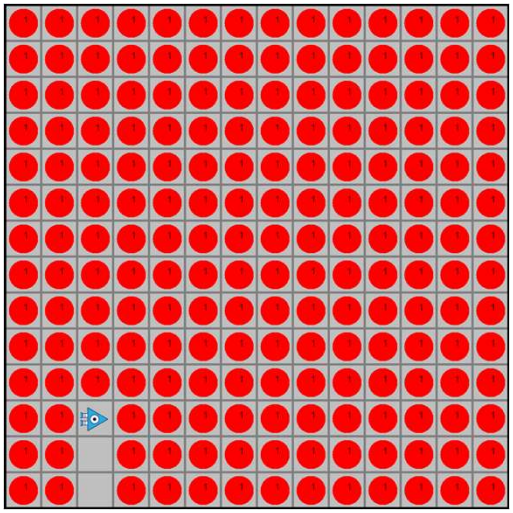
TECHNISCHE UNIVERSITÄT DARMSTADT



**Nach dem zweiten Schritt immer noch nicht, außer dass der Roboter seine Richtung *nicht* gewechselt hat.**

## Zweiter einfacher PacmanRobot

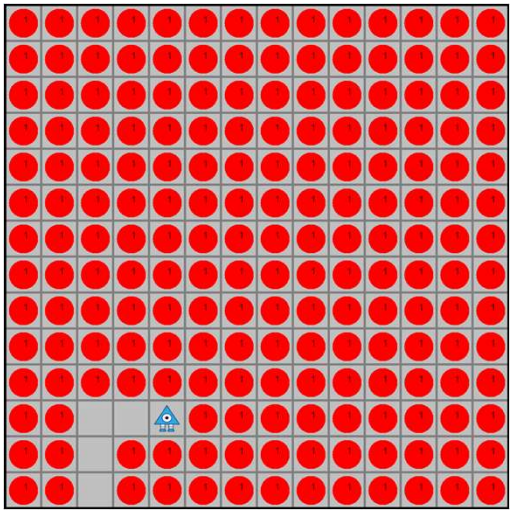
TECHNISCHE UNIVERSITÄT DARMSTADT



**Erst nach dem dritten Schritt sehen wir den Unterschied in Gänze: Dieser Roboter geht zwei Schritte in derselben Richtung vorwärts, bevor er seine Richtung ändert.**

## Zweiter einfacher PacmanRobot

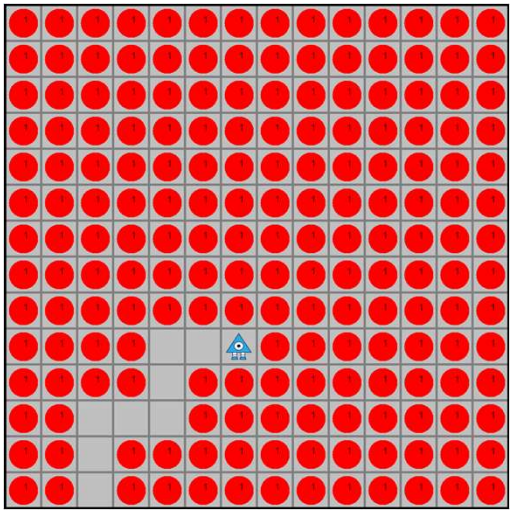
TECHNISCHE UNIVERSITÄT DARMSTADT



**Nach weiteren zwei Schritten sind weitere zwei Münzen in der neuen Richtung abgegrast, und erst danach wendet sich der Roboter wieder nach links.**

## Zweiter einfacher PacmanRobot

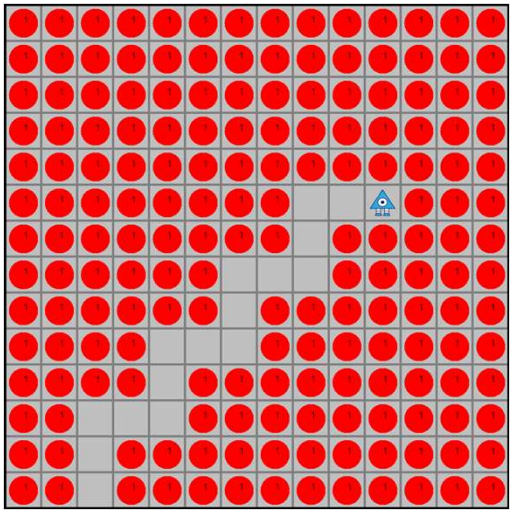
TECHNISCHE UNIVERSITÄT DARMSTADT



**Vier weitere Schritte, also zwei Richtungswechsel.**

## Zweiter einfacher PacmanRobot

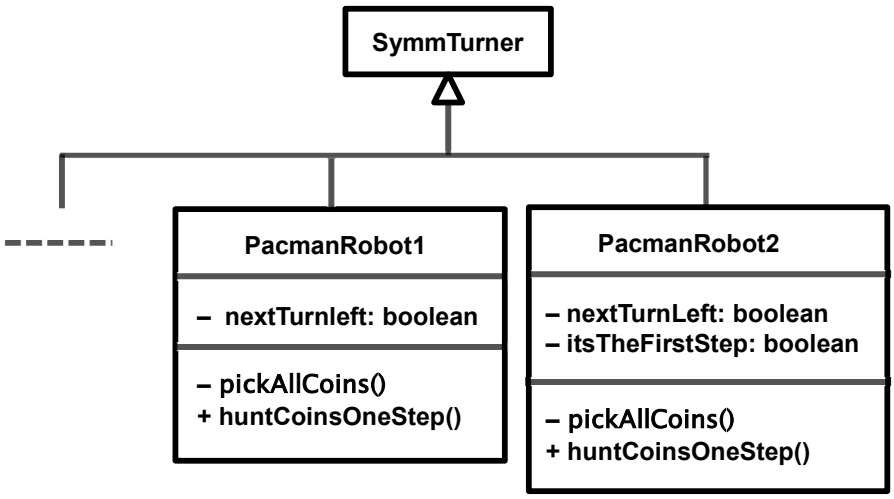
TECHNISCHE UNIVERSITÄT DARMSTADT



**Und so weiter.**

# Zweiter einfacher PacmanRobot

TECHNISCHE UNIVERSITÄT DARMSTADT




**Der Ausschnitt aus dem bisherigen UML-Klassendiagramm, um den es in diesem Kapitel geht. Ausgelassen sind die Klassen SlowMotionRobot und FastRobot aus Kapitel 01f, und SymmTurner ist nur in Kurzform dargestellt.**

# Zweiter einfacher PacmanRobot

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class PacmanRobot2 extends SymmTurner {

   private boolean nextTurnLeft;
   private boolean itsTheFirstStep;

   public PacmanRobot2 ( int x, int y,
                         Direction direction ) {
      super ( x, y, direction, 0 );
      nextTurnLeft = true;
      itsTheFirstStep = true;
   }

   .........
}
```

**Die Klasse PacmanRobot2 leiten wir also ebenfalls von SymmTurner ab.**

# Zweiter einfacher PacmanRobot


```
public class PacmanRobot2 extends SymmTurner {

  private boolean nextTurnLeft;
  private boolean itsTheFirstStep;

  public PacmanRobot2 ( int x, int y,
                        Direction direction ) {
    super ( x, y, direction, 0 );
    nextTurnLeft = true;
    itsTheFirstStep = true;
  }

  .........
}
```

**Bei PacmanRobot1 mussten nur zwei Fälle unterschieden werden, je nachdem, ob der Roboter sich als nächstes nach links oder nach rechts drehen soll. Hier müssen wir noch zusätzlich unterscheiden, ob der Roboter den ersten oder den zweiten Schritt in eine Richtung gemacht hat. Für Letzteres richten wir ein zweites boolean-Attribut ein und benennen auch dieses aussagekräftig.**

# Zweiter einfacher PacmanRobot


```
private void pickAllCoins () {
    while ( isNextToACoin() )
        pickCoin();
}
```

**Wie PacmanRobot1 hat auch PacmanRobot2 exakt dieselbe Methode, um alle Münzen auf dem momentanen Feld aufzunehmen. Wir haben sie bei PacmanRobot1 schon diskutiert, das brauchen wir hier nicht zu wiederholen.**

## Zweiter einfacher PacmanRobot

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public void huntCoinsOneStep () {
    !!! ausfuellen !!!
}
```

**In Klasse PacmanRobot2 brauchen wir ebenfalls eine Methode, die auf spezifische Weise vorwärtsgeht und Münzen aufnimmt. Da sie im Prinzip bis auf Details dasselbe tun soll wie Methode huntCoinsOneStep von Klasse PacmanRobot1, nennen wir sie auch gleich. In beiden Klassen ist die Methode huntCoinsOneStep public, void und parameterlos, also maximal ähnlich.**

# Zweiter einfacher PacmanRobot


```
public void huntCoinsOneStep () {
  move();
  pickAllCoins();
  !!! ausfuellen !!!
}
```

**Der Vorwärtsschritt und das Aufnehmen der Münzen auf dem neuen Feld sind auch völlig identisch zu PacmanRobot1.**

# Zweiter einfacher PacmanRobot

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public void huntCoinsOneStep () {
   move();
   pickAllCoins();
   !!! ausfuellen !!!
   itsTheFirstStep = ! itsTheFirstStep;
}
```

**Auf den ersten Schritt in eine Richtung folgt immer der zweite Schritt in dieselbe Richtung, und auf den zweiten Schritt in eine Richtung folgt immer der erste Schritt in die nächste Richtung. Daher ist es korrekt, in jedem Vorwärtsschritt die Information, ob es gerade der erste oder der zweite Schritt in eine Richtung ist, umzukehren.**

## Zweiter einfacher PacmanRobot

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public void huntCoinsOneStep () {
   move();
   pickAllCoins();
   if ( itsTheFirstStep )
      !!! ausfuellen !!!
   itsTheFirstStep = ! itsTheFirstStep;
}
```

**Aber nur in genau einem dieser beiden Fälle ist eine Drehung fällig.**

# Zweiter einfacher PacmanRobot

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public void huntCoinsOneStep () {
   move();
   pickAllCoins();
   if ( itsTheFirstStep ) {
     if ( nextTurnLeft )
        turnLeft();
     else
        turnRight();
     nextTurnLeft = ! nextTurnLeft;
   }
   itsTheFirstStep = ! itsTheFirstStep;
}
```

**Wieder je nachdem, ob nextTurnLeft momentan true oder false ist, geht die Drehung nach links beziehungsweise nach rechts.**

# Zweiter einfacher PacmanRobot

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public void huntCoinsOneStep () {
   move();
   pickAllCoins();
   if ( itsTheFirstStep ) {
     if ( nextTurnLeft )
        turnLeft();
     else
        turnRight();
     nextTurnLeft = ! nextTurnLeft;
   }
   itsTheFirstStep = ! itsTheFirstStep;
}
```

**Und nach jeder Drehung müssen wir wieder die Richtung für die nächste Drehung ändern.**

**Damit ist PacmanRobot2 fertig.**

TECHNISCHE UNIVERSITÄT DARMSTADT

# Sechste eigene Roboterklasse: Pacman-Roboter mit flexibel wählbarer Strategie

**Oracle Java Tutorials: Interfaces**
**(erste drei Abschnitte: Defining, Implementing, Using)**

**PacmanRobot1 und PacmanRobot2 haben sehr viele Gemeinsamkeiten und, bei Licht betrachtet, gibt es eigentlich nur einen einzigen Unterschied: die Strategie, wann gedreht werden soll und in welche Richtung dann gedreht werden soll. Alles andere ist bei beiden Klassen völlig identisch.**

**Wir werden jetzt sehen, wie man PacmanRobot1 und PacmanRobot2 zu einer einzigen Klasse PacmanRobot zusammenfassen kann, die alle Gemeinsamkeiten von beiden Klassen implementiert. Die Unterschiede werden dann geeignet ausgelagert. Das Konstrukt, in das die Unterschiede ausgelagert werden, heißt *Interface*.**

TECHNISCHE UNIVERSITÄT DARMSTADT

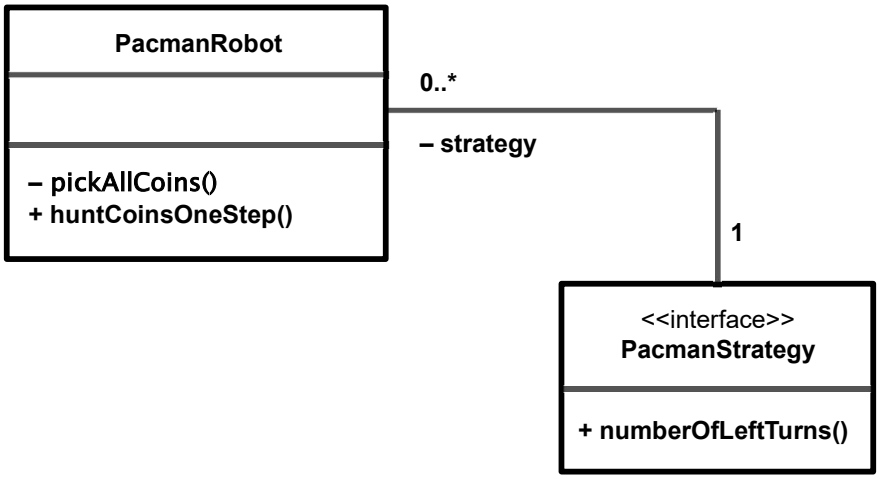
# Sechste eigene Roboterklasse: Pacman-Roboter mit flexibel wählbarer Strategie

**Oracle Java Tutorials: Interfaces**
**(erste drei Abschnitte: Defining, Implementing, Using)**

**In der Informatik spricht man in solchen Fällen von *Refactoring*: Das ursprüngliche Design mit PacmanRobot1 und PacmanRobot2 war ungut, weil die beiden Klassen sehr viel identischen Code gemeinsam hatten, was zur Folge hat, dass man bei jeder Änderung dieses Codes daran denken muss, den Code an beiden Stellen völlig identisch zu ändern. Das ist eine häufige Quelle von schwer zu findenden Fehlern. In diesem Abschnitt werden Sie das Ergebnis des Refactoring sehen: Der identische Code ist nur noch einmal vorhanden, Änderungen müssen nur noch an dieser einen Stelle durchgeführt werden, Fehler durch inkonsistente Änderungen sind damit ausgeschlossen.**

# Nichteingebaute Strategie

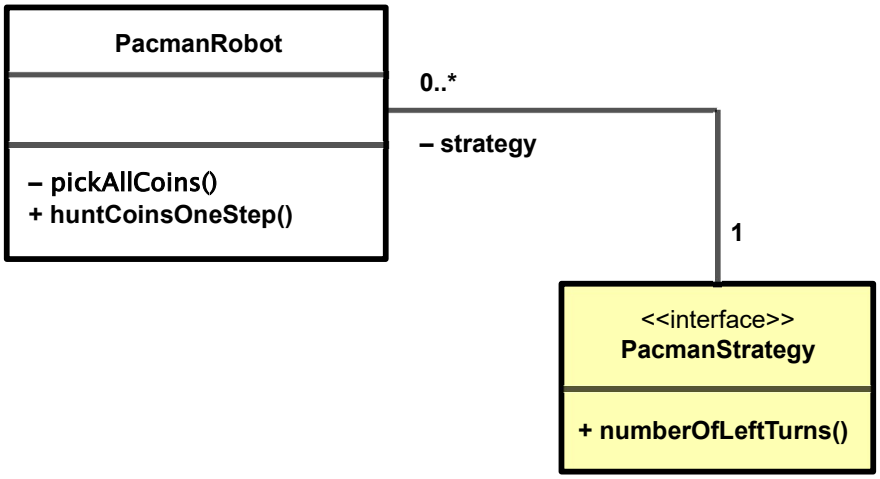
TECHNISCHE UNIVERSITÄT DARMSTADT




**Zuerst wieder das UML-Klassendiagramm.**

## Nichteingebaute Strategie

TECHNISCHE UNIVERSITÄT DARMSTADT



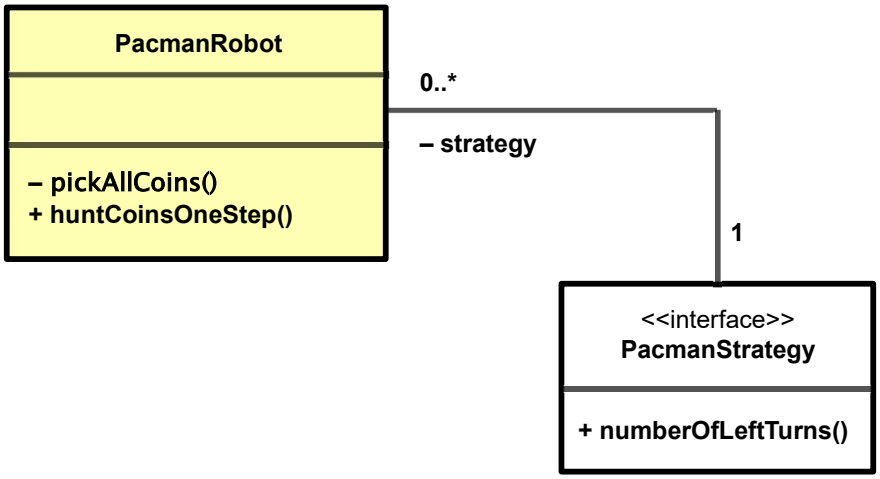

**Das Wort „interface“ in doppelten spitzen Klammern besagt, dass PacmanStrategy nicht eine richtige Klasse, sondern „nur“ ein Interface ist. Wir betrachten in diesem Kapitel nur die ursprüngliche, einfachste Form von Interfaces: Im Prinzip ist ein Interface in der einfachsten Form dasselbe wie eine Klasse, nur dass ein Interface erstens keine Attribute haben darf, sondern nur Methoden, zweitens alle Methoden public sind und drittens diese Methoden im Interface gar nicht implementiert, sondern nur *deklariert* werden. Was das genau heißt, werden wir gleich sehen.**

**In PacmanStrategy wird die Strategie für den Richtungswechsel in austauschbarer Form ausgelagert sein, so dass einmal die einfache Strategie von PacmanRobot1, ein anderes Mal die etwas kompliziertere Strategie von PacmanRobot2 dahinter stehen kann. Natürlich könnten wir dann auch beliebig viele weitere Strategien implementieren und mit PacmanRobot assoziieren.**

***Vorgriff*: In Kapitel 03b und 04c werden wir Interfaces systematisch betrachten.**

# Nichteingebaute Strategie

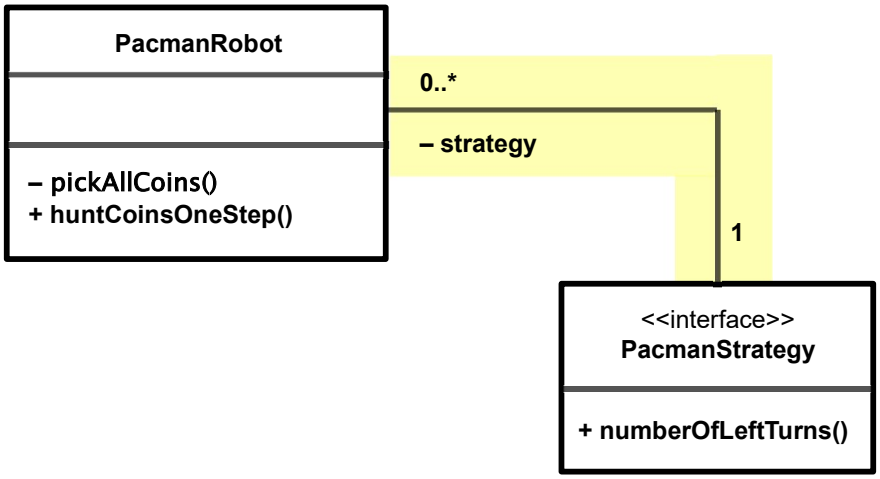
TECHNISCHE UNIVERSITÄT DARMSTADT




**Wie gesagt, wir haben jetzt nur eine einzelne Klasse PacmanRobot anstelle von PacmanRobot1 und PacmanRobot2. Die Gemeinsamkeiten von PacmanRobot1 und PacmanRobot2 finden sich genauso in PacmanRobot, die Unterschiede sind in PacmanStrategy ausgelagert.**

# Nichteingebaute Strategie

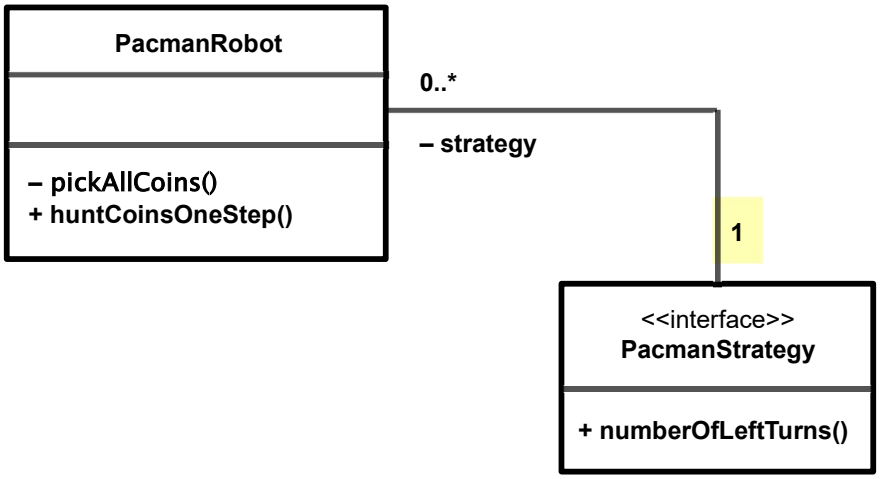
TECHNISCHE UNIVERSITÄT DARMSTADT




**Hier lernen wir eine neue Beziehung zwischen zwei Klassen (beziehungsweise so wie hier zwischen Klasse und Interface) kennen. Diese Beziehung heißt *Assoziation* und wird ohne Pfeil dargestellt. Andererseits kann eine Assoziation an beiden Enden *annotiert* werden, wie Sie es hier sehen. Die Annotation in dieser Graphik bedeutet:**

# Nichteingebaute Strategie

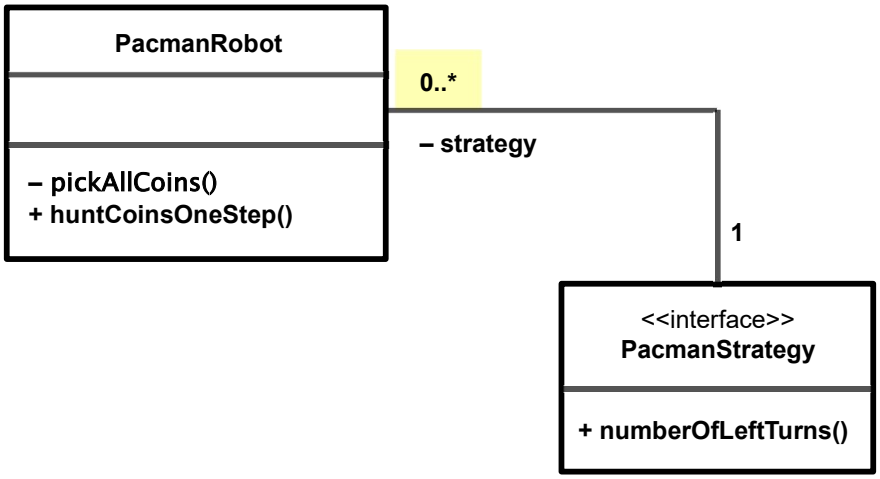
TECHNISCHE UNIVERSITÄT DARMSTADT




***Ein* Objekt von irgendeiner Pacman-Strategie ...**

# Nichteingebaute Strategie

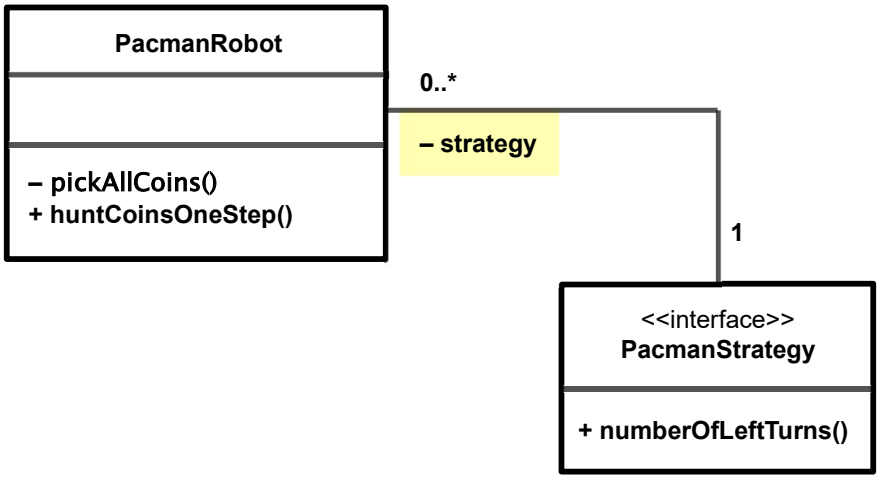
TECHNISCHE UNIVERSITÄT DARMSTADT




**... kann mit keinem oder beliebig vielen Pacman-Robotern assoziiert sein (wenn eine Pacman_Strategie mit mindestens einem Pacman-Robot assoziiert sein soll, dann müsste hier 1..* statt 0..* stehen).**

# Nichteingebaute Strategie

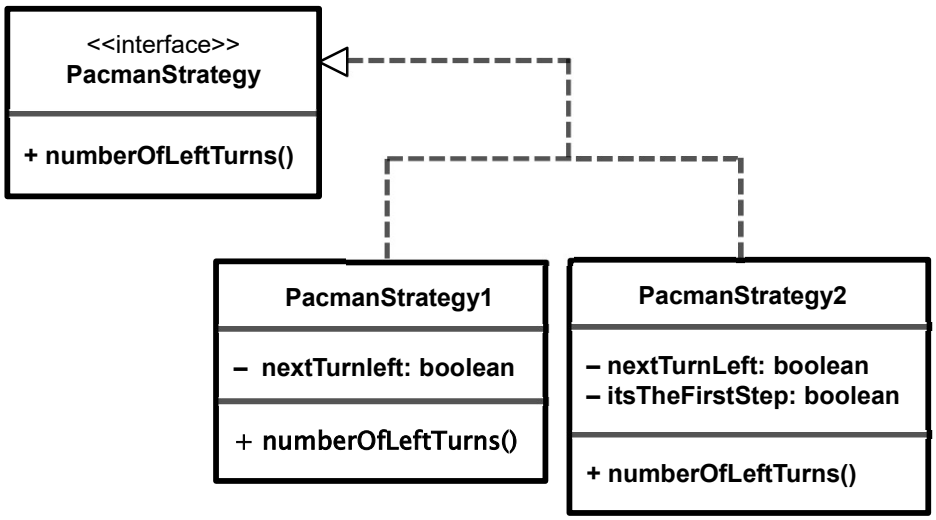
TECHNISCHE UNIVERSITÄT DARMSTADT




**In einem Pacman-Robot ist die Strategie ansprechbar unter dem Namen strategy. Wie bisher, zeigt auch hier das Minuszeichen an, dass dieser Verweis auf eine Strategie private ist. Diese Assoziation ist gewissermaßen einseitig, weil PacmanRobot einen Verweis auf „seine“ PacmanStrategy hat, aber umgekehrt nicht. Wir werden gleich sehen, dass das völlig ausreicht.**

# Nichteingebaute Strategie

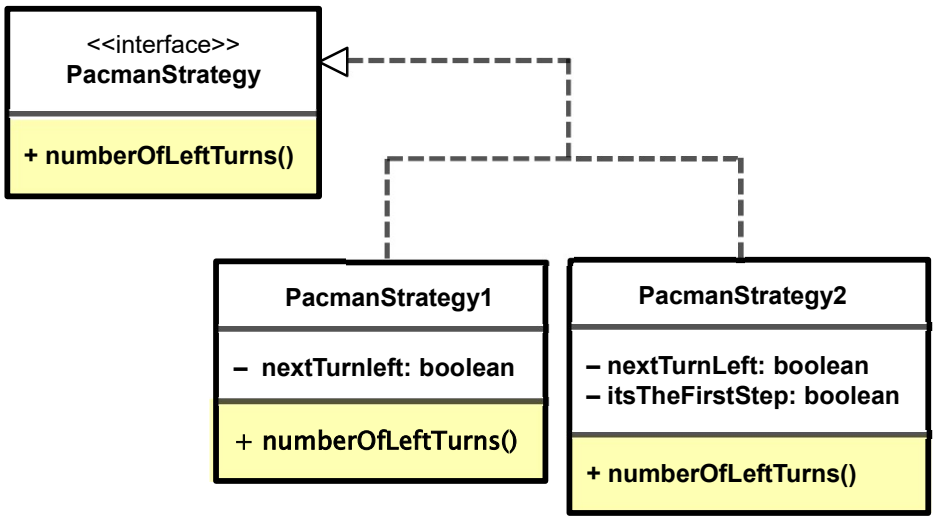
TECHNISCHE UNIVERSITÄT DARMSTADT




**Um die beiden Strategien aus PacmanRobot1 und PacmanRobot2 zu realisieren, definieren wir zwei Klassen PacmanStrategy1 und PacmanStrategy2. Diese Klassen sind im Prinzip von PacmanStrategy abgeleitet, so wie wir Klassen von anderen Klassen in Kapitel 01f abgeleitet haben. Aber bei einem Interface anstelle einer Basisklasse werden die Pfeile zur Unterscheidung gestrichelt gezeichnet, und die Sprechweise ist eine andere: Da PacmanStrategy ein Interface und keine Klasse ist, sagen wir nicht, dass PacmanStrategy1 und PacmanStrategy2 von PacmanStrategy *abgeleitet* sind, sondern wir sagen, dass PacmanStrategy1 und PacmanStrategy2 das Interface PacmanStrategy *implementieren*.**

# Nichteingebaute Strategie

TECHNISCHE UNIVERSITÄT DARMSTADT

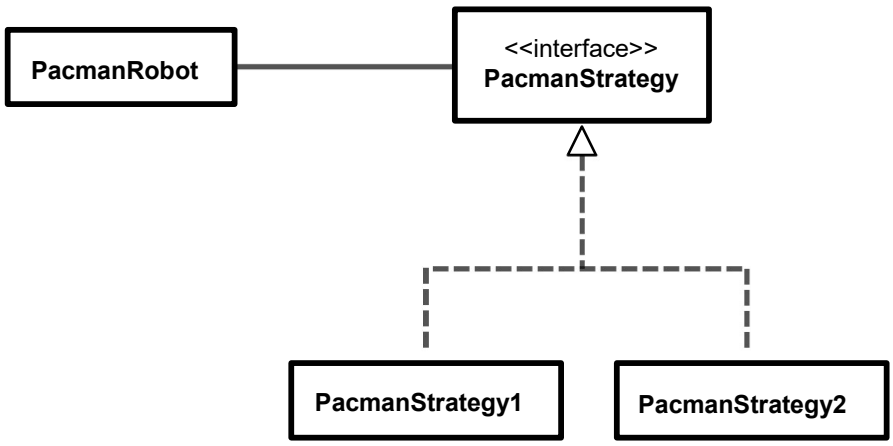



**Die Begriffsbildung, dass PacmanStrategy1 und PacmanStrategy2 das Interface PacmanStrategy „implementieren“, rührt wohl daher, dass die im Interface noch nicht implementierten Methoden in den beiden Klassen tatsächlich implementiert werden.**

***Vorgriff*: Im Abschnitt zu Interfaces in Kapitel 03c werden wir unter anderem sehen, dass die Methoden eines Interface auch nicht zwingend in seinen implementierenden Klassen implementiert werden müssen.**

## Nichteingebaute Strategie

TECHNISCHE UNIVERSITÄT DARMSTADT




**So sieht dann das gesamte Klassendiagramm unter Auslassung aller Details der einzelnen Klassen und des Interface aus.**

## Nichteingebaute Strategie

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class PacmanRobot extends SymmTurner {
    private PacmanStrategy strategy;

    public PacmanRobot ( int x, int y,
                         Direction direction,
                         PacmanStrategy theStrategy ) {
        super ( x, y, direction, 0 );
        strategy = theStrategy;
    }

    ..........
}
```

**Nun gehen wir an die Realisierung in Java. So wie PacmanRobot1 und PacmanRobot2, wird auch Klasse PacmanRobot wieder von SymmTurner abgeleitet.**

# Nichteingebaute Strategie

TECHNISCHE UNIVERSITÄT DARMSTADT

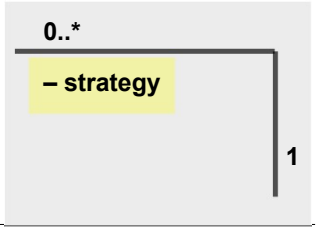
```
public class PacmanRobot extends SymmTurner {
    private PacmanStrategy strategy;

    public PacmanRobot ( int x, int y,
                         Direction direction,
                         PacmanStrategy theStrategy ) {
        super ( x, y, direction, 0 );
        strategy = theStrategy;
    }

    ..........
}
```

**Auch der Konstruktor mit Aufruf des Konstruktors der Basisklasse mit Zeilennummer, Spaltennummer, Richtung und null Münzen ist exakt derselbe wie bei PacmanRobot1 und PacmanRobot2.**

# Nichteingebaute Strategie

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class PacmanRobot extends SymmTurner {
    private PacmanStrategy strategy;

    public PacmanRobot ( int x, int y,
                         Direction direction,
                         PacmanStrategy theStrategy ) {
        super ( x, y, direction, 0 );
        strategy = theStrategy;
    }

    ..........
}
```




**Hier sehen Sie die Assoziation zwischen PacmanRobot und PacmanStrategy gemäß UML-Klassendiagramm von vor ein paar Folien, rechts unten noch einmal der relevante Ausschnitt daraus. Sie sehen auch schon, dass das in Java eigentlich gar nichts Neues ist: Die Assoziation ist nur in PacmanRobot realisiert, und zwar durch ein private-Attribut namens strategy vom Typ PacmanStrategy.**

**Die Annotationen an dem kleinen Ausschnitt rechts unten demonstrieren, warum Attribute, deren Typ eine Klasse oder ein Interface ist, anders in UML-Klassendiagrammen dargestellt werden als Attribute von primitiven Datentypen: Das Verhältnis zwischen zwei Klassen / Interfaces ist halt vielfältiger Natur, auch wenn es wie hier „nur“ durch ein Attribut in einer der beiden beteiligten Klassen realisiert ist; bei einem Attribut von einem primitiven Datentyp hat dieses Attribut in jedem Objekt der Klassen hingegen einfach nur jeweils einen bestimmten Wert des primitiven Datentyps, und das war‘s auch schon.**

# Nichteingebaute Strategie

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class PacmanRobot extends SymmTurner {

    ..........

    private void pickAllCoins () {
        while ( isNextToACoin() )
            pickCoin();
    }

    ..........
}
```

**Wir brauchen wieder Methode pickAllCoins, und zwar auch diese völlig identisch wie bei PacmanRobot1 und PacmanRobot2.**

# Nichteingebaute Strategie

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class PacmanRobot extends SymmTurner {

    ..........

    public void huntCoinsOneStep () {
        move();
        pickAllCoins();
        int numberOfLeftTurns = strategy.numberOfLeftTurns();
        for ( ... numberOfLeftTurns ... )
            turnLeft();
    }
}
```

**Methode huntCoinsOneStep ist jetzt relativ kurz.**

# Nichteingebaute Strategie

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class PacmanRobot extends SymmTurner {

    .........

    public void huntCoinsOneStep () {
        move();
        pickAllCoins();
        int numberOfLeftTurns = strategy.numberOfLeftTurns();
        for ( ... numberOfLeftTurns ... )
            turnLeft();
    }
}
```

**Ein Vorwärtsschritt und dann alle Münzen auf dem neuen Feld aufsammeln – auch dies völlig identisch zu PacmanRobot1 und PacmanRobot2.**

# Nichteingebaute Strategie

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class PacmanRobot extends SymmTurner {

    .........

    public void huntCoinsOneStep () {
        move();
        pickAllCoins();
        int numberOfLeftTurns = strategy.numberOfLeftTurns();
        for ( ... numberOfLeftTurns ... )
            turnLeft();
    }
}
```

**Für den Typ PacmanStrategy, den wir noch zu realisieren haben, legen wir hier fest, dass er eine Methode numberOfLeftTurns haben soll, die zurückliefert, wie viele Linksdrehungen der Roboter machen soll, um der Strategie zu folgen. Die for-Schleife unmittelbar darunter führt diese Anzahl Linksdrehungen aus.**

# PacmanRobot in Aktion

TECHNISCHE UNIVERSITÄT DARMSTADT

```
PacmanStrategy ps1 = new PacmanStrategy1 ();
PacmanStrategy ps2 = new PacmanStrategy2 ();

PacmanRobot pr1 = new PacmanRobot ( 1, 0, RIGHT, ps1 );
PacmanRobot pr2 = new PacmanRobot ( 3, 0, RIGHT, ps2 );

for ( ... 9 ... ) {
    pr1.huntCoinsOneStep();
    pr2.huntCoinsOneStep();
}
```

**Jetzt müssen wir noch PacmanStrategy und die beiden implementierenden Klassen für die beiden konkreten Strategien schreiben. Auf der letzten Folie haben wir schon festgelegt, dass eine Methode numberOfLeftTurns vorhanden sein soll. Um uns klarzumachen, was wir im Detail zu implementieren haben, schreiben wir erst eine beispielhafte Anwendung.**

# PacmanRobot in Aktion

TECHNISCHE UNIVERSITÄT DARMSTADT

```
PacmanStrategy ps1 = new PacmanStrategy1 ();
PacmanStrategy ps2 = new PacmanStrategy2 ();

PacmanRobot pr1 = new PacmanRobot ( 1, 0, RIGHT, ps1 );
PacmanRobot pr2 = new PacmanRobot ( 3, 0, RIGHT, ps2 );

for ( ... 9 ... ) {
    pr1.huntCoinsOneStep();
    pr2.huntCoinsOneStep();
}
```

**Die Klassen PacmanStrategy1 und PacmanStrategy2 sollen das Interface PacmanStrategy implementieren. Diese beiden Klassen implementieren wir gleich.**

# PacmanRobot in Aktion

TECHNISCHE UNIVERSITÄT DARMSTADT

```
PacmanStrategy ps1 = new PacmanStrategy1 ();
PacmanStrategy ps2 = new PacmanStrategy2 ();

PacmanRobot pr1 = new PacmanRobot ( 1, 0, RIGHT, ps1 );
PacmanRobot pr2 = new PacmanRobot ( 3, 0, RIGHT, ps2 );

for ( ... 9 ... ) {
    pr1.huntCoinsOneStep();
    pr2.huntCoinsOneStep();
}
```

**Wir legen hier erst einmal provisorisch fest, dass jede der beiden Klassen jeweils einen Konstruktor mit leerer Parameterliste haben soll. Es könnte sein, dass wir später im weiteren Verlauf der Implementierungsarbeiten feststellen, dass die Klasse PacmanStrategy1 oder die Klasse PacmanStrategy2 – oder beide – doch eher einen Konstruktor mit einem oder mehreren Parametern benötigen. Dann müssen wir an dieser Stelle eben später die benötigten Parameterwerte in das Klammerpaar einfügen.**

# PacmanRobot in Aktion

TECHNISCHE UNIVERSITÄT DARMSTADT

```
PacmanStrategy ps1 = new PacmanStrategy1 ();
PacmanStrategy ps2 = new PacmanStrategy2 ();

PacmanRobot pr1 = new PacmanRobot ( 1, 0, RIGHT, ps1 );
PacmanRobot pr2 = new PacmanRobot ( 3, 0, RIGHT, ps2 );

for ( ... 9 ... ) {
    pr1.huntCoinsOneStep();
    pr2.huntCoinsOneStep();
}
```

**Jetzt zwei Objekte der Klasse PacmanRobot, die wir soeben definiert haben.**

# PacmanRobot in Aktion

TECHNISCHE UNIVERSITÄT DARMSTADT

```
PacmanStrategy ps1 = new PacmanStrategy1 ();
PacmanStrategy ps2 = new PacmanStrategy2 ();

PacmanRobot pr1 = new PacmanRobot ( 1, 0, RIGHT, ps1 );
PacmanRobot pr2 = new PacmanRobot ( 3, 0, RIGHT, ps2 );

for ( ... 9 ... ) {
    pr1.huntCoinsOneStep();
    pr2.huntCoinsOneStep();
}
```

**Der letzte Parameter des Konstruktors von Klasse PacmanRobot hatte den Typ PacmanStrategy. Eine Klasse wie PacmanStrategy1 beziehungsweise PacmanStrategy2, die das Interface PacmanStrategy implementieren, kann dann anstelle von PacmanStrategy übergeben werden. Wenn dann in PacmanRobot die Methode numberOfLeftTurns aufgerufen wird, dann wird bei pr1 die Implementation von PacmanStrategy1 aufgerufen und bei pr2 die Implementation von PacmanStrategy2.**

# PacmanRobot in Aktion

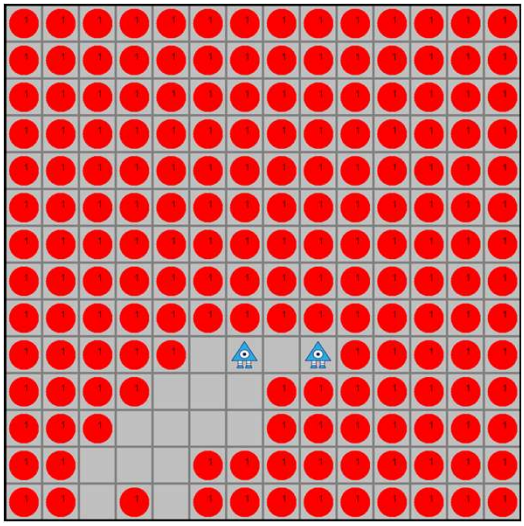
TECHNISCHE UNIVERSITÄT DARMSTADT

```
PacmanStrategy ps1 = new PacmanStrategy1 ();
PacmanStrategy ps2 = new PacmanStrategy2 ();

PacmanRobot pr1 = new PacmanRobot ( 1, 0, RIGHT, ps1 );
PacmanRobot pr2 = new PacmanRobot ( 3, 0, RIGHT, ps2 );

for ( ... 9 ... ) {
    pr1.huntCoinsOneStep();
    pr2.huntCoinsOneStep();
}
```

**So, und jetzt die beiden Roboter einfach loslaufen lassen!**

## PacmanRobot in Aktion

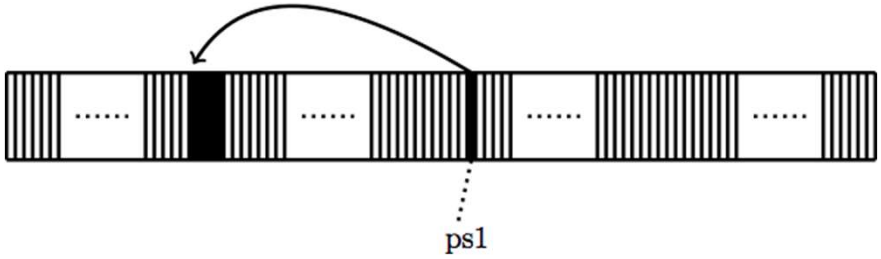
TECHNISCHE UNIVERSITÄT DARMSTADT



**Das soll herauskommen: Der Roboter pr1 durchläuft die Felder nach der ersten Strategie, der Roboter pr2 nach der zweiten Strategie.**

# PacmanRobot in Aktion

TECHNISCHE UNIVERSITÄT DARMSTADT

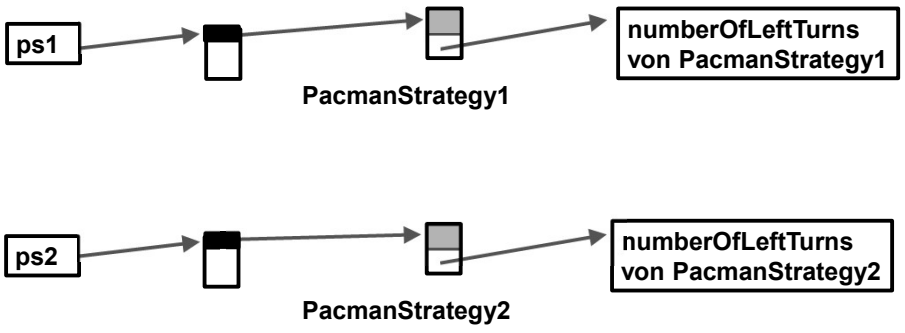



**PacmanStrategy ps1 = new PacmanStrategy1 ();**

**Das Bild, das ps1 und ps2 im Computerspeicher zeigen, ist exakt dasselbe wie bei Klassen.**

# PacmanRobot in Aktion

TECHNISCHE UNIVERSITÄT DARMSTADT




*Erinnerung*: In Kapitel 01f, Abschnitt „Allgemein: Methoden vererben / überschreiben“, haben wir schon solche Bilder gesehen zur Erläuterung, wie man sich das Ganze im Computerspeicher vorstellen kann.

Sie sehen, dass hier die Situation völlig analog ist: Die beiden Referenzen ps1 und ps2 verweisen jeweils auf ein Objekt der Klasse PacmanStrategy1 beziehungsweise PacmanStrategy2. Wie Objekte jeder anderen Klasse haben diese beiden Objekte jeweils einen anonymen Verweis auf die Daten zur jeweiligen Klasse. In den Klassen PacmanStrategy1 und PacmanStrategy2 ist jeweils nur eine einzige Methode implementiert, nämlich numberOfLeftTurns. Daher haben beide Methodentabellen jeweils nur ein Feld. Wird nun numberOfleftTurns mit ps1 aufgerufen, dann wird die Implementation in PacmanStrategy1 angesteuert, bei ps2 ist es eben die in PacmanStrategy2.

# Interface PacmanStrategy

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public interface PacmanStrategy {
    public int numberOfLeftTurns ();
}

public class PacmanStrategy1 implements PacmanStrategy {
    !!! ausfuellen !!!
}

public class PacmanStrategy2 implements PacmanStrategy {
    !!! ausfuellen !!!
}
```

**Jetzt implementieren wir diesen ominösen Typ PacmanStrategy und alles, was dahinter steckt.**

## Interface PacmanStrategy

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public interface PacmanStrategy {
    public int numberOfLeftTurns ();
}

public class PacmanStrategy1 implements PacmanStrategy {
    !!! ausfuellen !!!
}

public class PacmanStrategy2 implements PacmanStrategy {
    !!! ausfuellen !!!
}
```

**PacmanStrategy ist etwas, das wir bisher nicht kennen, ein Interface. Das sieht auf den ersten Blick aus wie eine Klasse, ist aber weit weniger als eine Klasse.**

**Wir halten hier erst einmal fest: Anstelle des Schlüsselwortes class bei Klassen oder enum bei Enumerationen steht bei Interfaces das Schlüsselwort interface.**

## Interface PacmanStrategy

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public interface PacmanStrategy {
    public int numberOfLeftTurns ();
}

public class PacmanStrategy1 implements PacmanStrategy {
    !!! ausfuellen !!!
}

public class PacmanStrategy2 implements PacmanStrategy {
    !!! ausfuellen !!!
}
```

**Hier ist die Methode numberOfLeftTurns, die wir schon von PacmanStrategy in Klasse PacmanRobot benutzt haben.**

**Aber in einem Interface werden Methoden nicht *implementiert*, sondern nur ohne Implementation *definiert*. Das heißt, es wird hier nur gesagt, dass diese Methode in Klassen, die das Interface PacmanStrategy implementieren, vorhanden sein soll, und welche Parameter und welchen Rückgabetyp diese Methode dann haben soll. Daher gibt es keinen Block von Anweisungen, sondern an dessen Stelle nur ein abschließendes Semikolon.**

**Damit sind übrigens das Interface PacmanStrategy und seine Methode numberOfLeftTurns vollständig definiert. Alles Weitere dann bei den Klassen, die das Interface implementieren und dadurch überall da verwendet werden können, wo PacmanStrategy erwartet wird.**

**Natürlich kann ein Interface auch mehrere Methoden haben.**

# Interface PacmanStrategy

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public interface PacmanStrategy {
    int numberOfLeftTurns ();
}

public class PacmanStrategy1 implements PacmanStrategy {
    !!! ausfuellen !!!
}

public class PacmanStrategy2 implements PacmanStrategy {
    !!! ausfuellen !!!
}
```

**Alle Methoden, die in einem Interface definiert werden, sind public; private und so weiter ist nicht erlaubt. Es ist eben nur ein Interface, zu deutsch Schnittstelle, also nur Methoden mit Außenwirkung, also public. Da sowieso alle Methoden public sein müssen, darf das public bei Interfaces vorneweg auch ausgelassen werden (bei Klassen natürlich nicht). Es ist allgemein üblich, das public bei Methoden von Interfaces auszulassen, und wir werden das ebenfalls von jetzt an so machen.**

## Interface PacmanStrategy

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public interface PacmanStrategy {
    int numberOfLeftTurns ();
}

public class PacmanStrategy1 implements PacmanStrategy {
    !!! ausfuellen !!!
}

public class PacmanStrategy2 implements PacmanStrategy {
    !!! ausfuellen !!!
}
```

**An dieser Stelle steht bei diesen beiden Klassen jeweils, dass das Interface PacmanStrategy von ihnen implementiert wird. Das Schema ist: Schlüsselwort implements gefolgt vom Namen des implementierten Interface. Wir werden gleich sehen, was das zu bedeuten hat.**

## PacmanStrategy1

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class PacmanStrategy1 implements PacmanStrategy {

   private boolean nextTurnLeft;

   public PacmanStrategy1 () {
      nextTurnLeft = true;
   }

   public int numberOfLeftTurns () {
      int result = nextTurnLeft ? 1 : 3;
      nextTurnLeft = ! nextTurnLeft;
      return result;
   }
}
```

**Das ist die *vollständige* Implementation der Klasse PacmanStrategy1, die unsere erste Strategie umsetzt, also dass die Richtung nach jedem Vorwärtsschritt gewechselt wird.**

**Wir haben uns das alles nicht nochmals neu ausgedacht, sondern uns einfach schamlos bei PacmanRobot1 bedient. Eben Refactoring: Der Code wurde aus PacmanRobot1 in PacmanStrategy1 überführt.**

## PacmanStrategy1

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class PacmanStrategy1 implements PacmanStrategy {
  private boolean nextTurnLeft;
  public PacmanStrategy1 () {
      nextTurnLeft = true;
  }
  public int numberOfLeftTurns () {
      int result = nextTurnLeft ? 1 : 3;
      nextTurnLeft = ! nextTurnLeft;
      return result;
  }
}
```

**Die altbekannte boolean-Variable nextTurnLeft aus Klasse PacmanRobot1 kommt in Klasse PacmanRobot nicht vor, sondern ist in die PacmanStrategy1 ausgelagert worden, weil sie ja nur für die Strategie zum Richtungswechsel gebraucht wird, und das ist nach dem Refactoring nicht mehr die Aufgabe vom PacmanRobot1 – die Klasse PacmanRobot1 existiert ja auch nicht mehr –, sondern von PacmanStrategy1. Die Variable nextTurnLeft wird im Konstruktor exakt so initialisiert wie in PacmanRobot1, die erste Drehung geht also auch hier nach links.**

## PacmanStrategy1

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class PacmanStrategy1 implements PacmanStrategy {
   private boolean nextTurnLeft;
   public PacmanStrategy1 () {
      nextTurnLeft = true;
   }
   public int numberOfLeftTurns () {
      int result = nextTurnLeft ? 1 : 3;
      nextTurnLeft = ! nextTurnLeft;
      return result;
   }
}
```

**Im Interface PacmanStrategy war diese Methode nur *definiert* worden. Hier müssen wir sie nun *implementieren*, um unsere Strategie zum Richtungswechsel zu realisieren. Name, Parameterliste und Rückgabetyp müssen exakt wie im Interface PacmanStrategy sein, und die Methode muss auf jeden Fall public wie in PacmanStrategy sein.**

## PacmanStrategy1

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class PacmanStrategy1 implements PacmanStrategy {

   private boolean nextTurnLeft;

   public PacmanStrategy1 () {
      nextTurnLeft = true;
   }

   public int numberOfLeftTurns () {
      int result = nextTurnLeft ? 1 : 3;
      nextTurnLeft = ! nextTurnLeft;
      return result;
   }
}
```

***Erinnerung*: Den Bedingungsoperator hatten wir schon in Kapitel 01b im gleichnamigen Abschnitt gesehen.**

**Der Bedingungsoperator ist bekanntlich ternär, hat also drei Operanden. Die ersten beiden Operanden werden durch ein Fragezeichen, die letzten beiden Operanden durch einen Doppelpunkt voneinander getrennt. Falls nextTurnLeft den Wert true hat, wird der Wert 1 in result gespeichert, sonst der Wert 3.**

# PacmanStrategy2

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class PacmanStrategy2 implements PacmanStrategy {

   private boolean nextTurnLeft;
   private boolean itsTheFirstStep;

   public PacmanStrategy2 () {
       nextTurnLeft = true;
       itsTheFirstStep = true;
   }

   public int numberOfLeftTurns () {

      !!! ausfuellen !!!

   }
}
```

**Jetzt unsere zweite Strategie, die nicht bei *jedem* Vorwärtsschritt einen Richtungswechsel vorschreibt, sondern nur bei jedem *zweiten*.**

# PacmanStrategy2

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class PacmanStrategy2 implements PacmanStrategy {
   private boolean nextTurnLeft;
   private boolean itsTheFirstStep;
   public PacmanStrategy2 () {
      nextTurnLeft = true;
      itsTheFirstStep = true;
   }
   public int numberOfLeftTurns () {
      !!! ausfuellen !!!
   }
}
```

**Attribut nextTurnLeft wie gehabt.**

# PacmanStrategy2

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class PacmanStrategy2 implements PacmanStrategy {
   private boolean nextTurnLeft;
   private boolean itsTheFirstStep;
   public PacmanStrategy2 () {
      nextTurnLeft = true;
      itsTheFirstStep = true;
   }
   public int numberOfLeftTurns () {
      !!! ausfuellen !!!
   }
}
```

**Für die zweite Strategie mussten wir uns ja schon in Klasse PacmanRobot2 zusätzlich merken, ob der nächste Schritt der erste oder schon der zweite in derselben Richtung ist, bevor es wieder an einen Richtungswechsel geht.**

# PacmanStrategy2

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class PacmanStrategy2 implements PacmanStrategy {

    private boolean nextTurnLeft;
    private boolean itsTheFirstStep;

    public PacmanStrategy2 () {
        nextTurnLeft = true;
        itsTheFirstStep = true;
    }

    public int numberOfLeftTurns () {
        !!! ausfuellen !!!
    }
}
```

**Jetzt müssen wir uns noch die Methode numberOfLeftTurns ansehen.**

## PacmanStrategy2

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class PacmanStrategy2 implements PacmanStrategy {
   .........
   public int numberOfLeftTurns () {
      !!! ausfuellen !!!
   }
}
```

**Wir bedienen uns dazu wieder schamlos, diesmal natürlich bei PacmanRobot2.**

# PacmanStrategy2

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class PacmanStrategy2 implements PacmanStrategy {
   .........
   public int numberOfLeftTurns () {
      !!! ausfuellen !!!
      itsTheFirstStep = ! itsTheFirstStep;
      !!! ausfuellen !!!
   }
}
```

**Wie bei PacmanRobot2 muss in jedem Schritt die Information, ob der nächste Schritt der erste oder der zweite in dieselbe Richtung ist, umgedreht werden.**

# PacmanStrategy2

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class PacmanStrategy2 implements PacmanStrategy {
   .........
   public int numberOfLeftTurns () {
      !!! ausfuellen !!!
      if ( itsTheFirstStep ) {
         !!! ausfuellen !!!
      }
      itsTheFirstStep = ! itsTheFirstStep;
      !!! ausfuellen !!!
   }
}
```

**Für die Frage, ob eine Drehung ansteht, ist wieder entscheidend, ob jetzt der erste oder der zweite Schritt in dieselbe Richtung kommt.**

# PacmanStrategy2

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class PacmanStrategy2 implements PacmanStrategy {
   .....
   public int numberOfLeftTurns () {
      !!! ausfuellen !!!
      if ( itsTheFirstStep ) {
         result = nextTurnLeft ? 1 : 3:
         !!! ausfuellen !!!
      }
      itsTheFirstStep = ! itsTheFirstStep;
      !!! ausfuellen !!!
   }
}
```

**Noch einmal der Bedingungsoperator wie bei PacmanStrategy1.**

# PacmanStrategy2

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class PacmanStrategy2 implements PacmanStrategy {
   .....
   public int numberOfLeftTurns () {
      !!! ausfuellen !!!
      if ( itsTheFirstStep ) {
         result = nextTurnLeft ? 1 : 3:
         nextTurnLeft = ! nextTurnLeft;
      }
      itsTheFirstStep = ! itsTheFirstStep;
      !!! ausfuellen !!!
   }
}
```

**Und nicht vergessen: nach jeder Drehung die Richtung der nächsten Drehung ändern.**

# PacmanStrategy2

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class PacmanStrategy2 implements PacmanStrategy {
   .....
   public int numberOfLeftTurns () {
     int result = 0;
     if ( itsTheFirstStep ) {
        result = nextTurnLeft ? 1 : 3:
        nextTurnLeft = ! nextTurnLeft;
     }
     itsTheFirstStep = ! itsTheFirstStep;
     return result;
   }
}
```

**Das zurückgelieferte Ergebnis ist 0, wenn itsTheFirstStep zu Beginn von numberOfLeftTurns false war, sonst 1 oder 3, je nachdem, ob nextTurnLeft true oder false war. Das entspricht exakt unserer zweiten Strategie: abwechselnd Links- und Rechtsdrehung in jedem zweiten Schritt und keine Drehung in den Schritten dazwischen.**