Übungen zu Algorithmen und Datenstrukturen für Medizintechnik
Sommersemester 2021
René Raab, Robert Richer

Lehrstuhl für Maschinelles Lernen und Datenanalytik
Department Artificial Intelligence in Biomedical Engineering (AIBE)
(https://www.mad.tf.fau.de)

Aufgabenblatt 5 vom 23. Mai 2021, Abgabe am 13. Juni 2021, 22:00 Uhr

---

Aufgabe 5.1: Theorie

Punkte siehe StudOn
OOP

Aufgabenstellung und Abgabe (individuell, nicht als Gruppe!) im StudOn.

Aufgabe 5.2: Pacman

65 Punkte
Objektorientierte Programmierung

In dieser Aufgabe werden Sie den Spieleklassiker *Pacman* programmieren. Der Spieler kontrolliert in diesem Spiel eine gelbe Spielfigur, welche von Monstern verfolgt wird. Ziel ist es, möglichst viele, auf dem Spielfeld verteilte, Punkte zu fressen, ohne dabei mit einem Monster zusammenzustoßen. Durch das Einsammeln von Lebenspunkten kann der Spieler öfters von den Monstern gefressen werden, ohne das Spiel zu verlieren.

Hinweis: Entnehmen Sie alle benötigten Informationen zu Attributsnamen, Sichtbarkeiten, Schnittstellen, Klassenbeziehungen, etc. dem UML-Klassendiagramm in Abbildung 1!

Hinweis: Wenn nicht anders angegeben, sollen alle Sichtbarkeiten für das Programm so restriktiv wie möglich gewählt werden.

1. Legen Sie ein neues Projekt `05-Pacman` an.

2. Wir stellen Ihnen für diese Aufgabe eine minimale Basisklasse `AudGameWindow` zur Verfügung, die einige grundlegende Funktionen wie das Anzeigen des Spielfensters zur Verfügung stellt. Da Sie aus technischen Gründen nicht direkt mit Klassen aus `java.awt` interagieren dürfen, stellen wir Ihnen für `Color` und `Graphics` noch die Klassen `AudColor` und `AudGraphics` zur Verfügung, die alle wichtigen Funktionen der AWT-Klassen bereitstellen. Bitte benutzen Sie ausschließlich diese bereitgestellten Klassen und greifen Sie **niemals** direkt auf Klassen aus `java.awt` zu. Importieren Sie diese auch nicht!

   Die bereitgestellen Klassen sind in der Datei `05-material.zip` enthalten, die Sie auf unserer StudOn-Seite herunterladen, entpacken und die Java-Dateien dann in Ihr Projekt einbinden können.

   Achtung: Diese Klassen dürfen nicht verändert werden und werden auch nicht mit abgegeben.

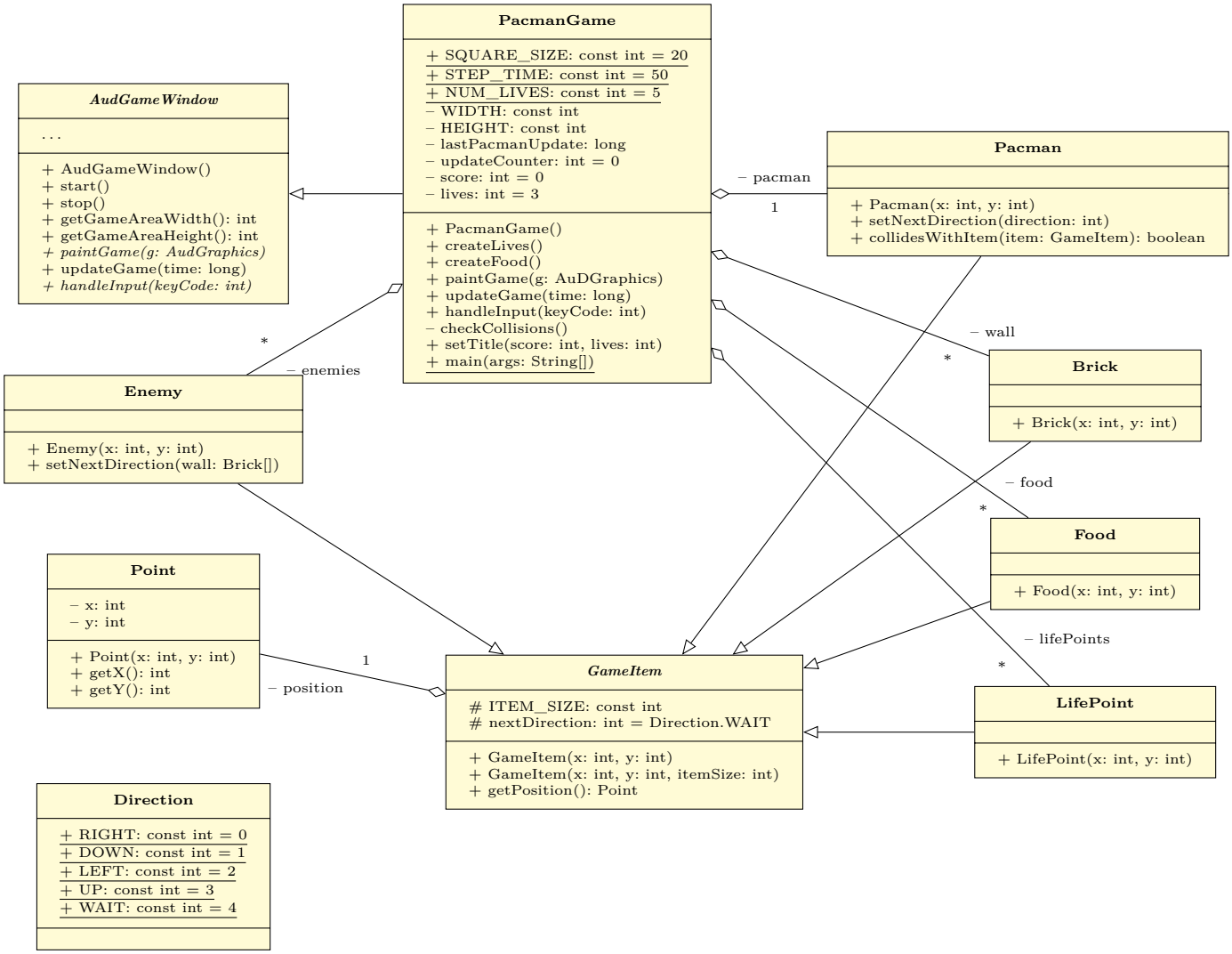

Abbildung 1: UML-Klassendiagramm des `Pacman`-Projektes

Graphics und Color
Informieren Sie sich in den Tafelübungs-Folien sowie in der Java-Dokumentation im Internet über die Klassen `java.awt.Graphics` und `java.awt.Color`!
Aufgrund der technischen Einschränkungen des EST dürfen Sie nicht direkt auf diese Klassen zugreifen sondern nur auf die bereitgestellten Klassen `AudGraphics` und `AudColor`, die alle wichtigen Methoden auch bereitstellen und genauso verwendet werden können.

3. **PacmanGame**

   In dieser Klasse sollen die Hauptfunktionalitäten des Spiels implementiert werden.

   a) Leiten Sie von der Klasse `AudGameWindow` eine neue Klasse `PacmanGame` ab. Lassen Sie alle geerbten abstrakten Methoden überschreiben. Diese neuen Methoden können Sie für den Moment noch leer lassen, wir werden sie später mit Inhalt füllen.

   b) Legen Sie in der Klasse eine `main`-Methode an. Erstellen Sie darin eine neue Instanz von `PacmanGame` und rufen Sie die (geerbte) Methode `start()` dieses Objektes auf.

   Wenn Sie das Programm ausführen, sollten Sie jetzt schon einmal ein leeres Fenster sehen.

   c) In der Methode `void paintGame(Graphics g)` wird das Spiel gezeichnet. Zeichnen Sie dort zunächst ein ausgefülltes Rechteck als Hintergrund. Die dafür benötigte Größe des Spielbereiches können Sie mithilfe der Methoden `getGameAreaWidth()` und `getGameAreaHeight()` abfragen.

   d) Legen Sie in `PacmanGame` einen Standard-Konstruktor an.

   e) Mit der Methode `setTitle(String)` können Sie den Titel des Spielfensters ändern. Überladen Sie die Methode, sodass sie die Übergabeparameter `int score` (der aktuelle Punktestand) und `int lives` (die Anzahl der verbleibenden Leben) besitzt. Die Methode soll daraus einen informativen Titel für das Spielfenster erzeugen (siehe Beispiel) und an die Methode `setTitle(String)` übergeben.

   `AuD-Pacman - Score: ⟨score⟩, Remaining Lives: ⟨lives⟩`

   f) Der Einfachheit halber werden alle Spielobjekte in ein quadratisches Gitter eingepasst. Der Pacman bewegt sich also beispielsweise in einem »Schritt« um ein ganzes Quadrat weiter. Im Folgenden werden wir dieses Gitter als »Gitter-Koordinatensystem« bezeichnen.

   Legen Sie in `PacmanGame` eine Konstante `SQUARE_SIZE` an, die die Kantenlänge eines Gitter-Quadrates in Pixeln speichern soll. Alle anderen Klassen des Projektes sollen auch ohne eine Instanz von `PacmanGame` auf diese Konstante zugreifen können. Weisen Sie ihr den Wert 20 zu!

   Benutzen Sie anschließend `SQUARE_SIZE` und die Methoden `getGameAreaWidth()` und `getGameAreaHeight()`, um im Konstruktor die Abmessungen des Gitter-Koordinatensystemes, also die Anzahl der Gitter-Quadrate in x- und y-Richtung zu bestimmen. Legen Sie die beiden Konstanten `WIDTH` und `HEIGHT` an, um diese Werte zu speichern.

4. **Point**

   Bevor wir mit der Implementierung der eigentlichen Spiel-Klassen beginnen, benötigen wir noch eine Hilfsklasse `Point`, die eine Position (x, y) auf dem Spielfeld speichert. Auf die beiden ganzzahligen Koordinaten soll von außen **nur** mit Hilfe von öffentlichen *getter*-Methoden zugegriffen werden können. Stellen Sie außerdem einen Konstruktor zur Verfügung, dem Anfangswerte für `x` und `y` übergeben werden können.

   Wichtig! Die Werte von x und y sind keine Pixel-Koordinaten, sondern beziehen sich auf die Koordinaten des *Spielgitters*! Pixel-Koordinaten werden **ausschließlich** verwendet, wenn es um das Zeichnen der Spiel-Oberfläche geht.

5. **GameItem**

   Für alle Spiel-Objekte lohnt es sich, eine gemeinsame, abstrakte Basisklasse `GameItem` anzulegen.

a) Legen Sie das Attribut `final int ITEM_SIZE` an, das **nur** für abgeleitete Klassen sichtbar sein soll. Diese Konstante wird die im Konstruktor übergebene Größe des Objektes (in Pixeln!) speichern. Legen Sie außerdem das private Attribut `Point position` an, welches die im Konstruktor übergebenen Koordinaten (x, y) speichern wird.

b) Im Konstruktor sollen der Klasse die Parameter `int x`, `int y` und `int itemSize` übergeben werden. `x` und `y` sind die Koordinaten des `GameItem`s und sollen im `Point`-Objekt `position` gespeichert werden. Von außen soll der Zugriff auf dieses Attribut mittels einer *getter*-Methode ermöglicht werden. Der Parameter `itemSize` soll in der Konstante `ITEM_SIZE` gespeichert werden.

c) Legen Sie einen zweiten Konstruktur an, der nur die beiden Parameter `int x` und `int y` besitzen und den Konstruktor mit drei Parametern aufrufen soll. Für `itemSize` soll die Standard-Gittergröße (also `SQUARE_SIZE` der Klasse `PacmanGame`) übergeben werden.

d) Alle von `GameItem` abgeleiteten Klassen müssen eine Methode `void paint(Graphics g)` zur Verfügung stellen. Legen Sie eine entsprechende abstrakte Methode an!

6. **Pacman**

Die Klasse `Pacman` ist für die Darstellung der Spielfigur verantwortlich.

a) Lassen Sie die Klasse `Pacman` von `GameItem` erben und lassen Sie alle abstrakten Methoden überschreiben.

b) Erstellen Sie einen Konstruktur `Pacman(int x, int y)` und rufen Sie darin den Konstruktor der Oberklasse mit den selben Parametern auf.

c) Bevor wir den Pacman zum Leben erwecken, wollen wir ihn erst einmal zeichnen. Setzen Sie dafür in der `paint(Graphics)`-Methode die Zeichenfarbe auf *gelb* und zeichnen Sie einen Kreis in das entsprechende Gitterquadrat. Verwenden Sie dabei für die Parameter `width` und `height` die in der Oberklasse definierte Konstante `ITEM_SIZE` und für die Parameter `x` und `y` die Konstante `SQUARE_SIZE`, mit der Sie auch die Gitter- in Pixel-Koordinaten umrechnen können!

d) Gehen Sie zurück zur Klasse `PacmanGame` und legen Sie ein privates Attribut für ein `Pacman`-Objekt namens `pacman` an. Ergänzen Sie den Konstruktor von `PacmanGame` so, dass dort ein neuer Pacman in der *Mitte* des Spiel-Fensters erstellt wird. Verwenden Sie dabei zwingend die Gitter-Koordinaten!

e) Erweitern Sie in `PacmanGame` die Methode `paintGame`, sodass der Pacman gezeichnet wird!

f) Wenn Sie das Programm nun ausführen, sollte der Pacman als gelber Kreis zu sehen sein.

7. **Bewegung**

Kommen wir nun zur Bewegung von Spielfiguren:

a) Zunächst benötigen wir eine Möglichkeit, die Bewegungsrichtung zu speichern. Da wir die Bewegungsrichtungen später auch noch für andere Spielelemente benötigen, wollen wir diese in einer eigenen Klasse `Direction` speichern. Legen Sie diese Klasse an, sowie die folgenden öffentlichen und statischen Konstanten: `RIGHT`, `DOWN`, `LEFT`, `UP` und `WAIT`. Weisen Sie ihnen aufsteigende, bei 0 beginnende `int`-Werte zu[1].

b) Nicht nur der Pacman soll sich bewegen können, sondern auch die Feinde (die wir später implementieren werden). Da alle Spielelemente von der Klasse `GameItem` erben werden, soll auch die beim nächsten Spielupdate verwendete Bewegungsrichtung in dieser Oberklasse gespeichert werden. Legen Sie daher ein Attribut `protected int nextDirection` an und

[1] Java bietet für diesen Zweck auch sogenannte *Enumerations*, die wir in dieser Aufgabe allerdings nicht verwenden.

initialisieren Sie es mit einem geeigneten Wert. Stellen Sie zudem eine *getter*-Methode zur Verfügung, die die Bewegungsrichtung zurückgibt. Die *setter*-Methoden werden wir jeweils in den abgeleiteten Klassen implementieren – natürlich nur bei den Spielelementen, die sich auch tatsächlich bewegen werden.

c) Legen Sie also eine *setter*-Methode `setNextDirection(int)` in der Klasse `Pacman` an, der ein Richtungswert übergeben wird. Falls dieser Wert keiner der fünf `Direction`-Konstanten entspricht, soll das Programm mit einer Fehlermeldung beendet werden.

d) Der Pacman und seine Feinde werden sich nicht tatsächlich kontinuierlich fortbewegen, sondern wir werden seine Position mehrmals pro Sekunde aktualisieren. Legen Sie eine neue öffentliche Methode `void step()` in `GameItem` an, die dafür zuständig sein soll, einen solchen Schritt auszuführen.

e) Die Figuren soll sich in alle vier Richtungen bewegen und stehen bleiben können. Dabei soll die Position – je nachdem welcher Wert in `nextDirection` steht – verändert werden. Bestimmen Sie dazu die neue Position der Spielfigur in Abhängigkeit der alten Position.

f) Im Hauptprogramm, der Klasse `PacmanGame`, muss `step()` jetzt natürlich noch aufgerufen werden. Die Häufigkeit soll über eine Konstante `STEP_TIME` eingestellt werden können, die die Zeit zwischen zwei Schritten in Millisekunden angibt. Ein guter Anfangswert könnten beispielsweise $50\,ms$ (also 20 Updates pro Sekunde) sein.

g) Die eigentlichen Veränderungen können nun in der Methode `updateGame(long)` der Klasse `PacmanGame` durchgeführt werden, die von der bereitgestellten Oberklasse automatisch aufgerufen wird — allerdings in unregelmäßigen Abständen. Es könnte also sein, dass seit dem letzten Aufruf kein Schritt, ein Schritt oder (bei besonders kurzen `STEP_TIME`s oder besonders hoher Systemlast) sogar mehrere Schritte ausgeführt werden müssen.

h) Um herauszufinden, wie viele Schritte fällig sind, speichern Sie den (geplanten) Zeitpunkt des letzten Updates in einem privaten `long`-Attribut `lastPacmanUpdate`. Im Konstruktor können Sie die Methode `System.currentTimeMillis()` (die die aktuelle Systemzeit in Millisekunden liefert) verwenden, um dafür einen Anfangswert zu setzen. Der Methode `updateGame(long)` wird als Parameter ebenfalls ein solcher Timestamp `time` übergeben, sodass Sie aus `time` und `lastPacmanUpdate` die verstrichene Zeit seit dem letzten Update berechnen können. Verwenden Sie `STEP_TIME`, um herauszufinden, wie oft der Pacman einen Schritt machen muss (⇝ Tipp: Schleife!). Rufen Sie entsprechend die `step`-Methode auf und erhöhen Sie nach jedem Update `lastPacmanUpdate` um `STEP_TIME`.

i) Wir wollen, dass sich der Pacman und seine Feinde jeweils abwechselnd bewegen, ihre jeweiligen `step()`-Methoden also in unterschiedlichen `updateGame`-Zyklen aufgerufen werden. Dies realisieren wir mit einer Zählvariablen `long updateCounter`, die bei jedem Aufruf der `updateGame(long)`-Methode um 1 inkrementiert werden soll. Stellen Sie dann sicher, dass sich der Pacman nur bei jedem zweiten `updateGame(long)`-Aufruf bewegt – im jeweils anderen Methodenaufruf werden sich dann die Feinde bewegen.

Hinweis: Wenn Sie das Programm jetzt starten, sollte sich der Pacman je nach gewählter Richtungseinstellung bewegen!

j) Damit der Spieler die Bewegung des Pacman beeinflussen kann, soll das Programm jetzt noch auf das Drücken der Pfeiltasten auf Ihrer Tastatur reagieren. Dies kann in der Methode `handleInput(int)` geschehen, die als Parameter den Tasten-Code einer gedrückten Taste übergeben bekommt. Die verschiedenen Tasten-Codes sind als öffentliche, statische Integer-Konstanten in der Klasse `java.awt.event.KeyEvent` definiert. Da Sie diese Klasse wieder nicht verwenden dürfen, haben wir Ihnen die wichtigsten Tasten-Codes in einer inneren Klasse von `AudGameWindow` mitgegeben, auf die sie mit `KeyEvent.VK_RIGHT`, `KeyEvent.VK_DOWN`,

`KeyEvent.VK_LEFT` und `KeyEvent.VK_UP` zugreifen können. Nutzen Sie diese Codes, um die gewünschte nächste Richtung des Pacman zu setzen. Verwenden Sie unbedingt die in `Direction` definierten Richtungskonstanten!

Testen Sie das Ergebnis!

8. **Brick**

Mit Objekten dieser Klasse sollen Mauern definiert werden, um das Spielfeld zu einem Labyrinth zu verwandeln und zu verhindern, dass der Pacman das Spielfeld verlassen kann.

a) Erstellen Sie die Klasse `Brick`, die von `GameItem` erben soll. Lassen Sie alle abstrakten Methoden überschreiben.

b) Legen Sie einen Konstruktor für `Brick` mit zwei Parametern für die x- und y-Koordinaten im Gitter-Koordinatensystem an. Denken Sie aber daran, dass Konstruktoren nicht vererbt werden und Sie auch den Konstruktor der Eltern-Klasse manuell aufrufen müssen!

c) Ein `Brick` soll immer genau ein Gitter-Quadrat ausfüllen und entsprechend als graues Quadrat gezeichnet werden. Setzen Sie diese Vorgaben in der `paint(Graphics)`-Methode entsprechend um.

d) Legen Sie nun in `PacmanGame` die zum Spiel benötigten Mauern an. Da diese sehr aufwändig zu erstellen sind, haben wir das für Sie übernommen! Laden Sie sich die Datei `GameEngine.java` aus StudOn herunter. Dort finden Sie die Methode `public static Brick[] generateWalls(int, int)`, die als Übergabeparameter die Breite und Höhe des Spielfeldes in Gitter-Koordinaten benötigt und daraus ein Array aus `Bricks` generiert. Legen Sie daraufhin in der Klasse `PacmanGame` ein entsprechendes Attribut `wall` an, das diese Mauern speichern soll und rufen Sie die `generateWalls(int, int)`-Methode im Konstruktor von `PacmanGame` auf. Zeichnen Sie anschließend alle Elemente von `wall` in der `paintGame(Graphics)`-Methode!

e) Ihr Spielfeld sollte nun aussehen wie in Abbildung 2.

9. **Kollisionen**

Wenn Sie nun spielen wollen, werden Sie feststellen, dass Sie noch alle Wände durchqueren können. Um dies zukünftig zu verhindern, müssen wir eine Kollisionsbehandlung für Mauern erstellen.

a) Erstellen Sie in der Klasse `GameItem` die beiden öffentlichen Methoden `boolean collidesWithWall (int direction, GameItem item)` und `boolean collidesWithWall(int direction, int x, int y)`. Ihr wird die Bewegungsrichtung, in der sich die Spielfigur als nächstes bewegen würde, übergeben, sowie entweder eine Referenz auf eine andere Spielfigur oder die x- und y-Koordinaten einer anderen Spielfigur.

b) Falls beim Schritt der sich bewegenden Spielfigur (also des Pacmans oder einer seiner späteren Feinde) erkannt wird, dass im Feld, das die Figur als *nächstes* betreten würde, eine Mauer steht, sollen die Methoden `true` zurückgeben. Falls kein Hindernis im Weg ist, soll `false` zurückgegeben werden.

Hinweis: Die Methode `collidesWithWall` wurde *überladen*, Sie müssen die Kollisionsbehandlung also nur in einer der beiden Methoden implementieren und diese aus der anderen heraus aufrufen. Überlegen Sie sich, welche Variante am sinnvollsten ist, also welche Methode die Kollisionsbehandlung implementieren und welche Methode dann die andere aufrufen soll!

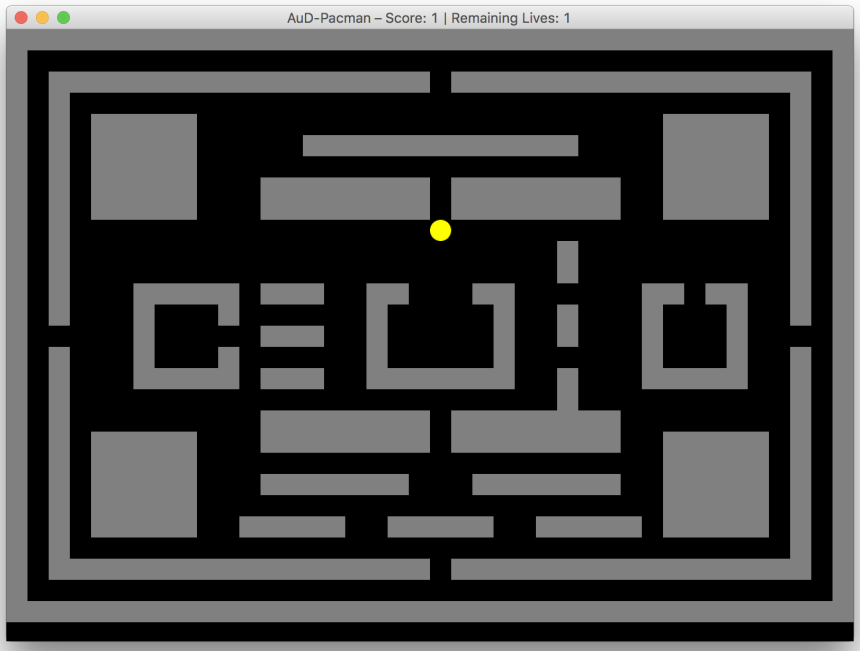

Abbildung 2: Spielfenster mit Pacman und Wänden

c) Erstellen Sie nun in `PacmanGame` eine Hilfsmethode `checkCollisions()`, die Sie nach jedem Schritt aus `updateGame(long)` aufrufen.

d) Überprüfen Sie darin für jeden Brick, ob eine Kollision vorliegt. Sollte dies der Fall sein, soll der Pacman vor der Mauer stehen bleiben.

e) Wenn Sie nun das Programm ausführen, werden Sie erkennen, dass der Pacman immer anhält, bevor er auf eine Mauer treffen würde. Jedoch ist bei nochmaligem Drücken der entsprechenden Pfeiltaste eine Durchquerung der Mauern möglich. Überlegen Sie sich eine Möglichkeit, dies zu verhindern!

Hinweis: Sie müssen dazu vermeiden, dass bei wiederholtem Drücken in Mauerrichtung das Attribut `nextDirection` wieder auf die Bewegungsrichtung gesetzt wird. Gegebenenfalls müssen Sie die Bewegungsrichtung des Pacmans in `handleInput` zurücksetzen!

f) Nun sollte eine Durchquerung der Wände in keinem Fall mehr möglich sein!

10. **Food**

Das Spielprinzip von Pacman ist es, möglichst viele Punkte zu fressen, ohne dabei selbst von den Feinden gefressen zu werden. Daher beschäftigen wir uns nun mit der Erstellung der Punkte auf dem Spielfeld.

a) Erstellen sie dazu eine neue Klasse `Food`, die von `GameItem` erbt. `Food` soll einen Konstruktor mit zwei Übergabeparametern besitzen. Der Durchmesser eines `Food`-Objektes soll halb so groß wie die Seitenlänge des Gitterquadrats sein und zentral im Quadrat seinen Mittelpunkt haben. Legen Sie eine Farbe Ihrer Wahl fest, in der `Food`-Objekte gezeichnet werden sollen.

b) Erstellen sie nun in der Hauptklasse `PacmanGame` die Punkte in einem 2D-Array `Food[][] food` und sorgen Sie dafür, dass diese auch angezeigt werden. Achten Sie dabei darauf, dass nur an freien Plätzen im Spielfeld Punkte angezeigt werden sollen, alle anderen Einträge des Arrays sollen `null` sein.

Hinweis: Eine Hilfsmethode zur Erstellung der Punkte (⇝ `void createFood()`) könnte hilfreich sein.

c) Wenn Sie ihr Programm nun ausführen, sollte auf jedem freien Platz im Spielfeld ein `Food`-Punkt angezeigt werden. Wenn Sie darüber hinweg laufen, wird ihnen auffallen, dass weder die Punkte verschwinden noch sich der Spielstand erhöht.

d) Fügen Sie der Methode `checkCollisions()` aus der Klasse `PacmanGame` eine Kollisionsbehandlung für die `Food`-Punkte hinzu. Legen Sie dazu die Methode `collidesWithItem(GameItem item)` in der Klasse `Pacman` an, die bei einer Kollision des Pacmans mit einem anderen `GameItem` `true`, ansonsten `false` zurückgibt.

Achtung: Im Gegensatz zu `collidesWithWall(...)`, bei der das *nächste* zu betretene Feld auf Kollisionen untersucht wird, soll die Methode `collidesWithItem(...)` das *aktuell* betretene Feld auf eine Kollision untersuchen!

Achten Sie darauf, dass nach der Kollision mit einem Punkt dieser verschwindet und die Punktzahl erhöht wird. Legen Sie dafür ein privates Attribut `score` an. Wenn Sie schon dabei sind, können Sie auch noch zusätzlich ein privates Attribut `lives` anlegen – dieses werden wir gleich benötigen.

e) Aktualisieren Sie nach jedem Aufruf von `ckeckCollisions()` den Titel des Spielfensters mittels `setTitle(int, int)`.

f) Sind alle Punkte gefressen (bei *unserem* Spielfeld 605, diese Anzahl kann jedoch variieren!), dann soll das Programm stoppen und eine Meldung ausgegeben werden, die die erreichte Punktzahl und Glückwünsche an den Spieler beinhaltet. Nutzen Sie dazu die Methode `showDialog(String text)`, die sie von `AudGameWindow` erben. Beenden Sie anschließend die Methode.

g) Ihr Programm sollte nun aussehen wie in Abbildung 3.

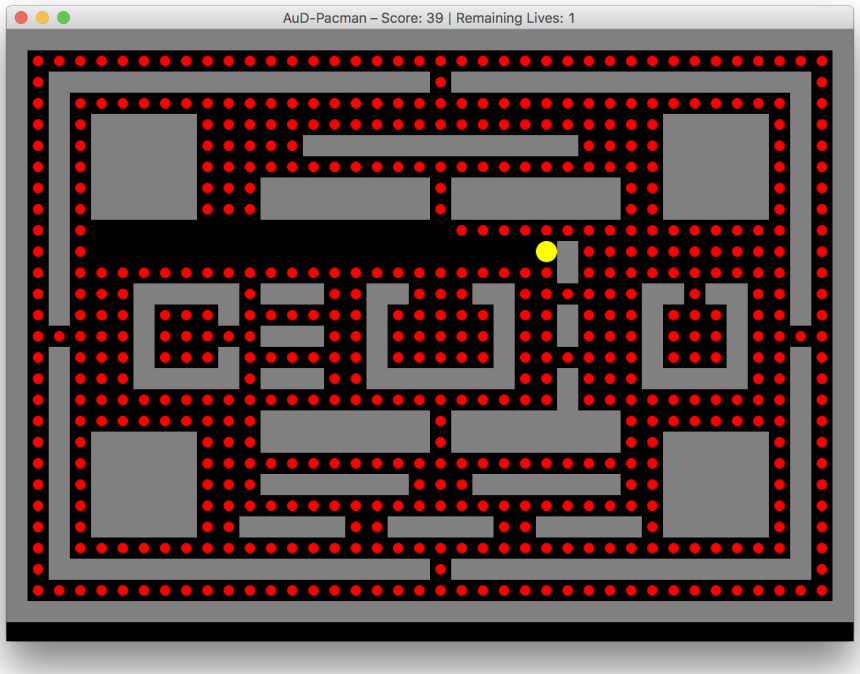

Abbildung 3: Spielfeld mit verteilten Punkten

Wichtig: Testen Sie an dieser Stelle ausführlich!

11. **LifePoint**

Beschäftigen wir uns nun mit der Erstellung zusätzlicher Lebenspunkte:

a) Erstellen die dazu eine neue Klasse `LifePoint`, die von `GameItem` erbt. Sie soll – ebenso wie `Food` – einen Konstruktor mit zwei Übergabeparametern besitzen und einen Konstuktor der Oberklasse aufrufen. Die Größe eines `LifePoint`-Objektes soll 16 Pixel betragen und zentral im Quadrat seinen Mittelpunkt haben.

b) Jeder `LifePoint` soll als farbiger Kreis dargestellt werden. Legen Sie dazu eine beliebige Farbe fest.

c) Erstellen Sie nun im Hauptprogramm `PacmanGame` fünf `LifePoints`, die Sie in einem Array abspeichern.

d) Die 5 `LifePoints` sollen dabei *zufällig* im Spielfeld verteilt werden, jedoch auf **keinen Fall** auf den selben Platz wie eine Mauer gesetzt werden. Lagern Sie die Erstellung der Lebenspunkte in eine Hilfsmethode `createLives()` aus, welche zuerst die zufällige Position bestimmt und dann den Lebenspunkt generiert und im Array speichert.

Hinweis: Eventuell hilft Ihnen eine `while`-Schleife (mit einer entsprechenden `boolean`-Variable) weiter, um eine neue Position zu finden, falls der `LifePoint` auf eine Mauer gesetzt werden würde.

e) Sorgen Sie nun in der `paintGame(Graphics)`-Methode dafür, dass die erstellen Lebenspunkte auf dem Spielfeld angezeigt werden.

f) Die `LifePoints` sollen natürlich genauso wie die `Food`-Objekte verschwinden, wenn der Pacman über das entsprechende Feld läuft. Fügen Sie diese Kollisionsbehandlung (unter Verwendung von `Pacman.collidesWithItem(...)`) zur Methode `checkCollisions()` hinzu.

g) Da Sie nun Lebenspunkte erstellt haben, können wir jetzt vom Attribut `lives` (die die aktuelle Anzahl an Lebenspunkten speichern soll) Gebrauch machen. Am Anfang des Spieles soll der Pacman 3 Leben haben, bei Kollision mit einem Lebenspunkt soll er natürlich jeweils ein Leben mehr bekommen.

Hinweis: Es werden nun Felder entstehen, auf denen sowohl Lebens- als auch Essenspunkte liegen. Je nachdem, in welcher Reihenfolge Sie die beiden Punkte in der Methode `paintGame(Graphics)` zeichnen, verdecken die Lebens- die Essenspunkte. Das ist aber nicht weiter schlimm.

12. **Enemy**

Fast geschafft! Jetzt müssen wir nur noch die Feinde erzeugen:

a) Erstellen Sie dazu eine neue Klasse `Enemy` als Unterklasse von `GameItem`. Der Konstruktor soll als Übergabeparameter die x- und y-Koordinaten im Gitter-Koordinatensystem bekommen. Die Feinde sollen Quadrate sein, die genau ein Gitter-Feld ausfüllen. Suchen Sie sich eine Farbe für die Darstellung der Feinde aus.

b) Erstellen Sie nun in der Klasse `PacmanGame` ein privates Attribut für einen Feind. Ergänzen Sie den Konstruktor entsprechend und legen Sie einen beliebigen Startpunkt im Spielfeld fest. Lassen Sie sich den eben erstellten Feind anzeigen.

c) Kümmern Sie sich nun um die Bewegung des Feindes! Diese funktioniert ähnlich zu der des Pacman. Nur ist hier zu beachten, dass der Computer den Feind steuert.

d) `Enemy` erbt die Methode `step()` von der Klasse `GameItem`. Allerdings müssen Sie noch die Methode `setNextDirection(Brick[])` implementieren, die als Übergabeparameter die Spielfeld-Mauern als `Brick`-Array bekommt und daraufhin die nächste Bewegungsrichtung des Feindes folgendermaßen festlegen soll: Die Bewegungsrichtung soll *zufällig* ausgewählt werden. Allerdings soll der Feind **nicht** in die Richtung gehen können, aus der er gerade kam (also wenn sich die Figur z. B. gerade nach rechts bewegt, soll sie im nächsten Schritt nicht nach links zurück können). Außerdem müssen natürlich **Kollisionen** mit den Mauern (⇝ `collidesWithWall(...)`) vermieden werden. Überlegen Sie sich, wie sich diese Vorgaben am besten umsetzen lassen!

e) Erwecken Sie nun den Feind zum Leben, indem Sie in der `updateGame(long)`-Methode zuerst seine nächste Bewegungsrichtung mittels `setNextDirection(Brick[])` festlegen und ihn dann mittels `step()` einen Schritt machen lassen.

> Wir bereits vorhin erwähnt, sollen sich der Pacman und seine Feinde jeweils *abwechselnd* – also in unterschiedlichen Aufrufen von `updateGame(long)` – bewegen!

f) Wenn Sie ihr Programm nun ausführen, sollte der sich Feind zufällig über das Spielfeld bewegen und dabei keine Wände durchqueren können.

g) Nun dürfte Ihnen jedoch auffallen, dass eine Kollision zwischen dem Pacman und dem Feind (noch) keinen Effekt hat. Fügen Sie daher eine Kollisionsbehandlung in die Methode `checkCollisions()` in der Klasse `PacmanGame` ein. Dabei soll bei jeder Kollision mit dem Feind (⇝ `collidesWithItem(GameItem)`) ein Leben abgezogen werden.

h) Sind alle Leben verbraucht, soll das Programm beendet werden. Es soll die Meldung "You died!" sowie der aktuelle Punktestand über die Methode `showDialog(String text)` ausgegeben werden. Verlassen Sie danach die Methode.

i) Wenn Sie ihr Programm nun ausführen, sollte es möglich sein, durch den Feind Leben zu verlieren und schlussendlich getötet zu werden.

j) Stocken Sie nun die Anzahl der Feinde auf fünf auf. Ersetzen Sie dafür das Attribut `Enemy enemy` durch ein `Enemy`-Array und passen Sie Ihr Programm entsprechend an. Sie können die Startpositionen der Feinde entweder selbst festlegen, oder die statische Methode `generateEnemies(int width, int height)` der Klasse `GameEngine` verwenden. Diese ist standardmäßig auskommentiert, vor Verwendung müssten Sie sie also erst einkommentieren.

k) Ihr Programm sollte nun so aussehen wie in Abbdildung 4:

Achtung! Achten Sie insbesondere darauf, dass sowohl die Lebenspunkte als auch die Punktzahl im Titel angezeigt und mitaktualisiert werden! Testen Sie das Programm ausgiebig, um eventuelle Fehler und Bugs zu erkennen und zu beheben!

13. Geben Sie anschließend die Dateien `Brick.java`, `Direction.java`, `Enemy.java`, `Food.java`, `GameItem.java`, `LifePoint.java`, `Pacman.java`, `PacmanGame.java`, und `Point.java` im EST ab!

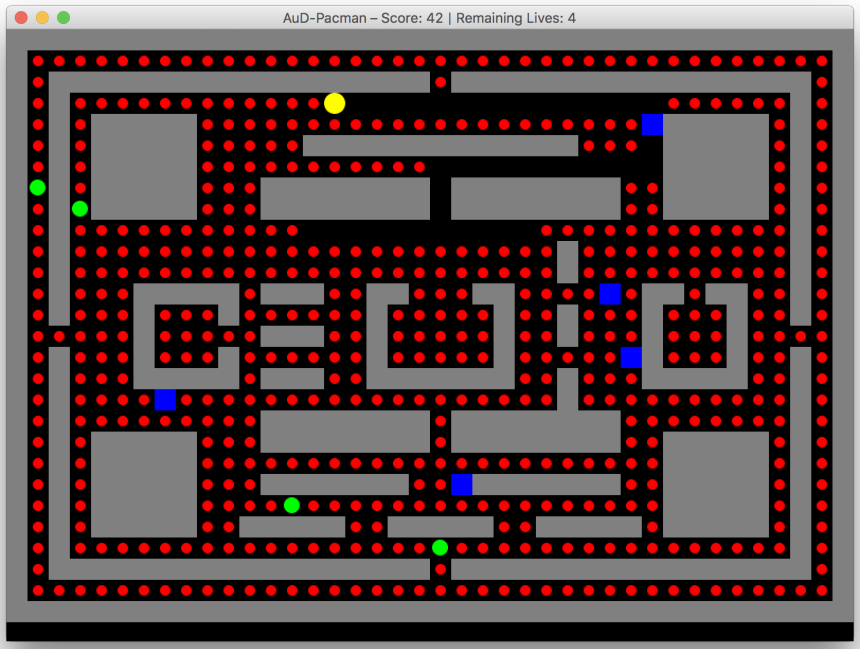

Abbildung 4: Fertiges Pacman-Spielfeld

Sollte Ihr Programm nicht übersetz- bzw. ausführbar sein, wird die Lösung mit 0 Punkten bewertet. Stellen Sie also sicher, dass IntelliJ IDEA keine Fehler in Ihrem Programm anzeigt, Ihr Programm übersetz- und ausführbar ist sowie die in der Aufgabenstellung vorgegebenen Namen und Schnittstellen *exakt* eingehalten werden. Geben Sie am Schluss die Dateien `Brick.java`, `Direction.java`, `Enemy.java`, `Food.java`, `GameItem.java`, `LifePoint.java`, `Pacman.java`, `PacmanGame.java` und `Point.java` über die EST-Webseite ab. Wenn Sie die Aufgabe zusammen mit einem Übungspartner bearbeitet haben, geben Sie im EST unbedingt dessen Gruppenabgabe-Code an! Kontrollieren Sie, ob Ihre Namen am Anfang aller Dateien angegeben sind – schreiben Sie im Quellcode Ihre Angaben in einen Kommentar. Im EST-Abgabesystem können Sie modifizierte Dateien mehrfach abgeben. Nur die zuletzt hochgeladene Version wird bewertet.