P3: Threads

Abgabetermin: 11. Mai 2025

Punktzahl: 42 + 5*

Inhalt

In diesem Projekt implementieren Sie eine Version des bekannten Spiels „Pac-Man[1]“. Der Fokus liegt auf der Anwendung von Threads, wobei jeder Thread eine spezifische Aufgabe in dem Spiel erfüllt. Die Spiellogik wird in einer Einzelspieler-Umgebung ausgeführt, in der kein Server notwendig ist.

Das Hauptprogramm verwaltet die verschiedenen Threads, die jeweils für eine spezifische Aufgabe verantwortlich sind. Der „PacMan“-Thread steuert die Bewegung von Pac-Man, der „Ghost“-Thread koordiniert die Bewegungen der Geister, ein „Collision“-Thread kümmert sich um die Kollisionserkennung und ein „Score“-Thread koordiniert die Punkteanzeige. Die Interaktion erfolgt durch die Pfeiltasten, mit deren Hilfe die Spielfigur horizontal oder vertikal bewegt werden kann. Das Ziel von Pac-Man ist es, alle Punkte einzusammeln, ohne von den Geistern gefangen zu werden. Eine Power-Pellet-Funktion ("Kraftpillen") ist in dieser vereinfachten Version des Spiels nicht implementiert.

Die Threads werden über das Hauptprogramm synchronisiert, welches den Spielstatus fortlaufend aktualisiert und eine reibungslose Darstellung des Spiels gewährleistet.
In der vorgegebenen Archivdatei `P3-vorgaben.zip` finden Sie bereits das Grundgerüst dieses Spiels. Hierbei fehlen aber einige Grundelemente für die grafische Oberfläche und das Thread-Management, welche Sie nun in den folgenden Aufgaben implementieren müssen.

In diesem Projekt sollen Sie zunächst die grafische Darstellung der Spielelemente sowie

[1] siehe z.B. https://de.wikipedia.org/wiki/Pac-Man

des Spielfelds (Aufgabe 1) und die Steuerung des PacMan-Objekts (Aufgabe 2) implementieren. Anschließend müssen Sie die Kollisionserkennung und die Geisterbewegung (Aufgabe 3) implementieren. In der letzten Aufgabe (Aufgabe 4) geht es dann darum Funktionen wie den Endscreen und die Punkteanzeige einzufügen, um das Spielgeschehen abzurunden.

Vorgaben

Die Vorgaben unter `P3_Vorgaben.zip` definieren bereits eine Grundstruktur, die Sie für Ihre Implementierung verwenden sollen. Darin sind mehrere Packages und ein zusätzliches Unterverzeichnis vorgegeben, die nachfolgend kurz erläutert werden.

Package `game`

Das `game`-Package implementiert die Hauptlogik und grafische Darstellung des Pac-Man-Spiels. Es verwaltet die verschiedenen Spielkomponenten wie das Spielfeld, die Geister, Pac-Man sowie den Punktestand und steuert den Spielablauf mithilfe von Threads.

- Die Klasse **GameLoop** steuert den zentralen Ablauf des Pac-Man-Spiels, indem sie die wichtigsten Threads startet und den Spielstatus verwaltet. Sie enthält das Attribut **gamePanel**, eine Referenz auf das **GamePanel**, welches die grafische Darstellung und Spiellogik des Spiels umfasst. Das Attribut **score** speichert den aktuellen Punktestand, während die Attribute **running** und **paused** (vom Typ `boolean`) den Spielstatus repräsentieren und anzeigen, ob die Threads laufen und ob sie pausiert sind.

  Die Methode **start()** startet die Threads. Mit **stopLoop()** wird das Spiel beendet, indem **running** auf **false** gesetzt wird. Um das Spiel anzuhalten, setzt die Methode **pauseGame()** das Attribut **paused** auf **true**, während **resumeGame()** das Spiel fortsetzt, indem **paused** wieder auf **false** gesetzt wird.

  Achtung: **pauseGame()** hat die Funktion, das Spiel im Falle einer Kollision (z.B. wenn Pac-Pan mit einem Geist gefressen wird) anzuhalten. Die Methode **stopLoop()** beendet die Threads.

  Die Methode **reset()** setzt die Positionen von Pac-Man und den Geistern zurück und aktualisiert die Anzeige im **gamePanel**, um den Spielzustand zurückzusetzen.

- Die Klasse **GamePanel** ist für die grafische Darstellung des Pac-Man-Spiels verantwortlich und enthält die wesentlichen Spielelemente sowie den zentralen Spiellogik-Loop. Sie umfasst das Attribut **pacMan**, das eine Instanz des Pac-Man-Objekts darstellt, und die Attribute **pinkGhost**, **orangeGhost**, **blueGhost** und **redGhost**, die jeweils eine der Geister-Instanzen repräsentieren. Das Attribut **listener**

ist ein **PacListener**, der die Benutzereingaben verarbeitet, und das Attribut **gameLoop** ist eine Referenz auf den **GameLoop**, der den Spielablauf steuert. Die Klassen für alle Spielobjekte (u.A. Geister und PacMan) befinden sich im Package `objects` (siehe unten).

Die Methode **initializeComponents()** initialisiert alle relevanten Spielelemente und erstellt die Instanzen von Pac-Man und den Geistern. Sie richtet ebenfalls den **PacListener** ein, der für die Steuerung der Pac-Man Spielfigur (aber die Pfeiltasten) zuständig ist. Die Methode **startGameLoop()** startet den zentralen **GameLoop**, um das Spiel in Gang zu setzen. Die **paintComponent(Graphics g)**-Methode wird überschrieben, um die Spielkomponenten zu zeichnen, indem sie die Methode **drawGame(Graphics2D g2)** aufruft, die für die detaillierte Darstellung der Spielobjekte zuständig ist.

- Die Klasse **GameWindow** erstellt das Fenster, in dem das Pac-Man-Spiel dargestellt wird und initialisiert das Spiel. Sie enthält das Attribut **gamePanel**, das eine Instanz der **GamePanel**-Klasse ist und die grafische Oberfläche sowie die Spielkomponenten darstellt.

  Die Methode **launch()** erzeugt eine Instanz von **GameWindow**, fügt dem Fenster das **gamePanel** hinzu und setzt die grundlegenden Fensterattribute (Titel, Größe etc.).

- Die Klasse **Maze** definiert das Spielfeld des Pac-Man-Spiels und speichert den aktuellen Zustand des Labyrinths. Sie enthält das statische Attribut **maze**, das ein zweidimensionales Array vom Typ **Type** ist und die Struktur des Labyrinths repräsentiert.

  Die Methode **calculateScore()** berechnet den Punktestand basierend auf den eingesammelten Punkten, indem sie die leeren Felder im Labyrinth zählt. Sie beginnt mit einem Ausgangswert von -6, um den Punktestand entsprechend der anfangs leeren Felder zu kompensieren, und erhöht diesen um 1 für jedes leere Feld im Array **maze**.

  Die Methode **drawMaze(...)** zeichnet das Spielfeld.

Package `main`

Dieses package wird zum Ausführen des Programms verwendet.

- Die Klasse **Main** enthält die **main(String[])**-Methode zum Ausführen des Programms, in welcher die **launch()**-Methode aus der **GameWindow**-Klasse aufgerufen wird.

package `objects`

Das **objects**-Package enthält alle wichtigen Spielobjekte für das Pac-Man-Spiel, darunter PacMan und die Geister. Es definiert die grundlegenden Eigenschaften der Spielfiguren, wie Position und Bewegungslogik, sowie die Verarbeitung von Benutzereingaben zur Steuerung. Zusätzlich enthält es Algorithmen für die Geisterbewegung, um die Spielumgebung dynamisch zu gestalten.

- Die Klasse **GameEntity** ist eine abstrakte Basisklasse, die die grundlegenden Eigenschaften und Methoden für die Spielobjekte **PacMan** und **Ghost** definiert. Sie speichert die Position der Spielobjekte durch die Attribute **x** und **y** und stellt die Methode **move(Direction direction)** bereit, um das Objekt in eine angegebene Richtung zu bewegen. Darüber hinaus enthält die Klasse Getter- und Setter-Methoden für die Koordinaten, sodass die Position der Spielobjekte angepasst werden kann. Außerdem wird in der Klasse die Konstante `resDir` definiert, welche den Pfad zu dem Unterverzeichnis `resources` (siehe unten) beinhaltet. Die Variable benötigen Sie später um Bilder aus diesem Verzeichnis zu laden.

- Die Klasse **Ghost** repräsentiert die Geister im Pac-Man-Spiel und erweitert die abstrakte Basisklasse **GameEntity**. Sie speichert die Farbe des Geistes im Attribut **color** und lädt das passende Bild für jeden Geist anhand der Methode **loadImages()**. Diese Methode weist jedem Geist basierend auf seiner Farbe ein spezifisches Bild zu, das aus dem `resources` Unterverzeichnis geladen wird. **Ghost** enthält zudem Getter- und Setter-Methoden für die Attribute **image** und **color**, um das Bild und die Farbe anpassen zu können. Zusätzlich enthält Sie die **drawGhost(...)**-Methode welche den Geist einzeichnet.

- Die Klasse **GhostAlgorithms** implementiert die Bewegungslogik der Geister im Pac-Man-Spiel. Sie enthält Methoden zur Steuerung der Geisterbewegung, basierend auf der Farbe und Art des Geistes. Die Methode **makeMove(Ghost ghost, PacMan pac)** bestimmt das Ziel der Geisterbewegung relativ zu Pac-Man und ruft spezifische Bewegungsstrategien auf, wie **redGhostMove(...)**, **blueGhostMove(...)**, **pinkGhostMove(...)** und **orangeGhostMove(...)**. Zusätzlich bietet **makeRandomMove(Ghost ghost)** eine zufällige Bewegungsoption, falls keine spezielle Strategie implementiert wurde. Die Bewegungslogik wird in verschiedenen Threads genutzt, um die Geister dynamisch durch das Labyrinth zu bewegen.

- Die Klasse **PacListener** ist ein Listener für die Benutzereingaben im Pac-Man-Spiel und verarbeitet Tastatureingaben zur Steuerung von Pac-Man. Sie enthält Referenzen auf **PacMan** und **GameLoop**, um die Spielsteuerung und -interaktion zu ermöglichen. Durch die Implementierung von Tastendruckereignissen weist **PacListener** Pac-Man eine Bewegungsrichtung zu und ermöglicht so die direkte Kontrolle des Spielers über das Pac-Man-Objekt.

- Die Klasse **PacMan** repräsentiert die Spielfigur Pac-Man im Spiel und erweitert die abstrakte Klasse **GameEntity**. Sie speichert die aktuelle Bewegungsrichtung im Attribut **direction** und stellt Getter- und Setter-Methoden bereit, um diese Richtung zu aktualisieren. Die Klasse lädt außerdem mit Hilfe der `loadImages` Methode verschiedene Bilder für Pac-Man in den vier Bewegungsrichtungen (oben, unten, links, rechts), um die grafische Darstellung je nach aktueller Bewegungsrichtung anzupassen. Zusätzlich implementiert die Klasse die **drawHeart(...)**-Methode, welche für die Darstellung der verbliebenen Leben zuständig ist und die **drawPac(...)**-Methode, welche für die Darstellung von Pac-Man zuständig ist.

Package `threads`

Das **threads**-Package enthält alle Threads, die die verschiedenen Spielkomponenten steuern. Diese Threads sorgen für die kontinuierliche Bewegung und Kollisionsabfragen der Spielfiguren, die Aktualisierung des Punktestands und die Synchronisation der Spielabläufe. Durch die parallele Ausführung der Threads können die Geister, Pac-Man und andere Spielzustände unabhängig voneinander aktualisiert werden, was das Spiel dynamisch gestaltet. Zu Übungszwecken werden die Threads hier auf mehrere verschiedene Arten erzeugt. Natürlich ist dies in "normalen" Projekten nicht der Fall, sondern wird einheitlich gemacht.

- Die Klasse **CollisionThread** ist dafür zuständig, Kollisionen im Pac-Man-Spiel zu überwachen und zu verwalten. Sie überprüft kontinuierlich, ob Pac-Man mit einem der Geister kollidiert, um daraufhin entsprechende Spielereignisse wie das Verlieren eines Lebens auszulösen.

- Die Klasse **GhostThread** steuert die Bewegung der Geister im Pac-Man-Spiel. In regelmäßigen Abständen ruft der **GhostThread** die Methode **moveGhosts()** auf, die basierend auf Pac-Mans aktueller Position die Bewegung jedes Geistes mithilfe von Algorithmen aus der Klasse **GhostAlgorithms** berechnet. Das **GamePanel** wird nach jeder Bewegung aktualisiert, um die Änderungen visuell anzuzeigen.

- Die Klasse **PacManThread** steuert die Bewegung von Pac-Man im Spiel und sorgt dafür, dass Pac-Man kontinuierlich in die zuletzt gewählte Richtung weiterläuft, sofern es zu keiner Kollision kommt. Hierzu wird die Position des Pac-Man in regelmäßigen Abständen aktualisiert und dafür gesorgt, dass das **GamePanel**aktualisiert wird, sodass die Bewegungen visuell dargestellt werden.

- Die Klasse **ScoreThread** aktualisiert den Punktestand des Pac-Man-Spiels und überwacht den Spielstatus. Als eigenständiger Thread läuft dieser kontinuierlich im Hintergrund und ruft regelmäßig die Methode **updateScore()** auf, um den aktuellen Punktestand anhand der eingesammelten Punkte zu berechnen. Zusätzlich prüft die Methode **checkGameStatus()**, ob Pac-Man alle Leben verloren hat oder alle Dots eingesammelt hat und beendet gegebenenfalls das Spiel. Der **ScoreThread** gewährleistet so eine konstante Überwachung und Anpassung des Spielstands, ohne den Spielfluss zu unterbrechen.

Package `utils`

Das **utils**-Package stellt Konstanten und Enums für das Pac-Man-Spiel bereit, darunter Spielgrößen, Geschwindigkeitseinstellungen und das Labyrinthlayout. Es definiert die Bewegungsrichtungen, Geisterfarben und Spielfeldtypen zur Unterstützung der Spiellogik und -struktur.

- **Constants**: Enthält grundlegende Spielkonstanten wie die Spielfeldgröße, Geschwindigkeitseinstellungen und das (initiale) Design des Labyrinths in **INITIAL_MAZE**.
- **Direction**: Ein Enum zur Festlegung der Bewegungsrichtungen (**UP**, **DOWN**, **LEFT**, **RIGHT**, **NONE**).
- **GhostColor**: Ein Enum zur Definition der Geisterfarben (**RED**, **PINK**, **ORANGE**, **BLUE**).
- **Type**: Ein Enum zur Kennzeichnung der verschiedenen Spielfeldtypen (**WALL**, **DOT**, **EMPTY**), die zur Strukturierung des Labyrinths verwendet werden, welches sich aus quadratischen Bausteinen (auch Kacheln genannt) zusammensetzt.

Unterverzeichnis `resources`

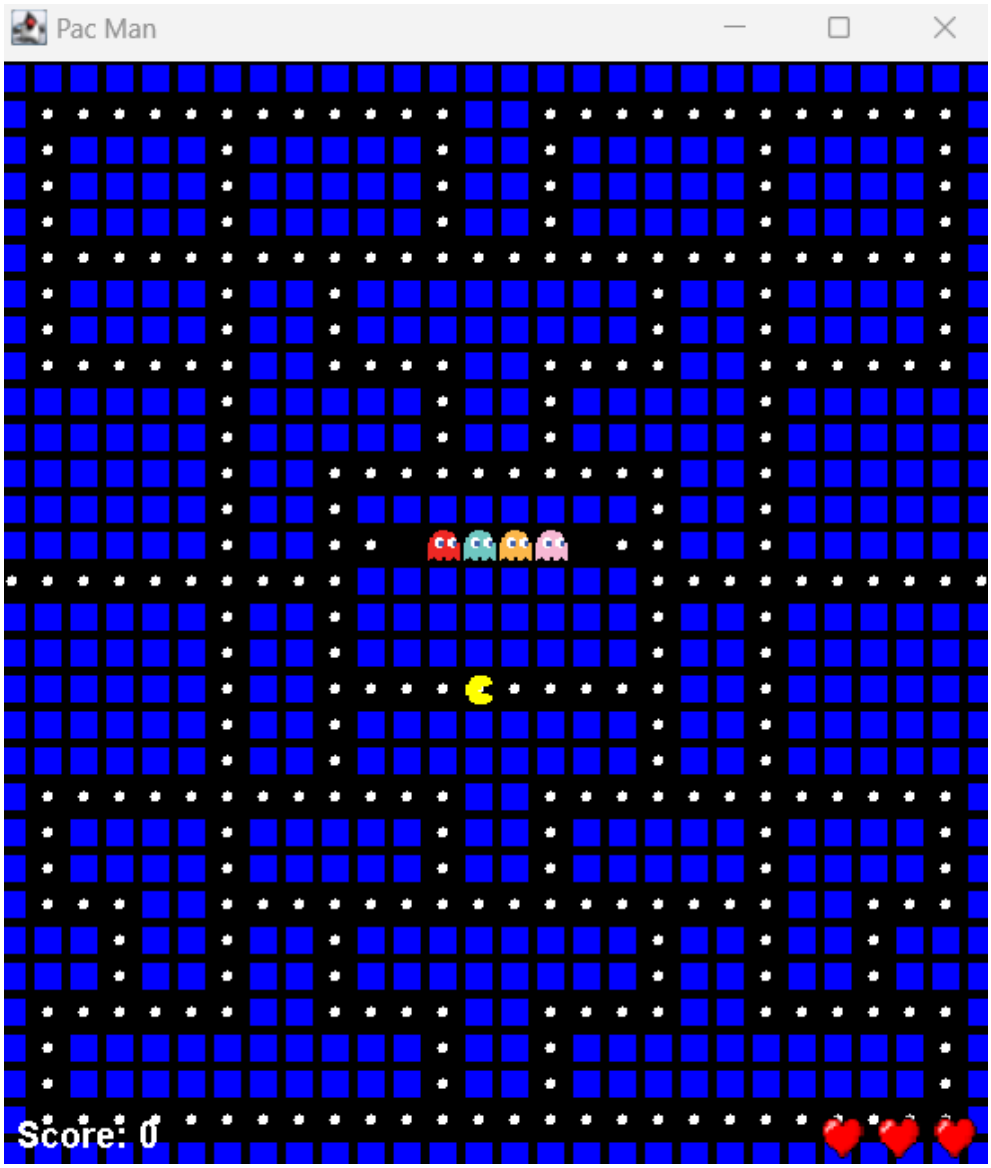
In diesem Verzeichnis befinden sich Bilder im **.jpg**-Format, welche in den Klassen **PacMan** und **Ghost** eingelesen werden.

Aufgabenstellung

Aufgabe 1: Grafik und Spielmechanik (18 P)

Das Ziel dieser Aufgabe ist es, die grafische Darstellung und Funktionalität des Pac-Man-Spiels zu vervollständogen. Dies beinhaltet die Implementierung von Methoden zur Kollisions- und Bewegungsüberprüfung sowie die visuelle Darstellung des Spielfelds, der Geister, des Pac-Man und seiner Lebensanzeige. Dabei sollen grundlegende Spielmechanismen wie valide Bewegungen, die Reaktion auf Kollisionen und die korrekte Anzeige von Spielobjekten in der Benutzeroberfläche entwickelt und integriert werden.

Nach erfolgreichem Implementieren der Aufgabe 1 sollte die GUI ungefähr so aussehen:



(a) drawMaze (4 P)

Ziel dieser Teilaufgabe ist es, das Spielfeld grafisch darzustellen. Die Methode `drawMaze(Graphics2D g2)` in der Klasse `Maze` soll dazu genutzt werden, um das zweidimensionale Array `maze` als sichtbares Spielfeld in der Benutzeroberfläche zu zeichnen. Das Spielfeld wird dabei durch ein regelmäßiges Gitter aus (unterschiedlichen) Rechtecken gebildet.

- Jede Zelle im Array `maze` repräsentiert einen rechteckigen Bereich auf dem Spielfeld und enthält einen Wert des Typs `Type` (`WALL`, `DOT` oder `EMPTY`).
- Zeichnen Sie für jede Zelle ein Rechteck an der passenden Stelle:
  - `WALL`: ein blaues gefülltes Rechteck (Tipp: mit leichtem Abstand zum Rand für bessere Optik).
  - `DOT`: ein kleiner weißer Punkt, zentriert im Feld (nutzen Sie dafür die bereits vorhandene Methode `drawDots`).
  - `EMPTY`: kann ausgelassen werden, da der Hintergrund schwarz ist.
- Die Größe eines Feldes ergibt sich aus der Konstante `Constants.TILE_SIZE`.
- Um die Positionen korrekt zu berechnen, verwenden Sie: `x = spalte * TILE_SIZE`, `y = zeile * TILE_SIZE`.
- Füllen Sie vor dem Zeichnen das gesamte Spielfeld mit einem schwarzen Rechteck, um sicherzustellen, dass keine vorherigen Inhalte sichtbar bleiben.

Hinweis: Orientieren Sie sich bei der Darstellung an dem Beispielbild unten. Ziel ist es, ein schachbrettartiges Layout zu erzeugen, in dem Wände, Punkte und leere Felder visuell klar voneinander zu unterscheiden sind.

(b) Pacman-Grafik implementieren (4 P)

In dieser Teilaufgabe soll die Darstellung des Pacman-Charakters umgesetzt werden. Ziel ist es, je nach aktueller Bewegungsrichtung das passende Bild des Pacman an der richtigen Position im Spielfeld zu zeichnen.

- Erweitern Sie die Methode `loadImages` in der Klasse `Pacman`, die von der abstrakten Klasse `GameEntity` erbt.
- Laden Sie die vier Bilder für die unterschiedlichen Bewegungsrichtungen (oben, unten, links, rechts) aus dem Ordner `Resources`. Nutzen Sie dafür die Klasse `ImageIO`.
- Initialisieren Sie vier entsprechende Variablen (`pacmanUpImage`, `pacmanDownImage`, `pacmanLeftImage`, `pacmanRightImage`) im Konstruktor.
- Vervollständigen Sie die Methode `drawPac` so, dass:

- die aktuelle Spielfeldposition des Pacman (x und y) korrekt berechnet wird (`x = getX() * TILE_SIZE`, `y = getY() * TILE_SIZE`),
- abhängig von der Bewegungsrichtung das passende Bild ausgewählt wird,
- das gewählte Bild an der korrekten Position auf das `Graphics2D`-Objekt gezeichnet wird.

Hinweis: Stellen Sie sicher, dass die Methode auch bei Richtungswechseln das richtige Bild verwendet und korrekt positioniert. Orientieren Sie sich dafür ggf. an der Methode `drawHeart`.

(c) Geister zeichnen und Bilder laden (3 P)

In dieser Teilaufgabe sollen die Geister im Spiel grafisch dargestellt werden. Ziel ist es, dass jeder Geist ein farblich passendes Bild erhält und korrekt im Spielfeld angezeigt wird.

- Erweitern Sie die Klasse `Ghost`, die von `GameEntity` erbt, um eine Methode `loadImage`, die abhängig von der `color` des Geists das zugehörige Bild lädt und der Instanzvariable `image` zuweist.
- Verwenden Sie zur Bildauswahl eine `switch`-Anweisung, mit der für jede Farbe ein String mit dem zugehörigen Pfadnamen generiert wird. Die Bilder befinden sich im Projekt-Unterverzeichnis `resources`.
- Laden Sie das Bild mithilfe von `ImageIO.read` und speichern Sie es in einem Attribut (z. B. `image`).
- Implementieren Sie die Methode `drawGhosts`, welche das aktuell geladene Bild an der richtigen Spielfeldposition zeichnet. Berechnen Sie dafür die Zeichenposition wie gewohnt über die Feldkoordinaten und `Constants.TILE_SIZE`.

Hinweis: Testen Sie die Methode mit mehreren Geistern in unterschiedlichen Farben, um sicherzustellen, dass das Laden und Zeichnen korrekt funktioniert.

(d) Bewegungsüberprüfung implementieren (4 P)

In dieser Teilaufgabe geht es darum, die Spiellogik um eine Überprüfung für gültige Bewegungen zu erweitern. Ziel ist es, zu erkennen, ob ein Objekt (Pacman oder Geist) sich in eine bestimmte Richtung bewegen darf.

- Implementieren Sie in der Klasse `GameEntity` eine Methode `isValidMove(Direction direction)`.
- Die Methode soll überprüfen, ob das Objekt bei einem Zug in die angegebene Richtung:

  - innerhalb des Spielfeldes (`maze`) bleibt,
  - nicht gegen eine Wand (`Type.WALL`) läuft.

- Für die Überprüfung der Spielfeldgrenzen können Sie die Höhe und Breite des Arrays `Maze.maze` nutzen.

- Ergänzen Sie zusätzlich die Klasse `GhostAlgorithms`:
  - Passen Sie die Methode `attemptMove` so an, dass ein Zug nur ausgeführt wird, wenn er laut `isValidMove` zulässig ist.
  - In diesem Fall soll die Methode `true` zurückgeben; andernfalls `false`.

Hinweis: Eine Kollision mit anderen Spielfiguren (Pacman oder Geister) soll in dieser Aufgabe noch nicht überprüft werden – das folgt in einem späteren Schritt.

(e) Kollisionserkennung implementieren (3 P)

In dieser Teilaufgabe soll eine einfache Methode zur Kollisionserkennung zwischen Spielfiguren implementiert werden.

- Fügen Sie in der Klasse `GameEntity` eine `statische Methode` mit dem Namen `collisionDetection` hinzu.

- Die Methode soll zwei Spielfiguren (z. B. `PacMan` und `Ghost`) als Parameter erhalten.

- Überprüfen Sie, ob sich beide Objekte aktuell auf derselben Position im Spielfeld befinden (d. h. identische x- und y-Koordinaten).

- Wenn eine Kollision vorliegt, geben Sie `true` zurück, andernfalls `false`.

- Optional: Sie können in der Methode z. B. direkt `pacMan.decreaseLives()` aufrufen, wenn dies sinnvoll erscheint – dies ist aber nicht zwingend Bestandteil dieser Aufgabe.

Zwischenziel Aufgabe 1:
Nach erfolgreicher Bearbeitung aller Teilaufgaben von Aufgabe 1 sollte die Benutzeroberfläche beim Ausführen der `main`-Methode der mitgelieferten Beispielabbildung entsprechen.

Aufgabe 2: Listener und Pac-Man-Thread (11 P)

Das Ziel dieser Aufgabe ist es, die Steuerung und Thread-Implementierung des Pac-Man-Spiels zu realisieren, um eine flüssige und unabhängige Bewegung des Pac-Man-Objekts zu ermöglichen. Dabei soll die Bewegung sowohl auf Nutzereingaben reagieren als auch automatisiert in festgelegten Zeitabständen erfolgen, wobei Synchronisation und zeitliche Beschränkungen beachtet werden. Durch die Integration eines Listeners und eines Threads wird die Spielmechanik erweitert.

(a) PacListener implementieren (5 P)

In dieser Teilaufgabe soll eine Steuerung für den Pacman implementiert werden, mit der Spielende die Bewegungsrichtung über die Pfeiltasten verändern können.

- Erweitern Sie die Klasse `PacListener`, sodass sie von der Klasse `KeyAdapter` nicht `KeyListener` erbt, um nur benötigte Methoden zu überschreiben
- Erstellen Sie nun zwei Hilfsmethoden:
  - `getDirectionFromKey(int keyCode)`: Diese Methode soll eine Richtung vom Typ `Direction` basierend auf dem übergebenen Tastencode zurückgeben. Verwenden Sie ein `switch`-Statement, um die Pfeiltasten (`VK_UP`, `VK_DOWN`, `VK_LEFT`, `VK_RIGHT`) auf die entsprechenden Richtungen (vom Typ `Direction`) abzubilden. Gibt es keine passende Taste, soll `null` zurückgegeben werden.
  - `keyUnequalsDirection(KeyEvent e)`: Vergleichen Sie die aktuelle Richtung von Pac-Man (`pacMan.getDirection()`) mit der durch den Tastencode bestimmten neuen Richtung (`getDirectionFromKey(e.getKeyCode())`). Gibt es einen Unterschied, geben Sie `true` zurück, sonst `false`.
- Fügen Sie ein boolean Attribut `hasMoved` zur Klasse hinzu.
- Überschreiben Sie die Methode `keyPressed(KeyEvent e)` in der Klasse `PacListener`, sodass auf Tasteneingaben sinnvoll reagiert wird. Gehen Sie dabei wie folgt vor:
  - Falls das Spiel pausiert ist (`gameLoop.paused`), setzen Sie es mit `gameLoop.resumeGame()` fort.
  - Prüfen Sie mit der Methode `keyUnequalsDirection(e)`, ob die gedrückte Richtung sich von Pac-Mans aktueller Bewegungsrichtung unterscheidet.
  - Bestimmen Sie anschließend mit `getDirectionFromKey(e.getKeyCode())` die gewünschte Richtung. Falls diese nicht `null` ist und der Zug in diese Richtung gültig ist (`pacMan.validMove(...)`), soll:
    - Pac-Mans Richtung mit `setDirection(...)` aktualisiert werden,
    - Pac-Man sich sofort in die neue Richtung bewegen (`move(...)`),

    - das Attribut `hasMoved` auf `true` gesetzt werden.

- Entfernen Sie nach erfolgreicher Implementierung die Kommentare im Konstruktor der Klassse `GamePanel`, sodass der Listener hinzugefügt wird.

(b) PacManThread erweitern (6 P)

In dieser Teilaufgabe soll ein eigener Thread für die kontinuierliche Bewegung des Pacman entwickelt werden. Der Pacman soll sich dabei automatisch in die zuletzt gewählte Richtung bewegen – unabhängig davon, ob gerade eine Taste gedrückt wird.

- Erweitern Sie die Klasse `PacManThread`, sodass sie von `Thread` erbt.

- Implementieren Sie dabei zunächst folgende Hilfsmethoden, bevor Sie die `run` Methode überschreiben.

- Implementieren Sie eine private Methode `canMovePacMan()`, die prüft, ob Pac-Man sich bewegen darf. Die Methode soll genau dann `true` zurückgeben, wenn:

  - Pac-Man sich noch nicht durch eine Nutzereingabe in der aktuellen Spielrunde bewegt hat (`!gamePanel.listener.hasMoved()`), **und**
  - die Bewegung in die aktuelle Richtung gültig ist (`gamePanel.pacMan.validMove(gamePanel.pacMan.getDirection())`).

  Verwenden Sie eine kompakte Rückgabe mit einem logischen UND-Ausdruck.

- Implementieren Sie die Methode `updateMazePosition()`, die Pac-Mans aktuelle Position im Labyrinth auf `Type.EMPTY` setzt. Gehen Sie dabei wie folgt vor:

  - Ermitteln Sie Pac-Mans aktuelle X- und Y-Koordinaten.
  - Setzen Sie das entsprechende Feld im zweidimensionalen Spielfeld-Array `Maze.maze` auf den Wert `Type.EMPTY`, um die alte Position zu „löschen“.

- Implementieren Sie die Methode `movePacMan()`, die Pac-Man bewegt und das Spielfeld entsprechend aktualisiert. Gehen Sie dabei wie folgt vor:

  - Synchronisieren Sie den Methodeninhalt auf dem Objekt `gamePanel.pacMan`, um parallelen Zugriff durch andere Threads zu vermeiden (mit einem `synchronized` Block).
  - Überprüfen Sie mit `canMovePacMan()`, ob eine Bewegung (in die aktuell gesetzte Richtung) erlaubt ist.
  - Falls die Bewegung erlaubt ist, führen Sie in folgender Reihenfolge die nötigen Schritte aus:
    1. Rufen Sie `updateMazePosition()` auf, um die alte Position im Labyrinth zu leeren.

2. Bewegen Sie Pac-Man in seine aktuelle Richtung mit `gamePanel.pacMan.move(gamePanel.pacMan.getDirection())`.
3. Leeren Sie die neue Position nach der Bewegung mit `clearNewPosition()`.

- Überschreiben Sie nun die Methode `run()`:
  - Solange `gamePanel.gameLoop.running true` ist, soll der Thread in einer Schleife laufen.
  - Prüfen Sie in jeder Iteration, ob das Spiel nicht pausiert ist. Falls ja:
    * Rufen Sie die Methode `movePacMan()` auf, um den nächsten Bewegungsschritt durchzuführen.
    * Setzen Sie anschließend den Zustand des Listeners über `gamePanel.listener.resetMove()` zurück.
  - Steuern Sie die Geschwindigkeit des Threads über `Thread.sleep(1000 / Constants.GAME_SPEED)`.
  - Achten Sie darauf, `InterruptedException` korrekt zu behandeln: Unterbrechen Sie den Thread sauber mit `Thread.currentThread().interrupt()`.

Zwischenziel Aufgabe 2:
Der Pacman lässt sich mit den Pfeiltasten steuern, bewegt sich kontinuierlich in die zuletzt gewählte Richtung, und Geister bewegen sich durch das Labyrinth. Kollisionen haben zu diesem Zeitpunkt noch keine Auswirkung.

Aufgabe 3: Geisterkollision und Spiellogik (5 P)

In dieser Aufgabe sollen Kollisionen zwischen Pacman und den Geistern erkannt und korrekt behandelt werden. Dazu implementieren Sie einen eigenen Thread, der regelmäßig den Spielfeldstatus prüft.

- Erweitern Sie die Klasse `CollisionThread`, sodass sie das Interface `Runnable` implementiert.
- Erstellen Sie zunächst zwei Hilfsmethoden:
- Erstellen Sie die Methode `isCollisionDetected()`, die `true` zurückgibt, sobald Pac-Man mit (mindestens) einem der vier Geister kollidiert. Verwenden Sie dazu die Methode `GameEntity.collisionDetection(...)` und einen passenden logischen Ausdruck.
- Erstellen Sie die Methode `checkCollisions()`, die überprüft, ob eine Kollision vorliegt. Falls ja, setzen Sie das Spiel zurück (`reset()`) und pausieren Sie es (`pauseGame()`). Synchronisieren Sie den Zugriff auf das `gamePanel`.

- Überschreiben Sie die Methode `run()`, sodass in einer Schleife regelmäßig auf Kollisionen geprüft wird:
  - Solange `gamePanel.gameLoop.running` `true` ist, soll `checkCollisions()` aufgerufen werden.
  - Verzögern Sie jede Schleifeniteration mit `Thread.sleep(500 / Constants.GAME_SPEED)`.
  - Behandeln Sie `InterruptedException`, indem Sie den Thread unterbrechen.

Zwischenziel Aufgabe 3:
Kollisionen mit Geistern werden zuverlässig erkannt. Bei einer Kollision wird das Spiel pausiert, das Spielfeld wird zurückgesetzt, und Pacmans Leben reduziert sich – die Spielfunktionalität bleibt erhalten.

Aufgabe 4: Spielstand und Endscreen (8 P)

In dieser Aufgabe sollen Sie den Punktestand des Spielers anzeigen und einen Endbildschirm implementieren, der nach Spielende angezeigt wird.

(a) Spielstand anzeigen (3 P)

- Erstellen Sie in der Klasse `GamePanel` eine Methode `drawScore`, die den aktuellen Punktestand anzeigt. Die Methode soll als Parameter einen Graphics2D Objekt besitzen, in den gezeichnet werden soll. Anmerkung. Dieser muss den Graphics Kontext des `GamePanel` Objekts darstellen, was beim Aufruf der Methode (siehe unten) gewährleistet werden muss.
- Der Score kann mithilfe der Methode `Maze.calculateScore()` berechnet werden.
- Zeichnen Sie den Score unten links in die Spielfläche. Wählen Sie eine passende Schriftart und -größe sowie eine gut lesbare Farbe.
- Rufen Sie die Methode `drawScore` innerhalb von `drawGame` auf.

(b) Endscreen anzeigen (3 P)

- Implementieren Sie in der Klasse `GamePanel` eine Methode `displayEndScreenIfGameOver`.
- Die Methode soll prüfen, ob das Attribut `gameLoop.running` auf `false` gesetzt ist. Falls ja:
  - löschen Sie die Spielfläche mit einem schwarzen Hintergrund,
  - zeichnen Sie eine große "Game Over"-Nachricht zentriert auf dem Bildschirm,

  - zeigen Sie den finalen Punktestand darunter an.

- Rufen Sie diese Methode ebenfalls in `drawGame` auf.

(c) Spiel beenden **(2 P)**

- Implementieren Sie in der Klasse `ScoreThread` die Methode `checkGameStatus()`, die das Spielende prüft. Beenden Sie das Spiel durch Aufruf von `gameLoop.stopLoop()`, wenn eine der folgenden Bedingungen erfüllt ist:

  - Pac-Man hat keine Leben mehr (`pacMan.getLives() <= 0`), oder
  - die aktuelle Punktzahl (`gameLoop.score`) ist größer oder gleich `Constants.MAX_SCORE`.

  Rufen Sie anschließend `gamePanel.repaint()` auf, um die Anzeige zu aktualisieren.

Zwischenziel Aufgabe 4:
Während des Spiels wird der aktuelle Punktestand angezeigt. Bei Spielende erscheint ein "Game Over"-Bildschirm mit dem finalen Score und das Spiel wird ordnungsgemäß gestoppt.

Bonusaufgabe: Suchalgorithmen für die Geister (*5 P)

- Entwickeln Sie eigene Ansätze für simple Suchalgorithmen, die das Verhalten der vier Geister steuern können. Aktuell führen diese jeweils durch Aufruf der Methode `makeRandomMove` zufällige Bewegungen durch. Hierzu müssen Sie in der Klasse `GhostAlgorithms` entsprechende Methoden ergänzen und die Aufrufe in den Methoden `xxxGhostMove` entsprechend ändern.

- Beschreiben Sie kurz den Algorithmus und dessen Funktionsweise. Hier sind einige mögliche Ansätze:

  - **Zufallssuche:** Der Geist bewegt sich zufällig in eine valide Richtung.
  - **Zielgerichtete Suche:** Der Geist bewegt sich auf den **PacMan** zu, indem er stets die Richtung mit der geringsten Distanz wählt.
  - **Suchmuster:** Der Geist folgt einem festen Bewegungsschema, z. B. durch das Spielfeld im Uhrzeigersinn.
  - **Strategische Umzingelung:** Der Geist versucht, den **PacMan** zu umzingeln, indem er eine Position wählt, die dessen Fluchtoptionen reduziert. Dies erfordert eine vorausschauende Berechnung möglicher Bewegungen.

- Denken Sie sich einen eigenen Algorithmus aus oder erweitern Sie die gegebenen.