Java Abschlussaufgabe

Von:

Jonas Knoll, Max Zeitler, Elena Solodova,

Maximilian Pazer, Robert Knabe, Federico Collasius

Inhaltsverzeichnis

1. Zielsetzung

Das Ziel des Programmierprojekts, war es eine ähnliche Spiellogik, wie in den klassischen Mario Party Spielen zu erschaffen. Dazu haben wir geplant, ein Hauptspiel mit verschiedenen kleineren Unterspielen (Minispielen) zu programmieren. Das Hauptspiel soll also zwischen den Minispielen koordinieren und je nach Spielablauf unterschiedliche Spiele auswählen. Jedes Gruppenmitglied sollte mindestens ein Minispiel programmieren, während wir die Hauptspielklasse untereinander aufteilten. Die Auswahl der Minispiele haben wir jedem selbst überlassen, wodurch jeder ein Spiel auswählen konnte, was seinen Fähigkeiten und natürlich Interessen entspricht. Jeder konnte für sein Minispiel entscheiden, welche Schwierigkeitsgrade es haben und wie stark es vom Original abweichen sollte, um das in der Vorlesung erworbene Wissen selbstständig anzuwenden.

Zu Beginn der Umsetzung haben wir gemeinsam einen Plan ausgearbeitet, ein Git-Repository erstellt und die ersten Klassen des Hauptspiels implementiert. Um gemeinsames Arbeiten zu ermöglichen, wurde auf Feature branches gearbeitet. Der Fortschritt wurde über die Commits gesichert. Bei aufkommenden Fragen stand niemand allein da und wir unterstützten uns so weit wie möglich.

Link zum GitHub Repository (öffentlich): https://github.com/DerEine117/marioParty

Das Ergebnis unserer Gruppenarbeit ist ein Hauptspiel, das gegen einen Computer gespielt wird, mit sechs verschiedenen Minispielen unterschiedlicher Komplexität.

2. Hauptgame

2.1 MainApplication

Die **Klasse "MainApplication**" ist der Einstiegspunkt für die Anwendung. Sie initialisiert die "**GameControlle**r"-Insatanz und erstellt die Szenen für die verschiedenen Spiele. Sie verwendet die „**StageChanger"-**Klasse**,** um die Szenen zu wechseln. Über diese Klasse kann die Anwendung ausgeführt und gestartet werden.

2.2 GameController

Die "**GameControlle**r"-Klasse verwaltet den Spielzustand, einschließlich der Spielerinstanzen und ihrer Münzenanzahl. Sie beinhaltet ebenfalls die Methoden zum Hinzufügen von Münzen für die Player Objekte. Von dieser Klasse erben, die Controller der Minispiele, somit können Minispielereignisse, wie Sieg oder Niederlage, Methoden der Player Objekte aufrufen und das System wird vernetzt. Damit die Spieler nur einmalig erstellt werden und jedes Minispiel auf dieselben Spieler zugreifen kann, werden diese in der „**GameControlle**r"-Klasse einmalig durch die statische Methode erstellt.

2.3 StageChanger

Der „**Stagechanger**" ermöglicht es, zwischen den einzelnen Szenen, also den Minispielen und dem Mainspiel zu wechseln. Die Klasse besitzt ein „**window**"-Attribut, das auf das Hauptfenster („**Stage**")

verweist. Diesem Window wird in späteren Schritten die jeweilige Szene zugewiesen. Das funktioniert durch die setScene()-Methode, an die beim Aufrufen der Scene-Parameter übergeben wird und welche dann die passende Szene setzt. Die Methode createStageController() initialisiert diesen.

2.4 MainGame (Maxim, Max, Elena)

Die Klasse "MainGame" verwaltet das Hauptspiel, einschließlich der Spiellogik und Interaktion **mit dem Board und Player Instanzen**. Das Spielbrett wird mithilfe der **initialize()-**Methode initialisiert und aufgebaut. Dazu gehört das Zeichnen des Hintergrunds, das Einrichten des Spielfelds und das Platzieren der Spieler auf dem Brett.
Die Methode **würfeln()** wird aufgerufen, wenn ein Spieler auf den Würfelknopf klickt. Diese Methode simuliert das Würfeln eines Würfels und aktualisiert den Spielzustand entsprechend der gewürfelten Zahl.
Anschließend aktualisiert die Methode **updateGameState()** den Spielzustand basierend auf der gewürfelten Zahl. Sie bewegt den Spieler auf dem Brett, überprüft, ob das neue Feld spezielle Eigenschaften hat (wie zusätzliche Münzen oder Hindernisse), und aktualisiert dementsprechend die Spielinformationen. Wenn beide Spieler gewürfelt haben, wird ein Minispiel gestartet.
Die Methode **startMiniGame()** wählt dann ein zufälliges Minispiel aus der Liste der verfügbaren Spiele aus und startet es. Wenn keine Minispiele mehr übrig sind oder ein Spieler das Ziel erreicht hat, wird das Spiel beendet.
Die Methode **setGameEnd()** zeigt einen Dialog an, der den Gewinner bekannt gibt und das Spiel beendet.
Die Klasse verwendet eine **Instanz der Drawer Klasse**, um verschiedene Elemente auf dem Spielbrett zu zeichnen, wie z.B. die Spielerbilder und die Würfelbilder.
Die Methode **updateLabels()** aktualisiert die Anzeige der Spielinformationen, wie z.B. die Anzahl der Münzen und die Position der Spieler.

2.5 Board (Maxim, Elena)

Die Board Klasse ist verantwortlich für die Verwaltung und Darstellung des Spielbretts. Die Board Klasse initialisiert das Spielbrett und legt die speziellen Felder fest. Sie verwendet eine 2D-Boolean-**Matrix visited** um zu verfolgen, welche Felder bereits besucht wurden. Sie hat auch **eine Map specialFields** um spezielle Ereignisse (wie das Gewinnen oder Verlieren von Münzen) auf bestimmten Feldern zu speichern. Die **setupBoard() -** Methode baut das Spielbrett auf, indem sie das **initializeBoard() und numberFieldsDFS()** Methoden aufruft. Die **initializeBoard() -** Methode erstellt ein GridPane und fügt Rectangle Objekte hinzu, um die Felder darzustellen. Die **numberFieldsDFS() -** Methode nummeriert die Felder in einer bestimmten Reihenfolge, indem sie eine Tiefensuche (DFS) verwendet. Die **setFieldStateBasedOnColor() -** Methode aktualisiert den Zustand der Felder basierend auf ihrer Farbe. Sie durchläuft alle Rectangle Objekte und setzt den Zustand des entsprechenden Field Objekts entsprechend der Farbe des Rectangle Objekts. Die **getFieldByNumber() -** Methode gibt ein Field Objekt basierend auf seiner Nummer zurück. Die **getFields()** - Methode gibt eine Map aller Field Objekte zurück. Die **getPlayerImageViewByCoordinates()** und **setPlayerImageViewByCoordinates()** Methoden werden verwendet, um die ImageView Objekte zu verwalten, die die Spieler auf dem Spielbrett darstellen. Die **removeField()** - Methode entfernt ein Feld vom Spielbrett. Sie wird verwendet, um Felder zu

entfernen, die nicht Teil des Spielwegs sind. In der Board Klasse wird ein **HashSet** verwendet, um die weißen Felder zu speichern. HashSet ist eine implementierung des Set Interfaces und ermöglicht das Speichern von eindeutigen Elementen in einer ungeordneten Sammlung. Ebenfalls werden mehrere **HashMaps** verwendet, um verschiedene Arten von Daten zu speichern, wie spezielle Felder, Rechtecke, Feldzustände, Felder, ImageViews und SpielerImageViews. HashMap ist eine implementierung des Map Interfaces und ermöglicht das Speichern von Schlüssel-Wert-Paaren.

2.6 Field (Maxim)

Die Field Klasse repräsentiert ein Feld auf dem Spielbrett. Sie speichert verschiedene Informationen über das Feld und bietet Methoden zum Abrufen und Ändern dieser Informationen: Die Field Klasse speichert die **Nummer des Feldes, die x- und y-Koordinaten des Feldes auf dem Spielbrett, den Zustand des Feldes, und ob das Feld einen Spieler enthält**. Die Field Klasse **bietet Getter und Setter Methoden** für alle ihre Attribute. Diese Methoden werden verwendet, um die Informationen des Feldes abzurufen und zu ändern. Die **setHasPlayer()** - Methode wird verwendet, um zu markieren, dass ein Spieler das Feld betreten oder verlassen hat

2.7 Drawer (Max)

Die **Drawer-Klasse** ist verantwortlich für das Zeichnen verschiedener Elemente im Spiel. Mithilfe verschiedener Methoden werden unterschiedliche Elemente auf dem Spielbrett gezeichnet, wie zum Beispiel der Hintergrund des Spiels auf einem gegebenen AnchorPane, das Bild eines Würfels und die Animation eines Würfelwurfs auf einem gegebenen ImageView usw.

2.8 Dice (Max)

Die **Dice-Klasse** repräsentiert einen Würfel im Spiel. Sie hat Methode roll(), die eine zufällige Zahl zwischen 1 und 6 generiert, um das Würfeln eines Würfels zu simulieren. Die Klasse ist mit der Methode **würfeln()** verknüpft und diese ruft Drawer-animation auf. Die Methode benutzt Thread für Würfeldauer.

2.9 Player (Maxim)

Die **Player-Klasse** repräsentiert einen Spieler im Spiel. Sie speichert verschiedene Informationen über den Spieler und bietet Methoden zum Abrufen und Ändern dieser Informationen. Die Klasse speichert den Namen des Spielers, die Position des Spielers auf dem Spielbrett, ob der Spieler ein Computer ist und das Bild des Spielers.
Die Player Klasse bietet **Getter und Setter Methoden** für alle ihre Attribute. Diese Methoden werden verwendet, um die Informationen des Spielers abzurufen und zu ändern. Die **move()-Methode** wird verwendet, um die Position des Spielers auf dem Spielbrett zu ändern, basierend auf der gewürfelten Zahl.

3. Minigames

3.1. Minigame 1 | TicTacToe (Robert)

Bei diesem Minispiel wurde das allgemein bekannte Spiel TicTacToe im Super Mario Stil implementiert. Genauso wie bei allen Minispielen in diesem Projekt, wird auch dieses über bestimmte Felder im Hauptspiel aufgerufen. Bei der Entwicklung wurde besonderer Wert auf die Objektorientierung sowie auf die Verwendung von aus der Vorlesung bekannten Konzepten gelegt. So wurden logische Objekte in eigene Klassen verlagert und so die Struktur/ Lesbarkeit verbessert. Ebenfalls ist der Code so auch durch andere Entwickler schnell durchschaubar und gegebenenfalls erweiterbar.

Das Ziel des Spiels ist es, auf dem 3x3 Spielfeld drei Felder vertikal, horizontal oder diagonal für sich zu beanspruchen. Der Spieler, der dies als erstes erreicht, gewinnt, und bekommt Punkte. Sollte das Spiel enden, ohne das ein Spieler gewinnt, so handelt es sich um ein Unentschieden und es werden keine Punkte verteilt. Der Menschliche Spieler spielt gegen einen Computerspieler, hinter dem sich der Minimax Algorithmus verbirgt. Hierbei handelt es sich um eine rekursive Funktion, die vom aktuellen Stand aus alle möglichen Ausgängen in der Zukunft „durchprobiert". Wie weit der Algorithmus dabei geht, wird durch die sogenannte Tiefe entschieden. In diesem Fall wurde mit einer Tiefe mit 4 und 5 experimentiert, da das Spiel für den menschlichen Spieler zwar gewinnbar, aber auch nicht zu einfach sein soll. Da bei einer Tiefe von 5 ein Gewinnen des menschlichen Spielers fast unmöglich ist, und es im besten Fall zum unentschieden kommen kann, wurde sich schlussendlich für eine Tiefe von 4 entschieden. Es werden also 4 Züge in der Zukunft ausprobiert.

Die Aufteilung in Klassen sieht folgendermaßen aus: Es gibt eine Klasse „Board", die aus einem zweidimensionalen Array aus 3x3 Feldern besteht, die wiederum Exemplare der Klasse „Field" sind. Jedes Feld hat dabei einen Zustand, der durch das Enum „Fieldstate" zur Verfügung gestellt wird. Es gibt eine draw() Methode, die ein board in die Konsole in einem menschlich leicht lesbaren Format zeichnet. Dies diente dem debugging, kann aber auch anderen Entwicklern helfen oder beim logging unterstützen. Die „Hauptklasse" dieses Minispiels ist der Controller nach dem Model-View-Controller (MVC) Prinzip, der in der Klasse „Game1Controller" liegt. Von diesem wird nicht nur der Spielablauf geregelt, sondern es wird auch die Klasse „ComputerPlayer" aufgerufen, die den MiniMax Algorithmus enthält. Ob ein Spieler gewonnen hat, wird über die ebenfalls vom Controller angesteuerten Klasse „GameEvaluator" entscheiden, die das Spielfeld („board") vertikal, horizontal und diagonal prüft. Schlussendlich gibt es noch die Klasse „Drawer", die dazu dient, das Spielfeld als Graphical User Interface (GUI) anzuzeigen und zu aktualisieren.

3.2. Minigame 2 | Balloon-Platzen (Max)

****Wegen der Lesbarkeit wird, da der Code englisch gehalten wurde, in der Dokumentation einheitlich das englische Wort Balloon statt Ballon verwendet****

Bei dem Spiel „Balloon-Platzen" (UML im Anhang) handelt es sich um ein Schnelligkeitsspiel, wessen inhaltlicher Fokus auf der Arbeit mit Threads liegt, um parallel und unabhängig voneinander Veränderungen an verschiedenen Objekten bzw. Elementen auf der Oberfläche vorzunehmen. Auch die Aspekte der Objektorientierung, wie Vererbung, abstrakte Klassen und Datenkapselung, spielten in diesem Spiel eine maßgebliche Rolle. Neben diesen großen Themen wurden in dem Minispiel auch kleinere Elemente der Vorlesungen wie z.B. Lamda-Expressions, Streams, Algorithmen oder auch Java FX integriert.

In dem Spiel „Balloon-Platzen“ muss der Spieler die Balloons platzen lassen bevor sie die Stacheln am oberen Ende des Fensters erreichen.
Die Balloons erscheinen dabei nach und nach und bewegen sich dann auf die Stacheln hinzu.
Die „BalloonGame“ Klasse ist dabei der Hauptcontroller für das Balloonspiel. Sie erbt von der abstrakten „GameController“ Klasse und implementiert das Initializable Interface. Sie verwaltet den Spielzustand und die Spiellogik.
Durch Klick auf den „startButton“ wird die „createBalloons()“ Methode aus der „IntializeBalloons“ Klasse aufgerufen und eine bestimme Anzahl von Balloon-Objekten(aus der Klasse Balloon) initialisiert. Diese bekommen bei Initialisierung eine zufällige Position am Boden des Fensters, eine zufällige Bewegungsgeschwindigkeit und eine zufällige Erscheinungsgeschwindigkeit zugewiesen.
Jetzt soll durch die Methode „sortBalloonsByDeploySpeed()“ mithilfe des Bubble-Sort-Algorithmus die verschiedenen Balloon-Objekte aus der ConcurrentLinkedQueue absteigend geordnet werden um ein mit der Zeit immer schwieriger werdendes Spiel zu erzeugen. Dies gelingt, da nun die Abstände in denen die Balloons “losgeschickt“ immer geringer werden.
Nachdem die Startphase nun abgeschlossen ist, wird die „gameLoop()“ Methode aufgerufen, sie ist der Hauptspielzyklus.
Sie startet einen neuen Thread, der durch alle erstellten Balloons iteriert. Für jeden Balloon wird ein weiterer Thread erstellt, der die Bewegung und das Zeichnen des Balloons auf dem Canvas steuert, solange der Balloon nicht geplatzt ist und das Spiel nicht beendet wurde.
Für das Ändern des Canvas und Einfügen der verschiedenen Bilder und Hintergründe wird die „Drawer“ Klasse verwendet. In dem „gameLoop()“ wird dabei nach jeder Bewegung eines Balloons das gesamte Canvas neugezeichnet, da ansonsten unschöne Überlappungen der Balloons entstehen.
Wenn ein Balloon die Oberseite des Canvas erreicht, hat der Spieler verloren, alle Balloons werden aus dem Spiel entfernt, das Spiel wird beendet und die Anzahl der verbleibenden Balloons wird aktualisiert.
Die „gameLoop“ Methode ist sehr komplex und verwendet mehrere Threads, um die Bewegung und Logik für jeden Balloon gleichzeitig zu verarbeiten. Dies ermöglicht es, dass mehrere Balloons gleichzeitig auf dem Bildschirm bewegt und aktualisiert werden können, ohne dass ihre Geschwindigkeit verändert wird, wenn zum Beispiel alle Balloons schon in Bewegung gesetzt wurden und der Thread nicht mehr für „deploy_speed“ ms schläft.
Das Spiel sollte, damit es nicht zu einfach wird, mit einem Trackpad und keiner Maus gespielt werden.

3.3. Minigame 3 | Hangman (Federico)

Hangman, entwickelt als Teil des größeren Mario Party-Projekts, ist ein Minispiel, das in Java unter Verwendung von JavaFX für die grafische Benutzeroberfläche umgesetzt wurde. Das Ziel des Spiels besteht darin, dass der Spieler ein geheimes Wort errät, indem Buchstaben einzeln geraten werden. Zu den in den Vorlesungen behandelten und im Spiel umgesetzten Themen gehören u.a.: Objektorientierung, ArrayLists, Lesen von Dateien über BufferedReader, Exceptions und GUI-Programmierung mit JavaFX.
Der Game3Controller ist das zentrale Element des Hangman-Spiels im Mario Party-Projekt, welches die Interaktion zwischen der Logik und der Benutzeroberfläche steuert. Dieser Controller leitet von der GameKontroller-Klasse ab und integriert spezifische Funktionen für Hangman. Im Konstruktor des Controllers wird das geheime Wort aus einer externen Datei (Hangman.txt) geladen und zufällig

ausgewählt. Das Wort wird im Spielinterface durch Unterstriche repräsentiert, wobei der erste und letzte Buchstabe zu Spielbeginn sichtbar werden, um den Spielern Hinweise zu geben. Die Methode handleGuess() verarbeitet die Buchstabeneingaben der Spieler. Sie prüft, ob die Eingabe gültig ist und ob der Buchstabe bereits versucht wurde. Je nachdem, ob der Buchstabe im Wort enthalten ist, wird das angezeigte Wort aktualisiert oder die Anzahl der verbleibenden Versuche reduziert. Die Methode isGameOver() überprüft, ob das Spiel beendet ist, sei es durch das vollständige Erraten des Wortes oder das Aufbrauchen aller Versuche. Bei Spielende gibt sie eine entsprechende Rückmeldung an den Spieler. Um die Spielinformationen aktuell zu halten, aktualisiert die Methode updateDisplay() kontinuierlich die Anzeige des geratenen Wortes, die Liste der verwendeten Buchstaben und die verbleibenden Versuche. Die Methode getSecretWord() wählt zufällig ein neues Wort für jede Spielrunde aus.
Die Hangman Klasse stellt,die grafische Benutzeroberfläche für das Spiel bereit. Sie lädt das FXML-Dokument, das die Struktur und das Layout der Benutzeroberfläche definiert, und zeigt es auf dem Bildschirm an. Die Anwendung setzt das Fenster mit dem Titel "Hangman Game" und sorgt dafür, dass alle Spielkomponenten korrekt angezeigt werden.
Das Haupt-UI des Spiels, definiert in der FXML-Datei game3-view.fxml , bietet eine grafische Benutzeroberfläche, auf der der Spieler Eingaben machen kann. Das Layout beinhaltet Textfelder für die Eingabe der Buchstaben und Labels, die den aktuellen Spielstatus anzeigen, wie geratene Buchstaben, verbleibende Versuche und Feedback zum Spielverlauf. Das Design der Benutzeroberfläche zielt darauf ab, dem Spieler visuelles Feedback zu geben, was durch den Einsatz von Stil- und Layout-Definitionen in FXML unterstützt wird.Die Interaktion des Spielers mit dem Spiel erfolgt über das Textfeld und einen Button, der das Rateereignis auslöst. Die Game3Controller Klasse verarbeitet diese Eingaben und aktualisiert den Spielzustand entsprechend.

3.4. Minigame 4 | Snake (Jonas)

In diesem Minispiel wurden einige Inhalte der Vorlesung berücksichtigt. Darunter Lambda Ausdrücke, Threads, Anonyme Klassen, Zeitmessung mit System current time millies, Streams usw.

Folgende Klassen wurden implementiert:

Snake

Jede Schlange der Snake Klasse besitzt unter anderem eine Liste von Körperteilen und die Startrichtung „d“ (rechts). Die Richtung bewirkt, dass die Schlange nicht nach links starten kann, da der Eventhandler in der Game4Controller Klasse, der die Tasten „WASD“ abhört, nur reagiert, wenn nicht die entgegengesetzte Richtung angegeben wird. Die Schlange soll sich schließlich nicht um 180 Grad drehen und in sich sein. Beim Erstellen einer Schlangeninstanz werden die ersten drei Körperteile erstellt und in die Liste gespeichert. Die Schlange hat einige Methoden. Eine Körperverlängerung, die ein neues Körperteil zur Liste hinzufügt. Vier „Move“-Methoden, die abhängig der Richtung den Schlangenkopf um eine Koordinate verschieben. Und einen Körperteilverschieber, der den Körper hinter dem Kopf herzieht. Dieser Methode werden die Koordinaten der „Move“-Methoden übergeben, welche die neue Position des Kopfes festlegen, dann geht eine Schleife über den restlichen Körper und setzt Körperteil 1 auf die Position des alten Kopfes. Körperteil 1 wird dann zum „alten Kopf“ und die Schleife wiederholt sich. Außerdem gibt es eine Methode, die checkt, ob Schlange sich selbst oder die Gameboarder berührt, wodurch das Spiel

verloren wäre und man sich nicht mehr bewegen kann. Da jedes Minigame unseres Mario Partys auch mehrmals aufgerufen werden kann, kümmert sich eine Reset Methode um die Zurücksetzung aller Spielfortschritte auf den Startzustand.

Die Schlange sollte im Mario Design erscheinen, weswegen ein benutzerdefiniertes Image von Wiggler der Raupe aus dem Mario Universum den Kopf und den Körper der Schlange darstellt. Die Images wurden auf die 50x50 Größe eines Schlangenelements zugeschnitten.

Fruit

Eine Frucht hat standardmäßig eine zufällige X- und Y-Koordinate, wodurch die Frucht beim Erstellen der Instanz immer auf einem zufälligen Feld erscheint. Die einzig interessante Methode dieser Klasse ist die „move"-Methode. Hier wird ein neues zufälliges Feld, innerhalb einer do-while-Schleife, für die Frucht ausgewählt, wenn sich Schlange und Frucht berühren. Die „while"- Bedingung der Schleife ist, dass die neue Fruchtposition nicht innerhalb der Schlange liegt.

Um auch hier im Mario Theme zu bleiben, ist ein benutzerdefiniertes „Pilz"-Image für die Frucht implementiert.

BackgroundBoard

Um auch einen Mario Hintergrund zu implementieren, habe ich eine Background Klasse erstellt. Diese Klasse wird initialisiert und erhält lediglich ein Anchorpane und ein Image mit Image View.

Sowohl Snake und Fruit als auch BackgroundBoard wurden mit einer „Draw" Methode auf ein Pane (je ein Pane pro Snake und Fruit), bzw Anchorpane (Backgroundboard) gezeichnet. Für Snake und Fruit wurden 2 identisches Panes übereinandergelegt. Das hat den Grund, dass das Snakepane jeden Gametick neu gezeichnet wird, die Frucht aber nicht neu gezeichnet werden soll, diese soll erst bei einer Berührung, also einem einmaligen Feldwechsel, neu gezeichnet werden.

Computerplayer

Weil Mario Party gegen andere Spieler gespielt wird und Snake ein Solo-Spiel ist, wurde eine Gegenspieler Klasse definiert. Diese Klasse erhält beim Instanziieren einen Wert „reachedLength", welcher die erreichte Länge der Schlange des Gegenspielers repräsentiert. Somit kann mein Spiel deutlich gewonnen oder verloren werden, denn das Ziel ist es Länger als der Gegner zu werden.

Game4Controller

Diese Klasse koordiniert das Minispiel und die einzelnen Klassen untereinander. Zuerst werden die oben beschriebenen Klassen instanziiert. Damit ein dynamisches Spiel entsteht arbeite ich mit einem Animationtimer. Dieser führt alle 125 Millisekunden ein Gametick aus, wodurch die Methode zur Bewegung der Schlange aufgerufen wird und diese darauf hin neu gezeichnet wird. Außerdem wird isGameOver() der Schlange ausgeführt. Um dem Spieler die Möglichkeit zu geben, die Schlange zu steuern ist eine längere Abfolge von Methoden notwendig. Es horcht dauerhaft ein Eventhandler die Tastatureingaben ab und je nach Eingabe (W-A-S-D) wird die zugehörige Direction der Schlange, per setter auf true gesetzt. Diese Richtungsvariablen erscheinen wie ein Umweg, sind aber notwendig, um eine kontinuierliche Bewegung zu ermöglichen. Denn würde durch den Eventhandler direkt die „Move"-Methode aufgerufen werden, würde sich die Schlange pro Tastenanschlag nur um ein Feld bewegen. Durch die Variablen der Richtungen, wird die Schlangen Methode updateSnakeMovement() aufgerufen, welche jetzt pro Gametick die „Move"-Methoden der

Schlangen Klasse aufruft, wodurch dann die Bodyparts neu gesetzt und gezeichnet werden. Für Spieler Informationen, updaten verschiedene Textboxen, die z.B. anzeigen, wie lang die Schlange ist und welche Länge erreicht werden muss oder ob man gewonnen, bzw. verloren hat. Falls das Spiel nochmal aufgerufen wird, muss alles resetet werden, um nicht beim alten Stand weiter zu spielen, weswegen eine Reset Methode die Schlange und Textfelder zurücksetzt.

3.5 Minigame5 | Sternenschlacht – Die Suche hinter den Wolken (Elena)

"Sternenschlacht" ist eine faszinierende Variation des traditionellen Spiels "Schiffe versenken", die in einem düsteren Szenario stattfindet. Der Gegner ist ein finsterer Dunkelheitskrieger, der seine Sternengruppen hinter den undurchdringlichen Wolken verborgen hat.

Das Ziel ist es, die versteckten Sterne zu finden, indem man geschickt die Wolken entfernt und nachsieht, ob sich dahinter eine Sternengruppe verbirgt. Gleichzeitig muss man schneller als der Computerplayer sein, der auch Sternengruppen entdeckt.

Die Methode **initialize()** der Game5Controller Klasse ist eine Initialisierungsmethode, die beim Start des Spiels aufgerufen wird. Zwei neue Spielbretter werden erstellt und das Spielbrett wird dem Computergegner zugewiesen. Schiffe für den Spieler und den Computer werden ebenfalls erstellt. Die Methode **createShips()** wird aufgerufen, um eine Liste von Schiffen zu erstellen. Diese Methode arbeitet mit der Methode **createShip()** zusammen. Die beiden Methoden erzeugen eine Liste von 5 Schiffen mit zufälligen Ausrichtungen und Längen von 1 bis 5. Die Gitterfelder werden mit den erstellten Spielbrettern und Schiffen initialisiert. Die Methode **initializeGrid()** wird aufgerufen, um das Gitter mit den Schiffen zu füllen. Ein Bild wird als Hintergrundbild für das Spielfeld gesetzt und schließlich wird die Anzahl der Schiffe für den Spieler und den Computer aktualisiert, indem die Methode **updateShipCount()** aufgerufen wird. Diese Methode aktualisiert die Anzeige der verbleibenden Schiffe für beide Spieler.

Um das Gitter mit den Schiffen zu füllen, wird die Methode **initializeGrid()** aufgerufen. Sie initialisiert das Spielfeld mit den gegebenen Schiffen und dem Spielbrett. Zuerst werden die Schiffe auf dem Spielbrett platziert. Für jedes Schiff in der Liste der Schiffe wird die Methode **placeShipRandomly()** aufgerufen, die das Schiff zusammen mit der Methode **placeShip()** der Board Klasse an einer zufälligen Position auf dem Spielbrett platziert. Diese Methode arbeitet mit der Methode **canPlaceShipAtPosition()** zusammen. Die beiden Methoden sorgen dafür, dass geeignete Position gefunden wird und, dass Schiff an dieser Position platziert wird. Dann wird über das 10x10 Gitter iteriert. Für jede Zelle im Gitter: Ein neues Canvas-Objekt wird erstellt und dem entsprechenden Array hinzugefügt. Wenn showShips wahr zurückgint und ein Schiff an der aktuellen Position ist, was durch die Methode **isOccupied()** der Board Klasse überprüft wird, wird ein Bild eines gelben Sterns geladen. Andernfalls wird ein Bild einer Wolke geladen. Ein Mausklick-Event-Handler wird zum Canvas hinzugefügt. Wenn auf das Canvas geklickt wird, wird überprüft, ob an der geklickten Position ein Schiff vorhanden ist. Wenn ja, wird das Bild auf dem Canvas zu einem roten Stern geändert und die **hit()** Methode der Klasse Schiff wird aufgerufen. Wenn nicht, dann wird das Bild entfernt. Danach wird der Computer seinen Zug machen. Die Methode h**it()** markiert Schiff als betroffen und die Anzahl der Treffer, die das Schiff erhalten hat, wird um eins erhöht. Anschließend wird überprüft, ob das Schiff gesunken ist, indem die Methode **isSunk()** aufgerufen wird. Diese Methode gibt true zurück, wenn die Anzahl der Treffer gleich der Länge des Schiffes ist, was bedeutet, dass alle Teile des Schiffes getroffen wurden und das Schiff somit gesunken ist. Wenn das

Schiff gesunken ist, wird die Methode **shipSunk()** der Klasse Game5Controller aufgerufen. Diese Methode aktualisiert das Spiel, um den Untergang des Schiffes zu berücksichtigen. Sie entfernt das gesunkene Schiff aus der Liste der Schiffe des Spielers oder des Computers. Danach aktualisiert sie die Anzahl der verbleibenden Schiffe und überprüft durch Aufruf der Methode **checkGameOver(),** ob das Spiel vorbei ist. Diese Methode ruft die Methode **allShipsSunk()** auf, um zu überprüfen, ob alle Schiffe eines Spielers gesunken sind. Wenn alle Schiffe eines Spielers gesunken sind, wird die Methode **showGameOverMessage**() aufgerufen, die eine Nachricht anzeigt, dass das Spiel vorbei ist und wer gewonnen hat.

Der Computergegner funktioniert nach "Paritäts-Schussstrategie". Die ComputerPlayer Klasse repräsentiert den Computer-Spieler. Sie enthält Logik für das Auswählen des nächsten Schusses und das Verfolgen der bereits besuchten Felder je nach festgestellter Ausrichtung. Der Konstruktor **ComputerPlayer()** initialisiert die Klasse mit dem Spielbrett des Gegners. Es initialisiert auch eine Menge von besuchten Feldern, einen Zufallsgenerator und Variablen, um den letzten Treffer und die Ausrichtung des letzten getroffenen Schiffes zu verfolgen. Die Methode **takeTurn()** wird aufgerufen, wenn der Computer an der Reihe ist. Sie wählt die Koordinaten des nächsten Schusses aus. Wenn der letzte Schuss ein Treffer war, versucht sie, in derselben Richtung weiter zu schießen, je nachdem ob das Schiff vertikal oder horizontal ausgerichtet ist. Wenn das Schiff horizontal ausgerichtet ist und getroffen wurde, dann schießt der Computer weiter nach rechts, wenn vertikal, dann weiter nach unten. Dabei wird die mithilfe der Methode **isFieldVisited()** überprüft, ob ein bestimmtes Feld bereits besucht wurde und mithilfe der Methode **isMiss()** überprüft, ob ein bestimmter Schuss ein Fehlschuss ist. Die Methode **markHit()** wird aufgerufen, um einen Treffer zu markieren. Sie aktualisiert die Koordinaten des letzten Treffers und die Ausrichtung des getroffenen Schiffes. Wenn kein vorheriger Treffer oder keine Suchrichtung gefunden wurde, schießt sie zufällig. Nachdem die Koordinaten ausgewählt wurden, überprüft sie, ob der Schuss ein Treffer ist, und markiert den Treffer, wenn dies der Fall ist.

Bei der Programmierung des Spiels wurden mehrere Themen der Programmiervorlesung berücksichtigt. Die berücksichtigten Themen sind: Vererbung; Objektorientierung; Modifier; Listen; Mengen; Lambdaausdruck, der als Event-Handler für Mausklick Ereignisse auf dem Canvas-Objekt dient.

3.6 Minigame6 | Minesweeper (Maxim)

Minesweeper ist ein klassisches Rätselspiel, bei dem der Spieler versuchen muss, alle Felder ohne Minen aufzudecken. Dabei kann der Spieler mit Linksklick die Felder öffnen und sieht dann in welchem Radius um dieses Feld eine Bombe ist. Alternativ werden mehrere Felder geöffnet. Falls der Spieler ein Feld mit einer Bombe öffnet, endet das Spiel. Ebenso endet es, wenn er zu viele und falsche Markierungen setzt. Ziel des Spieles ist es alle Bomben zu markieren und alle leeren Felder zu öffnen. Das Spiel wurde mit den folgenden Klassen implementiert:

Der Game6Controller fungiert als das Gehirn hinter dem Spiel. Es ist verantwortlich für die Koordination aller wichtigen Aspekte des Spiels 6, von der Initialisierung bis zum Ende. Wenn das Spiel gestartet wird, ruft der Game6Controller die Methode initialize() auf, die eine Instanz von MinesWeeperApp erstellt und das Spiel auf dem Schwierigkeitsgrad "easy" beginnt. Sobald das Spiel endet, sei es durch einen Sieg oder eine Niederlage, setzt der Game6Controller den Spielstatus entsprechend mit der Methode setGameOver(boolean gameOver, String message) und aktualisiert

die Anzeige für den Spieler. Darüber hinaus wird der Controller auch eine neue Partie starten, wenn der Spieler dies wünscht, durch Aufruf von newGame().

Das Board repräsentiert das Spielfeld selbst. Hier wird die eigentliche Spiellogik implementiert. Wenn das Spiel beginnt, ruft das Board die Methode startGame() auf, die das Spielbrett initialisiert und die Bomben platziert. Es überwacht auch fortlaufend den Spielstatus, um zu überprüfen, ob der Spieler das Spiel gewonnen hat, indem es alle nicht bombenhaltigen Felder aufdeckt. Sobald das Spiel vorbei ist, sendet das Board entsprechende Signale an den Game6Controller, der dann den Spielstatus aktualisiert und das Ergebnis anzeigt.

Die GameTimer-Klasse implementiert einen Timer als Thread, der die verstrichene Spielzeit anzeigt. Sie startet den Timer, wenn das Spiel beginnt, und stoppt ihn, wenn das Spiel gewonnen oder verloren wird.

Die Tile-Klasse repräsentiert ein einzelnes Feld auf dem Spielbrett. Jedes Feld kann entweder eine Mine enthalten oder eine Zahl, die die Anzahl der angrenzenden Minen anzeigt. Die Klasse verwaltet den Zustand des Feldes (aufgedeckt, markiert, Bombe) und stellt Methoden zum Aufdecken und Markieren bereit.

Diese beiden Komponenten, der Game6Controller und das Board, sind eng miteinander verbunden. Der Game6Controller fungiert als Regisseur, der die Aktionen des Spielers überwacht und die nötigen Anweisungen an das Board weitergibt, um das Spiel entsprechend zu aktualisieren. Das Board auf der anderen Seite ist das Herzstück des Spiels, das die Spiellogik implementiert und den Spielstatus verwaltet, während es mit dem Game6Controller kommuniziert, um den Spielverlauf zu steuern.

Zusammen bilden der Game6Controller und das Board das Rückgrat des Spiels 6, das die Grundlage für ein reibungsloses Spielerlebnis bildet. Ohne diese beiden Komponenten würden die anderen Klassen wie der GameTimer oder die Tile-Klasse nicht effektiv funktionieren können, da sie stark von der zentralen Spiellogik abhängen, die der Game6Controller und das Board bereitstellen.

4. Anhang

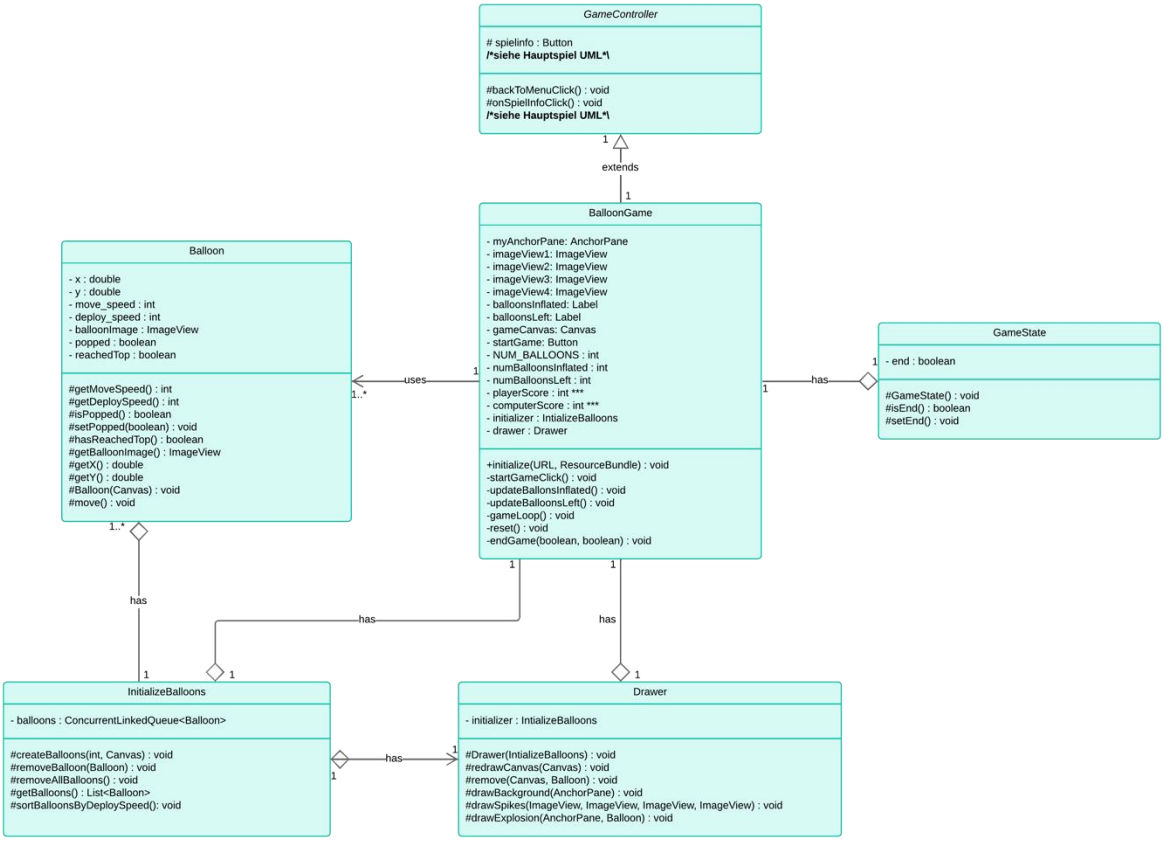

Abb1.: Klassendiagram Balloon-Platzen

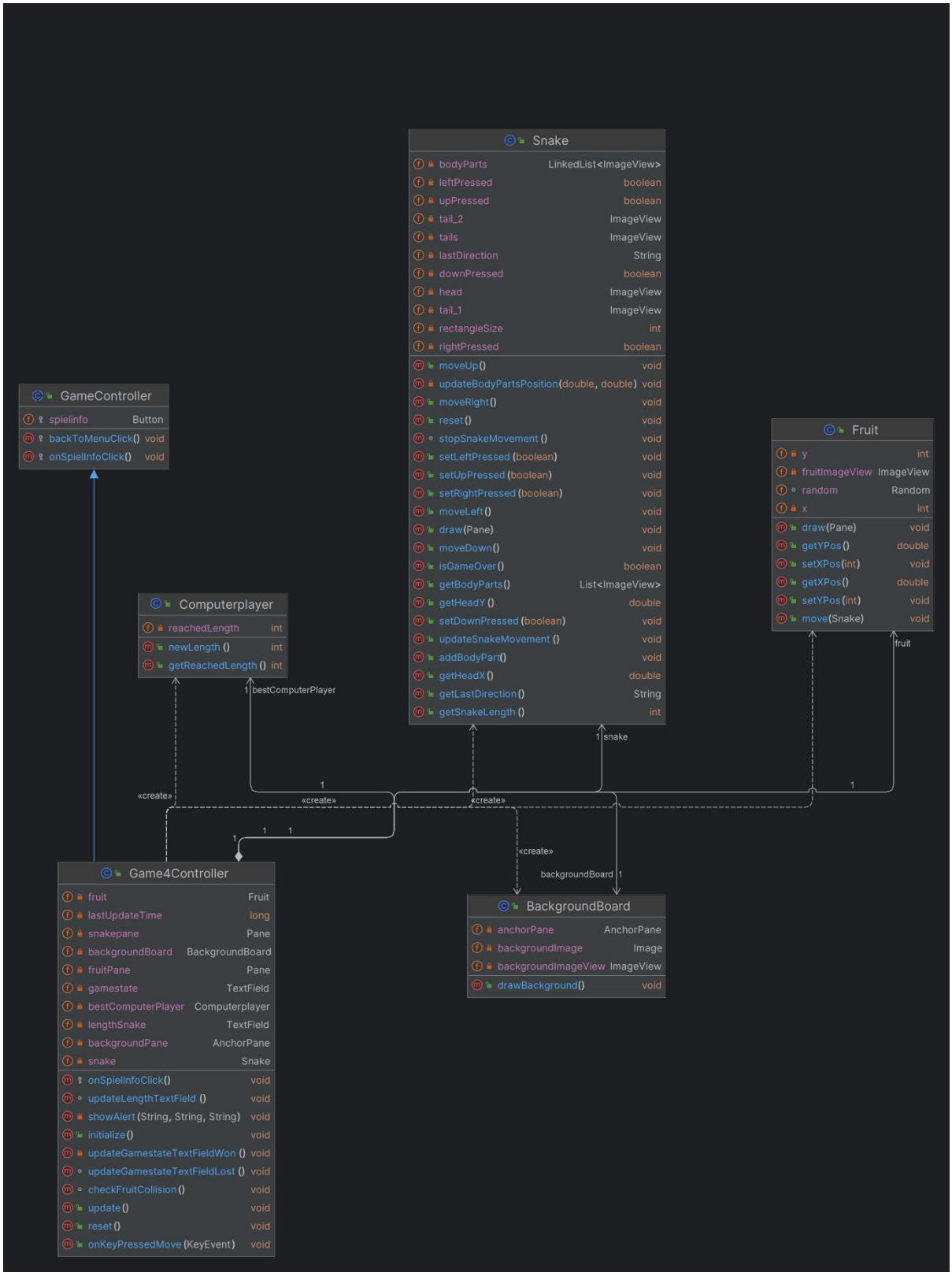

Abb.2: **Klassendiagram Snake**

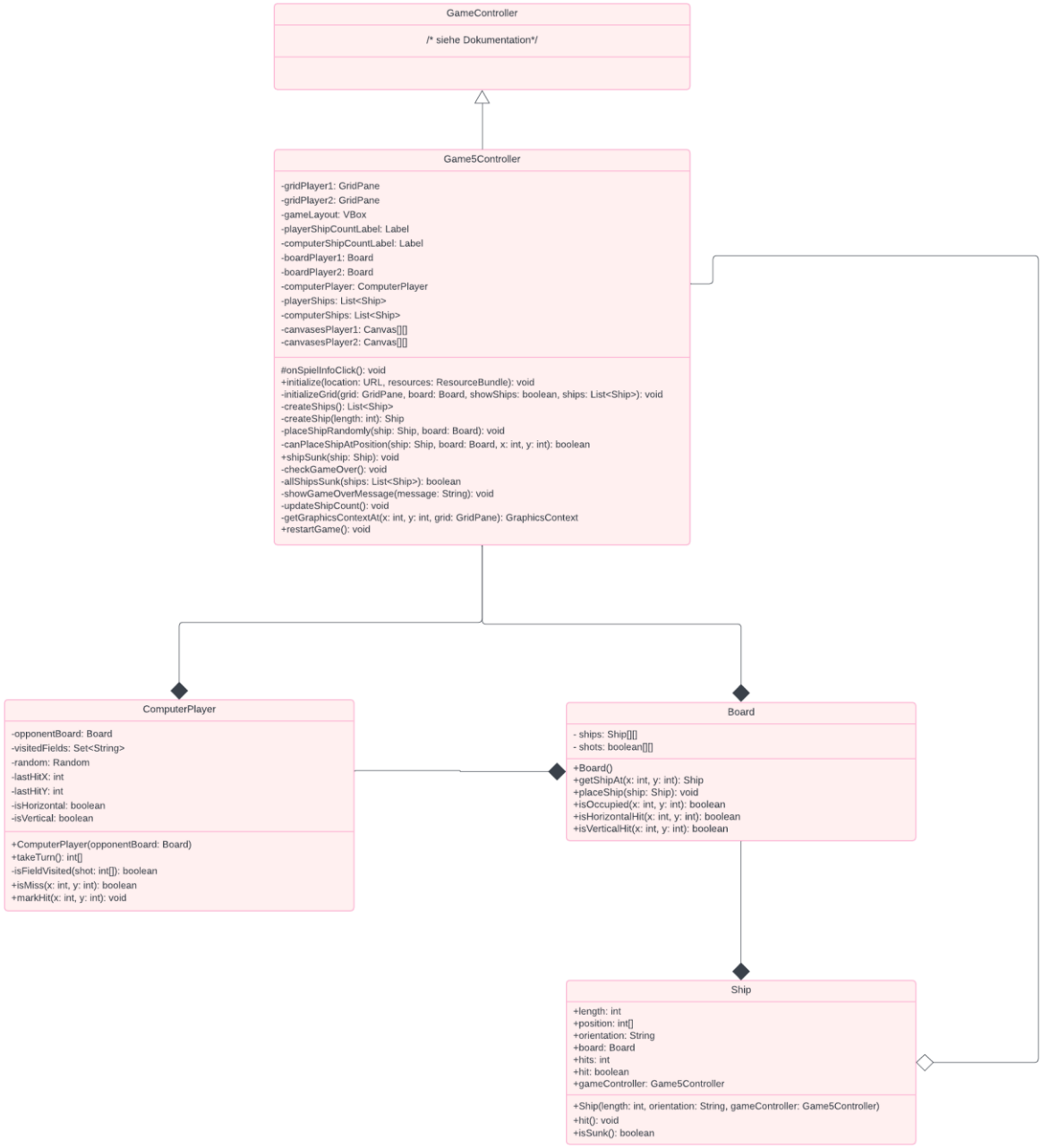

Abb.3: Klassendiagram Sternenschlacht

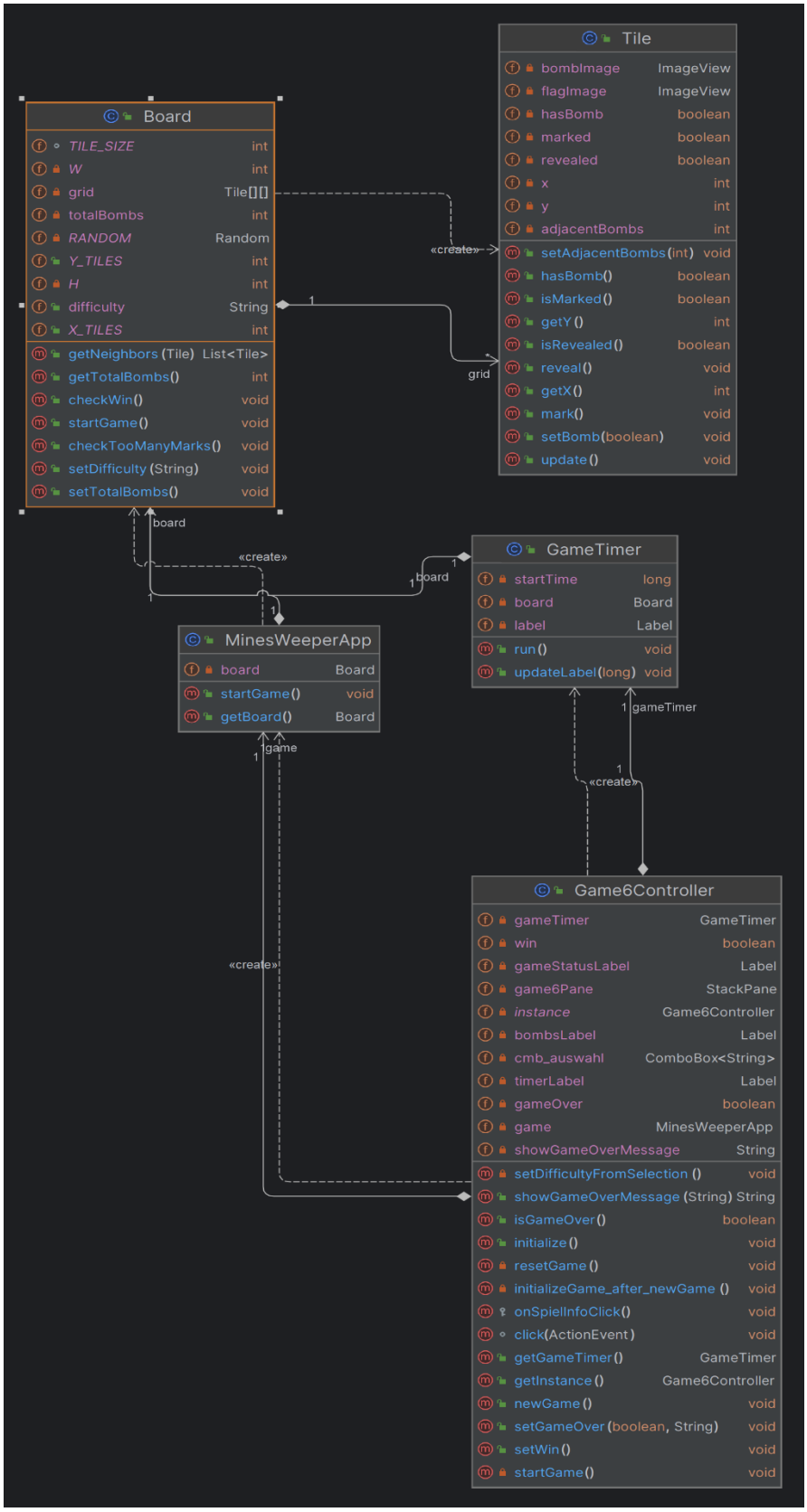

Abb.4: Klassendiagramm Minesweeper

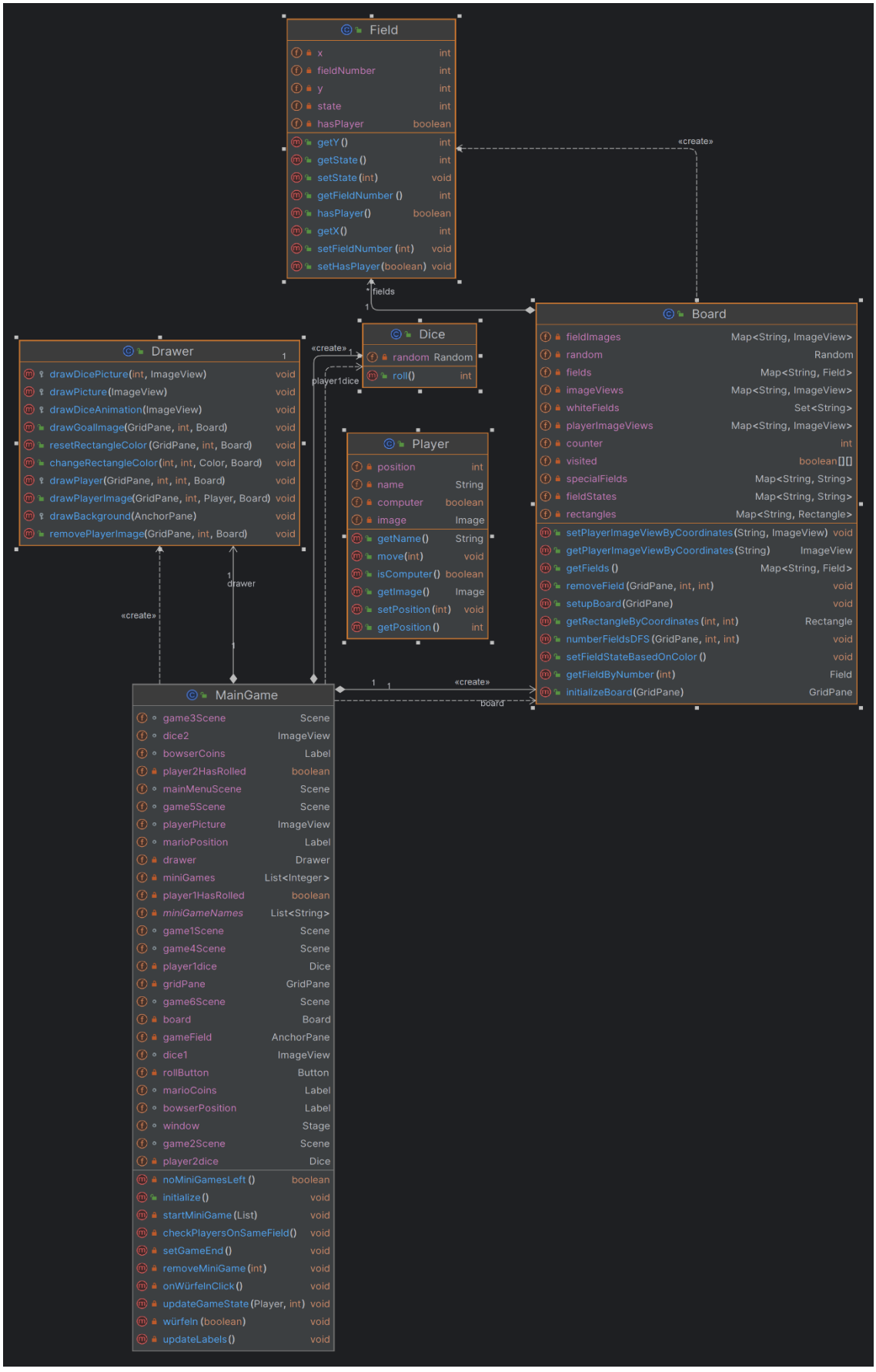

Abb.5: Klassendiagramm Maingame