Neue Spiel Implementieren

Memory

Erstellt von
Lehkdup Shöntsang

Wie man das Spiel herunterladen kann

Um das Spiel herunterzuladen, besuchen Sie bitte das entsprechende GitHub-Repository, in dem der Code bereitgestellt ist, und klonen Sie dieses lokal auf Ihren Rechner. Anschliessend können Sie das Spiel starten, indem Sie in Ihrer Entwicklungsumgebung oder über die Kommandozeile die Klasse **AigsServerApplication.java** ausführen. Dadurch wird der Server hochgefahren und das Spiel ist über den entsprechenden Port erreichbar (50005).

Repo: https://github.com/Lehkdup/AIMemoryGame

1. Einleitung

Dieses Dokument beschreibt den Aufbau (Design) und die wichtigsten Konzepte der gezeigten Java-Klassen für das „Memory“-Spiel.

2. Überblick über die Hauptkomponenten

Das System besteht aus folgenden Hauptbestandteilen:

1. **Game (Entity-Klasse)**
    - Repräsentiert den Zustand eines laufenden oder bereits beendeten Spiels.
    - Wird in einer Datenbank-Tabelle namens „games“ gespeichert (durch die JPA-/Hibernate-Annotationen).
    - Speichert u. a. folgende Informationen:
        - **token**: Eindeutige Kennung (String), um das Spiel wiederzuerkennen.
        - **gameType**: Der Typ des Spiels (z. B. Memory), als GameType verwaltet.
        - **difficulty**: Schwierigkeitsgrad als Long (z. B. 1 = leicht, 2 = schwer).
        - **options**: Beliebige weitere Optionen als String.
        - **score** und **scoreAI**: Punktestand des Spielers und der KI.
        - **result**: Spielausgang (true = gewonnen, false = verloren, null = läuft noch).
        - **aiMove**: Gibt an, ob die KI einen Zug durchführen soll oder nicht (z. B. 1 = KI ist am Zug).
        - **cardStates**: Zweidimensionales Array, das den aktuellen Status jeder Karte (verdeckt, aufgedeckt oder bereits gefunden) abbildet.
        - **cardValues**: Zweidimensionales Array, das den „Wert“ der Karte speichert (z. B. 1–10 bei Memory).
        - **knownCards**: Liste bereits bekannter Kartendaten in Form von Maps (für Smart-KI).

2. **MemoryGame (Implementierung von GameEngine)**
    - Zentraler Motor für das Memory-Spiel, implementiert das Interface `GameEngine`.
    - Enthält die Kernlogik:
        - **newGame(Game game)**: Initialisiert ein neues Memory-Spiel, legt Kartendecks an, mischt sie und setzt den Spielzustand zurück.
        - **move(Game game, HashMap<String, String> move)**: Verarbeitet einen Spielzug. Dabei werden Karten aufgedeckt, verglichen und ggf. Punkte vergeben. Im Anschluss kann die KI aufgerufen werden.
        - **isGameFinished(Game game)**: Überprüft, ob das Spiel beendet wurde (z. B. wenn alle Paare gefunden sind oder ein Spieler 10 Punkte erreicht hat).
        - **createShuffledDeck(int numberOfPairs)**: Erstellt und mischt das Kartendeck.
        - Interagiert direkt mit den Feldern aus der `Game`-Klasse (`cardStates`, `cardValues`, `knownCards`, `score`, etc.), um den Fortschritt und Zustand des Spiels zu aktualisieren.
    - Verwaltet Zustandsvariablen für Zwischenschritte.
3. **MemoryAi (Interface) und Implementierungen**
    - Definiert die Methode `makeMove(Game game, MemoryGame engine)`, über die eine KI einen Zug ausführt.
    - **RandomMemoryAi**: Eine einfache KI, die zufällig verdeckte Karten aufdeckt.
    - **SmartMemoryAi**: Eine fortgeschrittene KI, die sich bereits aufgedeckte Karten merkt und gezielt Kartenpaare sucht (verwendet das `knownCards`-Feld aus `Game`).

3. Zusammenspiel der Klassen

1. **Spielinitialisierung**
    - Der Client (z. B. ein Controller) ruft `newGame(game)` in der Klasse `MemoryGame` auf.
    - Dort werden die Arrays für Kartenwerte (`cardValues`) und Kartenstatus (`cardStates`) generiert, das Deck gemischt und alle notwendigen Felder (Punkte, Ergebnis etc.) auf ihre Startwerte gesetzt.
    - Dieses konfigurierte `Game`-Objekt wird anschliessend zurückgegeben und kann in der Datenbank persistiert werden.
2. **Verarbeitung von Zügen**
    - Ein Spielzug gelangt als `HashMap<String, String>` ins System (Methodenparameter `move`). Darin sind z. B. die Koordinaten der aufgedeckten Karten enthalten.
    - Die Methode `move(Game game, HashMap<String, String> move)` wertet die Koordinaten aus:

        1. **Kartenstatus prüfen** (ob bereits aufgedeckt oder gefunden).
        2. **Kartenwerte vergleichen**: Wenn zwei Karten übereinstimmen, wird der Status auf „gefunden“ gesetzt und Punkte werden vergeben.
        3. **Fehlversuche**: Werden zwei unterschiedliche Karten aufgedeckt, werden sie wieder verdeckt (Status=0) und die KI oder der nächste Spieler ist am Zug.
    - Am Ende jedes Zugs wird geprüft, ob das Spiel beendet ist (`isGameFinished`). Falls nicht und die `aiMove`-Flag aktiv ist, ruft das System eine KI auf.
3. **KI-Integration**
    - Anhand des `difficulty`-Werts in `Game` wird entschieden, ob `RandomMemoryAi` oder `SmartMemoryAi` verwendet wird.
    - Beide KIs rufen intern wiederum `move(game, moveMap)` auf, allerdings mit dem Parameter `isAi = "true"`.
    - Durch die KI-Zug-Logik werden verdeckte Karten aufgedeckt und ggf. der `knownCards`-Speicher der Smart-KI mit den entdeckten Infos aktualisiert.
4. **Datenbankanbindung**
    - Die Klasse `Game` ist eine JPA-Entity (`@Entity`, `@Table(name = "games")`). Beim Speichern bzw. Laden aktualisiert das Framework automatisch alle Felder (wie `cardStates`, `cardValues`, `score`, etc.).
    - Spezielle Datenstrukturen wie `knownCards` (ein `ArrayList<Map<String, Integer>>`) werden über einen eigenen Converter (`KnownCardsConverter`) in ein Datenbank-taugliches Format übersetzt.

4. Wichtigste Ideen im Code

1. **Trennung von Daten und Spielmechanik**
    - Die Klasse `Game` speichert lediglich den Spielstatus. Die eigentliche Logik (Karten mischen, Züge ausführen, KI-Methoden) steckt in `MemoryGame` (bzw. in den KI-Klassen).
2. **Speicherung und Wiederherstellung des Zustands**
    - Dank JPA/Hibernate kann der gesamte Spielstand (auch komplexe Arrays) in der Datenbank abgelegt werden. Dadurch sind Spielstände persistierbar und lassen sich später wieder fortsetzen.
3. **Erweiterbarkeit durch KI-Strategien**
    - Das KI-Verhalten ist durch das `MemoryAi`-Interface gekapselt und kann problemlos durch weitere Implementierungen (z. B. noch „schlauere“ KI) erweitert werden.
    - Zum Beispiel dass wenn die KI im KnownCards liste, keine Kartenpaare findet welche mal aufgedeckt wurden, führt er aktuell dann ein Zufalls Zug aus. Diesen Teil könnte man noch erweitern das dieser Zug auch strategisch wäre.
4. **Klare Statusdefinition**

- Das Feld cardStates verwendet integer-Codes (0 = verdeckt, 1 = aufgedeckt, 2 = gefunden), was die Logik in move(...) vereinfacht.

5. **Modularer Aufbau**
   - MemoryGame konzentriert sich auf das Standard-Memory-Regelwerk. Änderungen an der KI oder den Spielparametern (z. B. Anzahl Kartenpaare) bleiben davon relativ unabhängig.

5. Zusammenfassung

Das gezeigte System trennt die *Geschäftslogik* (MemoryGame und KI-Klassen) klar von den *Spieldaten* (Game). Für unterschiedliche Schwierigkeitsgrade oder andere Spielregeln lässt sich die Struktur leicht anpassen. Die KI-Klassen demonstrieren, wie das System flexibel KI-Strategien einbinden kann. Durch die Nutzung von JPA/Hibernate ist sichergestellt, dass alle Daten (einschliesslich des KI-Speichers knownCards) dauerhaft in der Datenbank vorgehalten werden können.

Wie man das Spiel testen/ spielen kann

Für das Spiel wurde eine Oberfläche gebaut jedoch kann man es über die Konsole Spielen bzw. Testen.

	Client Anfragen (curl anfragen)	Server Antwort
1	curl --location 'localhost:50005/users/register' \ --header 'Content-Type: application/json' \ --data '{ "userName":"legi", "password":"legi" }'	{ "userName": "legi", "password": "", "userExpiry": "2025-01-12T21:02:21.858382", "token": null }
2	curl --location 'localhost:50005/users/login' \ --header 'Content-Type: application/json' \ --data '{ "userName":"legi", "password":"legi" }'	{ "userName": "legi", "password": "", "userExpiry": "2025-01-12T21:02:56.549202", "token": "cf2c0b2e1ff8c00" }

3	curl --location 'localhost:50005/game/new' \ --header 'Content-Type: application/json' \ --data '{ "token":"cf2c0b2e1ff8c00", "gameType":"MemoryGame", "difficulty" :"2" }'	{ "token": "cf2c0b2e1ff8c00", "gameType": "MemoryGame", "difficulty": 2, "options": null, "score": 0, "scoreAI": 0, "result": false, "aiMove": 0, "gameMessage": "Spiel wurde erstellt", "cardStates": [[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]

		], "cardValues": [[2, 3, 9, 7], [4, 5, 10, 6], [2, 9, 3, 7], [8, 8, 10, 6], [4, 1, 5, 1]], "knownCards": [] }
4	curl --location 'localhost:50005/game/move' \ --header 'Content-Type: application/json' \ --data '{ "token": "cf2c0b2e1ff8c00", "row1": "0", "col1": "3", "row2": "3", "col2": "1",	{ "token": "cf2c0b2e1ff8c00", "gameType": "MemoryGame", "difficulty": 2, "options": null, "score": 0, "scoreAI": 0,

	"isAi": "false" }'	"result": false, "aiMove": 0, "gameMessage": "\n Keine gleichen paare gefunden.\nPosition Player: 0 3 Karte: 7\n position Player: 3 1 Karte: 8\n Keine gleichen paare gefunden.\nPosition AI: 2 2 Karte: 3\n position AI: 0 2 Karte: 9", "cardStates": [[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]], "cardValues": [

```
    [
      2,
      3,
      9,
      7
    ],
    [
      4,
      5,
      10,
      6
    ],
    [
      2,
      9,
      3,
      7
    ],
    [
      8,
      8,
      10,
      6
    ],
    [
      4,
      1,
      5,
      1
    ]
  ],
  "knownCards": [
    {
      "col": 3,
      "row": 0,
      "value": 7
    },
    {
      "col": 1,
      "row": 3,
      "value": 8
    },
    {
      "col": 2,
```

		"row": 2,
		"value": 3
		},
		{
		"col": 2,
		"row": 0,
		"value": 9
		}
		]
		}

Erläuterung

1. **User registrieren.**
2. **User anmelden und Token speichern**, um ein Spiel erstellen zu können.
3. **Neues Spiel erstellen**: Dazu den beim Login erhaltenen Token mitgeben und den Schwierigkeitsgrad angeben (1 = einfach, 2 = schwer).

   Zurück erhält man vom Server:

   - **gameType**: Welches Spiel man ausgewählt/erstellt hat.
   - **difficulty**: Bestätigung des Schwierigkeitsgrads.
   - **options**: *(null – kann ignoriert werden, für spätere Erweiterungen)*
   - **score**: Punktestand des Spielers.
   - **scoreAI**: Punktestand der KI.
   - **result**: *true*, wenn das Spiel beendet ist (z. B. für ein UI).
   - **gameMessage**: Nachrichten darüber, welche Karten aufgedeckt wurden und von wem, sowie weitere spielrelevante Rückmeldungen.
   - **cardStates**: Zeigt an, welche Karten gefunden sind (2) bzw. noch verdeckt (0).
   - **cardValues**: Enthält die Kartenwerte (Lösung) – nützlich für Tests oder ein UI.
   - **knownCards**: „Gedächtnis“ der Smart-KI, speichert bereits aufgedeckte Karten und ihre Positionen (sowohl eigene als auch die des Gegenspielers). Wenn ein Kartenpaar gefunden wurde, wird es aus der Liste entfernt.
4. **Karten aufdecken**: Man übergibt die gewünschten Kartenpositionen und kennzeichnet sich als Spieler. Als Ergebnis erhält man dieselben Informationen wie in Schritt 3, jedoch mit aktualisiertem Spielstand.