Workshop zu HexA, SOLID, Grasp

Ox: Ochse {m} *zool.*

Aufgabenstellung

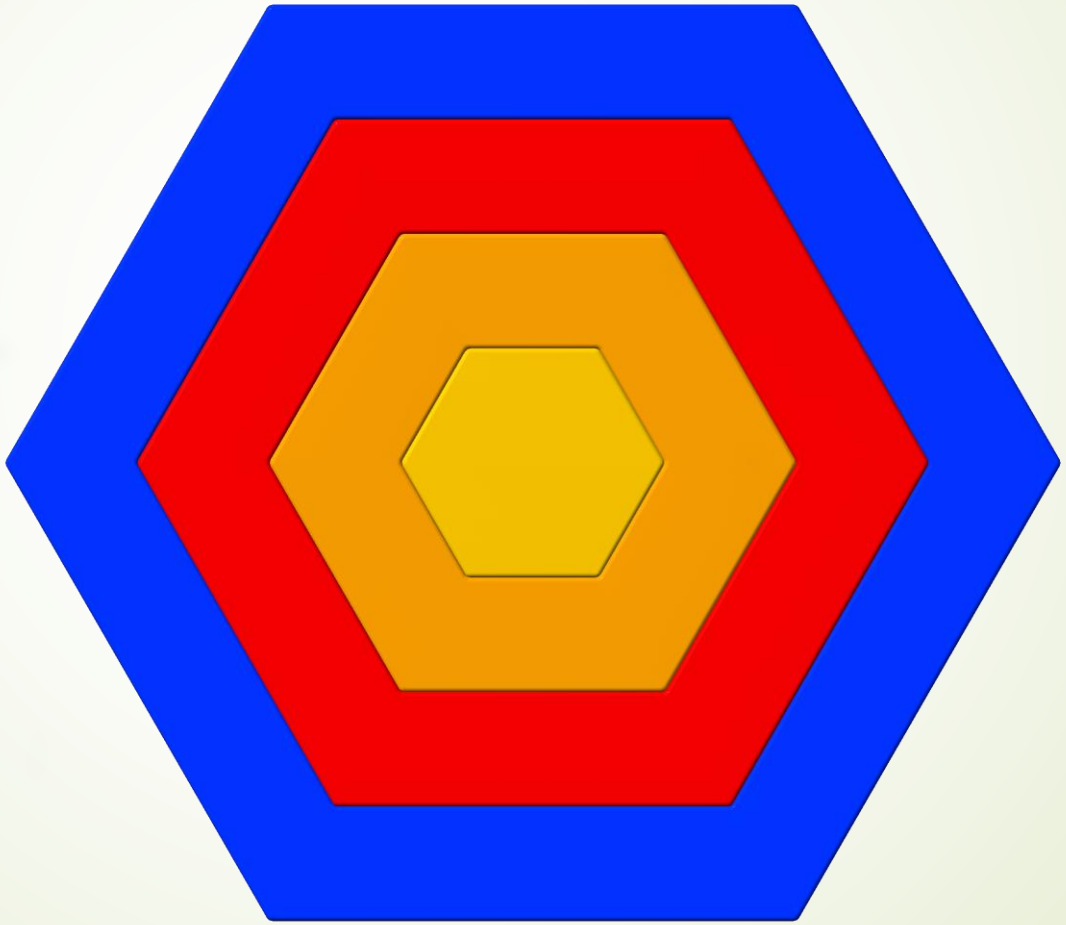
- Implementieren Sie eine Variante des Spiels „Bulls and Cows“
- Spielablauf:
  - Der Computer wählt ein Wort aus einem Wörterbuch und teilt dem Spieler mit, wie viele Buchstaben es enthält
  - Der Spieler rät ein Wort und erhält als für jeden Buchstaben folgende Information
    - Ein + wenn der Buchstabe an der richtigen Stelle vorkommt
    - Ein – wenn der Buchstabe in der Lösung an einer anderen Stelle vorkommt
    - Einen . wenn der Buchstabe nicht in der Lösung vorkommt

```
Guess the word!
The word to guess has 4 characters.
Please enter your guess:

able

Matches:
able
-..+

Please enter your guess:

gate

Matches:
able
-..+
gate
++.+

Please enter your guess:

game

Matches:
able
-..+
gate
++.+
game
++++

You won!
```

Varianten

- **Textbasierte Benutzeroberfläche,** GUI oder Web-basierte Oberfläche
- **Ein Spieler** oder mehrere Spieler
- Menschliche oder automatisierte Spieler (AI)
- Unterschiedlicher **Detailgrad** der Ausgabe

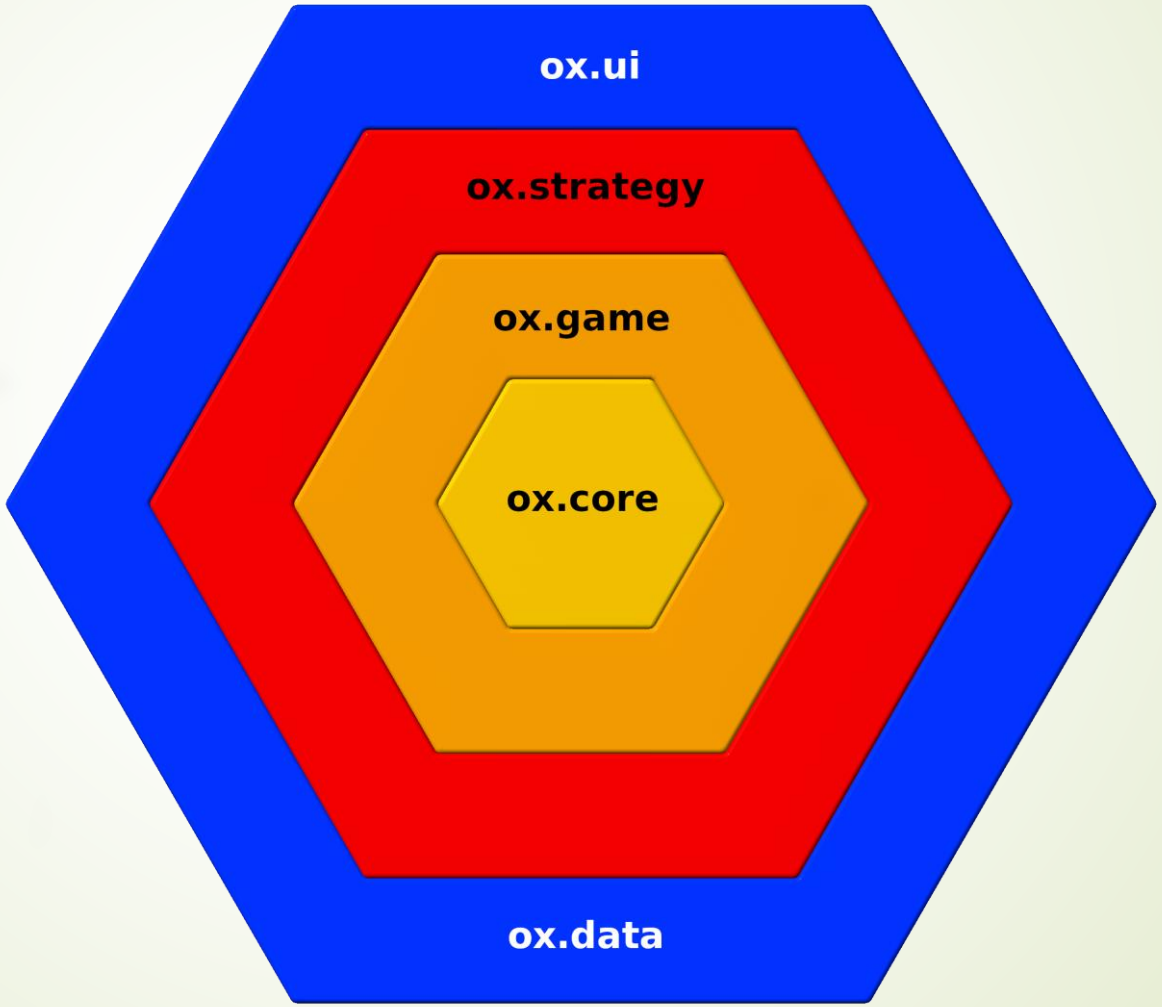
Welche Schichten kann man sich vorstellen?



Mögliche Schichtenarchitektur

- Denkbar sind folgende Schichten (die Java Packages entsprechen)
  - `ox.core` (Feststellen von Übereinstimmung zwischen Wörtern)
  - `ox.game` (Regeln und Ablauf des Spiels)
  - `ox.strategy` (Spieler, manuell und automatisch)
  - `ox.data` (Externe Daten, z.B. Wörterbuch oder Highscore-Liste)
  - `ox.ui` (Benutzerinterface)

Hexagonale Architektur



Implementierung der Core Schicht

- Implementieren Sie Interfaces/Klassen
  - CharacterMatch (Ist/Soll Vergleich zwischen einzelnen Zeichen)
    - char proposedChar()
    - boolean isPerfectMatch()
    - boolean isPartialMatch()
    - char describe() (+ für perfektes Match, - für partielles Match, . sonst)
    - Möglicherweise Unterklassen für die verschiedenen Match-Arten
  - Factory für CharacterMatch
  - Match (Vergleich zwischen Zeichenketten)
    - Enthält Liste von CharacterMatch-Objekten (für jedes Zeichen eines Wortes)
    - Ähnliche Funktionalität

Implementierung der Game-Schicht

- Klasse Game
  - setPlayer()
  - step() (Erlaubt dem Spieler seinen Zug zu machen)
  - proposeSolution(String solution) (Lösungsvorschlag vom Spieler)
  - updateGameState(Match mach)
  - isGameDecided(), wasGameLost(), wasGameWon()
  - getWordToGuess(), getWords()
  - Observer: onNewGame(), onNewMatch(), onGameWon(), onGameLost()
- Enum(?) GameState (NotStarted, InProgress, Won, Lost)
- Player
  - join(Game game)
  - proposeSolutionToGame()
- Dictionary
  - getRandomWord(int minLength, int maxLength)

Die Game Loop

```
Game::step() {
    player.proposeSolutionToGame();
}


<SomePlayer>::proposeSolution() {
    …
    theGame.proposeSolution(…);
}


Game::proposeSolution() {
    <Compute whether we found a match>
    <Notify the observers about the result>
    <Update the game state accordingly>
}
```

Implementierung der Strategy-Schicht

- Abstract Player
  - Enthält die Information, die jeder Player benötigt (Game)
- Manual Player
  - Liest Vorschläge vom Terminal ein,
- Random Player
  - Erzeugt ein zufälliges Wort einer vorgegebenen Länge

Mögliche Anwendungsfälle für Patterns

- Match: Factory
- Spielelogik: Observer
- Spieler: Strategy
- Wörterbuch, UI: Adapter
- (Das sind nicht die einzigen Anwendungsmöglichkeiten)

Automatisierte Spieler

- Einfachste Variante: erzeugt „Zufallswörter“ der vorgegebenen Länge aus den Buchstaben a bis z, ohne sich um die „Matches“ zu kümmern
- Verbesserte Variante: nimmt Wörter der vorgegebenen Länge aus dem Wörterbuch, rät nur Wörter die alle vorhergehenden „Matches“ respektieren
- Viele weitere Varianten, z.B.
  - Optimierte Variante des „Wörterbuch“-Spielers, der Wörter so auswählt, dass die Entropie maximiert wird
  - Spieler basierend auf Reinforcement Learning