Software Design

DI Christian Hartinger

Übung 2

Übung 2

- AssetRepository zu Verwaltung der Texturen (ggf. Partikeleffekte / Töne)
- Spieler kann sich über das Spielfeld bewegen
- Spieleraktionen werden angezeigt

AssetRepository

Problem: Laden von Texturen ist "teuer".

Immer wenn wir `` `new Texture(...);` `` verwenden, wird die Textur von der Festplatte geladen und für den Transfer zur Grafikkarte vorbereitet.

Für Position, Rotation etc. ist der `` `Sprite` `` verantwortlich, der die Textur jedoch nicht verändert.

AssetRepository

Mögliche Lösung: Singleton

Wenn wir eine Textur brauchen wäre es schön, immer die selbe Instanz zu bekommen. Hier bietet sich ein Singleton Pattern in form eines "AssetRepository" an.

Wenn das Repository alle Texturen kennt, kann man auch den Zeitpunkt des Ladens bestimmen. (zB Laden aller Texturen schon beim Laden des Spieles)

AssetRepository

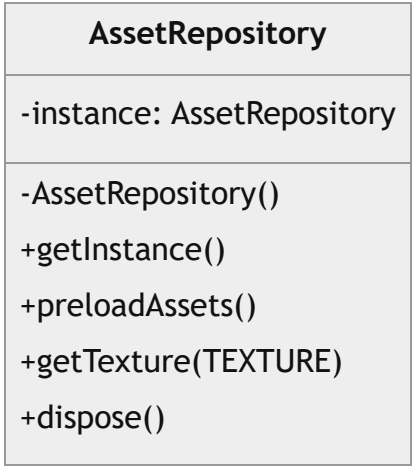
Aufräumarbeiten

Ebenso nicht behandelt haben wir, dass Texturen auch wieder "entladen" werden sollten wenn wir sie nicht mehr brauchen.
(`Texture.dispose()`).

Für die Übung reicht es, in der Methode `main.dispose()` alle Texturen wieder aufzuräumen.

AssetRepository

Beispiel



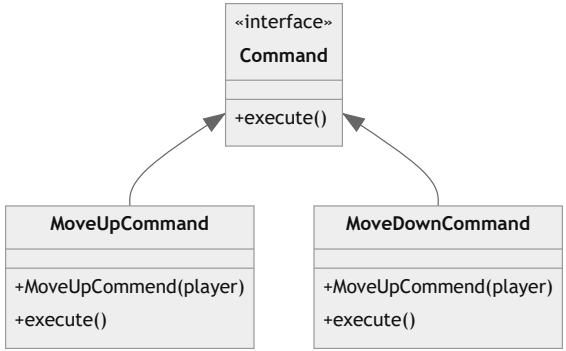
Game Input

Spieler soll sich auch über das Spielfeld bewegen können

- mittels `Command` Pattern (o.Ä.)
- zB vom GameInput ausgeführt - dieser darf nicht wissen wer `execute()` genau behandelt
- (kann in Main sein, kann Spieler selbst sein …)

Game Input

Beispiel



Game Input

Beispiel

```
@Override
public boolean keyDown(int keycode) {
    if(keycode == Keys.UP) {
      this.moveUpCommand.execute()
    }
}
```

Game Input

Überlegungen

- Wo kommt die Logik am besten hin?
- Wie erzeuge ich diese Commands und übergebe sie dem `InputAdapter`?

Spiel-Events

Beobachter & Stalker

Wir wüssten gerne, welche Aktion der Spieler als letztes ausgeführt hat.

Diese Information soll in der Konsole und optional auch im Interface angezeigt werden. Dies soll mit Hilfe von 1 (bzw, 2) `Observer` geschehen.

Spiel-Events

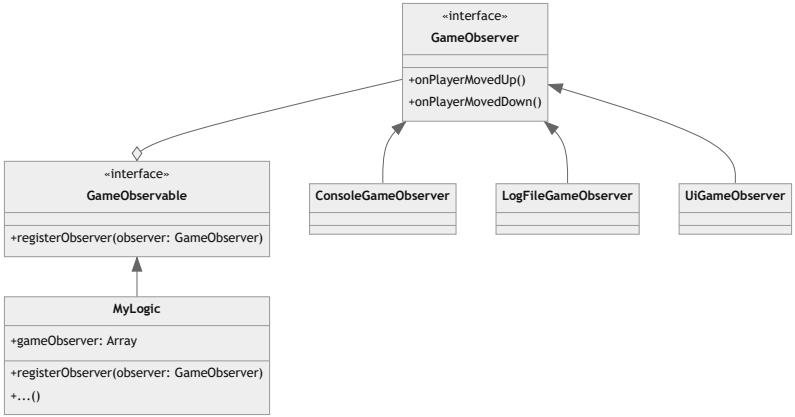
Warum Observer?

Welche Logik auch immer für die Bewegung zuständig ist (Player Klasse, Command, World, Logik, Main) soll nichts darüber wissen wie und ob eine Positionsänderung von einem anderen System weiterverarbeitet wird. Sie soll lediglich "beobachtbar" sein.

(Observer sind in der Regel genau das: Beobachter - sie greifen nicht in die Aktion ein)

Spiel-Events

Bespiel



Observer Bsp 2

```
public class PlayerMovement {
    private List<PositionObserver> observers = new ArrayList<>();

    public void addObserver(PositionObserver obs) {
        this.subscribers.add(obs);
    }

    public void removeObserver(PositionObserver obs) {
        this.observers.remove(obs);
    }

    public void setPosition(int x, int y, int direction) {
        for (PositionObserver obs : this.observers) {
            obs.update(this.positionX, this.positionY, direction);
        }
```

PositionObserver

```
public interface PositionObserver {
    public void update(int x, int y, int direction);
}
```

PositionObserver

```
public class PlayerPositionObserver implements PositionObserver {
    @Override
    public void update(int x, int y, int direction) {
        // do sth
    }
}

public class EnemyPositionObserver implements PositionObserver {
    @Override
    public void update(int x, int y, int direction) {
        // do sth
    }
}
```

Bewertungskriterien

- 25% - Singleton zum Assets erstellen
  - Texturen werden nur mehr 1x von der Festplatte geladen
- 25% - Der Spieler kann über das Command-Pattern bewegt werden
- 25% - Es gibt einen Observer
- 25% - Es gibt mehrere Observer:
  - Einen der die Positionen des Spielers in der UI anzeigt
  - Einen der die Positionen als Log ausgibt
  - Die Positionen vom Spieler und Gegner werden unterschiedlich ausgegeben