Programmentwurf

„Spielekiste“-Desktop App

für die Vorlesung

Advanced Software Engineering

an der

Dualen Hochschule in Karlsruhe

von

Jonas Barth & Eren Kilic

Dozent: Maurice Müller

Kurs: TINF18B5

Inhaltsverzeichnis

Einleitung

Dies ist die schriftliche Dokumentation, die Teil des Praxisprojektes ist. Als Codebasis wurden die Spiele

- Snake
- Brick Breaker
- Packman
- Flappy Bird

programmiert und müssen mit Hilfe erlernter Techniken aus der Vorlesung Advanced Software Engineering refactored werden. Die Dokumentation beschreibt dessen Anwendung und Gründe für die Anwendung.

Entwurfsmuster

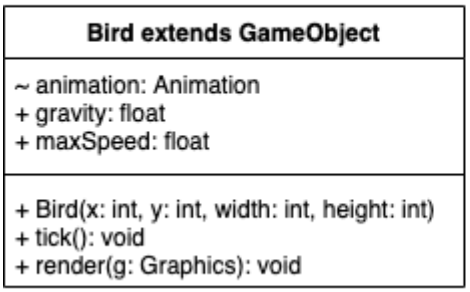
Die Aufgabe bestand darin, ein Entwurfsmuster auf den geschriebenen Code anzuwenden und vorher / nachher UML-Diagramme anzufertigen. Dabei sollen die gelernten Design Patterns angewendet und deren Einsatz begründet werden.
Folgend werden die einzelnen Design Patterns erläutert und deren Änderungen am Code hervorgehoben.

Singleton

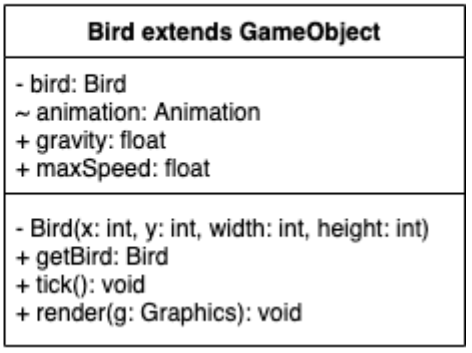
In Java zeichnet sich eine Singleton Klasse dadurch aus, dass sie nur ein einziges mal instanziiert wird. Von dieser Klasse existiert während der gesamten Programmlaufzeit nur ein einziges Objekt, das Singleton. Auf das Objekt kann mittels einer öffentlichen get-Methode zugegriffen werden. Beim ersten Zugriff per get-Methode wird das Singleton instanziiert. Folgende Aufrufe der get-Methode liefern immer das selbe Objekt zurück.

Da bei Flappy Bird jeweils nur ein einziger Vogel vom Nutzer bedienbar sein soll, bietet sich diese Klasse hervorragend an, um daraus eine Singleton Klasse zu machen. Dazu wurde der Konstruktor `private` gemacht und ist von außen nur noch `getBird()` zugreifbar.
Änderungen des Klassendiagramms der Klasse Bird:

Vorher



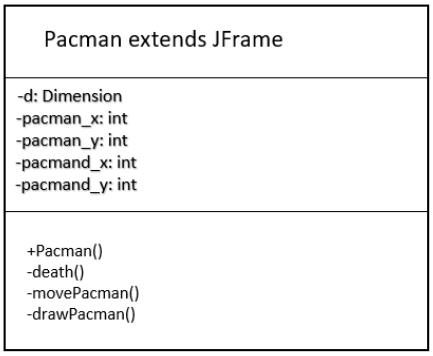
Nachher



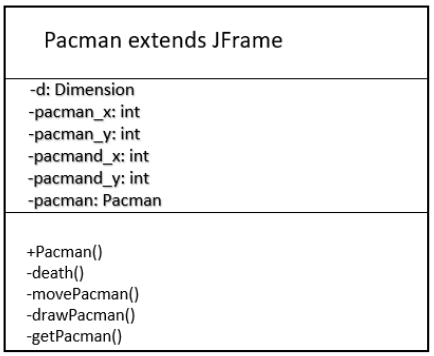
Der dazugehörige Commit ist hier zu finden:
https://github.com/job307/TINF18B5-SpieleKiste/commit/4b7d5c8bbef1ccf24763f1b33cd57524a77c9f25

Auch bei Pacman gibt es jeweils nur ein einzigen motiv der vom Nutzer bedienbar sein kann, daher bietet sich diese Klasse hervorragend an, um daraus eine Singleton Klasse zu machen. Dazu wurde der Konstruktor `private` gemacht und ist von außen nur noch `getPacman()` zugreifbar.
Änderungen des Klassendiagramms der Klasse Pacman:

Vorher



Nacher



Wie unten zusehen war der Code noch in public (hier der Code als Auszug, da es Probleme beim Commiten gab -> Eclipse):

```
public Pacman(){
        add(new Model());
}
```

Wurde aber in private geändert. Außerdem wurde die get-Methode ergänzt:

```
private Pacman(){
        add(new Model());
}


public static Pacman getPacman(){
        if(pacman == null){
                pacman = new Pacman();
                }
        }
        return pacman;
}



Pacman.pacman = Pacman.getPacman();
```

Immutable

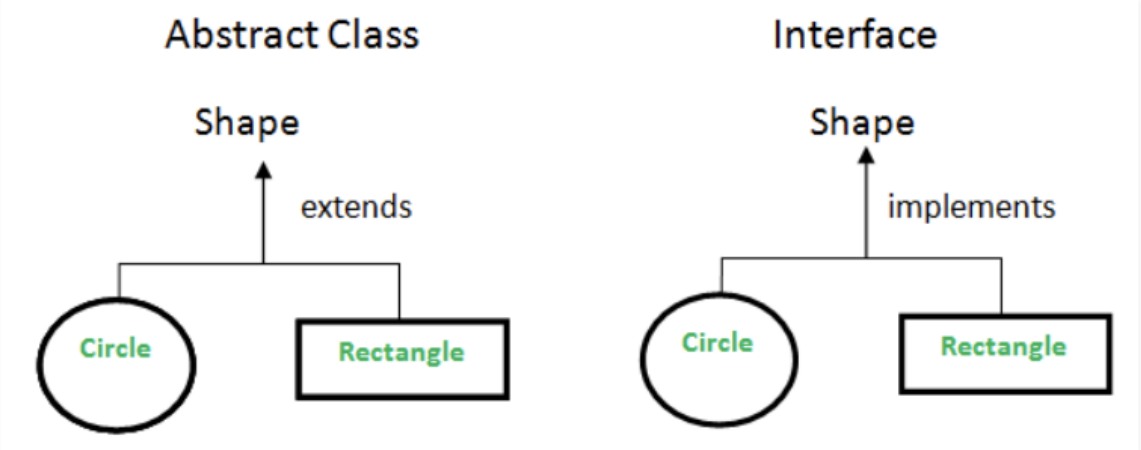
Nach der Instanziierung eines Objektes, darf es nicht mehr verändert werden können, um als unveränderliches (immutable) Objekt zu gelten. Dies sollte bei Objekten, die keine Änderungen benötigen, unbedingt angewendet werden, da es vor Manipulation schützt, die Performance verbessert und es können mehrere Threads problemlos gleichzeitig auf das Objekt zugreifen.

Bei Flappy Birds bietet sich dafür die Klasse Ground sehr an. Sie stellt den Boden / die Wiese, auf der das Spiel stattfindet dar. Dabei ist es nicht von Nöten, dass sich Werte der Klasse ändern. Die Klasse Ground besitzt nur private Attribute und besitzt keine set-Methoden. Da dies bei dieser Klasse schon immer so war, gibt es keinen Commit, der die Anwendung dieses Patterns zeigt.

Bei Pacman eignet sich in der Model-Klasse die levelData, da in der das Spielfeld bzw. der Aufbau (frei zugängliche Wege und Mauern) dargestellt ist. Der Aufbau wird größtenteils so bestehen bleiben, sodass da keine Änderungen vorgesehen ist. Die Implementierung wurde schon beim ersten Commit bzw. beim Programmentwurf entwickelt.

Interfaces

Interfaces ermöglichen den Java-Entwicklern Mehrfachvererbung anzuwenden. Ein Interface ist eine Schnittstelle, über die einer Klasse bestimmte Funktionen zur Verfügung gestellt werden. Um die Funktionen nutzen zu können, müssen sie aber erst von der Klasse implementiert werden. Das Interface gibt nur den Methodennamen und Parameter vor. Ein Interface trennt somit die Beschreibung von Eigenschaften einer Klasse von ihrer Implementierung.
Interfaces ähneln abstrakten Klassen sehr. Wie auch bei einem Interface kann von abstrakten Klassen keine Objekte direkt gebildet werden. Sie müssen erst durch eine abgeleitete Klasse / Schnittstelle erweitert werden. Erst von dieser können dann Instanzen erzeugt werden. Ein wichtiger Unterschied unterscheidet jedoch die Beiden. Während das Interface nur abstrakte Methoden besitzt, kann die abstrakte Klasse ebenso nicht-abstrakte Methoden haben. Diese müssen dann bei einer Implementierung der Klasse nicht erneut programmiert werden.



In Flappy Bird gibt es eine abstrakte Superklasse namens GameObject. Sie beschreibt alle beweglichen Objekte im Spiel und verfügt über deren Position, Form und Geschwindigkeit. Der Grund, warum sich für eine abstrakte Klasse entschieden wurde, ist, dass es nur zwei Methoden gibt, die bei der Implementierung erweitert werden müssen. Auch für diese Klasse gibt es kein Commit, da sie nie verändert wurde.

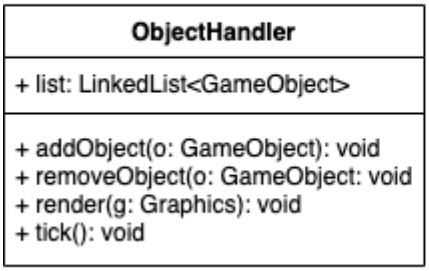
Bei Pacman werden die beweglichen Objekte, sei es die Positionen, die Geschwindigkeiten und das bewegen in der Model-Klasse dargestellt. Auch für diesen gibt es kein Commit, da sie nie verändert wurde.

Iterator

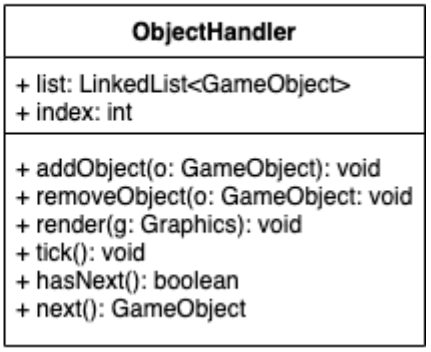
Ein Iterator durchläuft eine Sammlung von Objekten gleichen Datentyps. Dabei besteht er aus mindestens zwei Methoden: `next()` und `hasNext()`.Erst genannte Methode liefert das nächste Objekt der Sammlung und zuletzt genannte Methode sagt, ob noch ein weiteres Objekt in der Sammlung vorhanden ist.

Anwendung fand dieses Pattern in der Klasse ObjectHandler. Sie speichert während der Laufzeit die GameObjects, die sich im Spiel bewegen müssen (ihre Position ändern). Änderungen des Klassendiagramms der Klasse ObjectHandler:

Vorher



Nachher



Folgend der Commit, wie die beiden Methoden implementiert wurden:
https://github.com/job307/TINF18B5-SpieleKiste/commit/9ec25b45da98ca9110124a0d2369c36f6c7bc131

Leider konnte dieser Anwendungsfall weder bei Pacman noch bei Snake gefunden werden.

Delegate

Die eigene Klasse löst in diesem Fall die an sie gestellten Aufgaben nicht selbst, sondern übergibt die Aufgaben an die Hilfsklassen und überlässt ihnen das Lösen. Die eigene Klasse (Delegate-Klasse) kümmert sich nur um das Zurverfügungstellen der Methoden. Das eigentliche Implementieren findet in den Hilfsklassen statt.
Leider wurde ein solcher Anwendungsfall weder in Flappy Bird, noch in Brick Breaker gefunden.
Dieser Anwendungsfall wurde bei Snake zwischen der SnakeGame-Klasse und der GameFrame- Klasse gefunden. Hierzu gibt es kein commit, da an der nichts verändert wurde.

Domain Driven Design

Die Aufgabe zu Domain Driven Design (DDD) war, den Code auf Ubiquitous Language zu untersuchen und fünf Beispiele herauszusuchen. Anschließend sollten Repositories, Aggregates, Entities und Value Objects begründet und analysiert werden.

Ubiquitous Language

In beiden Spielen wurde jeweils eine Methode gefunden, deren Namen nicht passend genug ausgedrückt wurden, was zu Verständnisproblemen führen könnte, wenn sich jemand in die jeweiligen Methoden einarbeiten soll / will.
Darunter fiel:

- In Brick Breaker, Klasse Ball.java (Methode für power-ups)
  - `public void burn(int seconds)` wurde zu:
  - `public void setOnFire(int seconds)`
- In Flappy Bird, Klasse Button.java (Methode genutzt, um zu schauen, ob der Mauszeiger über dem Play-Button ist)
  - `public boolean checkCollision(int mouseX, int mouseY, Button btn)` wurde zu:
  - `public boolean mouseOver(int mouseX, int mouseY, Button btn)`

Der Link zum passenden Commit:
https://github.com/job307/TINF18B5-SpieleKiste/commit/b7ebc27d2a89dad88652252843b51d8150b53266

- In Snake, Klasse GamePanel.java (Methode für Aufbau des Spielfedes)
  - `public void draw(Graphics g)`
    hier in dieser Methode habe ich die Bezeichnung bzw. die Funktion (was es macht, mit einem Kommentar versehen,weil es so verständlicher ist und diese Ubiquitous Language vorbeugt)

Klasse GamePanel.java (Methode um zu überprüfen ob es irgendwo kollidiert hat)

- `public void checkCollisions()`
  hier in dieser Methode habe ich die Bezeichnung bzw. die Funktion (was es macht, mit einem Kommentar versehen, weil es so verständlicher ist und diese Ubiquitous Language vorbeugt)

Der Link zum passenden Commit:
https://github.com/job307/TINF18B5-SpieleKiste/commit/6c1788f0ec16f7900b05761df7b49f0597d2fbae

Repositories

Repositories werden verwendet, um die Businesslogik von der Datenbeschaffung zu trennen. Die Businesslogik selbst verwendet das jeweilige Repository um auf die Daten zuzugreifen mit dem Vorteil, dass alle relevanten Stellen denselben Code durchlaufen. Dieser muss dementsprechend nur an einer einzigen Stelle gewartet werden. Selbst Änderungen an der Datenbeschaffung bleiben der Businesslogik verborgen, da diese nur für das Repository relevant sind.

Bei Flappy Bird und Brick Breaker gibt es keine Datenbankanbindung. Flappy Bird erfordert sogar keinerlei Datensicherung. Bei Brick Breaker hingegen wird in Text-Files gespeichert, welche Level vom Spieler schon freigeschaltet sind und falls der Spieler eigene Levels erstellt, werden auch diese in den Text-Files hinterlegt, sodass sie auch nach einem Neustart des Spieles noch vorhanden sind. Die Klasse, die sich um das Erstellen, Löschen und Bearbeiten der Text-Files kümmert ist unter Brick-Breaker/src/main/bb/levelfiles/Files.java zu finden.

Auch bei Pacman und Snake gibt es keine Datenbankanbindung. Sowohl Datensicherung und Text-Files gibt es für beide Spiele nicht. Bei Pacman kommt man auf den nächst höheren Level, falls man den einen erfolgreich abgeschlossen hat. Bei beenden des Spiels wird der Endzustand nicht gespeichert, sodass man jedesmal von neu anfangen muss. Auch bei Snake wird der Endzustand nicht gespeichert, das Spiel läuft solange bis man tot ist.

Aggregates

Aggregation bezeichnet die Beziehung zwischen Klassen und Objekten. Sie wird auch HAS-A Beziehung genannt.
Bei Flappy Bird gibt es im handlers-Package die Klasse ObjectHandler.java. Diese hat unter anderem eine Liste von GameObjects, deren Position regelmäßig aktualisiert werden muss.

```
public class ObjectHandler {
   public static LinkedList<GameObject> list;

   public static void addObject(final GameObject o) {...}
   public static void removeObject(final GameObject o) {...}
   public static void render(final Graphics g) {...}
   public static void tick() {...}
```

Dank dieser Klasse muss nicht jedes Objekt einzeln aufgerufen werden, um aktualisiert zu werden.

Entities

Bei jedem Spiel gibt es einzelne Entitäten, die Objekte repräsentieren und ihnen Funktionen geben.

Bei dem Spiel Flappy Brid Gibt es die Entitäten Bird, Tube und Ground. Der Vogel ist vom Spieler bedienbar, indem er durch ein ausgelöstes KeyEvent einen “Flügelschlag” simuliert und seinen Fall bremst. Die Tubes stellen dabei das Hindernis für den Spieler dar, da es nur eine recht schmale Lücke zwischen den ihnen gibt, durch die der Spieler den Vogel navigieren muss.
Bei Brick Breaker gibt es die Entitäten Ball, Brick, Paddle und Powerup. Der Spieler bedient das Paddle, das einzig und allein nach rechts und links bewegt werden kann. Über dem Paddle befinden ich Bricks, Blöcke mit unterschiedlicher Anzahl an Leben. Ziel ist es einen Ball im Spiel zu halten, indem der Spieler ihn nicht unter sein bedienbares Paddle kommen lässt. Der Ball springt zwischen dem Paddle und den Blöcken hin und her. getroffene Blöcke verlieren Leben, bis sie zerstört sind.

Die Entitäten des Pacmans sind weißen Punkte, Geister und das Maze bzw. das Spielfeld. Der Pacman ist auch vom Spieler bedienbar, indem es durch ein ausgelöstes KeyEvent (TAdapter extends KeyAdapter)- keyPressed-Methode dargestellt wurde, wo die Benutzereingaben, in dem fall die Pfeiltasten (links-rechts, oben-unten) bedient werden können und damit am Spielfeld hin- und her navigieren kann. Dabei ist das Ziel am Spielfeld, die Punkte einzusammeln, ohne dabei von einem oder mehreren Geistern getötet zu werden. Das Spielfeld ist außen und in teilweise inneren Bereichen mit Mauern (dargestellt mit roter farbe) versehen.
Bei Snake besteht im Spielfeld ebenso ein Rand, indem sich man als Nutzer, den Snake navigieren muss. Entitäten des Snake-Spiels, der Apfel (appleX,appleY, appleEaten)l und der bodyParts.Dabei ist das Ziel, den roten Apfel (aussehen tut es wie ein roter Punkt) zu essen. Nach einem erfolgreichen essen eines Apfels, taucht der Apfel zufällig an einer anderen stelle. Auch hier werden die Benutzereingaben bzw. der Snak mit den Pfeiltasten gesteuert. Wichtig ist es nicht den Spielrand und sich selbst zu berühren.

Value Objects

Bei Entitäten ist es meist nützlich bei Initialisierungen oder set-Methoden die gegebenen Daten zu überprüfen und validieren. Gerade bei Webanwendungen mit Benutzereingaben können dabei schnell Fehler entstehen.

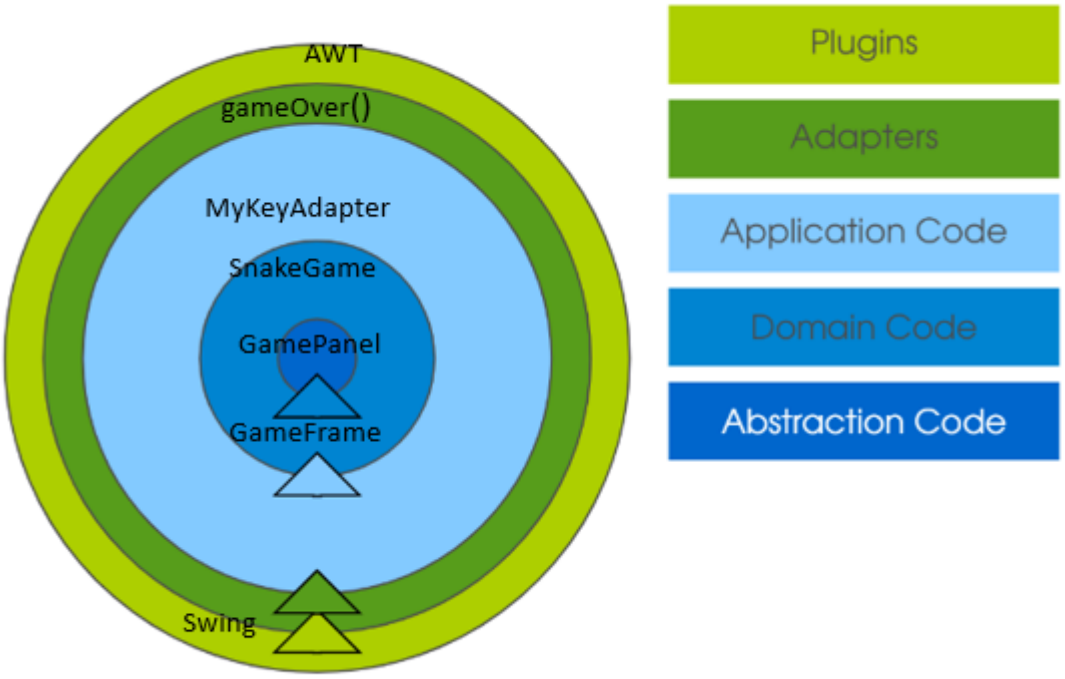
Generell besteht die Gefahr von falschen Nutzereingaben bei Flappy Bird und Brick Breaker nicht. Es werden auch nur in den allerseltensten Fällen Strings übergeben. Die einzigen Fehlerquellen wären falsch berechnete Koordinaten von Objekten. Deswegen überprüfen alle Entitäten des Spiels Brickbreaker die eingehenden Koordinaten, ob diese überhaupt auf dem Spiel-Fenster liegen.

Beim Pacman und Snake kann man bei Benutzereingaben fast nichts falsch machen. Da die Benutzereingaben relativ beschränkt sind. Was noch änderbar wäre, ist z.B. die Geschwindigkeit und die Anzahl der Geister (Ghost- bei Pacman).

Clean Architecture

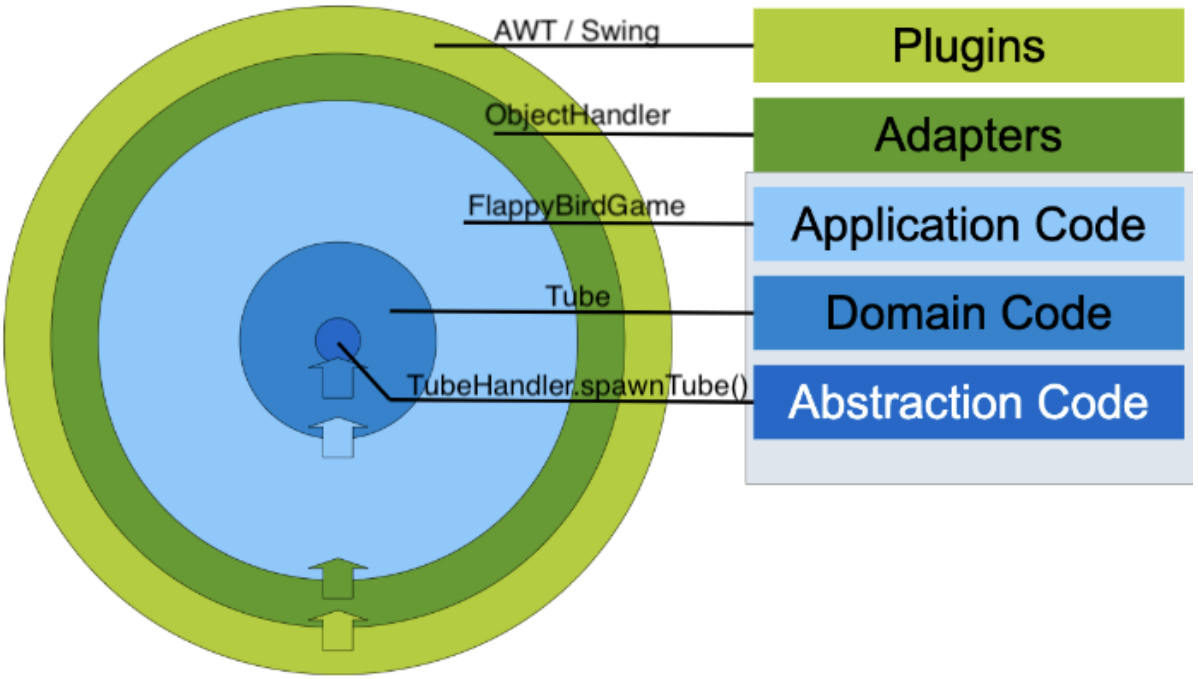
Nicht nur in Sachen Umwelt sollte Nachhaltig gedacht und gehandelt werden, sondern auch in der Softwareentwicklung. Um in der Softwareentwicklung nachhaltig zu agieren, sollte die Struktur bzw. die Architektur so ausgewählt werden, sodass Entscheidungen verzögert werden können, ohne dass man später negative Folgen bekommt. Meist werden gute Entscheidungen relativ spät getroffen, daher sollte genau überprüft und nachgedacht werden. Eine langfristige Architektur besitzt einen technologie unabhängigen Kern und behandelt jede Abhängigkeit als temporäre Leistung.

Bei Snake stellt sich folgende Clean Architecture heraus:



Für Snake besteht bei Plugins keine Datenbankanbindung und keine Drittsysteme, daher wurde es nicht im Schaubild dargestellt. Bei Snake kam eher die GUI in einsatz, insbesondere AWT und Swing. Für eine Desktop-Applikation reicht eine GUI, die mit Swing und AWT erzeugt wurde, vollkommen. Für das Adapters Schicht kam die gameOver Methode in Frage, da es eine einfache Anzeige für das Endergebnis im Spiel darstellt. Der MyKeyAdapter kann als Application Code gesehen werden, da an dem fast nix geändert werden sollte. Für das Domain Code wurde SnakeGame GameFrame eingefügt, da diese die Geschäftslogik umsetzen. Zuletzt wurde GamePanel als die Abstraction Code dargestellt.

Bei Flappy Bird sieht ein Beispiel für Clean Architecture folgend aus:



Die Methode `spawnTube()` aus der Klasse TubeHandler entscheidet zufällig, auf welcher Höhe die Lücke zwischen zwei Tubes erscheint, durch die der Vogel fliegen muss. Dabei werden auch direkt die dazugehörigen Tubes erstellt und auf die passende Höhe gebracht. Anschließend werden die Tubes dem ObjectHandler übergeben, dass regelmäßige Aktualisierungen stattfinden können. Die Anzeige der Objekte für den Spieler geschieht über AWT und Swing.

Legacy Code

Man kann Legacy Code auch als alter Code sehen, der irgendwann mal übernommen wurde. Das Problem dabei ist, dass der Code schwer änderbar und wartbar ist. Es kann sein, dass z.B. keine Tests für den Code geschrieben wurde, was das lesen oder verstehen des Codes nicht einfacher macht. Ein geschriebener Code/Software kann bzw. sollte nicht jahrelang gleichbleibend bestehen. Es sollte relativ einfach sein, den Code auch später neue Funktionalitäten einzufügen, zu Refaktorieren, die Struktur zu verbessern und das wichtigste Fehler zu beheben. Denn bei nicht vorgenommen Änderungen, sei es neue Klassen oder neue Methoden, kann es vorkommen, dass die alten Programmier bereiche mit der Zeit riesig und unverständlich werden.

Bei dem Spiel Flappy Bird wurde nach der Technik Sprout Classes refactored. Es handelt sich um die dadurch entstandene Klasse Window.java. Das macht die Klasse wiederverwendbar, übersichtlicher und deutlich offener gegenüber Erweiterungen. Der Commit, in dem die Technik angewendet wurde findet sich hier: https://github.com/job307/TINF18B5-SpieleKiste/commit/84891ba80ee8b08ddaf490a653e860bf79f19ee6

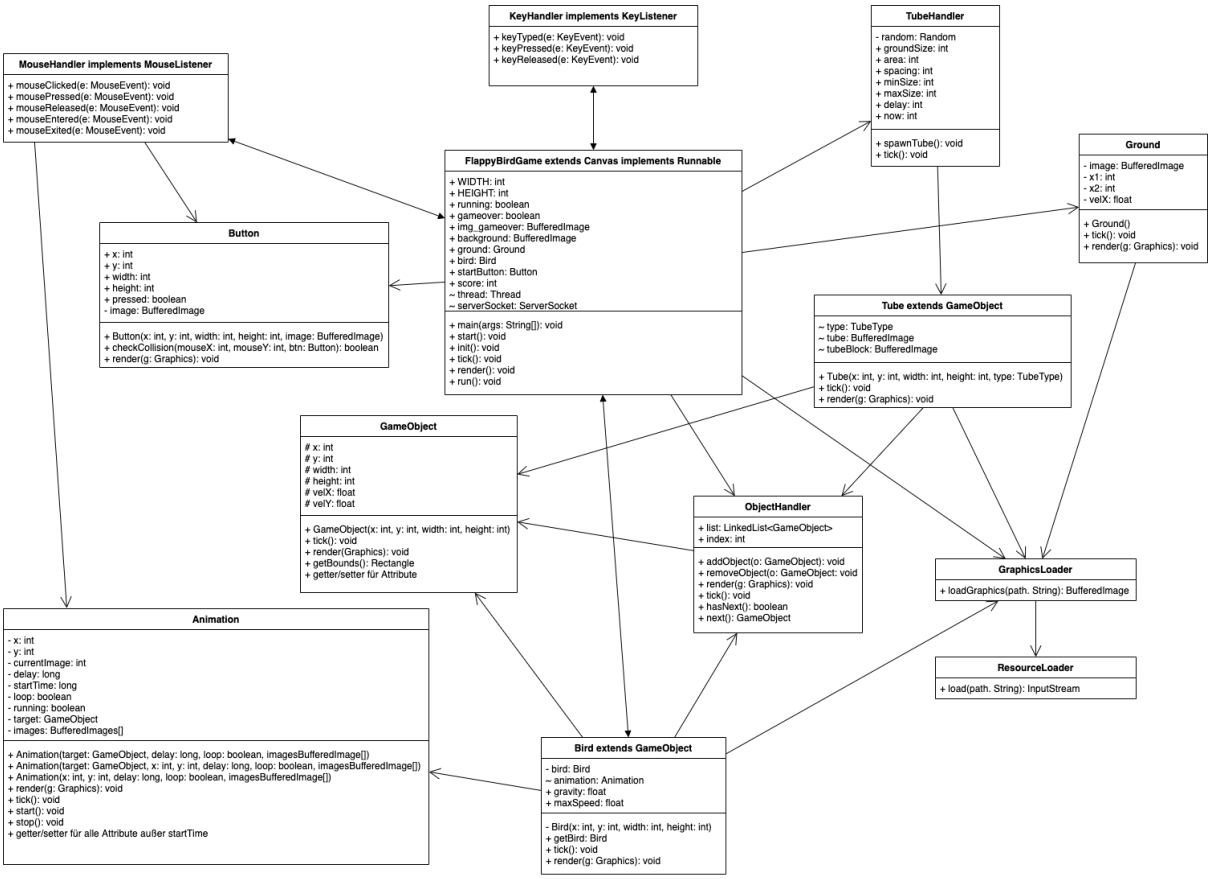
Refactoring

Refactoring anzuwenden hat viele Vorteile. Zum einen wird der Code übersichtlicher , da es “lesbarer” wird und die Wiederverwendung verbessert. Je länger und größer der Code, desto eher sollte darauf geachtet werden zu refaktorieren. Refactoring ermöglicht es die Software wartbarer zu machen und für einen “fremden” oder auch der neu in die Software/Code einlesen muss, einfacher es zu verstehen. Denn meistens entwickeln mehrere Personen parallel an einer Software, da ist es von großer Bedeutung, dass auch jeder im Team nachvollziehen kann, was der eine oder andere programmiert hat. Andererseits werden durch das Refactoring, auch die Fehler schneller und einfacher gefunden. Neue Funktionalitäten können demnach schneller entwickelt werden. Im großen und ganzen kann man erwähnen, dass Refactoring die Qualität des Codes verbessert.

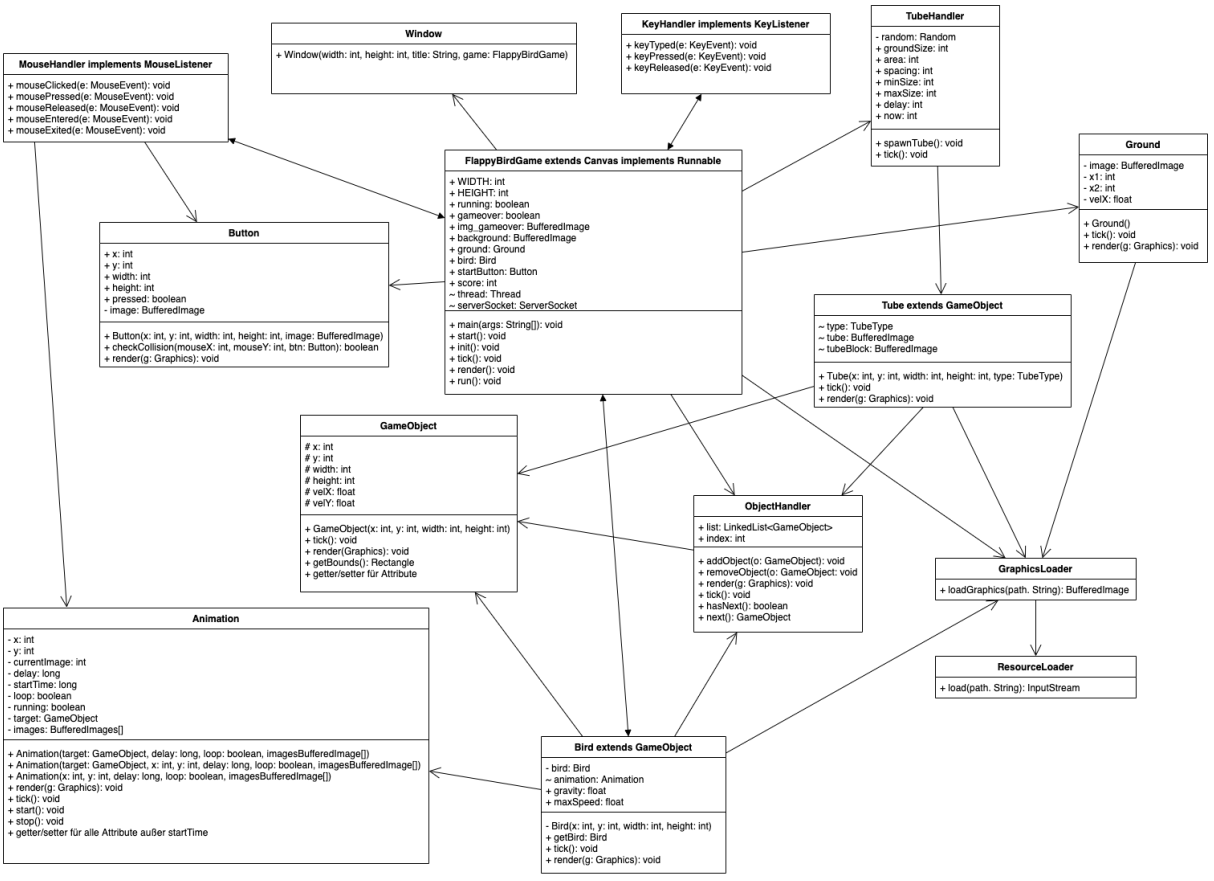
Bei Snake machte es sinn, an diesen Stellen (wie unten im link eingefügt) zu Refaktorieren, da es so deutlich übersichtlicher und lesbarer geworden ist.
Refactoring bei Snake (an mehreren Stellen):
https://github.com/job307/TINF18B5-SpieleKiste/commit/6c1788f0ec16f7900b05761df7b49f0597d2fbae

Bei Flappy Bird wurde eine neue Klasse erstellt, um Code auszulagern. Vorher wurde sich in der Game Klasse auch um die Erstellung des Fensters des Spiels gekümmert. Das Refactoring war nötig, da die Klasse Game sich schon darum kümmert, das Spiel in kurzen Abschnitten immer wieder zu aktualisieren und somit sich bereits um eine Aufgabe kümmert. Der zugehörige Commit ist hier zu finden:
https://github.com/job307/TINF18B5-SpieleKiste/commit/84891ba80ee8b08ddaf490a653e860bf79f19ee6

Vorher UML-Diagramm (Flappy Bird):



Nachher UML-Diagramm (Flappy Bird):



Programming Principles

Prinzipien gibt es nicht nur in bestimmten Bereichen des Lebens, sondern auch in der Softwareentwicklung. Man kann es als eine Art Leitfaden sehen. Doch Prinzipien entstehen nicht von heute auf morgen, es braucht Zeit und jahrelange Erfahrung.

SOLID

Single-responsibility Principle

Das Single Responsibility Principle sorgt für eine leichte Wartbarkeit der Software durch Reduktion der Aufgaben / Verantwortung einer bestimmten Klasse und folglich auch deren Größe.
Im Quellcode von Flappy Bird wurde wurde die Klasse `FlappyBirdGame.java` refactored, da sie mehrere Aufgaben zu lösen hatte und zu viel Verantwortung trug. Zudem war diese Klasse zu lang, was sie recht unübersichtlich gemacht hat. Durch das Refactoren wurde der Klasse eine Aufgabe abgenommen und daraus eine neu, übersichtliche, leicht wartbare Klasse (`Window.java`) geformt. Erläuterte Änderungen finden sich in folgendem Commit: https://github.com/job307/TINF18B5-SpieleKiste/commit/84891ba80ee8b08ddaf490a653e860bf79f19ee6

Open-closed Principle

Das Open-closed Principle beschreibt die Offenheit der Anwendung für Erweiterungen, aber die Geschlossenheit für Änderungen an bestehendem Code. Daher ist der gesamte Code des Flappy Bird Spiels in einer klaren Klassenstruktur gehalten, sodass Funktionen einfach erweitert werden können.

Liskov substitution Principle

Beim Liskov substitution Principle geht es um die logische Trennung verschiedener Klassen, wo sich gemeinsame Basisattribute und Funktionen feststellen lassen. Auch dies ist beim Spiel Flappy Bird umgesetzt worden. Die Entitäten Bird und Tube ähneln sich in ihren Attributen sehr, da beide mit Koordinaten und einer festen Höhe und Breite versehen sind. Beide extenden deshalb der Superklasse GameObject. Der einzige unterschied zwischen den beiden Objekten ist, dass sie sich in unterschiedliche Richtungen bewegen.

Interface segregation Principle

Das Interface segregation Principle kann nicht betrachtet werden, da keine in Code vorhanden sind. Stattdessen wurde eine abstrakte Klasse verwendet.

Dependency Inversion Principle

Bei dem Dependency Injection Principle werden Abhängigkeiten zwischen Methoden unterschiedlicher Klassen hergestellt, sodass diese in einer anderen Klasse verwendet werden können. Dieses Principle findet sich überall, wo eine Entität erstellt wurde. über diese Entität werden Methodenzugriffe aus der jeweiligen Klasse ermöglicht. Ebenso können

Methoden über Vererbung weitergegeben werden. Beispielsweise extendet die Klasse Bird dem GameObject, weswegen er über mehrere Funktionen / Methoden verfügt, als tatsächlich in der Klasse Bird deklariert wurden.

GRASP

Grasp ist dafür da, um die Zuständigkeiten zu klären. Genauer gesagt, derjenige der am meisten Wissen hat, soll für die neue Aufgabe zuständig sein.

Kopplung

Je geringer die Kopplung zwischen einzelnen Klassen, desto unabhängiger sind diese und umso schwerer wird es, durch Änderungen in einer Klasse, Fehler in einer anderen zu erzeugen. Deswegen gibt es beispielsweise bei Flappy Bird zusätzlich zur Klasse Tube einen TubeHandler, genauso wie es auch einen ObjectHandler gibt.

Kohäsion

Kohäsion ist dafür da, um den Code übersichtlicher und strukturierter anzuzeigen. Im Grunde zeigt Kohäsion die engen Verantwortlichkeiten von Elementen.

DRY

Das Dry-Prinzip schreibt den Entwicklern vor, dass man sich nicht wiederholen solle. Vor Allem sollte Redundanz vermieden oder wenns garnicht anders geht, reduziert werden. Das Prinzip findet sich im Flappy Bird Spiel wieder. Einer der Gründe für die eine abstrakte Klasse GameObject, im Gegensatz zu einem Interface, war es, dass nicht alle Methoden implementiert werden müssen, wenn ein Objekt von dieser Klasse abstammt.

Unit Tests

Tests gibt es nicht nur bei Hardware oder bei Autos, wie z.B. EuroNCAP, sondern auch bei der Softwareentwicklung ist das Testen sehr wichtig. Tests geben eine oder mehrere Auswertungen über das Produkt oder die Software. Sie schützen bestehende Funktionen und falls es Veränderungen geben sollte, wird es so schnell wie möglich erkannt. Dabei gibt es unterschiedliche variationen an Tests. Akzeptanztest, Integrationstest, Komponententests und Performancetest.

Bei Tests ist es wichtig die A-TRIP Regeln einzuhalten, um eine klare Struktur vorweisen zu können. A-TRIP steht für Automatic, Through, Repeatable, Independent, Professional. Nach diesen Regeln sollen Tests:

- einfach ausführbar sein
- alles notwendige überprüfen
- beliebig wiederholbar sein und müssen immer das selbe Ergebnis liefern
- keine Abhängigkeiten zu anderen Tests haben
- den selben Qualitätsstandards wie Produktivcode unterliegen

Die programmierten Spiele bieten nicht viele Möglichkeiten sinnvolle Tests zu schreiben. Dennoch wurden die wichtigsten Methoden einiger Klassen getestet:
https://github.com/job307/TINF18B5-SpieleKiste/commit/d90bbbdbedf2ebfe1971cf824738065c93211f59
Bezüglich dem Einsatz von Mockobjekten war es uns leider nicht möglich diese in die Tests einzubauen, da kein passender Anwendungsfall vorhanden ist. Es wurden weder zu mockende Datenbankanbindung oder API-Schnittstellen in den Spielen verbaut.

Fazit

Im großen und ganzen haben wir als Team versucht, so viele Anforderungen und Anwendungsfälle wie möglich durchzuführen und umzusetzen. Uns ist klar, dass wir nicht alles umsetzen konnten. Teilweise hat unser know-how gefehlt, aber es gab auch Bereiche, in der es nicht möglich war in den jeweiligen Spielen diese umsetzen. Es war für uns nicht einfach, für die Vorlesung “Advanced Software Engineering” in der kurzen und knappen Zeit (weil wir einfach zu viele andere Dinge für das Studium machen mussten) noch ein Programmentwurf zu gestalten, mit den entsprechen Anforderungen. Es wurde schon mehrfach angesprochen, aber wir könnens nur hoffen und Wünschen, dass die zukünftigen Studierenden in der Vorlesung “Advanced Software Engineering” statt einem Programmentwurf wieder eine Klausur schreiben dürfen.
Trotz all den Umständen und Schwierigkeiten gab es auch Momente, wo uns die Vorlesung mit Ihnen spaß gemacht. Wir müssen zugeben, dass wir auch ein schwieriger Kurs sind :)
Wir hoffen, dass auch Sie mit uns zufrieden waren und wünschen Ihnen alles Gute und bleiben Sie Gesund.