1 Aufgabe 1: Flappy Bird Quellcode

1.1 Anti-Patterns:

- **The Blob:** Ähnlich zum Beispiel in der Vorlesung haben wir mit der Klasse GamePanel einen Blob gefunden. Das GamePanel übernimmt hier die alleinige Verwaltung und dominiert somit den Programmablauf (ablauf-orientierter Entwurf). Sämtliche Datenklassen sind getrennt davon angeordnet. Dies führt zu einer engen Kopplung und erschwert die Wiederverwendbarkeit und Wartung des Codes

- **Poltergeist:** Mithilfe des IntelliJ-Tools für Diagramme und Analyse lässt sich ein UML-Diagramm der Klassenabhängigkeiten erstellen. Anhand dessen erkennt man einen zentral strukturierten Aufbau der Klassenabhängigkeiten, was nicht zu einem Poltergeist passt. Die zentrale Struktur zeigt, dass die Abhängigkeiten bewusst organisiert sind, könnte aber auch problematisch werden, wenn alles von einer Stelle abhängt.

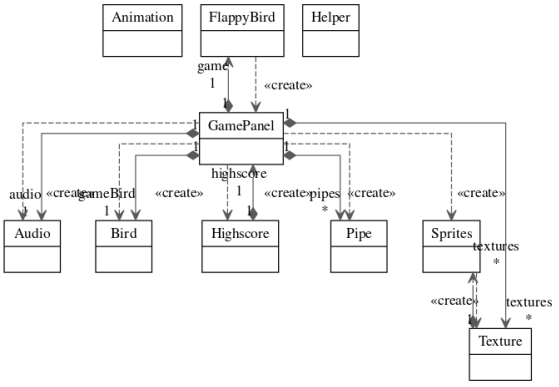

Abbildung 1: Ursprüngliches Klassen-Diagramm

- **Funktionale Zerlegung:** Wie bereits im obigen UML-Diagramm zu sehen, sind die einzelnen Funktionalitäten ausgelagert. Wir haben hier zwar keine explizite Benennung der Klassen nach Funktionsnamen, jedoch sind die Klassen teilweise sehr klein und haben oft nur eine genutzte Methode. Die Klasse Audio kümmert sich zum Beispiel nur darum, eine WAV-Datei abzuspielen, und ist dadurch sehr kompakt. Man kann also klar von einer funktionalen Zerlegung sprechen. Auch die Klasse Helper ist ein gutes Beispiel, da sie sich nur ums Öffnen von URLs kümmert.

1.2 Bad Smells:

- **Duplizierter Code:** Ein kurzer Blick in die größeren Klassen zeigt schnell einige doppelte Codezeilen. In der Klasse Sprites gibt es zum Beispiel viele Wiederholungen, vor allem bei der Erstellung von Texturen, der Verarbeitung von Schleifen für Punktzahlen und ähnlichen Aufrufen für Kategorien wie Birds oder Medals. Auch in der Klasse GamePanel tauchen beim Zeichnen von Buttons immer wieder ähnliche Codeabschnitte auf. Solche Wiederholungen könnten leicht in Funktionen ausgelagert werden, um den Code übersichtlicher und wartungsfreundlicher zu machen. In der GamePanel Klasse befindet sich auch Methoden wie *g.drawImage* welche oft aufgerufen werden, diese kann man ebenfalls einspaaren und als Methode auslagern.

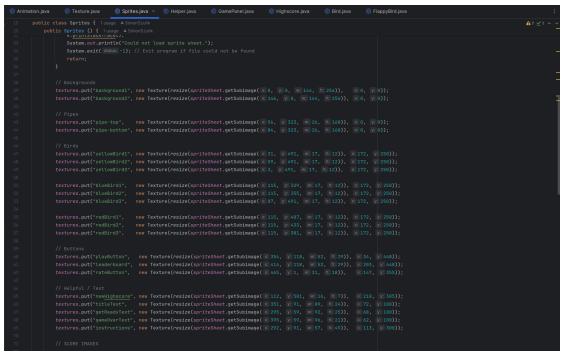

Abbildung 2: Dublizierter Code

- **Unangebrachte Intimität:** Die Klasse Sprites erstellt viele Texture-Objekte und fügt sie zu einer statischen HashMap hinzu. Es gibt eine starke Kopplung zwischen Sprites und Texture, da Sprites direkt Texture-Instanzen erzeugt und deren Konstruktorparameter kennt. Das macht den Code weniger flexibel, da Änderungen an der Klasse Texture hier schnell zu Problemen führen könnten. Auch die GamePanel Klasse greift teilweise unerwartet auf Bird-Funktionalitäten und Variabeln zu.

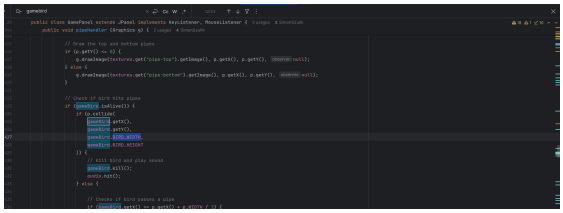

Abbildung 3: Schlechter Variablen-Zugriff

- **Keine Switch-Anweisungen:** Zur Analyse wurde in IntelliJ die Suchfunktion genutzt, um nach dem Keyword switch zu suchen und zentrale Stellen im Code zu überprüfen. Es gibt zwar einen Switch-Case in GamePanel.java, jedoch zeigt dieser keine direkten Hinweise auf einen Bad Smell. Die Zustandslogik ist klar strukturiert, unterscheidet sauber zwischen Spielzuständen und enthält keine überflüssige Logik. Solange die Anzahl der Zustände überschaubar bleibt, ist Refactoring nicht notwendig. Sollte der Code jedoch komplexer werden, könnte das State-Pattern helfen, die Übersichtlichkeit und Wartbarkeit zu verbessern.

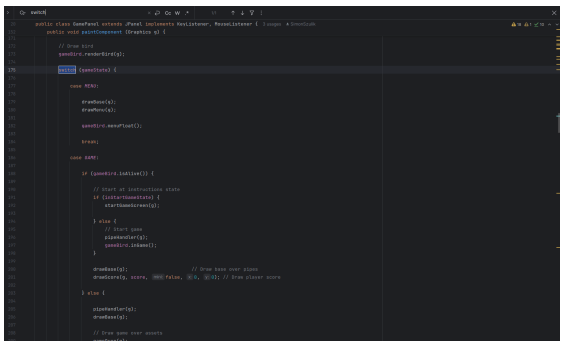

Abbildung 4: Switch-Case

- **Ausgeschlagenes Erbe** Eine schlechte Anwendung von Vererbung findet sich vor allem in der Bird-Klasse. Hier wird von JPanel geerbt, ohne dass diese Vererbung tatsächlich genutzt wird. Dadurch können auf Bird-Objekten Funktionen von JPanel aufgerufen werden, die aber in diesem Kontext keinen Sinn ergeben. In diesem Fall könnte man die Vererbung komplett entfernen und stattdessen eine Komposition verwenden, falls die Funktionalität von JPanel überhaupt benötigt wird.

1.3 Weiter Aufälligkeiten zum Code:

- **Unused Code:** Es gibt zahlreiche unbenutzte imports, welche ohne weiteres entfernt werden können (BSP: java.io.File in Sprites.java).
- **Unused Variables:** Ähnlich wie bei den Imports gibt es auch hier zahlreiche Klassenvariablen die nicht genutzt werden, oder schlecht genutzt werden. Teilweise erscheinen Felder komplett überflüssig und können leicht ausgelagert werden wie das MENU, GAME, was einfach durch einen Bool realisiert werden kann statt final static ints.
- **Viel zu Viel Kommentar:** Der Code enthält super viele unnötige Kommentare, was bei 500+ Zeilen teilweise sehr die Lesbarkeit kaputt macht. Kommentare würde ich komplett entfernen und nur auf Kommentare zurückgreifen wenn diese Komplexe Codestücke verständlicher machen.

2 Aufgabe 2: Alternative Architektur

2.1 Neue Architektur

Mein Vorschlag für eine neue Architektur sähe folgendermaßen aus. Die ganzen kleinen Helfer-Klassen werden so weit wie möglich in die bereits vorhandenen Klassen überführt und die Funktionen zu direkten Methoden der Klassen selber gemacht. Die GamePanel-Klasse wird verkleinert und bekommt eine Elternklasse, welche den Großteil der Arbeit wie das Zeichnen der verschiedenen Elemente beinhaltet. Es wäre auch möglich gewesen, die verschiedenen Game-States als eigene Panelklassen zu schreiben, welche zum Beispiel als Objekt an eine HauptKlasse wie das GamePanel übergeben werden, allerdings hätten wir dann wieder eine Vielzahl kleinerer Klassen und wieder eine Neigung zur funktionalen Zerlegung. Die Audioklasse wird in ihrer Abhängigkeit verschoben und gehört nun zum Bird selber und nichtmehr zu GamePanel. Die Struktur der Pipe und des Birds bleibt an sich gleich, das es doch Sinvoll ist die beiden Hauptobjekte der Klasse getrennt zu betrachten. Ich habe mich hier bewusst dazu entschieden keine *Game-Objekt* Oberklasse zu machen, von der beide erben, weil weil wir dann wieder zu viele unnötige Auslagerungen hätten. Auch wenn beide Klassen sich in ihreren Attributen wie der Größe etc. ähneln. Der Higscore-Klasse ist ebenfalls nichts hinzuzufügen; obwohl sie auch in die Game-Panel Klasse refactort werden könnte; was die Komplexität und die Aufgaben dieser dann aber wieder mehr an einen Blob errinern lässt.

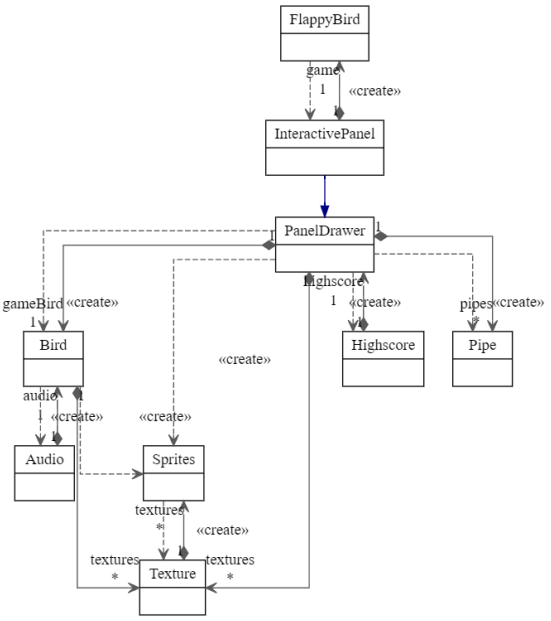

Abbildung 5: Architektur-Vorschlag

2.2 Vorgehen

Zuerst habe ich die Helper-Klasse entfernt und ihre Funktionalitäten in die GamePanel-Klasse integriert. Anschließend habe ich mich um die ungenutzten Vererbungen gekümmert, wie zum Beispiel das extends JPanel in der Bird-Klasse. Da keine spezifischen Funktionalitäten von JPanel verwendet wurden, konnte diese Vererbung problemlos entfernt werden. Mit diesen Änderungen wurden bereits mehrere Probleme wie mangelnde funktionale Zerlegung und schlecht eingesetzte Vererbung behoben. Um die Funktionale Zerlegung weiter einzugrenzen habe ich die Animation-Klasse aufgelöst. Diese bestant aus einer einzigen Methode (animate), welche nur einmal Anwendung fand (in Bird.renderBird)). Die Methode wurde in die Bird-Klasse verlagert.
Einige Beispiele für unangebrachte Intimität im Code wurden ebenfalls bereinigt. Funktionen, die nicht zur Zuständigkeit der Klasse gehörten, wurden in die jeweils relevanten Klassen verschoben. Beispielsweise war das zufällige Erstellen der Birds und deren Farben bisher Teil der GamePanel-Klasse. Diese Logik wurde in den Konstruktor der Bird-Klasse verlagert, wodurch unnötige Abhängigkeiten und Aufrufe auf Objektvariablen reduziert wurden. Dadurch entfällt auch die direkte Verbindung zwischen GamePanel und den Bird-Texturen, was den Code klarer und modularer macht. Die Audio Spuren-wurden sowohl in der GamePanel-Klasse als auch in der Bird Klasse verwendet verwendet. Die benutzten Audio Spuren sind nun teil der Methodik vom Bird. So wird audio.hit() dann aufgerufen wenn Bird.kill() benutzt wird. Äquivalent wird mit audio.point() und audio.jump() umgegangen, wobei für audio.point() eine neue Methode (public void score()) in Bird eingeführt wurde. Somit ist die nurnoch die Bird Klasse in direkter Verbindung mit der Audio Klasse. Auch habe ich überlegt die Audio-Klasse nun direkt in Bird einzubauen, aber meiner Meinung nach ist das abspielen von Audios doch eher eine Eigene Funktionalität die man stets erweitern kann, im gegensatz zur Helper-Klasse.
Ein Ansatz zur Verbesserung der Architektur war, die Klassen Sprites und Texture zusammenzuführen. Allerdings erfüllen diese beiden Klassen völlig unterschiedliche Aufgaben: Die Texture-Klasse dient als einfache Objektklasse, die Textur-Objekte bereitstellt, mit denen effizient gearbeitet werden kann. Die Sprites-Klasse hingegen extrahiert die verschiedenen grafischen Elemente aus den Ressourcen und speichert sie als die erwähnten Textur-Objekte. In der Sprites-Klasse wurden vor allem doppelter Code refaktoriert und die addTexture-Methode ausgelagert, um die Klasse übersichtlicher zu gestalten, insbesondere im Hinblick auf potenzielle Erweiterungen mit neuen Texturen. Ein alternativer Ansatz wäre, die Sprites-Klasse als Sammlung von statischen Hilfsmethoden zu implementieren, um die Abhängigkeit zwischen den Klassen zu vermeiden. Meiner Meinung nach ist dies allerdings nicht notwendig, da die Arbeit mit einem Sprites-Objekt unkompliziert bleibt. Zudem würde der statische Ansatz häufige Methodenaufrufe erfordern und weiterhin Abhängigkeiten zur Texture-Klasse beinhalten. Daher wurde an der bestehenden Struktur keine grundlegende Änderung vorgenommen. Um den Blob im Design aufzulösen bzw. versuchen zu die Klasse zu verkleiner, wurde eine neue Klasse namens PanelDrawer erstellt. Diese Klasse übernimmt sämtliche Aufgaben, die mit dem Zeichnen, Rendern und Aktualisieren von Sprites zu tun haben. Sie enthält alle dafür

notwendigen Methoden und Variablen und erbt von JFrame, um die entsprechenden JFrame-Aufgaben zentral zu bündeln. Alle Zeichen- und Renderfunktionen, die zuvor in der GamePanel-Klasse lagen, wurden in die PanelDrawer-Klasse ausgelagert.
Die GamePanel-Klasse ist nun deutlich kleiner und übersichtlicher. Sie erbt nicht mehr von JFrame, sondern von der neu eingeführten PanelDrawer-Klasse, wodurch die Verantwortung klar getrennt wurde. Die alte Blob Klasse, kümmert sich nun alleinig um das notify des jeweilligen Zeiger Fokus, sowie um die eigentliche Interaktion mit dem Feld; sprich sie implementiert immernoch die Key-/MouseListener und deren Methodik und heißt nun InteractivePanel. Als letztes habe ich mich um die unangebrachte Vielzahl des Whitespace gekümmert sowieo im die ganzen zu *offensichtlichen* Kommentare, und habe diese entweder entfernt oder umgestaltet.