0. Systemidee

Siehe: Software-Architektur IP1

Pipe It Up!

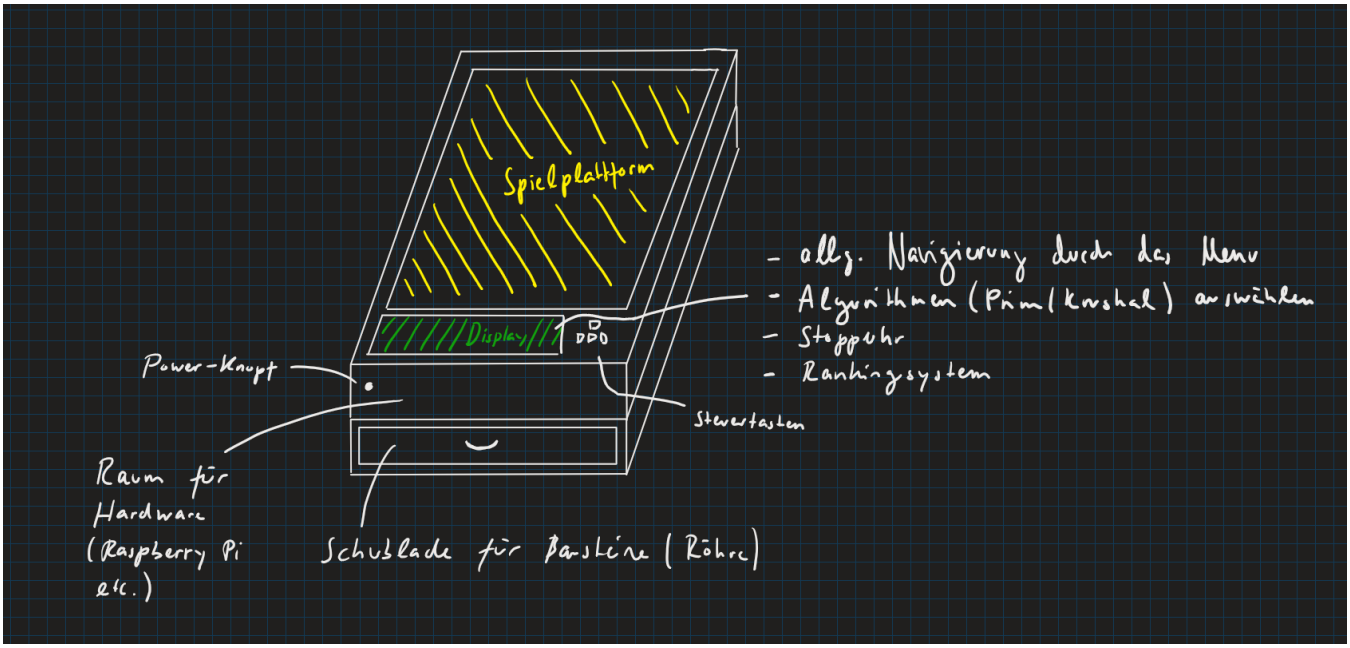
Pipe It Up! ist ein didaktisches Werkzeug um die gewichtete Graphentheorie zu erlernen.

Dieser Architekturüberblick ist dazu da, dass die wichtigsten Entwurfsentscheidungen nachvollzogen werden können. Er zeigt die Struktur der Lösung und das Zusammenspiel zentraler Elemente. Die Gliederung der Inhalte erfolgt nach der arc42-Vorlage.

Zielgruppe dieses Überblicks sind in erster Linie Projektmitglieder, die Informationen zur technischen Umsetzung suchen.

Diese Dokumentation soll als zentraler Ablage-Ort für alle technischen Konzepte und für Dokumentationen der Umsetzung gelten.

Skizze vom Produkt:



Pipe It Up! wird als physisches Brettspiel mit digitaler Steuerung implementiert. Die Benutzer sollen dabei physisch mit dem System interagieren können, um den didaktischen Nutzen zu maximieren. Das heisst Pipe It Up! ist ein Event basiertes System, mit einem physischen UI (User Interface). Dem User wird Feedback über LED's, Audio und Text auf dem DIsplay gegeben. Dies macht Pipe It Up! zu einem eingebetteten System.

Pipe It Up! wird als eigenständig lauffähiges System implementiert. Damit wird es mobil und ist nicht auf lokale Installationen oder Internetverbindung (mit Ausnahme dem Submitten der Highscore Daten) angewiesen.

1. Einführung und Ziele

1.1 Aufgabenstellung

Was ist Pipe It Up!?

- Pipe It Up! ist ein didaktisches Werkzeug zum Thema "Gewichtete Graphen".
- Es dient als spielerisches Werkzeug, um Studierenden das Thema der gewichteten Graphen näher zu bringen.
- Die Spielenden können ihr Wissen und ihre Fähigkeiten im Wettkampfmodus testen.
- Zufalls- und Selbstgenerierte Karten bieten wiederholenden Spielspass.

Wesentliche Features:

- Vollständige Implementierung des Prim und Kruskal Algorithmus.
- Enthält einen Erklärmodus, um den Prim und Kruskal Algorithmus zu erlernen.
- Führt eine Highscore, über welche man sich mit anderen Spielenden messen kann.
- Die gewichteten Graphen werden als physische Pipelines dargestellt, die zwischen zwei Knoten gelegt werden können.

1.2 Qualitätsziele

Die folgende Tabelle beschreibt die zentralen Qualitätsziele von Pipe It Up!, wobei die Reihenfolge eine grobe Orientierung bezüglich der Wichtigkeit vorgibt.

Qualitätsziel	Motivation und Erläuterung	Wie wird das Ziel erreicht?
Physisches (Brett)-Spiel / Robustheit	Für das einfache Verständnis soll das Spiel physisch in Form von einem Brett-Spiel umgesetzt werden. Die Pipelines und Nodes sollen physisch existieren und die Pipeline von Hand gesetzt werden können. Das Brettspiel soll physisch stabil sein, damit es nicht schnell kaputt geht und lange lebt. Die Dimensionen und das Gewicht des Spiels sollen so gehalten werden, dass es mobil / transportierbar ist. Zusätzlich soll das physische Brettspiel stabil und robust gebaut sein, damit es möglichst lange, trotz Abnutzung, noch funktioniert.	• Die Stabilität des Gehäuses wird durch ein stabiles Holz-Gehäuse sichergestellt. • Pipelines werden aus rostfreiem Stahl Röhren gefertigt. • Die Nodes werden als physische "Türmchen" 3D gedruckt.
Software UI	Für Testzwecke und zur Implementierung ist eine Software Repräsentation des UI's wichtig. Damit kann der Entwicklungs- und Testprozess vereinfacht werden.	• Die Darstellung des Graphens wird ebenfalls in einer Software-Form implementiert
Robuste Softwarelösung	Da das Spiel in physischer Form ist und lange Leben soll, soll die Software fehlertolerant implementiert sein und mit Fehler so gut wie möglich umgehen können (Hardware und Software Fehler). Sie muss ohne regelmässige Wartung auskommen und Logfiles oder eine Visualisierung für Probleme haben.	• Es wird Log4J zum Loggen verwendet, und in allen try-catch Blöcken werden Log-Meldungen eingebaut. • Wihcitge Ereignisse werden geloggt, um Fehlerquellen nachvollziehen zu können
Schnelles Feedback auf UserInput	Das System soll schnelles Feedback auf UserInput geben können, damit sich das ganze "snapy" anfühlt für die Benutzenden.	• Event-basiertes FrontEnd, dass ohne while(true) loops oder ähnliches auskommt und asynchron läuft, um den UI Thread nicht zu blockieren.
Einfache Installation	Pipe It Up! soll einfach installiert / aufgestellt werden können. Das einzige was nötig sein sollte, um "Pipe It Up!" in Betrieb zu nehmen ist das Stromkabel + (fakultativ) Ethernetkabel mit Internet einstecken.	• Externen Stromanschluss am Holzgehäuse, Internet wird gar nicht benötigt, da keine übermittlung der Highscores mehr passieren muss (Gemäss der Projektschiene IP12)

Die Qualitätsszenarien werden in 10. Qualitätsanforderungen konkretisiert und ihre Erreichung ausgewertet.

1.3. Stakeholder

Siehe Produktvision

2. Rahmenbedingungen

2.1 Technische Rahmenbedingungen

Rahmenbedingung	Hintergrund und / oder Motivation
Lauffähig auf Raspbery Pi 4B 8GB	Die Software muss auf einem Raspberry Pi 4B lauffähig sein. Diese Beschränkung kommt von CrowPi und von den Anforderungen an das Modul IP12
Java 11 als Programmiersprache	Definiert in IP12 - Technische Rahmenbedingungen. Da der Unterricht im 1. und 2. Semester auf Java beschränkt ist, und damit alle Projektmitglieder an der Software-Lösung mithelfen können wird im Rahmen von IP12 Java als Programmiersprache verwendet. CrowPi unterstützt zudem nur Java Version 11, daher wird die Software-Lösung auch in Java 11 umgesetzt.
SQLLite als Datenbank	Definiert in IP12 - Technische Rahmenbedingungen. Da man im 1. und 2. Semester noch nicht viele Technologien kennt, werden gewisse Technologien vordefiniert. Zudem ist SQLite eine Datei-basierte Datenbank und man benötigt daher keinen Datenbank-Server, welches das Setup komplizieren würde.
MQTT als Protokoll um HighScores zu submitten (obsolet)	Definiert in IP12 - Technische Rahmenbedingungen. Das fertige Projekt wird am Abschluss-Event öffentlich vorgestellt. Dabei gibt es einen Wettbewerb bei dem die Besucher bei allen fertigen Projekt-Exponaten Punkte sammeln können über ein Highscore System. Dies wird dann benutzt um den Sieger des Wettbewerbs zu küren. Die Highscores werden über das MQTT Protokoll an den FHNW Server gesendet.
Robuste Softwarelösung, die mit Fehlern (Exceptions) und Hardware Problemen umgehen kann.	Da das System voraussichtlich lange im Betrieb sein wird, ohne Wartungsmöglichkeiten, soll das System gut mit Fehlern in der Software und Problemen mit der Hardware umgehen können. Dabei sollen Software- und / Hardwarefehler in einem Logfile geloggt werden und optional noch visuell irgendwo angezeigt werden.

2.2 Organisatorische Rahmenbedingungen

Rahmenbedingung	Hintergrund und / oder Motivation
Team	Das Team setzt sich zusammen aus • Tino Heuberger (s) • Luca Plozner (s) • Aaron Bodenmann (s) • Föry Marc (s) • Samir Hauri (s) • Carl von Burg (s) • Nicola Liechti (s)
Zeitlicher Rahmen	Das Projekt startet mit dem Beginn des 1. Semster und erstreckt sich über 2 Semester, bis zum Ende des 2. Semesters
Prozess Model	Für die Umsetzung des Projektes wird RUP (Rational Unified Process) verwendet. Dies wurde von den Modulverantwortlichen so vorgeschrieben. Dabei handelt es sich um ein agiles und iteratives Vorgehen.

Entwicklungstools	Für die Java-Entwicklung werden IDE's wie IntelliJ, Eclipse, VsCode, Vim, ... benutzt. Damit Styling Rules IDE übergreifend gleich definiert sind werden .editorconfig files verwendet. Alle gängigen IDE's verstehen den .editorconfig Standard und für alle bekannten TextEditoren (VsCode, Vim, etc) gibt es Plugins dafür. Um die Abhängigkeiten zu Verwalten und um die Java Applikation zu kompilieren wird Maven genutzt.
Versionsverwaltung	Um die Zusammenarbeit und Versionierung zu vereinfachen, wird Git benutzt. Dazu wird von der FHNW ein gitlab Server zur Verfügung gestellt.
Deployment	Damit die Applikation auf dem Raspberry Pi deployt werden kann, wird das CI/CD System vom FHNW gitlab Server benutzt.
Qualitätssicherung	Für die Qualitätssicherung des Codes wird JUnit benutzt. Die UnitTests sollen bei jedem Pull Request in das master Repository laufen gelassen werden und nur merge-bar sein wenn diese alle erfolgreich durchgelaufen sind. Zudem wird die Code-Coverage im Pull Request gemessen und darf 80% nicht unterschreiten.
Dokumentation	Für die Dokumentation wird die Confluence Instanz der FHNW genutzt.

2.3 Coding Conventions

Coding Convention	Beschreibung
Google Java Style Guide	Der Code hält sich an die Google Java Styling Guideline sofern nicht anders definiert in .editorconfig oder in weiteren Coding Conventions definiert.
Triple A bei UnitTests	UnitTests sollem im "Tripple A" Schema implementiert werden. Heisst • **Arrange**: Vorbereiten aller Daten die man zur Testausführung braucht • **Act**: Ausführen der TestFunktion • **Assert**: die asserts bzw Tests für die Ausgeführte Aktion in act

2.4 Development Process

2.4.1 Git

Git wird als Tool für die Versions- und Codeverwaltung benutzt.

Kurz-Beschreibung	Beschreibung
Branches	Pro isolierte Funktionalität soll ein eigener Git-Branch erstellt werden, aka "**Feature Branch**". Der Branch Name soll dabei nach dem Schema "{UserStory}_{KurzBeschreibung}" erstellt werden. Bsp: "vt122106-10_CreateDbArtifacts"
Merge-Strategie mit master	Die Branches sollen alle mittel "**Rebase**" oder "**Squash-Merge**" mit dem Master gemergt werden. Dies verhindert unnötige Commits in der History, Rebase soll dem Squash vorgezogen werden, sofern die History vom Branch gut lesbar ist und den Entwicklungsprozess dokumentiert. Auch wenn mehrere Leute innerhalb vom gleichen Branch arbeiten, soll mit "git pull rebase" gearbeitet werden, um unnötige Merge Commits zu verhindern.
Reviews	Ein Branch darf nur in den master-branch gemergt werden, wenn min. 1 andere Person welche nicht Author vom Branch ist, den code reviewed hat und absegnet. Zusätzlich wird jeder Pull Request (oder in Gitlab "Merge Request) mittels CI/CD getestet. Dabei werden folgende Kriterien überprüft: • Kompillierfähig • Code Coverage darf 90% nicht unterschreiten • Alle UnitTests müssen grün sein

Test Package	Die Tests zu einem Package sollen im Package der zu testenden Logik + .tests sein und die Klase mit "Tests" enden. Bsp: Tests für ch.fhnw.ip12.pipeitup.TestClass ch.fhnw.pipeitup.tests.TestClassTests
Git LFS	Git LFS (Git Large File System) soll verwendet werden für binary Daten, wie z.b. Photos, .jar files, etc
Git Commits	Die Git Commits sollen kurz und prägnant beschreiben was geändert wurde, dabei wird nicht in der Vergangenheit geschrieben! Commits sollten atomar sein und wenn möglich vertikale Slices, und keine horizontale Slices darstellen (aka sie sollten revertbar sein ohne nachfolgende commits kompilierunfähig zu machen). Wenn die Commit Message zu lang wird, kann man eine leere Zeile einfügen und dann in Prosa beschreiben was gemacht wurde. Wichtig: Commit Messages sind zur Erklärung des Entwicklungsprozesses da. "Updated fileXy.java" bringt nichts, ist redundant da man dies ja bereits im Content des Commits sieht. Die Commit Message sollte die User-Story im Namen enthalten. Bsp commit message: VT122106-10: Setup Db connection string

2.4.2 CI / CD

CI/CD steht für "Continuous Integration / Continuous deployment". Dies bezeichnet ein automatisch Systme, welches auf Branches automatisch gewisse Aktionen ausführen kann.

Dazu zählt z.b. das automatische Deployen der Software wenn etwas neues in den master branch gemerged wird.

Die einzelnen CI/CD Prozesse werden **Pipelines** genannt. DIese werden innerhalb von GitLab gehostet und ausgeführt.

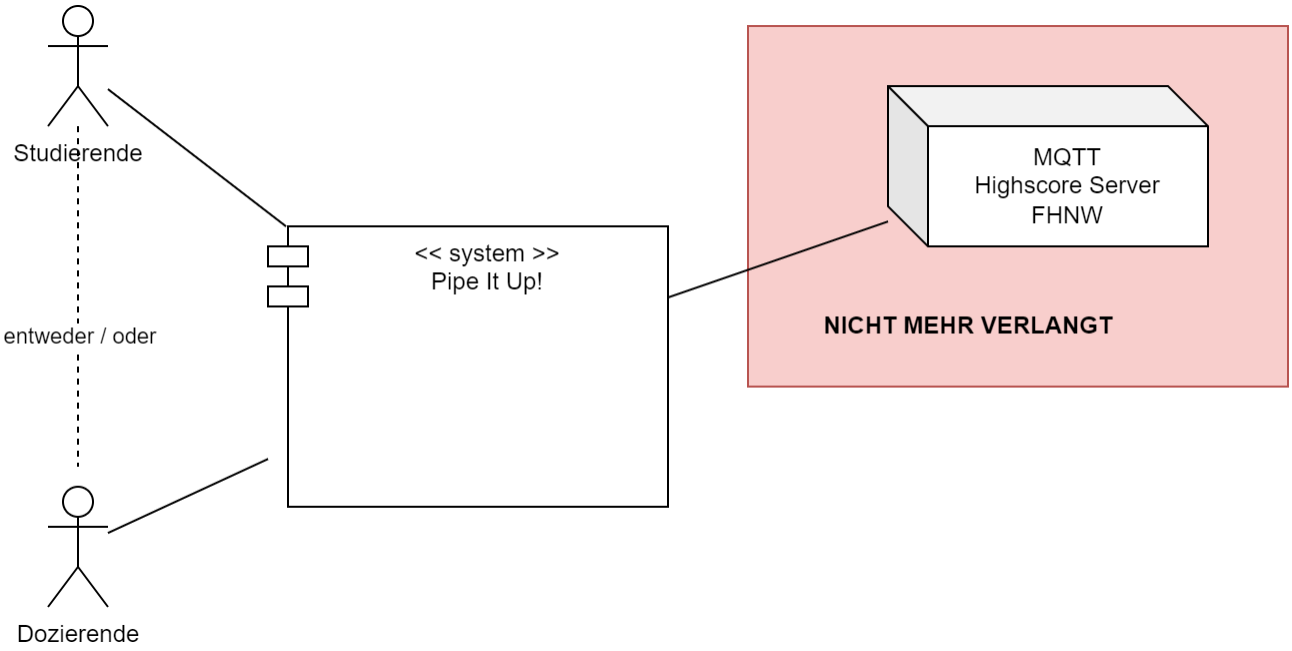
Dabei werden folgende Pipelines für das Projekt erstellt:

Pipeline	Beschreibung
CodeCoverage	Testet ob die CodeCoverage von einem git branch mindestens 90% ist
Deploy	Kompiliert & Deployt den aktuellen master branch auf den Raspberry Pi
UnitTestRunner	Lässt alle UnitTests laufen

3. Kontextabgrenzung

Dieses Kapitel beschreibt das Umfeld von Pipe It Up!. Für welche Benutzer ist es da, und mit welchen Fremdsystemen interagiert es?

3.1 Fachlicher Kontext



Studierende (Benutzende)

Die Studierenden verwenden Pipe It Up! zum Erlernen der gewichteten Graphentheorie. Spezifisch um die beiden Lösungs-Algorithmen "Kurskal" und "Prim" zu erlernen (zusammen mit dem Erklärmodus). Zudem können die Studierenden per Highscore im Spielmodus gegeneinander antreten und Challenges mit verschiedenen Graphen-Layouts probieren.

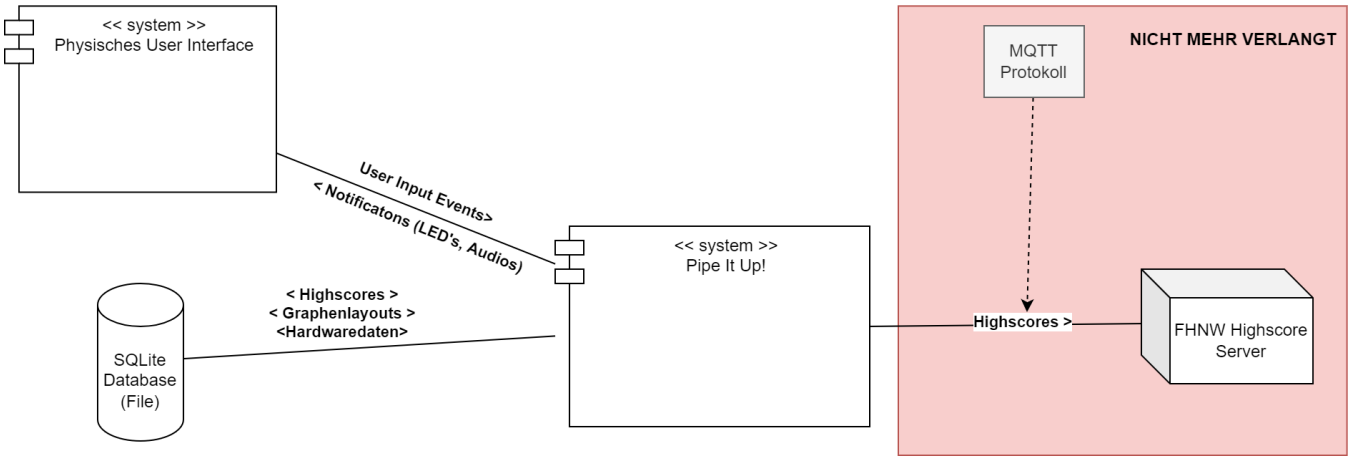
Dozierende (Benutzende)

Die Dozierenden benutzen das System im Unterricht zur didaktischen Vermittlung der gewichteten Graphentheorie. Ausserdem können die Dozierenden eigene Graphenlayouts definieren und speichern, Um gewisse Problematiken oder Challenges für den Unterricht zu definieren, welche die Studierenden dann zu lösen versuchen können.

MQTT Highscore Server der FHNW (Fremdsystem) *(nicht mehr aktuell)*

Die FHNW stellt einen Highscore Server zur Verfügung, an welchen man die Highscores über das MQTT Protokoll submitten kann. Diese Highscores werden dann benutzt um einen Sieger beim öffentlichen Abschlussevent des IP12 Projektes zu küren.

3.2. Technischer Kontext



MQTT Highscore Server der FHNW (Fremdsystem) *(nicht mehr aktuell)*

Siehe Kapitel 3.1

SQLite Database (Fremdsystem)

Für persistente Datenspeicherung wird eine SQLLite Datenbank verwendet. Dabei werden folgende Daten in der Datenbank gespeichert:

- **Highscore:** Eine Liste von Spielern und erreichter Punktzahl im Spielmodus
- **Graphenlayouts**: Vordefinierte (oder selbst definierte) Layouts für den Spielmodus (nicht Teil des ersten MVP)
- **Hardwaredaten**: Benötigte Daten zu Hardwareadressen von Buttons, LEDs und Port Expanders.

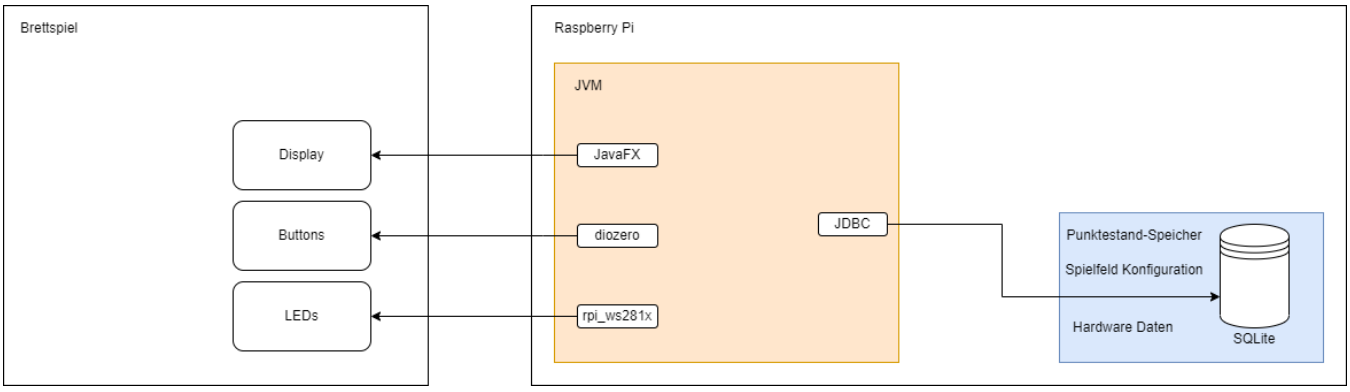
4. Lösungsstrategie

4.1 Einführung in die Strategie

Die folgende Tabelle ergänzt die Qualitätsziele von Pipe It Up! Aus Abschnitt 1.2 mit passenden architektonischen Entscheidungen.

Qualitäts Ziel	Lösungsentscheidung
Physisches (Brett)-Spiel / Robustheit	• Das Spiel wird in Form eines physischen Brettspieles umgesetzt. • Damit es stabil ist, wird der Rahmen aus Holz konstruiert • An wichtigen, strukturellen Punkten soll die Konstruktion stabilisiert werden. • Es muss noch Platz zum Verbauen der Elektronik haben
Software UI	• Simulation des physischen Brettspieles • Implementation eines GUI-Interfaces • Austauschbar mit physischem UI zur Lauf- oder Kompilationszeit • Dient auch als Ausweichslösung, falls der Zusammenbau des physischen Spiels zu komplex wird.
Robuste Softwarelösung	• Muss mit Hardware und Softwarefehler umgehen können • Logging aller Exceptions, und Nachrichten auf den Leveln "Info", "Warning", "Error", "Critical" • Sollte aber nicht zu viel loggen, sonst ist Lebensdauer der SD-Karte stark eingeschränkt • Soll trotz Exceptions noch weiterhin (eingeschränkt) funktionieren
Schnelles Feedback auf UserInput	• Das System muss sich Responsive anfühlen • UI in eigenem Thread laufen lassen • Effiziente Implementierung von Prim und Kruskal
Einfache Installation	• Alles über einen Stromstecker mit Strom versorgen

Eine grobe Skizze der Umsetzung, mit Libraries, Fremdsystemen und physischen Geräten sieht wie folgt aus:



4.2 Aufbau von Pipe It Up!

Pipe It Up! Wird als Java Applikation dass über eine main Methode gestartet wird realisiert. Es besteht grob aus folgenden Teilen:

- User Interface Layer (FrontEnd)
  - Physisches Brettspiel

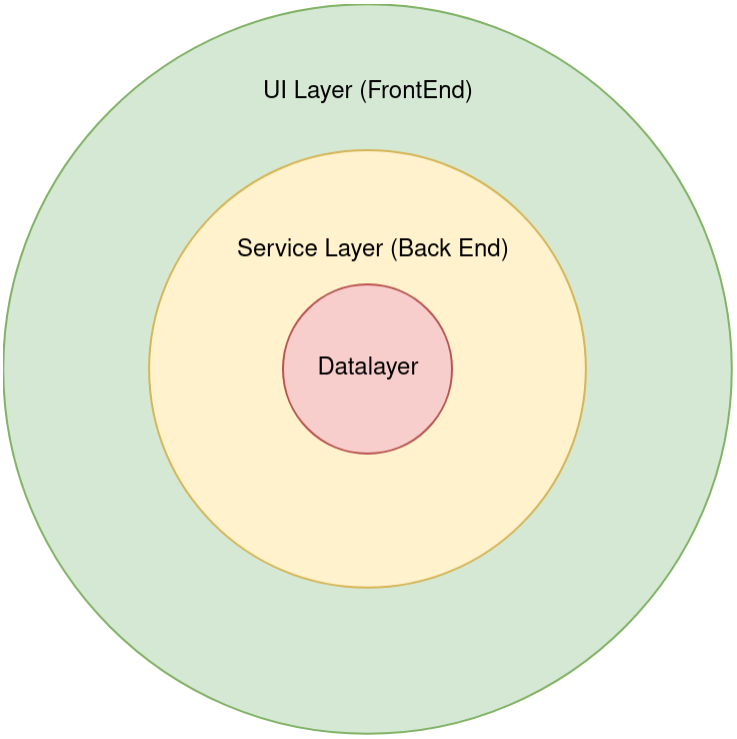
    - Software UI
- Service Layer (BackEnd)
    - Spielmodus (Logik)
    - Erklärmodus (Logik)
    - Prim Implementation
    - Kruskal Implementation
- Datenbank (Data Layer)
    - Daten persistieren / laden von Datenbank
    - Hardwareadressen für Implementierung des Userinterfaces
    - Highscore

Diese Zerlegung minimiert die Abhängigkeiten zwischen den einzelnen Teilen, und ermöglicht es die Implementation einzelner Teile auszutauschen. Der Aufbau orientiert sich am SOLID Principle und zwischen den einzelnen Teilen soll nur mittels Interfaces kommuniziert werden. Die Befolgung des SOLID Prinzips erlaubt es auch einfach automatisierte Tests für den Code zu schreiben.

Der Informationsaustausch zwischen den einzelnen Teilen wird über Value-Objekte / Models sichergestellt. Dies sind "dumme" Objekte die keine Logik, sondern nur Daten beinhalten. Heisst nur read-only Felder und sonst keine Methoden / Logik. Die einzelnen Value-Objekte werden dabei in dem Teil definiert, der sie auch "befüllt".

Alles wird wenn möglich über **Interfaces** implementiert, vor allem zwischen den einzelnen Teilen werden nur Interfaces zur Kommunikation genutzt.

Die Architektur folgt dabei ebenfalls dem Zwiebel-Schalen-Modell (teilt Systemarchitektur in verschiedene Layers auf):



Ein Layer darf immer nur auf Objekte / Schnittstellen der direkt untergeordneten zugreifen. Dies stellt sicher dass die Single Responsibility aus dem SOLID Prinzip zwischen den Teilen nicht verletzt wird. Das ausführende Programm (welches eine statische Main Methode besitzt) wird dann die einzelnen Schichten Instanzieren,

Mittels Dependency Injection wird diese Abgrenzung erzwungen. Jeder Layer hat seine eigene Dependency Injection Konfiguration (Mapping von Interface X wird implementiert von Klasse Y) was es ermöglicht, die Implementation Package intern zu halten und gar nie für Aussenstehende Packages sichtbar zu machen.

4.3 Raspberry Pi

Der Raspberry Pi wird so konfiguriert, dass er automatisch mit bekannten wifi's verbindet. Die bekannten WIfis können im setup.shFile (https://gitlab.fhnw.ch/ip12-21vt/ip12-21vt_pipelinesystem/pipe-it-up/-/blob/master/pi/setup.sh) eingetragen werden mittels der "SetupKnownWifi" Funktion (siehe Ende des files).

Dies erleichtert die Entwicklung und Problembehandlung des Gerätes.

Des weiteren kann man die aktuelle Version von Pipe It Up auf dem Raspberry Pi direkt mit einem einfachen Shell Script updaten und einrichten.

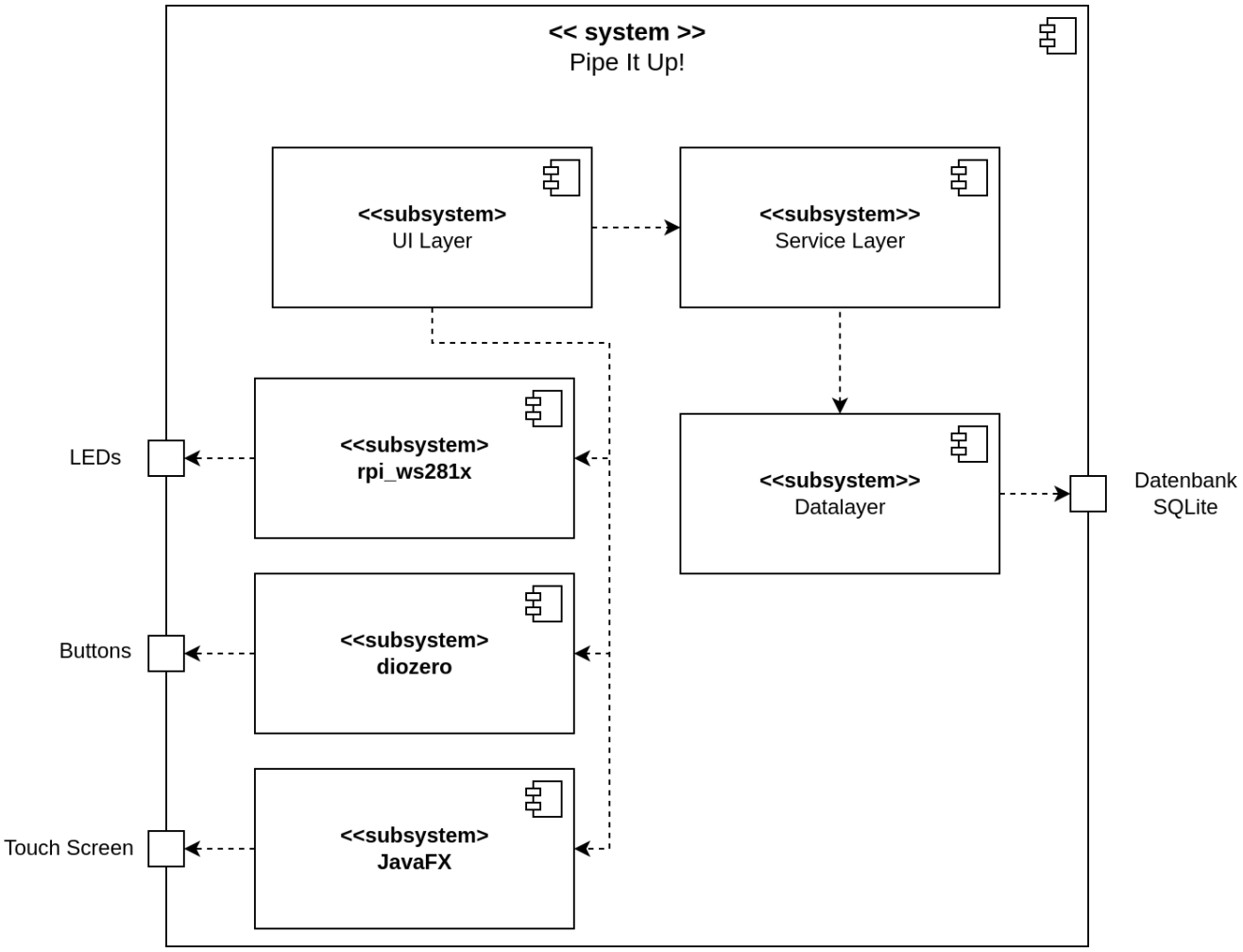
Das Bash-Skript sollte auch auf allen apt basierten Linux-Systemen funktionieren (wie Ubuntu, Mint, ...)

Das Setup Script kann hier gefunden werden: https://gitlab.fhnw.ch/ip12-21vt/ip12-21vt_pipelinesystem/pipe-it-up/-/tree/master/pi

5. Bausteinsicht

5.1 Systemansicht Ebene 1

Pipe It Up! zerfällt wie im Diagramm unten dargestellt in mehrere Subsysteme. Die gestrichelten Pfeile stellen fachliche Abhängigkeiten der Subsysteme untereinander dar (“x -> y” für “x ist abhängig von y”). Die Kästchen auf der Membran des Systems sind Interaktionspunkte mit Außenstehenden.



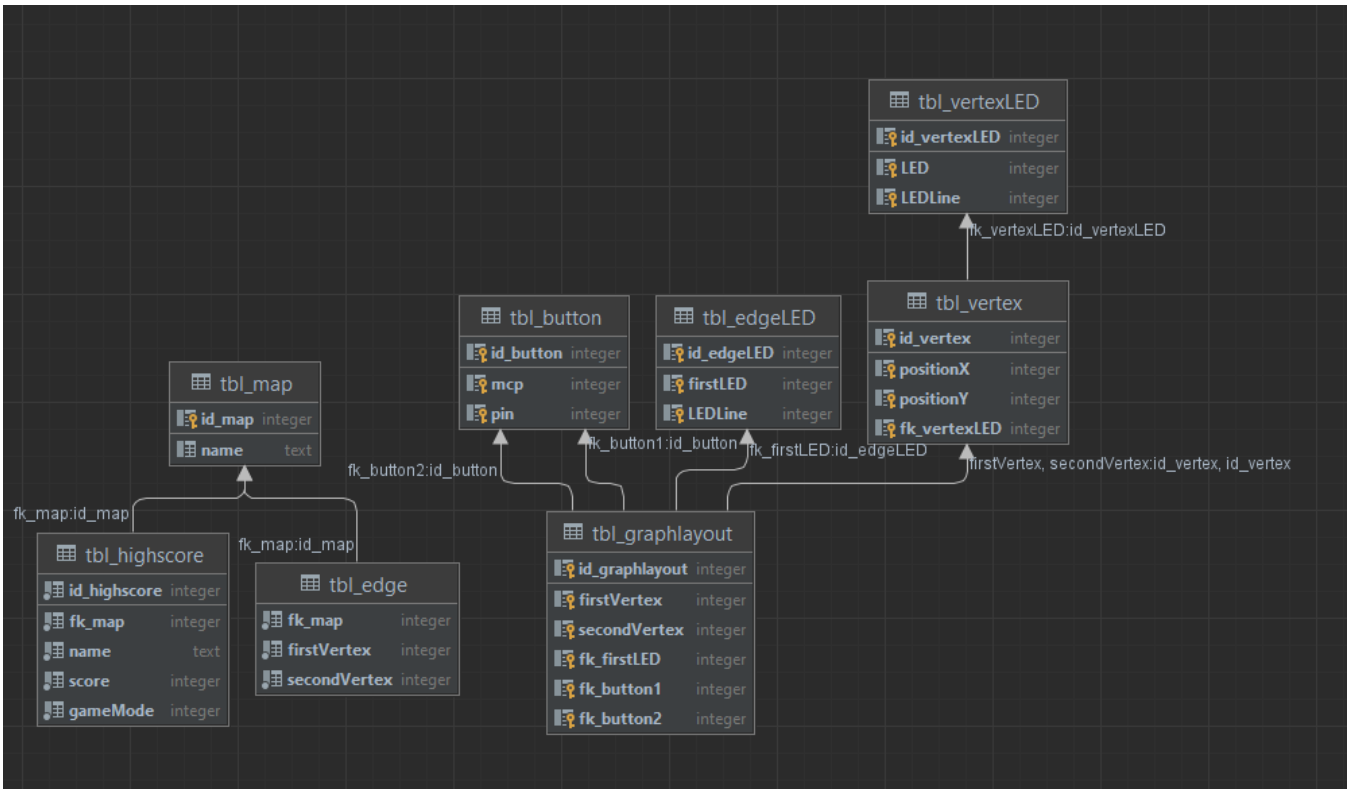
5.2 Data-Layer

Der Data Layer ist zuständig für das Laden und Speichern von Daten wie Graphen-Layouts und Highscores.

Die Java Klassen dazu befinden sich im Java Sub-Package "data".

5.2.1 Datenbank

In der Datenbank werden verschiedene Daten zum Spiel gespeichert.



- **tbl_button:** Enthält MCP und Pin der Buttons (siehe Elektronik).
- **tbl_edgeLED:** Enthält die Hardwareadressen der LEDs, welche Kanten sind (siehe Elektronik).
- **tbl_vertexLED:** Enthält die Hardwareadressen der LEDs, welche Knotenpunkte sind (siehe Elektronik).
- **tbl_graphlayout:** Enthält das feste Layout des Spiels (alle Kanten und Knoten), sowie FKs zu den Hardwaredaten der jeweiligen Kante.
- **tbl_vertex:** Enthält die Koordinaten der Knotenpunkte, welche für das Software-UI benötigt werden, sowie einen FK zu den LED-Hardwaradressen des Knotenpunktes.
- **tbl_map:** Enthält vorgefertigte Karten mit fixem Layout (nicht Teil des ersten MVP)
- **tbl_edge:** Enthält die Kanten-Daten für die vorgefertigte Karten (nicht Teil des ersten MVP)
- **tbl_highscore:** Enthält die Highscore-Daten. (Muss nicht zu tbl_map gebunden sein, aber kann)

5.3 Logic-Layer

Im Logic-Layer findet sich sämtliche Business Logik die für das Spiel benötigt wird.

Es beinhaltet

- Prim und Kruskal Algorithmen
- Generieren / Laden von Zufallsgenerierten Karten
- Speichern / Laden von Highscore Einträgen (Weitergabe an den Daten-Layer)

Lädt Daten durch die Interfaces definiert im Data-Layer.

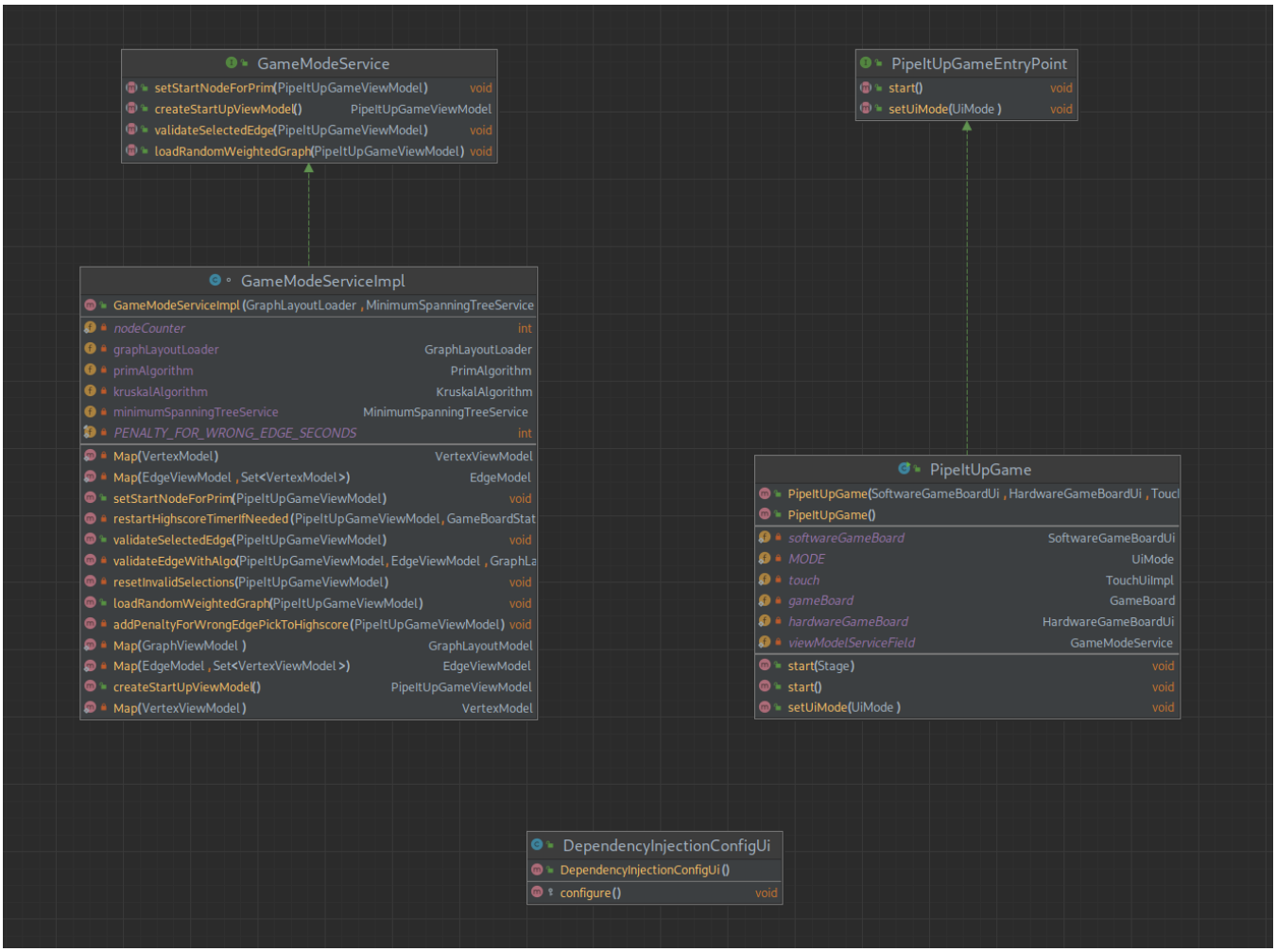
5.4 UI-Layer

Der UI-Layer ist selbst noch einmal in zwei Teile aufgeteilt:

- Der Presentation-Model / UI-Logik Teil
- Die Views

5.4.1 UI-Logik Teil

Der UI-Logik ist im Package ***ch.fhnw.ip12.pipeitup.ui*** zu Finden und besteht vor allem aus den folgenden Klassen:



- **GameModeService**:
  - Orchistrierungsservice welcher die UI-Logik für das Pipe-It-Up Game enthält und das ViewModel (oder auch Presentation Model) erstellt und managed
    - Schnittstelle des UI's zum Logik Layer
- **PipeItUpGame**:
  - Entry-Point des Java-FX UI's
  - Startet die jeweiligen UI's wie
    - TouchUI
    - SoftwareGameBoard
    - HardwareGameBoard
  - Stellt sicher dass die Methode "stop()" auf dem jeweiligen Gameboard aufgerufen wird beim Beenden der Applikation.

5.4.2 Views

Sämtlicher Code der mit den Views zusammenhängt ist im Package ***ch.fhnw.ip12.pipeitup.ui.views*** zu finden

Dabei wurde ein Sub-Package pro View-Typ erstellt. View-Typen sind:

- **touch**:
  - Das Java-FX GUI welches auf dem Touchscreen angezeigt wird
  - Pro FXML View (Im Resources Ordner) gibt es einen Controller im Sub-Package "controllers"
- **gameboard**:
  - Das Frontend für die Darstellung des Gameboards (Graphenlayout). Dabei gibt es zwei verschiedene Implementierungen, jeweils wieder in eigenen Sub-Packages:
    - **software**: Die Software Implementation des Graphen-Frontends als Java-FX GUI
    - **hardware**: Implementation für das Brettspiel, mit der Steuerung der LED's und Buttons
  - Alle Implementation nutzen das gleiche View-Model welches vom GameModeService erstellt und gemanaged wird.

Alle Front-Ends werden Event-basiert über Listeners aus Java-FX Properties gesteuert.

5.4.2.1 Touch

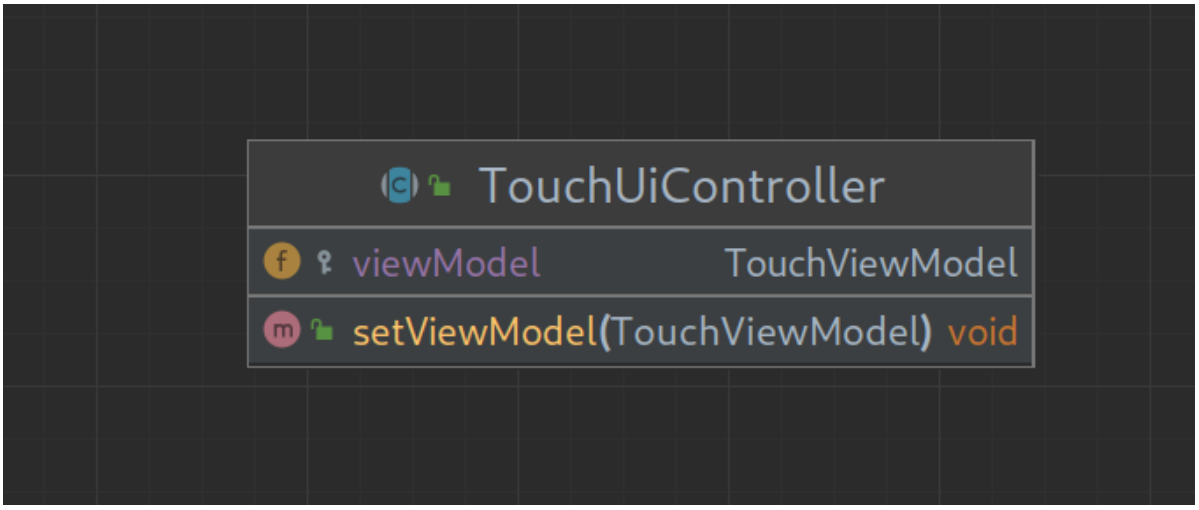
Als Entry Point für das Touch UI gilt das Interface ***TouchUi*** bzw dessen Implementation ***TouchUiImpl***.

Diese Klasse übernimmt das Mapping vom ***TouchScene*** enum auf fxml Files und rendert das Java-FX UI

Für jede Scene im Touch UI gibt es einen Eintrag im ***TouchScene*** Enum!

Die Controllers die jeweils zu einer View gehören, beinhalten dann View-spezifischen Code, wie ***onButtonClick()*** Methoden / Events und ändern dabei den Zustand des ViewModels.

Das View Model wird dabei jedem Controller beim Laden der Scene durch ***TouchUiImpl*** übergeben. Damit dies möglich ist wurde eine abstrakte Klasse ***TouchUiController*** erstellt:



Jeder Controller erbt von dieser abstrakten Klasse, und ***TouchUiImpl*** setzt beim Laden der Scene mittels "setViewModel".

So hat man im jeweiligen Controller dann Zugriff auf das ViewModel über ***this.viewModel***

5.4.2.2 Gameboard

Die View für das Gameboard (ob Software oder Hardware gesteuert) ist lediglich dazu da den Inhalt aus dem ***GameBoardViewModel*** darzustellen. Abgesehen davon sollten die Views / Controllers keinerlei Logik enthalten.

Dies ermöglicht ein einfaches Austauschen des FrontEnds, falls z.b. mal ein neues Gerät gebaut wird, LED's oder Buttons ausgetauscht werden, etc.

Software-Gameboard

Das Software-Gameboard ist ein Java-FX GUI welches per Canvas den Graphen rendert. Um Knoten und Kanten werden Hitboxen berechnet, welche beim mouse-click dazu dienen zu überprüfen was angeklickt wurde.

Um das Software-UI zu starten muss man PipeItUp mit dem Konsolen Parameter ***--softwareUi*** starten.

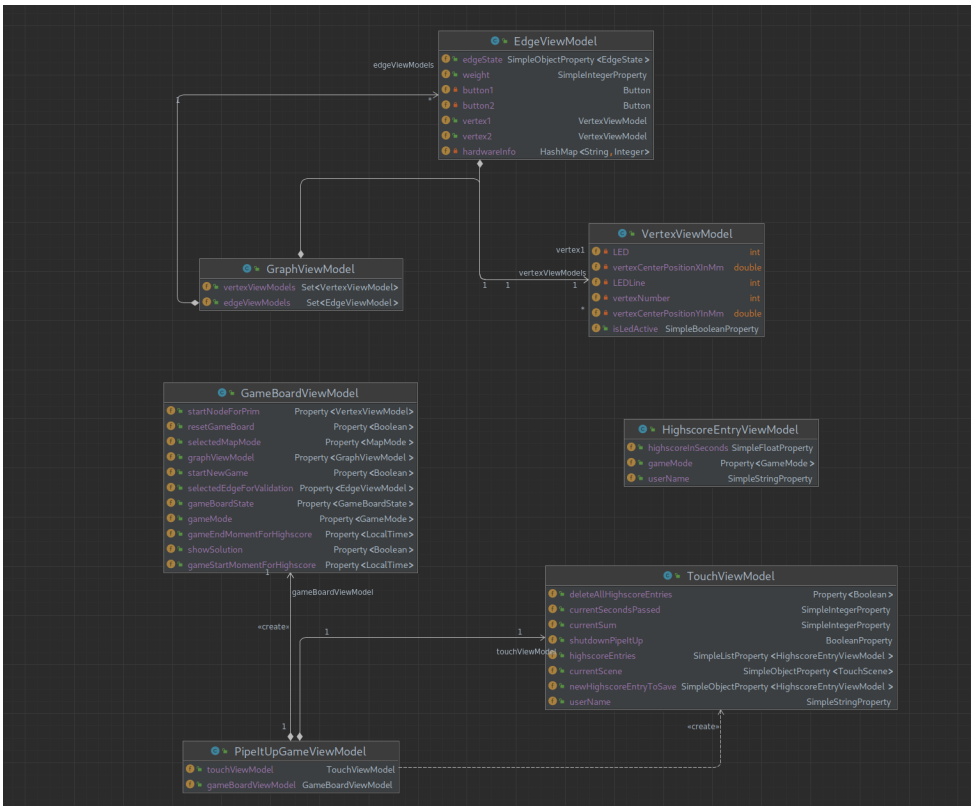
Hardware-Gameboard

Diese Implementation benutzt die verbunden LED's und Buttons aus dem Brettspiel für die Anzeigen und Steuerung des ***GameBoardViewModel***. Für mehr Informationen zu der Elektronik siehe Elektronik

5.4.3 ViewModel

Das ViewModel dient als Presentation Model für die Views. Dabei wird unterteilt in **TouchUiViewModel** und **GameBoardViewModel**.

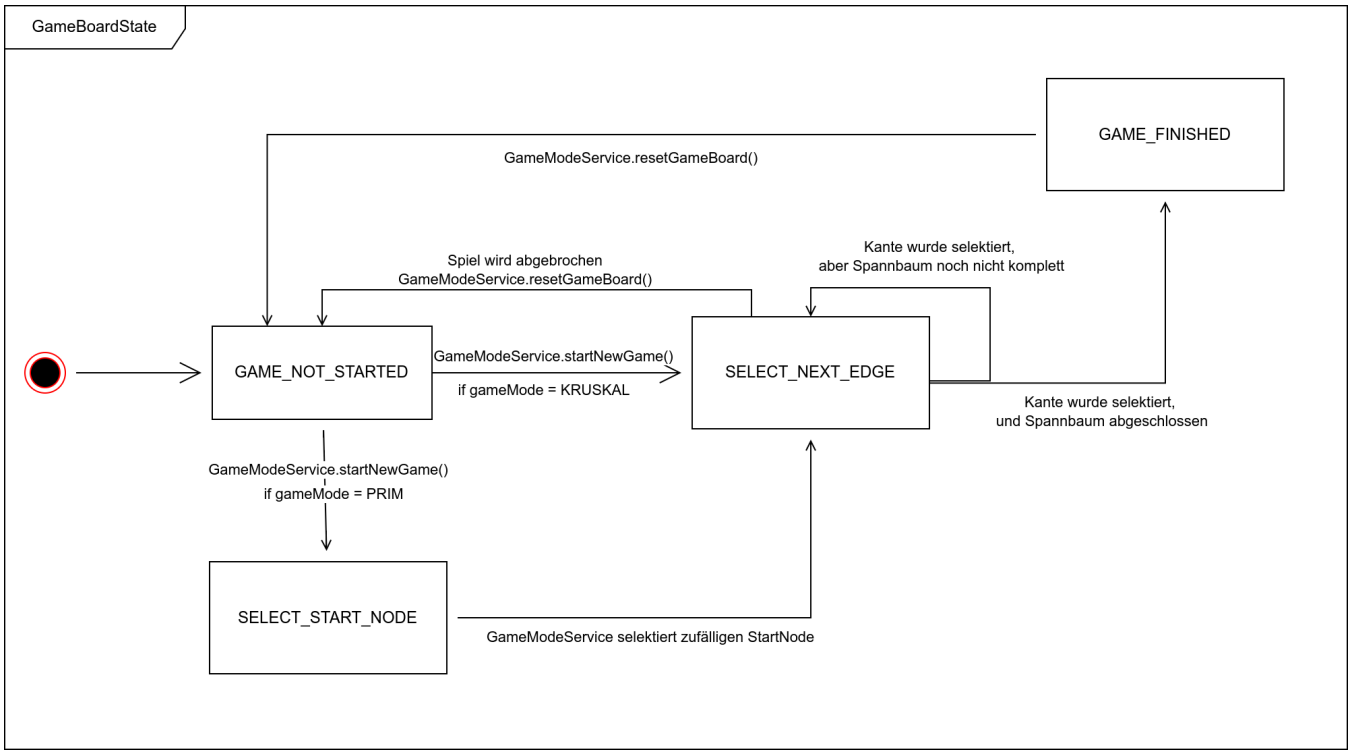
Der GameModeService hat listeners auf dem ViewModel, um auf Requests / User Input reagieren zu können.


EdgeViewModel
edgeState SimpleObjectProperty <EdgeState >
weight SimpleIntegerProperty
button1 Button
button2 Button
vertex1 VertexViewModel
vertex2 VertexViewModel
hardwareInfo HashMap <String , Integer>
edgeViewModels
VertexViewModel
LED int
vertexCenterPositionXInMm double
LEDLine int
vertexNumber int
vertexCenterPositionYInMm double
isLedActive SimpleBooleanProperty
vertex1
vertexViewModels
GraphViewModel
vertexViewModels Set<VertexViewModel>
edgeViewModels Set<EdgeViewModel>
GameBoardViewModel
startNodeForPrim Property <VertexViewModel>
resetGameBoard Property <Boolean>
selectedMapMode Property <MapMode >
graphViewModel Property <GraphViewModel >
startNewGame Property <Boolean>
selectedEdgeForValidation Property <EdgeViewModel >
gameBoardState Property <GameBoardState >
gameMode Property <GameMode >
gameEndMomentForHighscore Property <LocalTime>
showSolution Property <Boolean>
gameStartMomentForHighscore Property <LocalTime>
HighscoreEntryViewModel
highscoreInSeconds SimpleFloatProperty
gameMode Property <GameMode >
userName SimpleStringProperty
TouchViewModel
deleteAllHighscoreEntries Property<Boolean>
currentSecondsPassed SimpleIntegerProperty
currentSum SimpleIntegerProperty
shutdownPipeItUp BooleanProperty
highscoreEntries SimpleListProperty <HighscoreEntryViewModel >
currentScene SimpleObjectProperty <TouchScene>
newHighscoreEntryToSave SimpleObjectProperty <HighscoreEntryViewModel >
userName SimpleStringProperty
gameBoardViewModel
«create»
touchViewModel
«create»
PipeItUpGameViewModel
touchViewModel TouchViewModel
gameBoardViewModel GameBoardViewModel

6. Laufzeitsicht

6.1. *Statemachine GameBoardState*

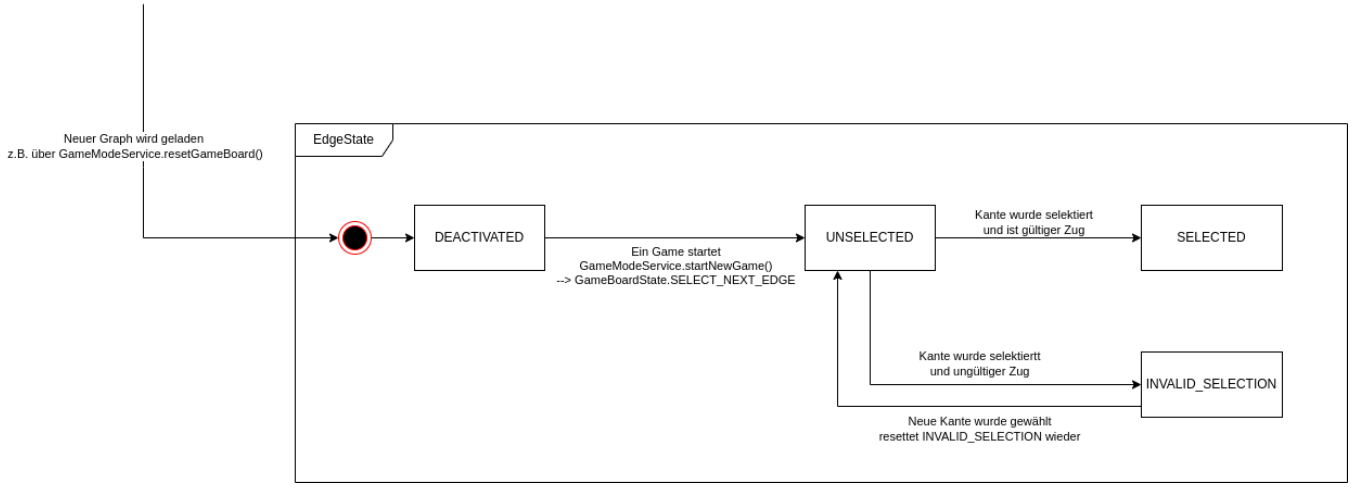
Der GameBoardState beschreibt den aktuellen Zustand in welchem sich das Spielfeld befindet, bzw beschreibt in welcher Spielphase man sich aktuell befindet und hat folgende Statemachine:



6.2. *Statemachine EdgeState*

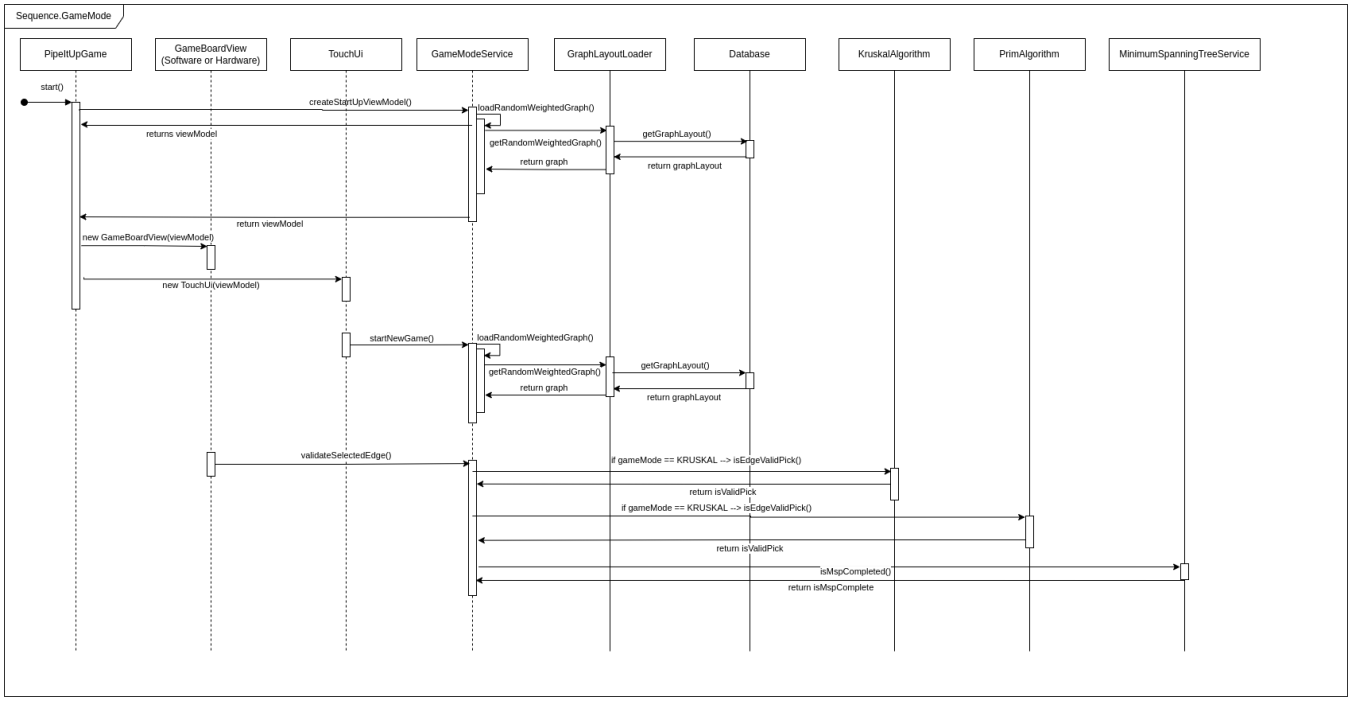
Die Kanten besitzen ebenfalls eine eigene Statemachine für die Abbildung des Zustandes einer Kante. Dazu kann der Enum "EdgeState" folgende States annehmen:

- **DEACTIVATED**: Eine Kante die deaktiviert ist (kein Gewicht anzeigen soll bzw in einem neutralen Zustand ist)
- **UNSELECTED**: Kante zeigt Gewicht an, ist jedoch unselektiert (LEDs leuchten blau)
- **SELECTED**: Kante zeigt Gewicht an und wurde selektiert (LEDs leuchten grün)
- **INVALID_SELECTION**: Kante zeigt Gewicht an und wurde fälschlicherweise selektiert (LEDs leuchten rot)
- **BLINKING**: Die Kante zeigt das Gewicht an und die LEDs blinken dabei in einer goldenen / gelben Farbe. (Wird nur im Erklärmodus benutzt, hat im Spiel keine Funktion)



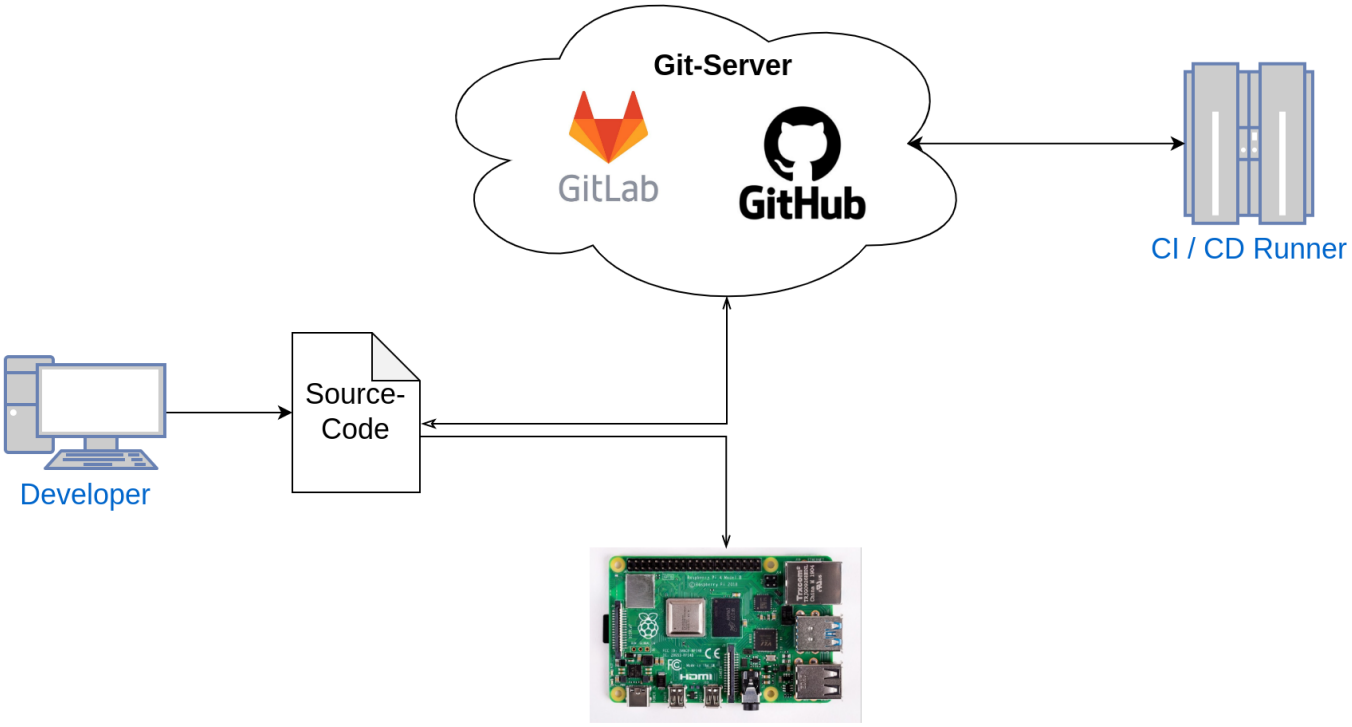
6.3. Laufzeitdiagramm Spielmodus

Eine grobe Übersicht über den Ablauf des Spielmodus sieht wie folgt aus:



7. Verteilungssicht

7.1 Infrastruktur Ebene 1



Raspberry Pi 4B

Begründung	*Das pushen des Codes auf den Raspberry Pi passiert nicht über die CI/CD Pipeline automatisch, sondern muss manuell getriggert werden.* *Der Grund dafür ist, dass der Raspberry Pi selbst kein Internet hat und nicht immer online ist. Hierzu wurde ein Maven Profil mit dem Namen "Install" erstellt, welches den Code auf den Pi hochlädt vom aktuellen PC aus (Bedingung ist, dass man im gleichen WLAN / Netzwerk ist wie der Raspberry Pi und dessen IP kennt).* *DIe CI/CD Runner überprüfen ob ein gepushter Code funktioniert und die Code Coverage mindestens 80% ist. Wenn nicht kann man seinen Merge Request nicht mit dem Master Branch mergen.*
Qualitäts- und /oder Leistungsmerkmale	• Nutzung von CI/CD für Code Coverage und Sicherstellung der Kompillierfähigkeit • Raspberry Pi 4 braucht keine Internetverbindung • Versionierung durch Git (Gitlab oder Github)

7.2 Verteilungssicht Ebene 2

7.2.1 Raspberry Pi 4

Das Setup-Script setzt den Raspberry Pi wie folgt auf:

Pfad	Beschreibung
~/packages/PipeItUp	Klon des Git-Repositories von PipeItUp (https://github.com/RononDex/PipeItUp)
/etc/sudoers	Überschreibt die Default sudoers mit unserer custom Config.
/etc/xdg/autostart/pipeitup.desktop	Autostart Config, welches beim Starten des Pi4 aufgerufen wird.

Maven Install kreiert folgende Ordner & Files:

Pfad	Beschreibung
~/deploy	Beinhaltet alle distibution Jars, welche für die Ausführung des Games benötigt werden.

8. Konzepte

8.1 Dependency Injection

Sämtliche Klassen innerhalb von Pipe-It-Up können in folgende Kategorien aufgeteilt werden:

- **Value-Objects**: "Dumme" Objekte ohne jegliche Logik (mit Ausnahme von getter / setter). Dienen zum Halten von Daten / Zustände und Datenaustausch zwischen verschiedenen Services
- **Services**: Stateless-Klassen, die Methoden enthalten und deklarierte Interfaces implementieren. Stateless heisst konkret, dass Abhängigkeiten auf andere Services durch Konsturktorparameter deklariert werden (und nie mit ... = new ...() selbst initialisiert werden) für Dependency Injectio. Zudem heisst es, dass die Klassen keine States haben, bzw. wenn eine Aktion auf ein Value-Objekt ausgeführt werden soll, so ist das Value-Objekt ebenfalls Teil der Parameterliste der Methode, und wird nicht z.b. von einem globalem Feld referenziert.

Dies ermöglich es die Abhängigkeiten zwischen Services über den Konstruktor auszudrücken. Z.b.

```
        private MinimumSpanningTreeService minimumSpanningTreeService;

        @Inject
        public PrimAlgorithmImpl(MinimumSpanningTreeService minimumSpanningTreeService) {
                this.minimumSpanningTreeService = minimumSpanningTreeService;
        }
```

Mit dem **@Inject** wird dem Dependency Injection Framework mitgeteilt, dass es diesen Konstruktor verwenden darf um eine selbstständig eine Instanz zu erzeugen wenn es benötigt wird.

Ein weiterer Vorteil von Dependency Injection ist, dass nur die Interfaces public sein müssen, die Implementation können package-private sein und müssen von aussen nicht sichtbar sein. Dadurch wird eine höhere Stufe an Kapselung aus dem SOLID Prinzip erreicht und die Aufrufende Komponente hat direkt keinen Zugriff auf die Implementierung.

Jedes Interface, zusammen mit dessen Implementation, wird pro Layer in einer Dependency Injection Config registriert. Dabei wird Guice als Dependency Injection Framework genutzt.

Die Konstruktoren der Service-Klassen werden mit dem Attribute @Inject versehen, um dem Framework zu signalieren die Abhängikeiten sollen aufgelöst und automatisch bei Bedarf mitgegeben werden.

Eine solche Dependency-Injection config kann dann wie folgt aussehen:

```
@ExcludeTypeFromJacocoGeneratedReport
public class DependencyInjectionConfigLogic extends AbstractModule {

        @Override
    protected void configure() {
        bind(GraphLayoutLoader.class).to(GraphLayoutLoaderImpl.class);
        bind(MinimumSpanningTreeService.class).to(MinimumSpanningTreeServiceImpl.class);
        bind(PrimAlgorithm.class).to(PrimAlgorithmImpl.class);
        bind(KruskalAlgorithm.class).to(KruskalAlgorithmImpl.class);
    }

}
```

Das heisst es wird immer ein Mapping zwischen dem Interface und der Implementation gemacht. Bzw. das Dependency Injection Framework (Guice) weiss nun so, wo die Implementation eines Interfaces zu finden ist und kann dieses Instanzieren wenn ein Service im Konstruktor nach einer solchen Abhängikeit verlangt.

8.2 Unit Testing mit Mocking

Siehe Code Testing

8.3 Schichten Layering

Zwischen den einzelnen Layers wird jeweils immer nur über Interfaces kommuniziert. Die Implementationen der Interfaces werdem immer package private deklariert.

Nach aussen sollen nur Interfaces sichtbar sein! Das Dependency Injection Framework (Guice) wird dann die Implementation der Interfaces dem Konstruktor von Services übergeben.

9. Entwurfsentscheidungen

9.1 Wie wird Kompatibilität und Identische Funktionsweise vom Physischen Board UI und vom JavaFX Software GUI sichergestellt?

Zur Fragestellung:

Für die einfachere Entwicklung wird ein Software GUI benötigt. Dies erlaubt es Code zu simulieren und zu testen, ohne dass man das physische Brettspiel bei sich lokal bereithalten muss. Dies erlaubt es das Parallelisieren des Programmieraufwandes und gestaltet gleichzeitig die Qualitätssicherung einfacher. Zudem erlaubt es bei einem Total-Ausfall des Brettspieles, das Spiel immerhin noch in Software Form zu nutzen auf einem PC.

Wie wird jedoch sichergestellt, dass eine Code Änderung sich auf dem physischen Brettspiel und dem Software UI gleich verhält?

Relevante Einflussfaktoren:

- Qualitätsziel: Software UI
- Qualitätsziel: Robuste Software Lösung
- Randbedingungen
- Code-Architektur

Annahmen

- Es wird nur ein physisches Brettspiel entwickelt, welches Parallelisierung beim Programmieren schwierig macht.
- Das physische Brettspiel wird irgendwann, wegen mechanische Abnutzung, nicht mehr betriebsfähig sein.

Betrachtete Alternativen

- Keine Software-UI Lösung anbieten
- Nur mit Console-Output für Entwicklungszwecke arbeiten
- Qualität nur mit Test-Coverage sicherstellen.

Entscheidungen Software

Die Qualität nur über Test-Coverage sicherzustellen wurde ziemlich schnell als nicht ausreichend identifiziert. Das Problem bei dieser Alternative wäre, dass auch 100% Code Coverage heisst nicht zwingend ass man alle Eventualitäten abgedeckt hat. Desweiteren können auch beim Schreiben der Unit-Tests Fehler passieren, was dann zu falschem Qualitätsbewusstsein führen würde.

Da blieben noch die zwei Varianten mit dem Console-Output und einem JavaFX UI. Das heisst es wurde zwingend nötig, dass man die UI Logik strikt von jeglicher Businesslogik trennt. Diese Trennung erreicht man durch das Beschrieben Zwiebelschalen-Modell in 4. Lösungsstrategie - IP12-21vt_Pipelinesystem - fhnw Confluence20.

Mit Console-Output ist man jedoch stark limitiert in der Darstellung. Da sowieso schon JavaFX für die Darstellung auf dem integrierten Touchscreen verwendet wird, liegt JavaFX nahe, denn die Technologie wird so sowieso schon im Projekt eingesetzt.

Die JavaFX erlaubt es gleichzeitig auch das Produkt Pipe-It-Up! auch im Falle eines totalen Ausfalls des physischen Brettspiels weiterhin zu nutzen. Mit der Entscheidung für JavaFX werden somit gleich zwei Ziele erfüllt: das Parallelisieren der Programmier-Aufgabe und Sicherstellung der Langlebigkeit des Produktes.

9.2 Library zur Hardware-Steuerung

Für die Steuerung der LEDs haben wir uns für das "rpi-ws281x-java"-Library entschieden. Es wurden mehrere Libraries in der ersten Projektwoche angeschaut, viele funktionierten jedoch nicht mit Java 11 oder hatten wenig Funktionen. Das gewählte Library ist speziell für den Raspberry Pi und unseren LED-Stripe-Typ ausgelegt.

Zur Steuerung der Buttons und IC Expanders wurde das "diozero"-Library gewählt. Es ist bei weitem das am umfangreichste Library, dass wir gefunden haben. Zudem ist eine gute Dokumentation des Libraries, was bei anderen Libraries mangelnd war. In der Dokumentation wird die Installation und Verwendung ausführlich und mit Beispielen gut erklärt. Die Ansteuerung von Buttons über die IC Expanders wurden mit dem Library erfolgreich getestet.

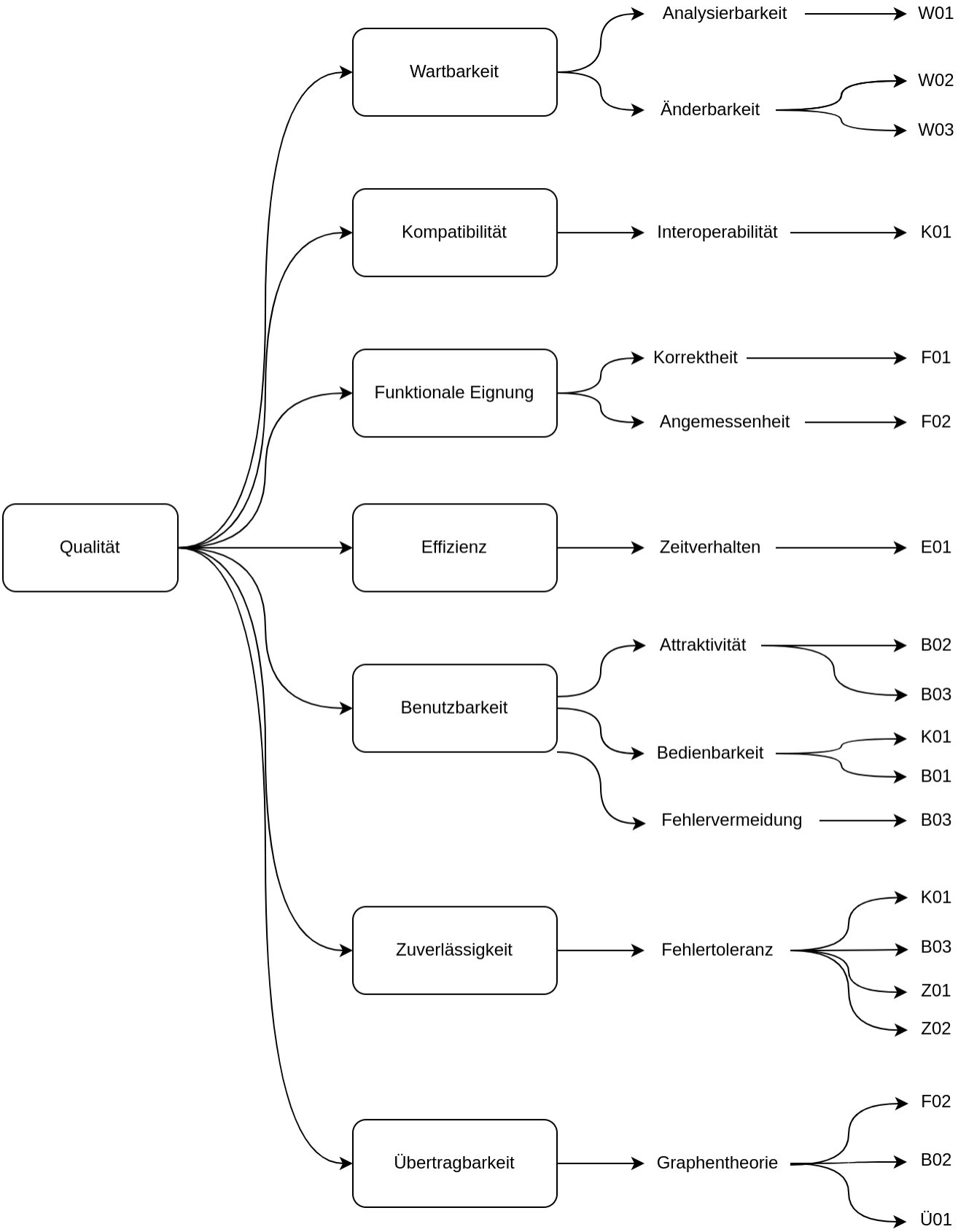
9.3 Datenbank

Die Datenbank speichert neben dem Highscore nur statische Daten. Wir haben uns entschieden, diese Daten trotzdem in der SQLite Datenbank zu speichern. Dadurch ist alles in einem File und die jeweiligen Daten können trotzdem durch die Auswahl des Tables gezielt ausgelesen werden. Zudem wird die Performance nicht einschränkt, da SQLite die Daten direkt von der Datei ausliest und keine Serververbindung nötig ist, wie bei herkömmlichen Datenbanken.

Mit der Kundin wurde evaluiert, dass sich der Aufwand, einen Filter für die Namen in der Highscoretabelle zu implementieren, nicht lohnen würde. Das Spiel wird grösstenteils von der Kundin überwacht. Zudem ist die Highscoretabelle zurücksetzbar.

10. Qualitätsanforderungen

10.1. Qualitätsbaum



10.2. Qualitätsszenarien

Die Anfangsbuchstaben der Bezeichner (IDs) der Szenarien in der folgenden Tabelle stehen jeweils für das übergeordnete Qualitätsmerkmal, W beispielsweise für Wartbarkeit. Diese Bezeichner finden auch im Qualitätsbaum Verwendung. Nicht immer lassen sich die Szenarien eindeutig einem Merkmal zuordnen. Sie treten daher mitunter mehrmals im Qualitätsbaum auf.

ID	Szenario
W01	Jemand mit Grundkenntnissen in Java mit JavaFX möchte das Pipe-It-Up! verstehen. Die Person kann innerhalb von maximal 30 Minuten relevante Codestellen identifizieren.
W02	Falls der eingebaute Raspberry Pi kaputt geht, kann man mit max. 2h Aufwand einen neuen Raspberry pi installieren.
W03	Zur einfachen Wartung soll der integrierte Raspberry Pi mit bekannten WLAN-Verbinden oder einen Hotspot starten, falls kein bekanntes WLAN gefunden wird.
K01	Im Falle dass das Brettspiel z.b. durch Abnutzung defekt ist, kann eine Software Repräsentation des Spielfeldes genutzt werden um das Pipe-It-Up! zu spielen.
F01	Pipe-It-Up! bewertet Spielzüge gemäss dem Algorithmus von Prim bzw Kruskal und meldet gemachte Fehler dem Spielenden.
F02	Der Erklärmodus erlaubt es einem Studierenden, jeweils den Prim- und Kruskal-Algorithmus in maximal 10 Minuten zu verstehen.
E01	Die Darstellung auf dem Touch-Screen und die Reaktionszeiten beim Aktivieren einer Kante auf dem physischen Spielbrett beträgt nicht länger als 200ms.
B01	Das Spiel ist einfach in Betrieb zu nehmen, alles was für Starten des Spieles benötigt wird, ist das Einstecken des Stromkabels
B02	Die Studierende verstehen innerhalbt von weniger als 1 Minute wie man das Spiel bedient.
B03	Die Studierende erhalten visuelles Feedback auf Ihre Aktionen, die klar und unmissverständliche Signal geben.
Z01	Wenn ein physischer Defekt an z.B. einem Knopf auf dem Spielfeld auftritt, kann das Spiel immer noch benutzt werden mit der betroffenen Kante deaktiviert.
Z02	Pipe-It-Up! Kann mit unerwarteten Fehlern umgehen und soll z.B. bei einem Problem mit einem physischen Knopf nicht abstürzen, sondern normal (eingeschränkt) weiter funktionieren. Die Fehler werden in einem Log-File zur Analyse geloggt.
Ü01	Die vermittelte Theorie aus dem Erklärmodus können auf den Unterricht des Moduls "Mathematische Grundlagen der Informatik (mgli)" angewandt werden

11. Risiken und technische Schulden

Die folgenden Risiken wurden zu Beginn des Vorhabens identifiziert. Sie beeinflussten die Planung der ersten beiden Konstruktionsphasen massgeblich

11.1 Risiko: Physisches UI wird zu komplex / zeitaufwändig (grösstes / wichtigstes Risiko)

Unser Know-How in Elektronik und Holzbau ist nicht stark ausgebildet. Dadurch ergibt sich ein Risiko, dass das Zusammenbauen des Physischen Brettspiels zu wenig Know-How vorhanden ist und für unser Projektteam zu komplex wird.

Zusätzlich ist die Anzahl an Nodes und Verbindungen ein Risiko. Wenn diese Zahl zu gross wird, gibt es zu viel zeitlichen Aufwand die LED's, Knöpfe und das Spielfeld zu bauen.

11.1.1 Eventualfallplanung

Es könnte ein weniger Komplexes Spielfeld (weniger Knoten und Verbindungen) benutzt werden, welches die manualle Arbeit zum zusammenbauen erheblich reduziert.

11.1.2 Risikominderung

Durch vorzeitigen Prototypen mit ein paar wenigen Knoten und Kanten könnten der zeitliche Aufwand abgeschätzt werden und das Wissen für den Zusammenbau angeeignet werden.

In der Projektwoche einen ersten Prototypen erstellen, heisst es müssen bis dann Hardware Teile für den Prototypen zur verfügbar sein vor Ort.

11.2 Risiko: Aktivierung der Kanten durch das Legen von Pipelines nicht umsetzbar

Der Mechnismus, der sicherstellt dass man den Knopf auf einer Kante ebenfalls über eine Pipeline aktivieren kann, könnte sich als relativ komplex / nicht umsetzbar herausstellen.

Zudem ist unklar wie gut Knöpfe für dies überhaupt funktionieren, ob nicht evt, mit Sensoren gearbeitet werden soll. Dies könnte aber Probleme mit dem Budget geben.

11.2.1 Eventualfallplanung

Kanten können nur über den Knopfdruck aktiviert werden. Aktivierung durch das Legen von physischen Pipelines ist nicht möglich.

11.2.2 Risikominderung

Frühzeitiger Prototyp erstellen, um zu sehen was wie gut funktioniert und ob es überhaupt realistisch ist so wie wir es uns vorstellen.

Verschiedene Varianten ausprobieren

11.3 Lieferengpässe für physische Teile

Wegen Corona kann es zu Lieferengpässen kommen. Teile können eventuell nicht rechtzeitig geliefert werden, was unseren Zeitplan durcheinander bringen würde und uns für das Zusammenbauen des Spiels weniger Zeit geben würde.

11.3.1 Eventualfallplanung

Graphenkomplexität reduzieren wenn möglich, ansonsten auf Softwarelösung ausweichen

11.3.2 Risikominderung

Frühzeitig bestellen, und wenn möglich nur wenn an Lager.

Glossar

Was	Beschreibung	Kommentar
Pipe It Up!	Der Name des Spiels / Produktes	
Prim	Ein Algorithmus um alle Kanten möglichst günstig in einem gewichteten Graphen zu verbinden https://en.wikipedia.org/wiki/Prim%27s_algorithm	
Kruskal	Ein weiterer Algorithmus, um gewcihtete Graphen möglichst günstig zu lösen (alle Knoten zu verbinden) https://en.wikipedia.org/wiki/Kruskal%27s_algorithm	
SWOT	SWOT steht für die Analyse von **S**trengths / **W**eaknesses resp. **O**pportunities / **T**hreats und dient der Ermittlung der im Team vorhandenen Fachkompetenzen.	Wird in https://www.cs.technik.fhnw.ch/confluence20/pages/resumedraft.action?draftId=48013634&draftShareId=53465ad4-d072-4098-ac7c-20194f7d10b6& erwähnt
SOLID	Standard für Objektorientierte Software-Architektur Siehe https://www.digitalocean.com/community/conceptual_articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design#dependency-inversion-principle	
Gameboard	Beschreibt den Teil des Frontends, welches den Graphen darstellt, mit Knoten und Kanten etc. Hier gibt es ein Software-Gameboard und ein Hardware-Gameboard	