### **Aufbau technischer Infrastruktur:**

**Backend**

Das Backend erfüllt zwei Aufgaben: Das Ausliefern des Clients und bereitstellen von Ressourcen in Form von URLs. Wird die Adresse des Backend in die Adresszeile des Browsers eingegeben, liefert das Backend die Anwendung bzw. Die SPA (Single Page Applikation) an den Browser aus. Das Backend kann dann API-Aufrufe von dem Client entgegennehmen und liefert als Antwort Daten im JSON-Format.

Beispiel: Der User klickt auf einen Button, wodurch eine Frage und Antwort im Quiz gelöscht werden soll. Der Client sendet dann eine HTTP-Anfrage an das Backend, dieses löscht die entsprechende Frage und Antwort aus der Datenbank und sendet die aktualisierte Liste von Fragen und Antworten, im JSON-Format, an den Client zurück. Der Client rendert die aktualisierte Liste und stellt sie dar.

Damit ein User die Privilegien hat Änderung vorzunehmen, muss das Backend wissen, von wem die Anfragen kommen. Der User kann sich Einloggen, wobei auf dem Backend eine Session erstellt wird, wenn das Einloggen erfolgreich war. Der Client erhält eine Session-ID, die dieser bei jeder Anfrage mit an das Backend sendet, welches anhand der Session-ID die entsprechende Session laden kann. Anhand der Session kann dann festgestellt werden, von wem die Anfrage gemacht wurde und welche Daten ausgeliefert werden müssen und können.

**Plattform**

Das Backend wird auf der Node.js Plattform ausgeführt. Node.js ermöglicht es, dass JavaScript außerhalb des Browsers ausgeführt werden kann. In Node.js können Module geladen werden, um bspw. Einen HTTP-Server bereit zu stellen.

**Module**

Das HTTP-Modul ist für die Anfragen und Antworten (Request, Response) über das HTTP-Protokoll verantwortlich.

Express-Session dient der Erstellung für Sessions und wird verwendet, Damit sich Benutzer einloggen können

Bei eingehenden Anfragen mit dem HTTP-Header: “Content-Type: application/json” wird der Inhalt des Payloads (Nutzdaten) der Anfrage, durch den Body-Parser in ein Objekt geparst. Der Zugriff auf die Daten ist dadurch sehr einfach, da alle Felder in dem Objekt body bereitgestellt werden, bspw. request.body.password.

Das eingesetzte Framework ist Express.js (ebenfalls als Modul bereitgestellt), mit dem bspw. Automatisch die HTTP-Header gesetzt werden und das Routing sehr erleichtert wird.

Die Aufgaben des Frontend sind: Darstellung des GUI, Tätigen der API-Anfragen an das Backend bzw. Web-API, Darstellung des GUI auf Basis der Backend Antworten aktualisieren.

Als Datenbank wird MongoDB eingesetzt. Diese ist eine nicht relationale Datenbank und basiert daher nicht auf Tabellen, sondern Collections, in denen Einträge gemacht werden können, die als Documents bezeichnet werden.

**Frontend**

Das Frontend wird mit dem Framework Vue.js erstellt. In Vue.js wird die Logik sowie das Styling und der HTML-Code jeweils in einer Datei zusammengefasst. Diese Dateien nennt man Components.

Für das Routing auf wird der Vue-Router genutzt. Dieser lädt die entsprechende Komponente anhand der, in der URL eingegebenen, Route.

Eine Navigationsbar wird verwendet, um alle entsprechenden Komponenten anzeigen zu lassen. Der Router wird bei jeder Nutzung eines Links in der Navigationsbar die entsprechende Komponente Laden.

Für die API-Calls wird eine JavaScript-Datei erstelle, die dann in jeder Komponente, die Anfragen an die API senden muss, eingefügt werden kann.

Als Store wird ebenfalls eine JavaScript-Datei genutzt, in dieser werden sowohl alle relevanten Benutzerdaten sowie Daten, die zur Darstellung der Lobby benötigt werden, gespeichert.

Vue.js stellt während der Entwicklung einen lokalen Server bereit. Sobald die Entwicklung abgeschlossen ist, wird das Frontend kompiliert. Das Ergebnis des Kompilierens sind die CSS und JavaScript Dateien sowie Assets. Diese werden dann von dem Backend bzw. Der API für die Benutzer ausgeliefert, sobald die Webseite aufgerufen wird.

Die Applikation mit allen Abhängigkeiten wie Frontend, Backend und Datenbank wird auf dem vorher konfigurierten Server bereitgestellt.

### **Eingesetzte Programmiersprache und Modelle:**

**JavaScript als Programmiersprache:**

Da in Web-Browsern die Interpretierte Sprache JavaScript verwendet wird, hat sich das Team geeinigt, diese zu verwenden. Ein weiteres Argument JavaScript zu verwenden ist, dass dieses auch außerhalb des Browsers, mit Hilfe von Node.js, ausgeführt werden kann. Somit kann der Server ebenfalls in JavaScript Implementiert werden, was zu einer verringerten Komplexität führen kann.

**Eingesetzte Werkzeuge**

Da der Lead des Entwicklerteams bereits Erfahrungen im Bereich Web-Entwicklung hat, schlug dieser eine Auswahl an möglichen Werkzeugen dem Team vor. Im Folgenden sollen diese kurz erläutert werden.

**Node.js als Server-Plattform:**

Als Plattform für die Server-Anwendung bzw. API, hat sich das Team für die Plattform Node.js entschieden. Das Hauptargument war, dass der Lead des Entwicklerteams bereits Erfahrungen mit der Serverseitigen Programmierung gemacht hat. Ein weiteres Argument für die Entscheidung zur Nutzung von Node.js war, dass Sowohl im Bereich des Backend als auch des Frontend die gleiche Programmiersprache verwendet werden kann.

**Vue.js für den Client:**

Dem Team wurde das Grundkonzept dieses Frameworks vorgestellt und hat sich abgestimmt, dieses Framework für die Implementierung des Clients zu Verwenden. Das Hauptargument für die Verwendung des Frameworks war, dass alle Aspekte der Web-Entwicklung, in Bezug auf das Erstellen von HTML-Code, die Implementierung der Logik und das Erstellen des Layouts in Dateien zusammengefasst werden können. Diese Dateien werden als Components bezeichnet und können in anderen Components integriert werden, was die Modularität und somit die Wiederverwendbarkeit von Code erhöht.

**NPM, um alle notwendigen Pakete zu installieren:**

Mit dem Package-Manager NPM ist es ein leichtes alle notwendigen Pakete, inkl. Dem Framework Vue.js und Node.js zu Installieren. Daher wurde vom Team abgestimmt, diesen zu verwenden.

**GIT als Versionierung-Tool:**

Das Hauptargument ein Versionierungssystem zu verwenden war, dass Teammitglieder an Komponenten Arbeiten können und ihre Arbeit als Version für andere Teammitglieder zur Verfügung stellen können. Somit ist gewährleistet, dass jedes Teammitglied immer die aktuelle Version zur Verfügung steht.

### **Initialisierung der Software:**

Der Projektordner mit dem Namen iu\_quiz wurde erstellt. Dieser enthält das Verzeichnis Server und Client.

Im Verzeichnis Server wurde mit $ npm init –y die Datei package.json erstellt. In dieser Datei werden alle Abhängigkeiten aufgelistet, so dass mit dem Befehl npm install diese automatisch heruntergeladen und im Verzeichnis node\_modules abgelegt werden. Außerdem wurde in diesem Ordner die Datei index.js erstellt, die als Einstiegspunkt für den Server dient. Im Verzeichnis Server wurden das Unterverzeichnis lib erstellt, in dem wiederum die Verzeichnisse routes und store angelegt wurden. Damit wurde die Verzeichnisstruktur für den Server angelegt.

Im Verzeichnis Client wurde mit $ npm init vue@latest die Verzeichnisstruktur für den Client angelegt. Nach dem Ausführen des Kommandos befindet sich die Datei Index.html im Verzeichnis Client. Diese Datei kann als Einstiegspunkt für den Client angesehen werden, da diese alle CSS- und JavaScript-Dateien referenziert. Zusätzlich wurde das Verzeichnis src angelegt. In diesem Verzeichnis befinden sich die Verzeichnisse Views, Components und router. In Views werden im weiteren Entwicklungsprozess Komponenten gespeichert, die die einzelnen Seiten der Anwendung darstellen. Diese Komponenten werden im weiteren Verlauf als Views bezeichnet. In Components werden solche Komponenten abgespeichert, die in Views und Komponenten referenziert werden können. In router werden alle Routen angegeben, damit zwischen den Views navigiert werden kann. Damit wurde die Verzeichnisstruktur des Clients angelegt.

### **Initialisierung der Software:**

Der Projektordner mit dem Namen iu\_quiz wurde erstellt. Dieser enthält das Verzeichnis Server und Client.

Im Verzeichnis Server wurde mit $ npm init –y die Datei package.json erstellt. In dieser Datei werden alle Abhängigkeiten aufgelistet, so dass mit dem Befehl npm install diese automatisch heruntergeladen und im Verzeichnis node\_modules abgelegt werden. Außerdem wurde in diesem Ordner die Datei index.js erstellt, die als Einstiegspunkt für den Server dient. Im Verzeichnis Server wurden das Unterverzeichnis lib erstellt, in dem wiederum die Verzeichnisse routes und store angelegt wurden. Damit wurde die Verzeichnisstruktur für den Server angelegt.

Im Verzeichnis Client wurde mit $ npm init vue@latest die Verzeichnisstruktur für den Client angelegt. Nach dem Ausführen des Kommandos befindet sich die Datei Index.html im Verzeichnis Client. Diese Datei kann als Einstiegspunkt für den Client angesehen werden, da diese alle CSS- und JavaScript-Dateien referenziert. Zusätzlich wurde das Verzeichnis src angelegt. In diesem Verzeichnis befinden sich die Verzeichnisse Views, Components und router. In Views werden im weiteren Entwicklungsprozess Komponenten gespeichert, die die einzelnen Seiten der Anwendung darstellen. Diese Komponenten werden im weiteren Verlauf als Views bezeichnet. In Components werden solche Komponenten abgespeichert, die in Views und Komponenten referenziert werden können. In router werden alle Routen angegeben, damit zwischen den Views navigiert werden kann. Damit wurde die Verzeichnisstruktur des Clients angelegt.

Thomas Sieber

### **Erste Versionierung:**

Nach der Initialisierung der Software, wurde auf der Homepage: [https://www.github.com](https://www.github.com/) ein Repository mit dem Namen iu\_quiz erstellt. Die lokal initiierte Software iu\_quiz wurde das erste mal versioniert und auf das erstellte Repository auf github hochgeladen. Für das Team war das Verwenden einer versionierungs Applikation relativ neu, daher gab der Lead des Entwicklerteams eine Einführung und stellte die wichtigsten Kommandos vor. Zusätzlich wurde eine Guideline erstellt, die Konzepte der Versionierung näherbringen und zusätzlich als Referenz dienen sollte.

Nachdem sich Alle Teammitglieder mit der Versionierung mit GIT vertraut gemacht hatten, haben diese das Repository mit dem Kommando $ git clone, lokal gespeichert.

**Implementierung des Servers/API:**

Das Team hat entschieden, dem Lead des Entwicklungsteams die grundsätzliche Programmierung der API zu überlassen, da dieser in seiner Freizeit einige Erfahrung mit der Entwicklung im Bereich Backend gesammelt hatte.

Die Module Express.js, Cors, Body-Parser, nodemon und Express-Session wurden, mit npm, installiert.

Da die Datei index.js im Verzeichnis server der Einstiegspunkt des Programms ist, wird diese mit $ nodemon index.js gestartet. Nodemon startet die Anwendung Node.js bei jeder Änderung neu, sodass ein manuelles Neustarten, bei Änderungen an dem Server, nicht mehr notwendig sein wird.

In der Datei index.js wurde der HTTP-Server implementiert. Dieser erwartet eine Callback-Funktion, die bei jeder Anfrage aufgerufen wird, um diese zu bearbeiten. Diese Callback-Funktion soll im weiteren Verlauf als API bezeichnet werden. Die API wurde dabei separat erstellt und in index.js referenziert, damit diese vom HTTP-Server verwendet werden kann.

**Konfiguration der API:**

In das Modul createApi.js wurde eine Instanz des Frameworks Express.js erstellt. Diese Instanz stellt die API dar und diese Instanz soll im weiteren Verlauf konfiguriert und anschließend zurückgegeben werden. Dabei wurde im Modul createApi.js die gleichnamige Funktion erstellt, in der alle Konfigurationen durchgeführt und alle Module, die zur Konfiguration dienen, referenziert werden.

Im Modul createApi.js wurden die Module: Cors, Body-Parser, Express.js und Express-Session referenziert. Cors wird eingesetzt, um Eingehende Anfragen beantworten zu können. Da der Client während der Entwicklung von einem anderen Server ausgeliefert wird, ist dieses Module während der Entwicklung notwendig. Wenn eine Anfrage Daten in Form von JSON enthält, die sich im Payload der Anfrage befinden, werden diese als JavaScrip-Objekt geparst.

**Erstellung der Routes:**

Für jede Anfrage soll eine eigene Route in der API erstellt werden. Bei Aufruf einer solchen Route, soll eine entsprechende Händlerfunktion aufgerufen werden. Diese Händlerfunktionen sollen sich in separaten Modulen befinden, die dann in der API referenziert werden können.

Die Route “/create-lobby” und für diese die Händlerfunktion createLobby wurde erstellt. In createLobby.js wurde die Benutzer-Validierung Implementiert. Um eine Lobby erstellen zu können, wurde ein Lobby-Store erstellt (lobbyStore.js). Dieser wurde in index.js instanziiert und als Referenz der API weitergereicht, sodass sie jeder Händlerfunktion als Parameter übergeben werden kann. In lobbyStore.js wurde die Funktion addLobby implementiert. Mit Hilfe dieser Funktion, war es nun möglich, eine neue Lobby zu erstellen, wenn die Route “/create-lobby” aufgerufen wird. Für Fragen und Antworten wurde ein Wörterbuch erstellt und die Struktur in Form von Platzhalter vorgegeben. Diese Struktur wurde dem Team vorgestellt, sodass das Team mit der Erstellung der Fragen und Antworten beginnen konnte. Die Funktion findLobby wurde erstellt um eine Lobby anhand einer ID finden zu können. FindLobby wurde mit der Funktionalität erweitert, sodass, wenn keine Lobby gefunden wird, eine neue Lobby erstellt wird.

Der Lead des Entwicklerteams entschied sich, aufgrund von Zeitdruck, gegen die Nutzung von einer Datenbank für die Benutzer. Um dennoch das Registrieren und Einloggen zu ermöglichen, hat dieser einen Store implementiert, in dem die Nutzerdaten nur im Arbeitsspeicher bestehen. Diesem “In Memory Store” wurde die Funktion addUser implementiert, damit sich Benutzer Registrieren konnten.

Die Route “/add-player-to-lobby” und die dazugehörige Händlerfunktion addPlayerToLobby.js wurde erstellt.

Die Route “/start-quiz” und die entsprechende Händlerfunktion wurden erstellt. Diese soll lediglich die Variable lobby.istRunning auf True setzte, wobei lobby der Lobby-Store ist. Überprüfung, ob Lobby existiert wurde, implementiert.

Die Route “/evaluate-answer” und die dazugehörige Händlerfunktion evaluateAnswer.js wurde erstellt. Implementierung des Vergleichs der Korrekten Antwort und der vom Benutzer getätigten Antwort. Dabei wurden einige Variablen im Lobby-Store erzeugt, die für den Vergleich benötigt werden.

Die Router “/fetch-game-data" und die zugehörige Händlerfunktion fetchGameData.js wurde erstellt. Implementiert wurden Server-Sent-Events in fetchGameData.js und ein Intervall, indem alle 500 ms alle notwendigen Lobby-Daten an den entsprechenden Client sendet.

**Implementierung der Game Loop:**

Es wurde entschieden, dass eine Spielschleife für die kontinuierliche Aktualisierung der Daten erstellt werden sollte. Dabei sollte per Iteration jeweils eine Lobby, die in einer Wörterbuch-Datenstruktur gespeichert ist, Aktualisiert werden. Da durch die Händlerfunktion der Route /fetch-game-data in einem regelmäßigen Zyklus alle in dem Game Loop aktualisierten Lobbys an die Benutzer gesendet werden, muss sich die Game Loop nicht weiter darum kümmern.

Das Modul gameLoop.js wurde das erste Mal erstellt aber mit dem Frontend noch nicht getestet. Vielmehr wurde eine erste grobe Version der Spielschleife geschrieben, bei der auszugehen ist, dass sie Fehler enthält und noch weiterentwickelt werden muss, da wahrscheinlich nicht alle Aspekte, während der Aktualisierung der Lobby-Daten berücksichtigt wurden. Es viel auf, dass statt eines Wörterbuches zu Speicherung der Lobbys zu nutzen, ein Array besser geeignet gewesen wäre, da der Zugriff mittels dem jeweiligen Key umständlich war. Eine Funktion wurde implementiert, die den gewünschten Key zurückgibt, um auf die aktuelle Frage zugreifen zu können. Es wurden zwei if-Anweisungen implementiert, in denen jeweils der kooperative- und der kollaborative Spielmodus behandelt werden. Nach jeder Beantwortung einer Frage, wird nun das Ergebnis in die jeweilige Lobby eingetragen. Ein Fehler wurde behoben, bei dem der Timer weiterlief, auch wenn alle Fragen beantwortet wurden. Während des Testens viel auf, dass die Lobby-Daten nicht synchron waren, da es vorkam, dass Spieler Fragen übersprungen hatten. Dieser Fehler wurde durch das Hinzufügen von Bedingungen behoben.

Nach der Implementierung des kollaborativen Modus, war durch Evaluierung klar, dass der kooperative Modus sehr ähnlich sein wird. Der Unterschied besteht im Grunde, dass nur ein Benutzer, nämlich der Ersteller der Lobby, die jeweilige Antwort auf die jeweilige Frage geben kann. Diese Funktionalität wurde auf der Client-Seite realisiert, da nur überprüft werden musste, ob es sich um den Ersteller der Lobby handelt. Ist das nicht der Fall, wird der Bestätigung Button einfach deaktiviert. die Funktionalität des kollaborativen Modus wurde aus Zeitgründen einfach übernommen, wobei die Prüfung der einzelnen Nutzer weggelassen wurde, sodass nur die Antwort vom Ersteller der Lobby evaluiert werden konnte.

Bei einem Review versuchte der Lead des Entwicklerteams dem Team die Funktionsweise von Programmcode-Abschnitten zu erläutern. Wegen der Komplexität wurde ihm schnell bewusst, dass ohne zusätzliches Anschauungsmaterial und weiteren Erklärungen von JavaScript und Web-Entwicklung nicht möglich war. Daher wurde entschieden, dass ein Großteil der Programmierung dem Lead zu überlassen. Um das Entwicklerteam mehr in dem Prozess der Entwicklung mit einzubeziehen, entschloss sich der Lead, Todos in Form von Kommentaren im Quellcode zu schreiben. Es wurde eine kleine Einführung in JavaScript, mit Focus auf “truthy-” und “falsy-values”, gegeben, da die meisten Todos Validierungen von Benutzereingaben sein werden. Die ersten Todos wurden geschrieben und dem Team vorgestellt. Die Idee mit den Todos war ein guter Vorschlag, da diese schnell von dem Entwicklerteam abgearbeitet werden. Weitere Todos wurden vom Team abgearbeitet und vorgestellt.

**Implementierung des Clients:**

Alle Routen, die in der Applikation benutzt werden, um eine eigenständige Seite anzuzeigen, wurden in der index.js Datei im Verzeichnis “iu\_client/client-app/src/router/” vorläufig, anhand der Architektur, eingetragen. Zu jeder Route wurde eine Komponente erstellt. Diese Komponenten vereinigen Logik und Darstellung. Alle Komponenten die eigenständig eine Seite Darstellen und nicht in anderen Komponenten verwendet werden, wurden als views bezeichnet. Im Laufe der Entwicklung wurden viele views durch Komponenten ausgetauscht, da es oft ausreichend war, diese views als Komponenten zu referenzieren.

Für den Umgang mit dem Framework gab der Lead des Entwicklerteams eine Einführung und beschrieb die Grundlegenden Elemente einer Vue-Applikation. Vermittelt wurde die Struktur der Komponenten und wie diese durch den Router aufgerufen werden können.

Als Navigation hatte sich das Entwicklerteam für eine einfache Navigationsbar entschieden. Diese wurde als Komponente angelegt und in der Hauptkomponente App.vue referenziert, so dass sie auf jeder Seite verfügbar ist. Die Links zu den Views wurden eingetragen. Für das Layout hat sich das Team für die CSS-Eigenschaft “display: Flex;” entschieden, da diese ein responsive Design ermöglicht. Benutzer können sich nun Ausloggen, wenn sie auf den Button Logout klicken. Der Benutzer wird dann auf die Seite zum Einloggen weitergeleitet.

Die Datei store.js wurde angelegt. In Dieser wurden Daten des Benutzers sowie Daten der Lobby eingetragen. Da bei einem Start eines Spieles in periodischen Zeitabständen eine aktualisierte Lobby gesendet wird, wird die veraltete ständig durch die aktualisierte ausgetauscht.

Um eine Lobby erzeugen zu können, wurde die Komponente CreateLobby.vue erstellt. In dieser Komponente wurde die Funktion createLobby erstellt, die eine Anfrage an den Server sendet zwecks Erzeugung einer Lobby auf dem Server. Der Lead Entwickler entschied sich eine Funktion zu erstellen, die Anfragen generalisiert. Diese Funktion namens call, wurde in der Komponente apiCall.vue implementiert und nutzt das Modul Axios um HTTP-Anfragen Senden zu können. Jetzt war es möglich, durch das Aufrufen von call in der Funktion createLobby, eine Lobby auf dem Server zu erstellen. Die Funktion startSinglePlayer wurde in CreateLobby.vue implementiert, um ein Einzelspiel starten zu können.

Damit einer bestehenden Lobby beigetreten werden kann, wurde die Komponente joinLobby.vue erstellt. In dieser Komponente wurde die Funktion joinLobby implementiert, in der die call Funktion mit den Entsprechenden Parametern aufgerufen wird, um einen Benutzer in die Lobby-Store auf dem Server zu speichern.

Sowohl createLobby.vue als auch joinLobby.vue wurden in der view LobbyView.vue referenziert. Diese view wurde dahingehend implementiert, dass der Benutzer entscheiden kann eine Lobby zu erstellen, oder sich einer bestehenden Lobbe anschließen kann. Sowohl createLobby als auch joinLobby waren zuvor als views erstellt worden. Während des Entwicklungsprozesses ist aber aufgefallen, dass die Umsetzung dieser views als Komponenten für einen übersichtlicheren Umgang mit der Anwendung sorgt.

Für die Umsetzung des kollaborativen Modus wurde die view KollaborativView.vue erzeugt. Sobald ein Quiz gestartet wird, muss der Server den Aktuellen Lobby-Store an alle Benutzer bzw. Clients senden. Für diesen Zweck wurde die Funktion startFetchGamedata in der Datei handleGame.js erstellt. Diese Funktion wurde dahingehend implementiert, dass sie als erstes eine Dauerhafte Verbindung zum Server herstellt. Dafür wurde die Technologie Server Sent Events (SSE) eingesetzt. Hier musste der Lead Entwickler sich als erstes mit dieser Technologie auseinandersetzen, um den notwendigen Programmcode sowohl auf dem Client als auch auf dem Server schreiben zu können. Neben den Aufruf der funktion startFetchGamedata in KollaborativView, wurde die Funktion startQuiz implementiert. In Dieser Funktion wird call aufgerufen, um den Server zu signalisieren, dass das Quiz gestartet wurde.

Die Umsetzung des kooperativen Modus war der des kollaborativen Modus sehr ähnlich. Hierfür wurde wieder eine view mit dem namen KooperativView.vue erstellt. Die Funktionen sind die gleichen, der unterschied ist, dass der kooperative Modus auch als Einzelspieler Modus geeignet ist und dass nur der Ersteller eine Antwort abgeben kann. Dies wurde ermöglicht, indem der Antworte Button nur für den Ersteller “klickbar” ist, für alle anderen ist dieser Button ausgegraut.

Der Spielablauf wird durch dich view FragenView.vue dargestellt. In dieser wurden die Funktion evaluateAnswer implementiert in der die Funktion call aufgerufen wird. Call sendet die vom Benutzer getätigte Antwort an den Server. Zusätzlich werden hier die vom Server empfangenen Lobby-Daten visualisiert.

Auch hier hat der Lead des Entwicklerteams Todos in Form von Kommentaren verwendet, die dann vom Team abgearbeitet und getestet wurden. Die meisten Todos waren Benutzereingaben Validierungen.

Dem Lead des Entwicklerteams viel auf, dass der Programmcode unleserlicher geworden ist, sich aber entschloss, diesen Code nicht mehr um zu schreiben, da dafür nicht mehr ausreichend Zeit zur Verfügung stand. Aus zeitlichen Gründen und wegen der Tatsache, dass keine Zeit für eine Umstrukturierung des Quellcodes vorhanden war, geling es dem Lead nicht einen Fehler zu beheben, der gelegentlich auftritt, wenn im kollaborativen Spiel Modus zeitnahe Antworten abgegeben werden.

Am Ender der Implementierung angekommen, gab es noch einige Bugs, die aber auf Grund der Zeitknappheit nicht behoben werden konnten. Das Team entschied, da es sich um einen Prototyp handelt, die Implementierung abzuschließen.