Marco Kluin
Matr. Nr.: 2181477
E-Mail: Marco.Kluin@hhu.de
GitHub: xayfuu
Infowiss. U. Sprachtech. 4. Semester

Projekt Bromi

Eine gamificated Sprach- and Vokabeltrainings-App für Android

Inhaltsverzeichnis

1 Einleitung

Im Rahmen des Projektseminars „I4: Angewandte Informationswissenschaft“ soll ein eigenständiges Projekt von der Idee, über die Planung, bis zur Ausführung eigenständig erarbeitet werden. Hierbei sollen eigene Interessen in das Projekt einfließen und gleichzeitig eine gewisse Herausforderung gegeben sein, die man sich stellen muss. Diese Herausforderung soll etwas sein, was bisher im Studium der Informationswissenschaft und Sprachtechnologie noch nirgendwo behandelt oder als Projekt durchgeführt wurde. Die Relevanz zu den behandelten Themen im Studiengang muss hierbei ebenso vorhanden sein.

Für mein Projekt möchte ich eine Android App programmieren, welches Vokabeltraining für Sprachen mit Spielelementen vereint, um die Motivation und das Lernergebnis des Nutzers spielerisch zu fördern (Gamification). Die App soll den Namen *Bromi* tragen.

2 Projektplan

2.1 Grundvoraussetzungen

Neben der eigentlichen Programmierung der App, über die ich später nochmal genauer sprechen werde, gibt es eine Handvoll Grundlagen, welche ich zunächst lernen muss, bevor ich damit überhaupt anfangen kann. Die größte Herausforderung, und eines meiner Hauptziele mit diesem Projekt, ist das Erlernen der Grundzüge der Android-App-Programmierung. Da ich mich noch nie damit beschäftigt habe, muss ich mich damit zunächst ausführlich beschäftigen und Tutorials, Übungen und Dokumentationen suchen. Da die Entwicklung ausschließlich mit Java stattfindet, muss ich auch zunächst mit Java erneut vertraut machen, sowie eine Android IDE und das Android SDK herunterladen und installieren (auf Windows). Wir halten also fest:

- **Hauptziel: Android-App-Programmierung erlernen**
- Tutorials, Übungen und Dokumentationen über App-Entwicklung ersuchen und durchforsten
    - XML Android Manifest
    - Android XML Layouts and their connectivity to Java code via IDs
    - Activity-Management and Activity-Lifecycles (`onCreate()`, `onStart()`, `onResume()`, `onStop()`, `onPause()`, `onRestart()`, `onDestroy()`)
    - Buttons
    - Saving data to Android Devices
    - User Profiles and User Sign-Ins with e.g. Google Account
    - User Input
- Android IDE und Android SDK herunterladen und installieren
- Android Libraries anschauen
- Android Emulator einer bestimmten Version verwenden
- Java wiederholen und JDK installieren, da ich einen neuen PC habe
- Erstellung von .jar-Dateien anschauen
- Und das alles auf Windows (PATH Variable)
- Links:
    - Google Doc: https://developer.android.com/training/basics/firstapp/index.html
    - Official Android IDE: https://developer.android.com/studio/index.html
    - Android Code Beispiele: https://github.com/JStumpp/awesome-android https://github.com/codepath/intro_android_demo/tree/master/app

2.2 Aufbau der App

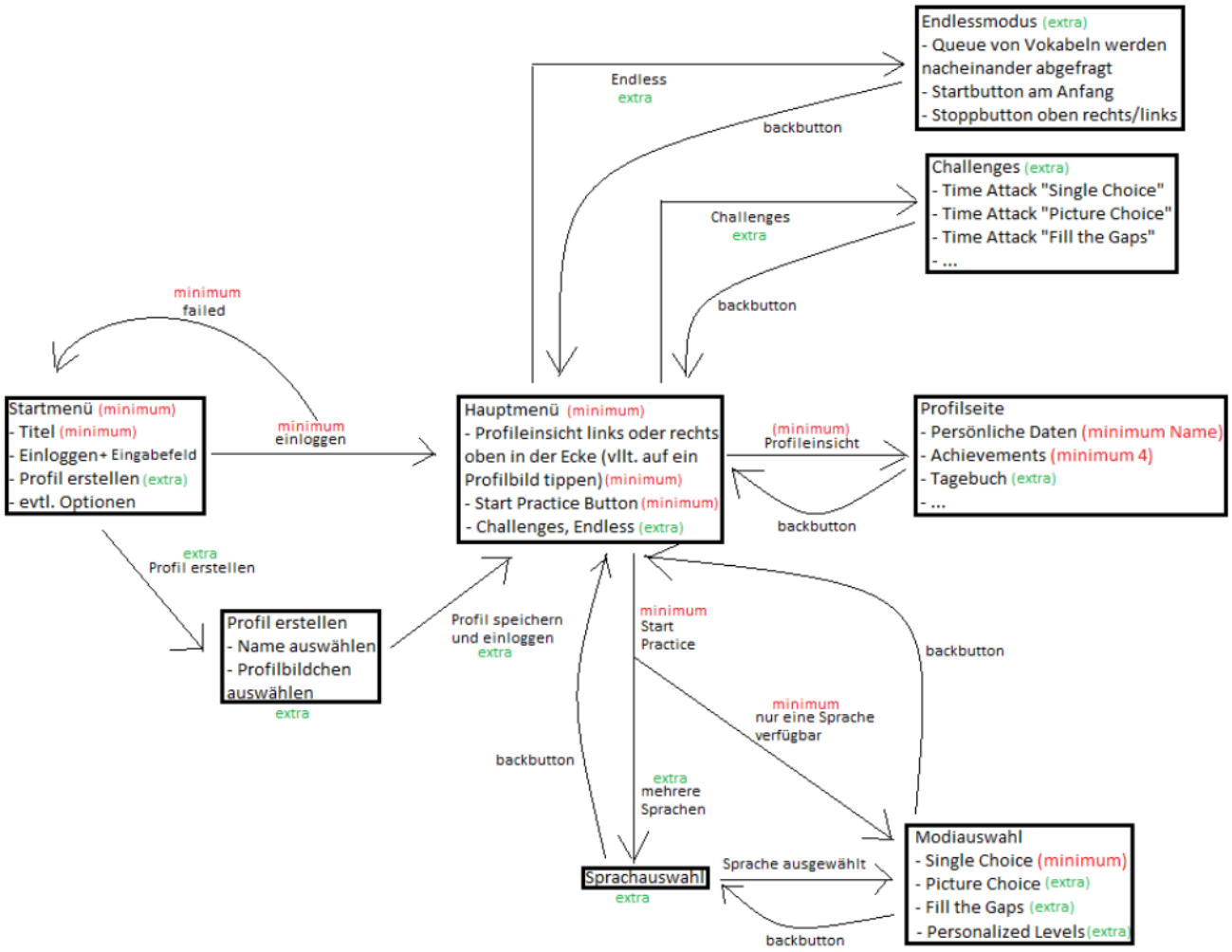
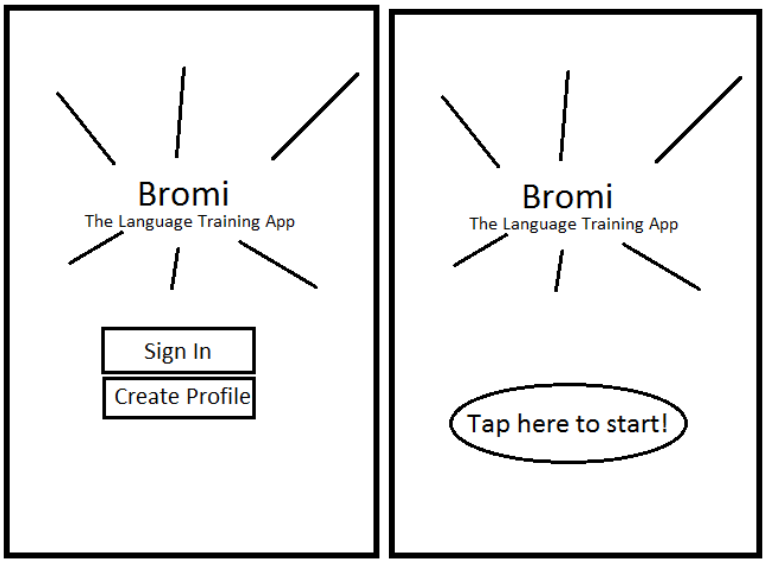


Abb. 1: Visualisierung der Struktur von *Bromi*

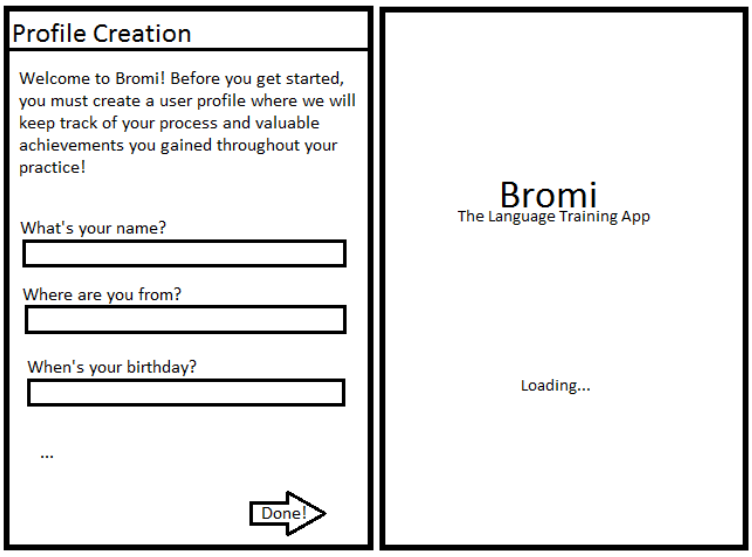
Abb. 1 verdeutlicht den Aufbau der App und was ich für die App in Planung habe, sobald ich bereit bin, mit der Entwicklung anzufangen. Hierbei möchte ich die Aufmerksamkeit auf die beiden Vermerke „extra“ und „minimum“ lenken. Alle Punkte und Verbindungen, welche mit „minimum“ vermerkt wurden, werde ich auf jeden Fall in das Projekt und die Entwicklung miteinbinden. Alle mit „extra“ vermerkten Punkte werden als Bonusaufgaben angesehen, welche ich auch programmieren möchte, sofern die Zeit da ist. Wenn die Zeit nicht da ist, weil ich beispielsweise zu lange gebraucht habe, mir die Androidprogrammierung anzuschauen oder aus irgendeinen anderen Grund, werde ich in Betracht ziehen, diese Extras wegzulassen, um mich nicht zu sehr zu belasten. Im Folgenden werde ich auf alle Ideen eingehen und die Vermerke aus Strukturgründen direkt mitangeben. Ich habe ebenfalls temporäre Funktionsnamen angegeben, welche für die jeweiligen Punkte verwendet werden können. **Nicht-verwendete, geänderte Funktionen oder neue Funktionen werden in der Dokumentation am Ende des Projekts angegeben!**

2.2.1 Startmenü



Konzept 1 Konzept 2

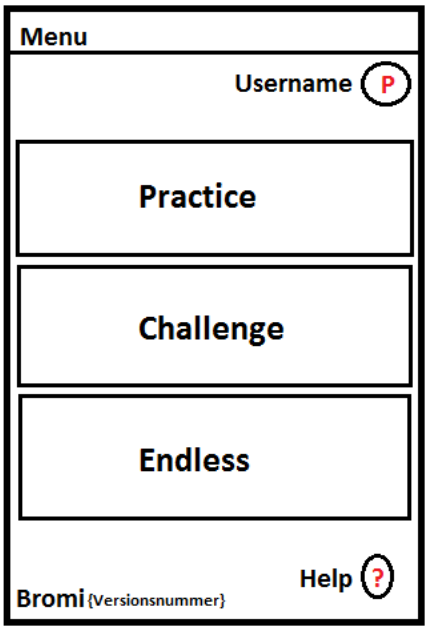
- [m] Startmenü soll angezeigt werden wenn die App gestartet wird
  `{showTitle(), buttonLogIn()}`
- [e] Beim ersten Start soll ein Profil erstellt werden (evtl. Vorgefertigtes Profil nehmen zu Minimalzwecken)
  `{createProfile(), saveProfile(), showLoadingScreen()}`
- [m] Bei jedem weiteren Start der App soll das Profil automatisch geladen werden (evtl. Vorgefertigtes Profil)
  `{loadProfile()` oder `getProfile(), showLoadingScreen()}`



Profilerstellung Ladebildschirm

- [m] Bei erfolgreichem Laden/Erstellen soll dann das Hauptmenü (2.2.2) geöffnet werden
  `{buttonOk(View currentLayer), showMainMenu()}`

2.2.2 Hauptmenü



Hauptmenü

- **[m] P** Button oben rechts führt zur Profileinsicht (2.2.4)
  `{showUserProfile(), loadAchievements(), loadUserInfo(), loadExperience()}`

- **[m] Practice** Button führt zur Sprachauswahl mit **selectedKey 1** (2.2.3)
  `{buttonPractice(), showLanguageSelectionMenu(), setSelectedKey(int key), getSelectedKey(), checkAvailableLanguageSets(), setScrollPane()}`

- **[e] Challenge** Button führt zur Sprachauswahl mit **selectedKey 2** (2.2.3)
  `{buttonChallenge(), showLanguageSelectionMenu(), setSelectedKey(int key), getSelectedKey(), checkAvailableLanguageSets(), setScrollPane()}`

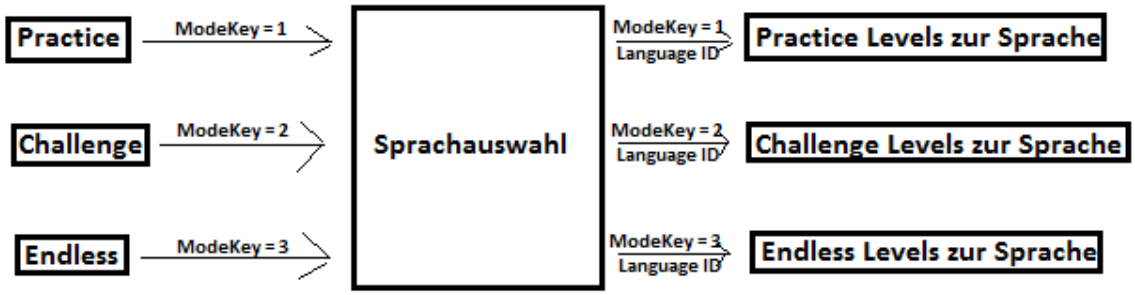
- **[e] Endless** Button führt zur Sprachauswahl mit **selectedKey 3** (2.2.3)
  `{buttonEndless(), showLanguageSelectionMenu(), setSelectedKey(int key), getSelectedKey(), checkAvailableLanguageSets(), setScrollPane()}`

- **[e] ?** Button soll Hinweise (2.2.5) anzeigen,

2.2.3 Sprachauswahl

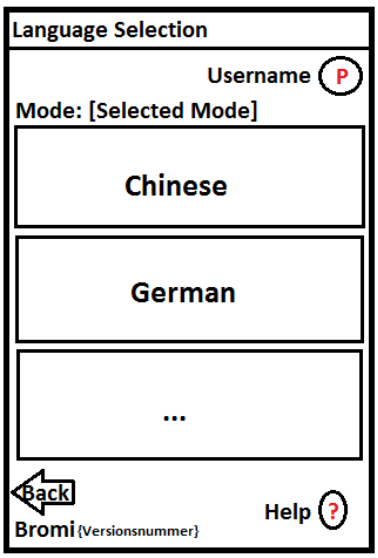
- [m] Je nachdem, welchen Modus der User im Hauptmenü (2.2.2) ausgewählt hat, soll das Programm einen Key erhalten, damit der Modus im Screen nach der Sprachauswahl nicht verloren geht. Die Sprachauswahl soll für alle Modi schließlich gleich sein.
  `{showLanguageSelectionMenu(), setSelectedKey(int key), getSelectedKey()}`

- [e] Zusätzlich zum Key aus dem vorherigen Punkt soll es eine Language ID für jede Sprache geben, die verfügbar ist, damit dass Programm zwischen den Sprachen unterscheiden kann und dann die Menüs für Practice, Challenge und Endless entsprechend korrekt öffnet
  `{checkAvailableLanguageSets(), setCurrentLangID(int id), getCurrentLangID()}`



MenuKeys und Language IDs zugunsten der Unterscheidung in visualisierter Form

- [m] Mindestens eine Sprache muss verfügbar sein, welche auch ohne Sprachauswahlmenü geladen werden kann (benötigt trotzdem den Modi Key)
  `{loadLang(int id), setCurrentLangID(int id), getCurrentLangID()}`

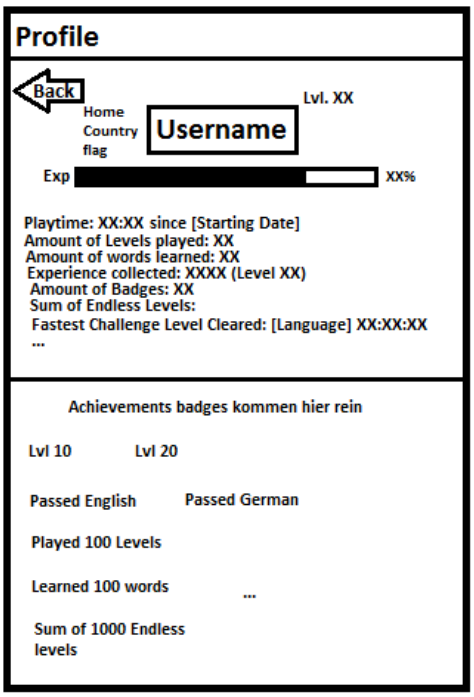


Sprachauswahlmenü

- [m] **P** Button siehe Profil 2.2.4
- [e] **?** Button siehe Hinweisbutton 2.2.5

- **[m] Back** Button siehe Backbutton 2.2.6
- **[e] Sprache** Buttons führen dann zum jeweiligen Menüscreen von Practice (2.2.7), Challenge (2.2.8) und Endless (2.2.9)

2.2.4 Profil



Profilbeispiel

- **[e/m]** Verschiedene Statistiken zum Spiel – All diese Daten werden beim Spielen upgedated und gespeichert
  `{saveProfile(), loadProfile(), setPlayTime(…), getPlayTime, setLevelsPlayed(…), getLevelsPlayed(), setWordsLearned(…), getWordsLearned(), setTotalExperience(…), getTotalExperience(), setAmountOfBadges(…), getAmountOfBadges(), setSumOfEndlessLevels(…), getSumOfEndlessLevels(), setFastesChallengeClear(…), getFastestChallengeClear(), showProfile(), showPlayTime(), showLevelsPlayed(), showWordsLearned(), showTotalExperience(), showAmountOfBadges(), showSumOfEndlessLevels(), showFastestChallengeClear()}`

- Profildaten am besten im Programm als Variablen in einer User-Klasse speichern, damits nicht immer wieder neu geladen werden muss

- [e] Länderflagge neben Namen anzeigen – Flaggen am besten aus einer Quelle herunterladen und in einen Ordner packen und die Dateinamen als Array im Programm speichern und abrufbar machen
  {getUserLocation(), showLocationFlag()}

- [m] Namen anzeigen
  `{showProfile(), showName()}`

- [m] EXP und Level siehe Gamification (2.3)

- [e/m] Badge/Achievement werden dort angezeigt (mindestens 4 Achievements) – siehe Gamification (2.3)

- [e] Benötigt evtl. Scrollpane

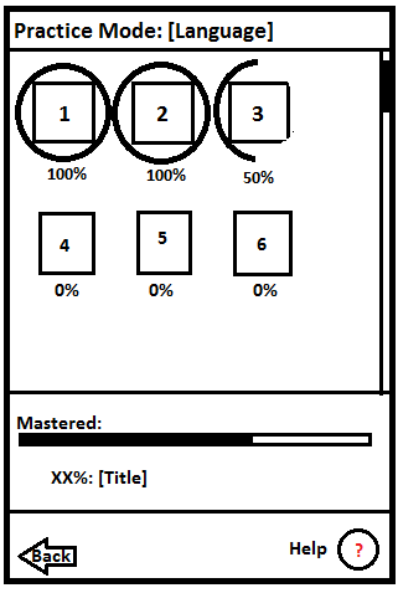
2.2.5 Hinweisbutton

- [e] Um dem Nutzer einen besseren Einstig in die App zu ermöglichen, soll es zu den wichtigsten Menüs Hinweisbuttons geben
- [e] Anstatt hier tausend verschiedene Helpfunktionen zu erstellen, wird der Hinweistext am besten in einer Hashliste mit key, value Paaren gespeichert und bei Bedarf für das jeweilige Layer aufgerufen (wenn Android Layers eine ID haben können, sollte dies relativ einfach sein; ansonsten selbst eine ID variable erstellen, welche sich mit jedem Menü ändert).
  `{buttonHelp(), showHelp(View currentLayer), formatString(String str), createStringPane(String str, View currentLayer)}`
- [e] Hinweise sollen als Pop-up erscheinen und keine neue Activity in dem Sinne aufrufen

2.2.6 BackButton

- [m] Es soll möglich sein, über einen kleinen Button auf das vorherige Menü zu gelangen
- [m] Die Backbuttons sind hierbei statisch auf das jeweils vorherige Menü verlinkt und basieren *nicht* auf etwas wie einen Verlaufsarray (z.B. Profilbackbutton führt immer zum Hauptmenü, Sprachauswahlbackbutton führt auch immer zum Hauptmenü, etc.)
  `{backButton(), showBackButton(int height, int width)}`

2.2.7 Practice Modus



- **[e/m]** Zu einer Sprache soll es **mindestens 5 Level** im Practice Mode geben {langPracticeLevels(), loadLevelData(), startLevel()}

- **[e/m]** Es soll verschiedene Abfragearten geben (Single Choice, Fill in the Gaps, Picture Single Choice, …). Hiervon soll **mindestens Single Choice** implementiert werden.
  Die Fragen werden am besten in einer JSON Datei gespeichert oder in irgendeinem anderen Format zum einfachen Zugriff. Hierbei soll zu jedem Wort eine richtige Antwort gespeichert werden und eine Menge falscher Antworten (wenn möglich, wie bei Single Choice), welche für den Randomizer gebraucht wird. Die Level werden durch eine ID identifiziert und besitzen weiterhin eine QuestionType ID, welche zum Unterscheiden der Fragetypen im Programm benötigt wird (wenn es mehr als ein Abfragetyp gibt), um verschiedene FrageUIs zu jedem Typen zu generieren. Beispiel:

  - **{levelId1+QuestionType;[**
    {„**Word1**“; „SingleChoiceAnswer“},
    {„**Word2**“; „SingleChoiceAnswer“},
    {„**Word3**“; „SingleChoiceAnswer“},
    {„**Word4**“; „SingleChoiceAnswer“},
    {„**Word5**“; „SingleChoiceAnswer“},
    ]}
  - **{levelId2+QuestionType;[**
    …
    ]}
  - …
  - **{wrongAnswerSetId+QuestionType;** [WrongAnswer1, WrongAnswer2,WrongAnswer3, WrongAnswer4, …
    ]}

Die JSON Dateien sollen hierbei in einer Ordnerhierarchie geordnet werden, da die Dateigröße und -struktur eines einzelnen Levels sonst zu groß und komplex wird. Weiterhin wird so ein einfacheres Zugreifen auf die verschiedenen Sprachen und deren Levels einfacher, wie zum Beispiel könnte man das Format der Directory-Adresse variabel machen „`~/LevelData/LangX/TypeY.json`“.

Beispiel der Ordnerhierarchie:

- Leveldata
  - English oder LanguageID
    - SingleChoice.json
    - FillTheGap.json
    - …
  - German oder LanguageID
  - Chinese oder LanguageID
  - …

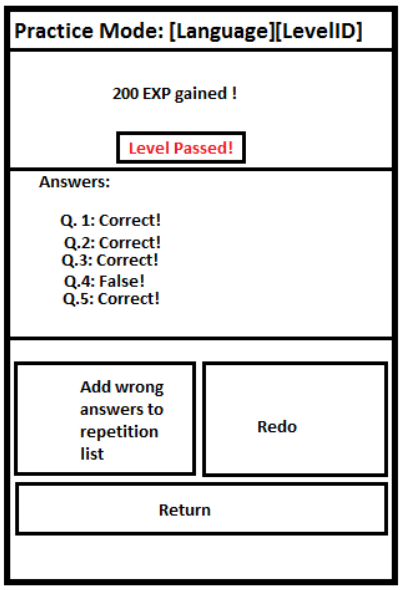
```
{loadLevel(int levelID, int languageID), loadQuestions(String
type, int levelID, int languageID), randomizer(arr[]
questions, arr[] wrongAnswers, int QuestionTypeID),
showLevel(arr[] randomized, int levelID, int languageID, int
QuestionTypeID), levelLoop(), buttonLevel()}
```

- **[e/m]** Alle Levels sollen mindestens 5 verschiedene Vokabeln einführen und trainieren. Zu jedem Wort soll jeweils eine Frage zu jeder Abfrageart, die es gibt, implementiert werden, sodass sich die Summer der verfügbaren Fragen auf $5 * |Abfragearten|$ beschränkt. Als Minimum wären also jeweils 5 Single Choice Fragen zu jedem Vokabular pro Level angemessen.

- **[e]** Wenn es dann mehrere Abfragearten gibt, so sollen alle Abfragedaten geladen werden und mit einem **Randomizer** zu einem zufälligem Level gewürfelt werden, wie es bereits erwähnt wurde.

- **[e]** Zu jedem Level soll es eine Fortschrittsanzeige geben, welches trackt, ob der Nutzer alle Fragen in diesem Level mindestens jeweils einmal richtig beantwortet hat (auch beim Randomizer). Der Fortschritt soll im Nutzerprofil gespeichert werden, damit dieser nicht verloren geht.
  ```
  {loadProgress(int languageID), saveProgress(int languageID),
  showProgress(int levelID, int[] initialCoordinates),
  drawCircle()}
  ```

- **[e/m]** Ein Ergebnisbildschirm welches die Performance bei einem Level zeigt, die Anzahl erhaltener EXP und die Möglichkeit bietet, das Level zu wiederholen, zurück ins Menü zu gehen, oder die Levels zu einer sogenannten „Repetition List“ hinzuzufügen.




```
{saveAnswer(), showResultScreen(arr[] answers),
computeExperience(arr[] answers), redoButton(),
returnButton()}
```

- **[e]** Die **Repetition List** ist eine Funktion, welches falsch beantwortete Wörter in einer Liste speichern soll, welche entweder als einfache Liste abrufbar ist (ähnlich zu einem Vokabelheftchen) oder als eigenes Level verfügbar sein soll. Die Repetition List sollte im Falle der Implementierung als eigene JSON Datei gespeichert werden.

```
{createRepetitionList(), addToRepetitionList(),
removeFromRepetitionList(), saveRepetitionList(),
loadRepetitionList(),
```

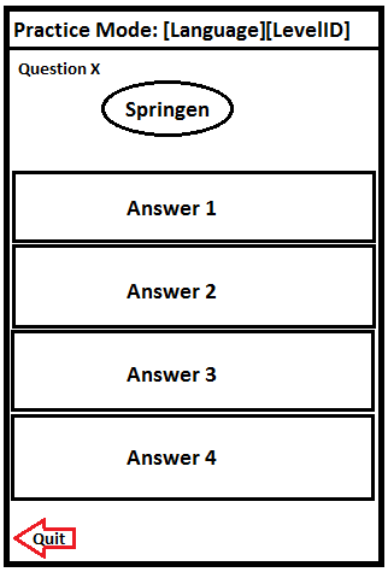
- **[m]** EXP siehe Gamification (2.3)

- **[e]** Ein Hinweisbutton (2.2.5)

- **[e]** Eventuell soll eine Scrolleinheit implementiert werden

- **[m]** Ein Backbutton zum vorherigen Menü (2.2.6)

- **[m]** Single Choice Level Aussehen:



2.2.8 Challenge

- **[e]** Wird geplant, wenn ich die Minimalvoraussetzungen erfüllt habe
- **[e] Grundidee:** Es soll sich um einen Time-Attack-Modus handeln, bei welchem der Nutzer unter Zeitdruck Vokabelfragen beantworten muss

2.2.9 Endless

- **[e]** Wird geplant, wenn ich die Minimalvoraussetzungen erfüllt habe
- **[e] Grundidee:** Der Nutzer soll solange Fragen beantworten, bis keine mehr verfügbar sind, der Nutzer keine Lust mehr hat oder der Nutzer 10 Fragen falsch beantwortet hat.

2.3 Gamification

- **[m]** Es soll ein simples EXP-System geben. Der Nutzer erhält durch das Lösen von Levels Erfahrungspunkte, beim Erreichen von XX% Fortschritt bei einem Level oder einer Sprache, beim Spielen von Challenges, beim Spielen von Endless Levels, beim Erhalten von Badges/Achievements, usw. Die EXP sollen natürlich mit dem Nutzerprofil gespeichert werden und entsprechend in verschiedenen Menüpunkten angezeigt werden.

  ```
  {loadExp(), saveExp(), checkUserLevel(), setUserLevel(),
  getUserLevel(), computeUserLevelProgressBar(),
  showProgressBar(), getTotalExpSum(), …}
  ```

- **[m]** Es soll Badges/Achievements geben, welche zu verschiedenen Situation vergeben werden können. Dazu gehört z.B. das Erreichen eines bestimmten Levels, das meistern einer Sprache, das Lösen einer bestimmten Anzahl von Fragen, das Richtighaben einer bestimmten Anzahl von Fragen, das Erreichen einer bestimmten Spielzeit etc. Badges sollen natürlich im Userprofil gespeichert werden und angezeigt werden als kleine Grafik. Als Minimum würde ich 4 Badges machen wollen, ohne großen Aufwand für Grafik und Animationen.

  ```
  {checkBadges(), saveBadges(), loadBadges(), showBadges()}
  ```

- **[e]** Jedes Level im Practice Modus soll eine eigene Fortschrittsleiste haben, welches sich genau dann komplett aufgefüllt hat, wenn der Nutzer alle möglichen Fragen für ein Level richtig beantwortet hat. Mit dem Randomizer sollte dies etwas dauern und die Motivation des Nutzers trotzdem beibehalten.

- **[e]** Für Endless und Challenge soll es Leaderboards geben oder eine Möglichkeit, die höchsten und besten erreichten Statistiken auf dem Profil anzeigen zu lassen.

2.4 Liste der Tools und Werkzeuge

- Java
- JDK 8+
- Java JAR
- Android SDK
- Android IDE
- JSON for Java
- Evtl. eigenes Samsung Galaxy
- GitHub

3 Sonstige Notizen