

**Power Pirates**

Universität Kassel

Fachbereich FB16 - Elektrotechnik/Informatik

Fachgebiet ComTec

Dokumentation Code-Camp(iOS/Objektiv-C)

Team:

Domminik Waas

Tim Veith

Christoph Sandrock

Jannik Raabe

Inhalt

[Einleitung 3](#_Toc523429109)

[Applikation Frontend 4](#_Toc523429110)

[Aufbau der Screens 4](#_Toc523429111)

[Start Screen 4](#_Toc523429112)

[Main Screen 5](#_Toc523429113)

[Shop Screen 5](#_Toc523429114)

[Lager Screen 6](#_Toc523429115)

[Minispiel Screen 6](#_Toc523429116)

[Developer Screen 6](#_Toc523429117)

[Grafik und Musik 7](#_Toc523429118)

[Minispiele 9](#_Toc523429119)

[Schiffssause 9](#_Toc523429120)

[Schatzsuche 10](#_Toc523429121)

[Applikation Backend 11](#_Toc523429122)

[Notification Manager 11](#_Toc523429123)

[Push-Mitteilung erstellen 11](#_Toc523429125)

[Push-Mitteilung löschen 11](#_Toc523429126)

[Queue 11](#_Toc523429127)

[Bedürfnisse 12](#_Toc523429128)

[Initialisierung 12](#_Toc523429129)

[Game Loop 12](#_Toc523429130)

[Wünsche erfüllen 13](#_Toc523429131)

[Tests 13](#_Toc523429132)

[Datenstruktur 14](#_Toc523429133)

[Datenverwaltung 14](#_Toc523429134)

[Datenbank Manager 14](#_Toc523429135)

[Inhalt und Klassen 15](#_Toc523429136)

[TypeDef 17](#_Toc523429137)

[Fazit 18](#_Toc523429138)

# Einleitung

In der einwöchigen Veranstaltung Code-Camp wurde die Aufgabe gestellt eine „Tamagotchi“ App zu erstellen. Durch diese App sollten wir einen tieferen Einblick in die Programmierung mobiler Geräte und den integrierten Sensoren bekommen.

Am ersten Tag haben wir uns Gedanken zur Struktur das Thema und den Ablauf unserer App gemacht.

Die restliche Woche wurde intensiv genutzt um unsere Ideen zu implementieren, nach Ablauf der Woche waren wir mit den Grundanforderungen soweit fertig. Als Ziel für die restliche Zeit hatten wir uns Bug fixes und nette Erweiterungen vorgenommen.

# Applikation Frontend

Im folgenden Kapitel „Applikation“ wird eine Übersicht über die einzelnen Screens der Applikation gegeben.

## Aufbau der Screens

Um die Screens und die Zusammenhänge der Screens untereinander zu erstellen - wie das Bewegen von einen auf den anderen Screen - haben wir den Interface Builder und Storyboards benutzt, mit denen man simpel die Struktur der App und der Screens entwickeln kann.

Der Main-, Shop-, Lager-, Minispiel- und Developer Screen sind zusammen in einer Tab Bar enthalten, sodass man einfach und übersichtlich zwischen diesen Screens wechseln kann, da dies die Screens sind, die man am häufigsten benutzt.

Zusätzlich haben wir für jeden Screen eine eigene ViewController Klasse erstellt, da jeder Screen andere Funktionalitäten hat und auf ihnen unterschiedliche Dinge angezeigt werden.

## Start Screen

Nachdem man das Spiel gestartet hat, gelangt man nach dem Intro Screen der App zum Start Screen.

Je nachdem, ob man schon einen Spielstand hat oder nicht, erscheint entweder ein Register Screen oder ein Continue Screen.

Auf dem Register Screen kommt man, wenn man noch keinen Spielstand hat. Hier kann man seinem Piraten einen Namen geben und somit einen neuen Piraten erstellen.

Falls man schon einen Spielstand hat, gelangt man auf den Continue Screen. Dort wird einem das Gold, das Level und der Name seines Piraten angezeigt und durch das Klicken auf „Start Game“ startet man seinen gespeicherten Spielstand.

Um zu entscheidet, welcher Screen angezeigt werden soll, wird in der Datenbank nachgeschaut, ob schon ein Pirat angelegt ist oder nicht. Je nachdem wird dann das jeweilige Storyboard zum starten ausgewählt. Dies war eine einfache Lösung, um das Problem zu lösen, dass man 2 verschiedene Startmöglichkeiten hat: Ist ein Charakter schon angelegt oder nicht?

## Main Screen

Nach dem Fortfahren oder Erstellen des Piraten ist der Main Screen der erste Screen, den man sieht.

Dort werden alle wichtigen Informationen über den Piraten selbst angezeigt, wie Gold, sein Leben und sein Level in Form seines Aussehens.

Ein weiterer großer Punkt, der hier angezeigt wird, sind die aktuellen Bedürfnisse des Piraten. Diese werden in Textform in der Mitte des Bildschirmes angezeigt, falls es irgendwelche aktuellen gibt.

Um diese erfüllen zu können, haben wir ein Dropdown Menü mit Buttons erstellt, mit denen man den Piraten zum Beispiel füttern kann.

Probleme bei der Anzeige hat hier das Dropdown Menü gemacht, da es nicht standardmäßig in XCode vorhanden ist. Gelöst haben wir das Problem, in dem wir einfach eine H-Box mit Buttons erzeugt haben, die man über einen anderen Button anzeigen und wieder verstecken kann.

## Shop Screen

Einen wichtigen Teil des Spiels stellt der Shop dar, da man fast nur dort die Items bekommt, die man zum Erfüllen der Bedürfnisse benötigt.

Die zu kaufenden Items werden auf dem Bildschirm dargestellt und sind eigentlich Buttons, damit dem Benutzer beim Klicken auf das Item der Preis, der Name, die bereits vorhandene Anzahl des Items und ein Kauf- und Verkauf Button für das jeweilige Item angezeigt wird.

Außerdem wird im oberen rechten Teil des Bildschirmes die vorhanden Anzahl an Gold angezeigt, damit man sehen kann, ob man sich die Items überhaupt leisten kann.

Besonders hervorzuheben ist dabei, dass diese ganzen Informationen der Items generisch durch eine Funktion im Quellcode erzeugt werden, der man nur die Informationen über das angeklickte Item mitgeben muss.

## Lager Screen

Der 3. Screen, der über die Tab Bar erreichbar ist, ist das Lager. Dort kann man die Anzahl seiner Items sehen, die durch eine Zahl über dem entsprechendem Item angezeigt wird.

Ein weiteres lustiges Feature, dass wir uns ausgedacht haben, ist das Gehen auf Raubzüge, was bei einem echten Piraten natürlich nicht fehlen darf. Dafür muss man auf den „Plündern“-Button im Lager klicken, wodurch der Pirat entweder ein zufälliges Item oder Gold bekommt oder ein Leben verliert. Die Erfolgsquote ist dabei abhängig vom Level des Piraten, sprich: Je höher das Level des Piraten ist, desto höher ist die Chance ein Item zu bekommen.

## Minispiel Screen

Der letzte für den Benutzer wichtige Screen ist der Minispiel Screen. Dort werden alle vorhandenen Minispiele angezeigt, die man von dort aus auch Spielen kann. Über die Minispiele selbst wird im Punkt „3. Minispiele“ mehr berichtet und hier nicht weiter darauf eingegangen.

## Developer Screen

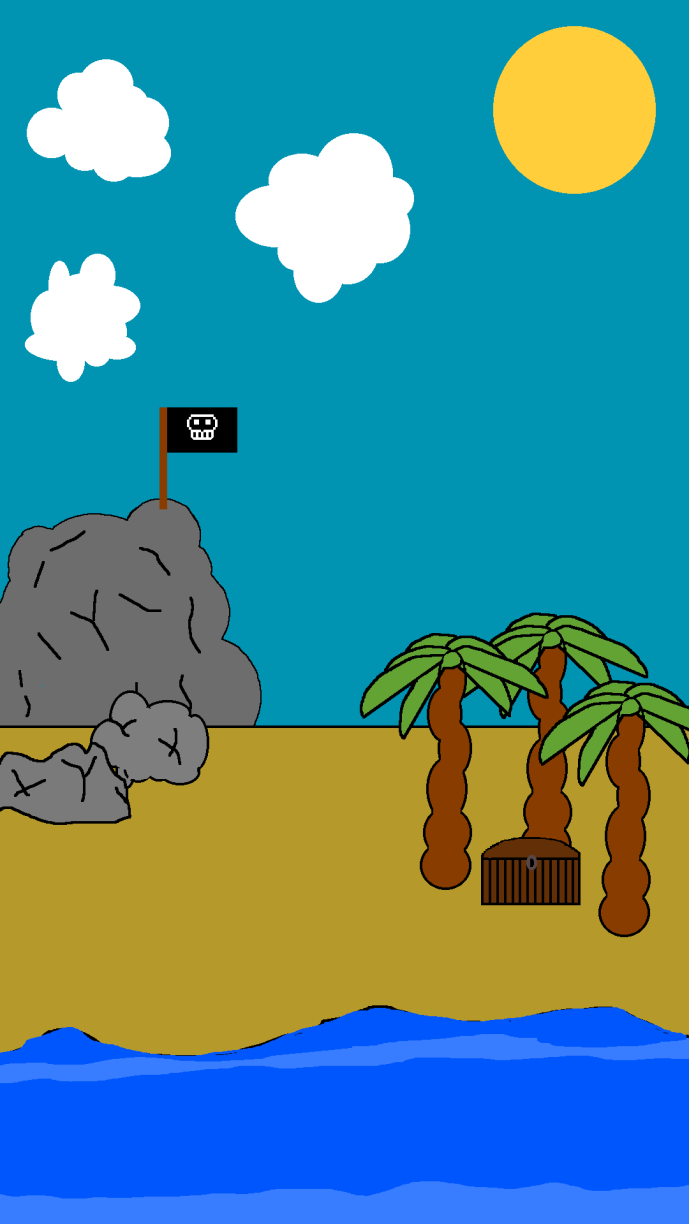
Ganz am Ende der Tab Bar befindet sich der Developer Screen, welcher dafür entwickelt wurde gewisse Aktivitäten triggern zu können, um Funktionalitäten der App einfacher zu testen.

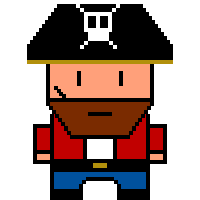
Dies ist durch das Klicken der jeweiligen Buttons möglich, auf denen steht, was beim Klicken passiert.

## Grafik und Musik

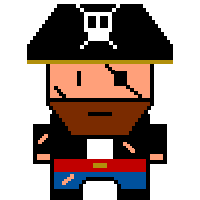
Alle Grafiken/Sprites in der App sind durch das Programm Piskel (https://www.piskelapp.com/) entstanden und selbst erstellt. Piskel ist ein freier Online Sprite Editor. Ein Sprit ist ein Grafikobjekt, in diesem Projekt bestehend aus .png Dateien, das von der Grafikhardware über das Hintergrundbild bzw. den restlichen Inhalt der Bildschirmanzeige eingeblendet wird.

Für alle Themen wurde ein Hintergrund angefertigt.

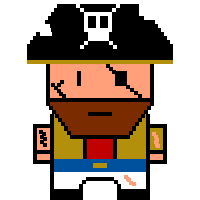
Startbildschirm + Hauptbildschirm Spielfigur(Tamagotchi)



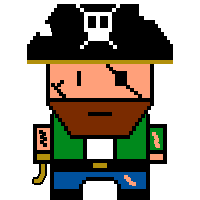
Level 1



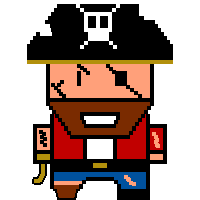
Level 2



Level 3

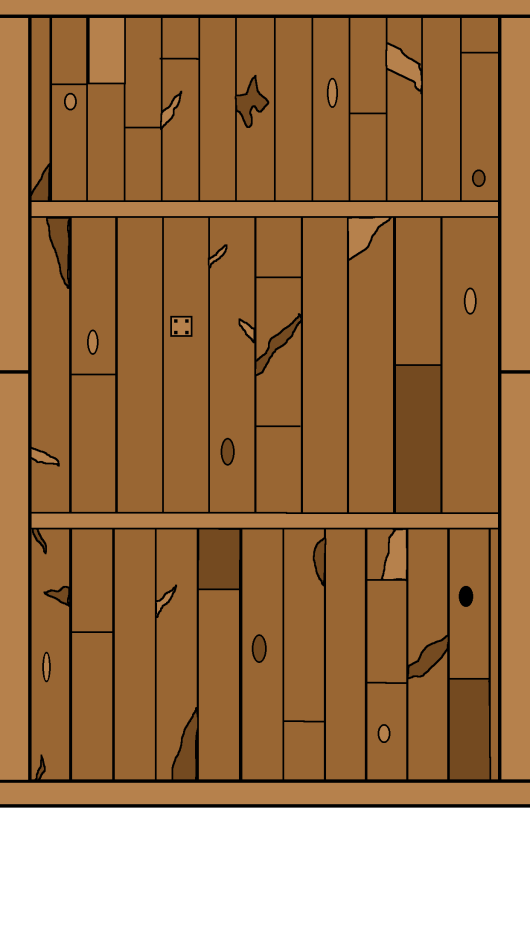


Level 4



Level 5

Laden Lager



Animationen

Die Animationen, wie die Wellenbewegung des Startbildschirms, entstehen durch das übereinanderlegen der Bilder wie bei einer GIF. Jedes nachfolgende Bild ist minimal anders zum Vorgänger und so wird eine Bewegung im Bild vorgetäuscht.

Man legt ein Array an und legt dort alle Bilder in der Reihenfolge wie man sie zeigen möchte ab. Durch das durch laufen dieses Arrays und das anzeigen der verschiedenen Bilder entsteht der Animationseffekt. Hierzu verwenden wir das UIImageView Objekt. Dies zeigt ein Bild oder eine Sequenz an, wobei die Dauer, wie lange ein Bild angezeigt wird, unsere FPS bestimmen.

Musik

Als Hintergrundmusik läuft eine „8-Bit“ Version der Titelmelodie von Fluch der Karibik in Dauerschleife. Hierfür wurde das AVAudiplayer Objekt verwendet.

## Minispiele

Um die App spannender zu gestalten, haben wir uns entschieden Minispiele als zusätzliche Features hinzuzufügen, damit auch die Geschicklichkeit des Benutzers gefordert wird.

Die Minispiele sind über den Minispiel Screen erreichbar, welcher alle verfügbaren Minispiele anzeigt.

### Schiffssause

Das erste Spiel ist das Geschicklichkeitsspiel „Schiffssause“, bei dem es darum geht mit einem Schiff einer gewissen Anzahl an auf einem zukommenden Objekten, wie Inseln oder Seemonstern, auszuweichen.

Der Spielverlauf ist folgendermaßen:

* Als erstes kommt man auf den Willkommensbildschirm des Spieles.
* Wenn man dort auf den Bildschirm tippt gelangt man zum Spielbildschirm und das Spiel beginnt sofort.
* Gegenstände kommen auf den Spieler zu und dieser muss ihnen ausweichen oder kann gelegentlich Goldstücke einsammeln, die dem Piraten gutgeschrieben werden
* Dabei wird die Geschwindigkeit der Objekte nach und nach schneller, was das Spiel im Verlauf immer schwieriger macht.
* Berührt der Spieler eines der Objekte wird das Spiel langsam beendet und die Anzahl der Ausgewichenen Gegenständen wird dem Spieler angezeigt
* Anschließend gelangt er wieder zum Startbildschirm.

Das Design ist dabei relativ einfach gehalten, um das Spiel und die Steuerung so intuitiv wie möglich zu lassen, zum Beispiel durch die 2 großen Steuerungsknöpfe.

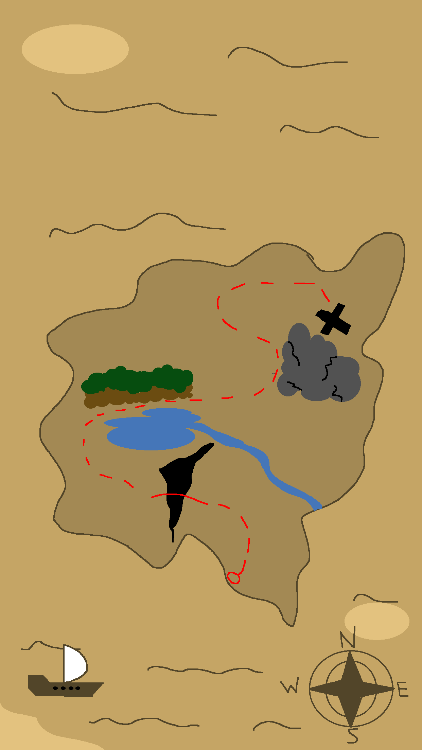
Als Framework für das Spiel haben wir SpriteKit verwendet, da es das Standard Framework für einfache 2D Spiele für IOS ist und sich einfach in die App einbinden lässt.

Die Erstellung des Spieles ließ sich anschließend in 3 Bereiche einteilen:

* Das Erstellen des Grundgerüsts
* Den Spieler/Schiff erschaffen und es bewegen können
* Die Objekte erstellen, denen man ausweichen muss

Wir hatten uns anfangs überlegt die Steuerung des Spielers nicht über Knöpfe, sondern über das Neigen des Handy zu realisieren. Da man die App aber nur auf dem Simulator und keinem echten Gerät testen kann und sich das Neigen nicht so einfach für Spiele auf dem Simulator testen lässt, haben wir diese Idee wieder verworfen.

### Schatzsuche

Im diesem Minispiel geht es um eine Art Schatzsuche. Der Spieler muss hier mit dem Handy 100 Meter laufen um den Schatz in Höhe von 10 Münzen zu erhalten.

Technisch wird das ganze mit dem GPS-Sensor umgesetzt, da dies über den Emulator mit der „StreetRun“-Funktion simulierbar ist. Sobald man auf „Start“ drückt sieht man auf dem Display wie viele Meter man noch laufen muss. Die App berechnet nach jeder neuen Standort-Aktualisierung die Distanz in Metern zwischen dem letzten Standort und dem neuen und verrechnet die neuen Meter mit den bisher zurückgelegten. Sobald man die 100 Meter zurückgelegt hat, bekommt man die 10 Münzen gutgeschrieben und man hat die Möglichkeit die Schatzsuche erneut zu starten.

Ebenfalls ist es möglich die App zu minimieren, denn der LocationManager merkt sich alle Standortveränderungen. Genauso gut ist es bei aktiviertem GPS Sensor auch möglich zu einem späteren Zeitpunkt die Suche zu starten (wenn sie nach App-Start bereits einmal gestartet wurde) und sich die bis zu Neustart zurückgelegten Meter einmalig auf die neue Schatzsuche gut schreiben zu lassen.

Beispiel: Wenn man zwischen den Schatzsuchen 30 Meter gelaufen ist und dann die neue Schatzsuche startet, muss man nur noch 70 Meter zurücklegen.

*Tipp: Wenn man auf den „Zurück“-Button drückt, um sich zum Beispiel um seinen Piraten zu kümmern, werden die bisher geschafften Meter zurückgesetzt. Deswegen ist es zu empfehlen zu laufen, bevor man „Start“ drückt um später mit den nebenbei gelaufenen Metern direkt in eine neue Suche zu starten. Das hat die Ursache, dass wir für die Schatzsuche keine extra persistente Speicherung erstellen wollten (um die bisher zurückgelegten Meter zu speichern). Genauso startet man deshalb auch nach kompletten App-Neustart wieder bei 100 verbleibende Metern.*

# 

# Applikation Backend

## Notification Manager

## 



Über die Klasse *NotificationManager* können Push-Mitteilungen in das Notification Center des Geräts eingetragen werden und daraus gelöscht werden.

Vom Notification Center aus werden sie zur angegebenen Zeit automatisch angezeigt.

### Push-Mitteilung erstellen

Die Methode *createPushNotification* erstellt eine neue Push-Mitteilung im Notification Center. Als Parameter werden die Zeit, zu der die Mitteilung erscheinen soll sowie der Inhalt der Nachricht übergeben. Falls die Zeit in der Vergangenheit liegt oder die App keine Berechtigung für Push-Mitteilungen hat, wird die Mitteilung nicht erstellt. Als ID wird die Zeit verwendet, die eindeutig ist, da nie zwei Mitteilungen mit gleicher Zeit erstellt werden können.

### Push-Mitteilung löschen

Mitteilung werden mit Hilfe der Methode *removePushNotification* wieder gelöscht. Dabei werden sie anhand ihrer ID identifiziert. Außerdem können mit der Methode *cleanPushNotifications* alle Einträger der App in das Notification Center gelöscht werden, was bei einem Neustart des Spiels passiert.

### Queue

Die Nachrichten werden immer nur dann verschickt, wenn die App gerade nicht aktiv ist. Dabei sind wir auf das Problem gestoßen, dass in dieser Zeit keine neuen Einträge im Notification Center erstellt werden können. Um das zu umgehen, werden die Wünsche immer „auf Vorrat“ erstellt, während die App aktiv ist. Es werden für jedes Bedürfnis zwei Einträge erstellt: einer für den Start und einer für den Ablauf.

## Bedürfnisse

Der Pirat hat vier Bedürfnisse: Er will Fleisch essen, Wasser trinken, Rum trinken und seinen Vitaminbedarf mit Obst decken, um nicht an Skorbut zu erkranken.

|  |  |
| --- | --- |
| **ID** | **Name** |
| 0 | Hunger |
| 1 | Durst |
| 2 | Alkohol |
| 3 | Vitamine |

### Initialisierung

Die Bedürfnisse werden am Spielstart einmalig zufällig mit der Methode *initDesires* initialisiert, indem eine Warteschlange mit fünf Einträgen erstellt wird (einer pro Leben). Dabei wird jedem Eintrag eine zufällige ID (siehe Tabelle), eine Startzeit und eine Ablaufzeit zugewiesen. Für die Zeiten werden in der TypeDef Intervalle für die Zeit zwischen Start und Ablauf eines Wunsches und zwischen dem Ablauf eines Wunsches und dem Start des nächsten angegeben. In diesen Intervallen werden gleichverteilte Zufallszahlen erstellt, die auf die aktuelle Zeit bzw. die zuletzt generierte Zeit addiert werden und so die nächste Start- oder Ablaufzeit bilden. Die Ablaufzeit eines Bedürfnisses liegt also immer vor der Startzeit des nachfolgenden, sodass keine zwei Bedürfnisse gleichzeitig aktiv sein können.

### Game Loop

Während des Spiels wird durch einen Timer in der *AppDelegate* durchgängig mit Hilfe der Methode *checkStatus* überprüft, ob gerade ein Wunsch abläuft. Dazu wird die Datenbank ausgelesen und die Ablaufzeit jedes der eingetragenen Wünsche mit der aktuellen Zeit verglichen. Falls mindestens eine Ablaufzeit erreicht ist, ändert sich der Text der Sprechblase des Piraten entsprechend und ihm wird ein Leben abgezogen. Bei nicht aktiver App wird eine Push-Mitteilung angezeigt, die den Verlust eines Lebens mitteilt. Ein neuer Wunsch wird hierbei nicht erstellt, da der Pirat jetzt ein Leben weniger hat und man deshalb auch einen Wunsch weniger scheitern lassen darf, bis der Pirat stirbt.

Im *MainGameViewController* wird geprüft, ob eine Startzeit erreicht ist. Falls das der Fall ist, erscheint eine Denkblase über dem Piraten, in der er dem Nutzer seinen Wunsch mitteilt. Falls die App nicht aktiv ist, wird eine Push-Mitteilung angezeigt, die schon vorher erstellt wurde.

### Wünsche erfüllen

Die Bedürfnisse werden über die entsprechenden Buttons im Menü erfüllt. Sobald einer der Buttons gedrückt wird, wird durch die Methode *fulfillDesire* geprüft, ob das Item noch im Lager vorhanden ist. Wenn das der Fall ist, wird der Bestand um eins verringert - auch, wenn das Item gerade gar nicht gefordert ist.

Da die Push-Mitteilung für den Verlust eines Lebens schon erstellt ist (siehe Notification Manager), wird diese hier wieder gelöscht, wenn der Wunsch tatsächlich erfüllt wurde. Anschließend wird ein neuer zufälliger Wunsch in die Liste aufgenommen, der wie bei der Initialisierung erstellt wird, allerdings ausgehend von der Ablaufzeit des am spätesten ablaufenden Wunschs.

Für jeden erfüllten Wunsch erhält der Pirat Erfahrungspunkte. Dadurch kann er Level aufsteigen, was sein Aussehen verändert und dem Nutzer kleine Vorteile bringt.

### Tests

Es gibt zwei Tests für die Wünsche: man kann einen Wunsch aktivieren und einen aktiven Wunsch ablaufen lassen. Neue Wünsche zu erstellen ist nicht möglich, da dies im Spiel immer nur dann passiert, wenn davor ein anderer erfüllt wurde (Ausnahme: 1. Spielstart). Wenn man einen Wunsch erstellt ohne vorher zu löschen, kann es mehr Wünsche als Leben geben, sodass der Pirat nach seinem Tot bei inaktiver App weiterhin Wünsche angibt. Außerdem kann ein neu erstellter Wunsch, der sofort aktiv ist, die Regel verletzen, dass immer nur genau ein Wunsch aktiv ist, was zu Fehlern führt.

Beim Aktivieren eines Wunsches wird der zeitlich nächste Wunsch aus der Datenbank herausgesucht. Wenn er bereits aktiv ist, passiert nichts. Wenn er allerdings noch inaktiv ist, wird die Ablaufzeit und die ID des Wunsches gespeichert und er wird komplett gelöscht. Danach wird ein neuer Wunsch mit gleicher ID und Ablaufzeit erstellt. Als Startzeit wird die aktuelle Zeit plus 3 Sekunden festgelegt. Der Grund dafür, dass der Wunsch gelöscht und neu erstellt wird ist, dass das Verändern eines Wunsches, besonders mit den zugehörigen Push-Mitteilungen, deutlich komplizierter ist.

Dadurch, dass der Wunsch erst in 3 Sekunden aktiv wird, kann man so auch die Push-Mitteilungen testen, wenn man die App direkt schließt.

Wenn man den Ablauf eines Wunsches testet, wird zuerst geprüft, ob aktuell ein Wunsch aktiv ist. Wenn nicht, wird dies im Log vermerkt und es passiert nichts. Andernfalls wird das aktuelle Bedürfnis mit der Methode *getActiveDesire* aus der Datenbank ausgelesen und zwischengespeichert. Anschließend wird- wie bei dem anderen Test- der aktive Wunsch gelöscht und ein neuer mit gleicher Startzeit und ID erstellt. Als Ablaufzeit wird die aktuelle Zeit plus 5 Sekunden festgelegt, um das Bedürfnis noch erfüllen zu können. Die Startzeit liegt dabei in der Vergangenheit, was nur bedeutet, dass keine Push-Mitteilung mehr für den Start ausgegeben wird.

## Datenstruktur

### Datenverwaltung

Zum persistenten speichern und verwalten der Daten haben wir uns für eine SQLite Datenbank entschieden. ­

### Datenbank Manager

Als Schnittstelle für die Kommunikation mit der Datenbank wurde ein Datenbank Manager implementiert.

Dieser stellt zum einen allgemeine Funktionen für die jeweiligen Klassen zur Verfügung, als auch die Methoden zur initialen Tabellen Erstellung. Die vorher manuell angelegte Datenbank liegt bereits im Projektordner. Der Datenbankname muss beim Anlegen eines DBManager-Objekts an die Funktion „initWithDatabaseFilename“ übergeben werden. In dieser Funktion wird der Verzeichnispfad zu der Property „documentsDirectory“ hinzugefügt und die Datenbank-Datei falls notwendig in das Dokumenten-Verzeichnis kopiert.

Darüber hinaus gibt es noch die Funktion „runQuery“, die im Folgenden Erläutert wird, da sie ein Stück weit das Herzstück des DBHelper’s ist. Diese nimmt einem alle Schritte die zum Ausführen einer Datenbank-Anfrage nötig sind. Zum einen erstellt sie ein Ergebnis-Array und ein Spaltennamen-Array. Dieses ermöglicht es einen später statt einem Zugriff per Index einen Zugriff per Spaltennamen.

Bei den Anfragen wird eine allgemeine Unterscheidung zwischen ausführbaren Anfragen und nicht ausführbaren gemacht, also ob es lediglich eine Datenabfrage ist, oder ob etwas in der Datenbank ausgeführt wird. Die nötige Information dafür bekommt die „runQuery“-Funktion dadurch, dass wir sie meistens indirekt über die Funktionen „loadDataFromDB“ und „executeQuery“ aufrufen.

Diese geben eine BOOL Variable an die Funktion, in der steht, ob die Anfrage ausführbar ist oder nicht. Über diesen Wert wird in der „runQuery“-Funktion nun eine Fallunterscheidung gemacht.

Der Rest der Funktion besteht nun nur noch aus dem Auslesen der Daten und dem Befüllen der beiden angelegten Arrays.

### Inhalt und Klassen

Die Datenbank besteht im Allgemeinen aus drei Tabellen, die in dieser Version des Projektes in keiner Beziehung miteinander stehen. Gegen diese haben wir uns entschieden, weil wir in unserer App grundsätzlich nicht erlauben wollen, dass auf einem Gerät mehrere Piraten gleichzeitig angemeldet sind und ohne diese Funktionalität wären die Beziehungen nur ein Overhead (bei jeder Anfrage auf eine der Detailtabellen müsste immer ein Join über den Piraten eingebaut werden).

In den folgenden Abschnitten werden unsere drei Tabellen zusammen mit deren Funktionalitäten vorgestellt.

#### Piraten

In der Piraten-Tabelle werden sämtliche Informationen zum aktuellen Piraten verwaltet.

Der Spieler wählt zu Beginn eines neuen Spiels einen Namen aus, zusätzlich bekommt jeder neue Pirat eine bestimmte Anzahl an Leben, ein Startlevel und einen Standard Alkoholpegel zugewiesen (wo festgelegt wird welchen Wert diese Werte haben sollen und wie es umgesetzt wird siehe Kapitel TypeDef). Das Feld „geschaffteBeduerfnisse“ gibt an wie viele Bedürfnisse der Pirat seit seinem Erstellen erfolgreich erledigt hat. Dieses Feld ist auch der Richtwert für einen Level-Aufstieg.

Nach jedem geschafften Bedürfnis wird innerhalb des Piraten-Objekts die Methode „checkLevelUp“ aufgerufen, die nach ebenfalls in der TypeDef abgelegten Werten prüft, ob genug Bedürfnisse für einen Level-Aufstieg erfüllt sind. Selbiges geschieht für den Alkohol-Pegel, wo entschieden wird, ob der Pirat betrunken ist oder nicht.



Die Piratenklasse hat neben diesen Funktionen auch noch die einzelnen Funktionen, seine jeweiligen Attribute zu erhöhen oder zurück zu setzen.

Eine zusätzliche Eigenschaft dieser Klasse ist, dass der Pirat bei Spiel Start seine Klassen-Attribute mittels der „loadData“ Funktion aus der Datenbank übernimmt, wodurch wir zur Laufzeit nur eine lokale Kopie des Piraten haben und uns so ein paar auslesende Zugriffe auf die Datenbank ersparen können. Nach jeder Änderung wird die Datenbank parallel zur lokalen Instanz geupdatet um bei einem möglichen Spielabsturz nicht die Daten des Piraten-Objekts zu verlieren.

Zusätzlich gibt es noch die Funktion „saveData“, die im Allgemeinen zu Testzwecken da war, falls man bei der Entwicklung gerne ein Attribut ändern wollte, wofür es keine Funktion gab und man somit das Objekt direkt verändert hat. Mit der „saveData“-Funktion konnte man somit das komplette Objekt in die Datenbank einspeichern.

#### Lager

In der Lager-Tabelle werden die einzelnen Waren verwaltet.



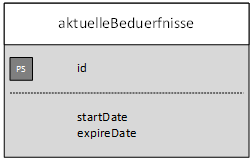
Die Waren an sich sind fix und werden bei einem neuen Spiel mit Hilfe der TypeDef-Datei in die Datenbank geschrieben. Genauso hat jeder neue Spieler auch eine gewisse Startanzahl. Das Geld wird hier ebenfalls als Ware behandelt und verwaltet. Hier sind die Kosten „1“ und die Anzahl bestimmt den Kontostand.

Das Lager-Objekt wird genauso wie das Piraten-Objekt mittels der „loadData“-Methode gefüllt. Innerhalb der Klasse gibt es noch die Lagerveraltungsfunktionen „buy“ um ein Item im Shop zu kaufen, „give“ um dem Spieler ohne weiteres ein Item zu geben (wird für die Raubzug-Option gebraucht), „sell“ um ein Objekt aus seinem Lager zu verkaufen (Verkaufspreis berechnet sich aus der Hälfte des Einkaufspreises) und „useItem“ um ein Item zu benutzen und dadurch Wünsche des Piraten zu erfüllen. Nach jedem Aufruf einer solchen Methode, die nach Überprüfung der erforderlichen Mittel auch erfolgreich ist, wird die „update“-Methode aufgerufen, die die Datenbank direkt aktualisiert.

Auch das Lager hat die „saveData“-Funktion. Diese hat den gleichen Zweck wie im Piraten.

#### Aktuelle Bedürfnisse

In der Tabelle aktuelle Bedürfnisse wird die Warteschlange der Wünsche des Piraten gespeichert.



Diese wird wie im Kapitel Bedürfnisse beschrieben initialisiert und während des Spiels aktualisiert, was immer direkt in die Datenbank geschrieben wird.

Wie beim Pirat und dem Lager werden die aktuell gespeicherten Daten mit Hilfe der loadData Funktion gelesen.

Die Bedürfnisse werden nicht zwischengespeichert, sodass während des Game Loops immer wieder aus der Datenbank gelesen wird

## TypeDef

Die „TypeDef.h“-Datei haben wir angelegt, um dort verschiedene Konstanten anzulegen.

Beispielsweise werden dort die Werte gespeichert, die bei Initialisieren eines neuen Spielstandes in die Datenbank eingetragen werden. Auch die Array-Indizes für das „supllie“-Array im Storage-Objekt werden hier angelegt. Dies ermöglicht es, nicht jedes Mal das Array durchlaufen zu müssen und nach dem richtigen Eintrag zu filtern, sondern direkt über die Konstante darauf zu zugreifen. Dadurch, dass auch die Datenbank nach diesem Schema erstellt wird, werden die Indizes immer stimmen.

Auch sämtliche Limits, wie Levelaufstieg-Grenzen, werden hier festgelegt.

# 

# Fazit

Es wurde eine Tamagotchi-App entwickelt, die als Thema in einer Piratenwelt spielt. Der Benutzer hat die Möglichkeit sich um seinen eigenen Piraten zu kümmern und zu versorgen. Der Pirat hat wie bei einem Tamagotchi üblich regelmäßig Bedürfnisse, die bei Erfüllung sogar nach einer gewissen Zeit einen LVL aufstieg ermöglichen. Jedoch bei nicht erfüllen den Tod bedeuten. Zudem hat der Pirat die Möglichkeit sich mit 2 Minispielen die Zeit zu vergnügen wodurch er sogar Münzen verdienen kann.

Ein geeignetes Datenmodell wurde geplant und erstellt. Ebenfalls eine SQL-Datenbank zum Speichern der Daten und eine animierte Benutzeroberfläche wurden entwickelt.

Trotz der reingesteckten Mühe gibt es noch Raum für Ideen und Verbesserungen. Wie z.B.:

* Mehrere Benutzer pro Handy
* Verbesserte grafische Animationen
* Zur Steuerung des Minispiels „Schiffssause“ Handybewegungen verwenden
* Mehr Sensoren verwenden wie z.B. Sprachsteuerung eines Papageien
* Lootboxen
* und noch ein paar mehr..