System design document (SDD) for project *QuizWalk*

Version: *QuizWalk, iteration 5*
Date: *2013-05-26*
Authors: *Hampus Forsvall, Markus Norén, Roy Nard, Johanna Hartman.*

1 Introduction
1.1 Design Goals
1.2 Nomenclature
2 System Design
2.1.1 Overview
2.1.2 Android paradigm
2.1.3 Domain Model
2.1.4 Event Handling/Presentation layer
2.1.5 Persistence
2.1.6 Resources
2.2 Software decomposition
2.2.1 General
Decomposition into subsystems
2.2.3 Layering
2.2.4 Dependency analysis
2.2.4 Concurrency issues
2.2.5 Persistent data management
2.5 Access control and security
2.6 Boundary conditions
3 References
4 APPENDIX

1 Introduction

QuizWalk is an application for Android-enabled devices with GPS. (See RAD for more

information)

1.1 Design Goals

To create a completely independent and flexible model domain and interface that simplifies future re-implementations for competing smartphone frameworks (e.g. iOS and Windows Phone). Moreover, to enable easy data-sharing in a client-server situations by using the common SQL standard and JSON as syntax of choice, meant for HTTP object-transfers over Internet.

Finally, to achieve as high-level abstraction of model and data-persistence as the Android SDK permits.

1.2 Nomenclature

See references for more detailed explanations:

- **Java**: general-purpose object-oriented static programming language
    - **Java SDK**: Software development kit for Java, compiles Java into bytecode (.class files).
    - **JCF**: Java Collection Framework.
- **GUI**: Graphical User Interface
- **Client-device**: Will be used to refer to the device executing the application QuizWalk.
- **MVP**: Model-View-Presentation software architecture pattern based on Model-View-Controller (MVC).
- **TDD**: Test Driven Delelopment
- **Android (Android OS)**: Unix-based operating system created by google
    - **Android SDK**: Software development kit for Android. compiles bytecode (into .dex files) from Java SDK-generated .class files.
        - **ACF (Android Core Framework)**: Provides access to the device functionality and is the paradigm one much use for Android.
        - **Dalvik-VM**: Virtual-Machine that runs dalvik-compatible bytecode. Also the only entry point for any application running on Android.
        - **Android Components**: The different entities that constitute an Android application. *Activities, Services, Content Providers and Broadcast receivers* are defined as the different components in an Android application.
        - **SQLite**: The only private database that ACF provides. Follows the SQL-standard.
- **JSON**: JavaScript Object Notation, open standard for data interchange which also is “readable by humans”.
- **JUnit**: Unit testing framework for Java used in Test-driven development.
- **Google Maps**: World-Wide Web-mapping service by Google.

    - **Google Maps API**: Application programming Interface to access Google Maps functionality and integrate it to other applications.
- **Libraries/Frameworks**: External code, either compiled or source code that contains general-purpose subroutines.
    - **GSON**: High-level Java library to serialize and deserialize objects to and from JSON-format.
    - **ORMLite**: High-level Object Relational Mapping between Java objects and SQL.
    - **Guava (Google Guava)**: General-purpose common procedure library. Extends the JCF, provides null-wrappers and other functionality.
    - **Google Play Services**: Library for Android-applications enabling usage of Google services (such as Google Maps API)

2 System Design

2.1.1 Overview

The application is at this iteration implemented for Android-devices. Domain-model is however general-purpose and independent from a conceptual object oriented modeling perspective. A heavily-modified MVP-pattern was used. Mostly to retain the integrity and independence of the domain model when forced to work in the Android SDK paradigm (which restricts usability of MVC-patterns)

All fields are final and objects in the model are immutable except one single stateful field in the main model class (`QuizWalkGame`). Local persistence is done through the provided SQL-implementation in ACF.

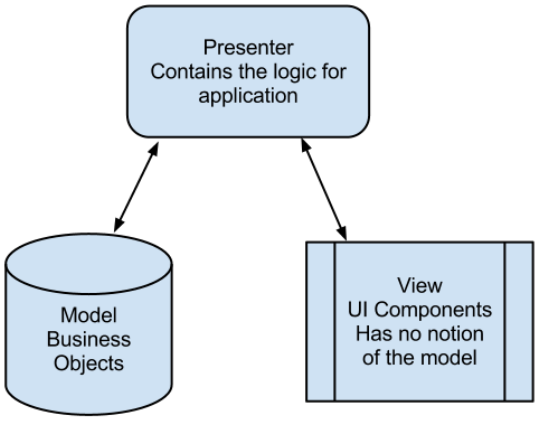


Fig 1: General conceptual model of the Model-View-Presenter pattern

2.1.2 Android paradigm

The application is implemented for the *Android ADK*, hence they are totally dependent of the *Android Collection Framework*. *Android OS* applies the principle of *least privilege* for any application it runs. Every application runs by default in its own process and unique user in the *UNIX*-based *Android OS*. This means they also run in their own instance inside of *Dalvik-VM*. An *Activity* in *Android OS* are by themselves *View* and *Control components* (from a *MVC*-perspective) that have their own life cycle - which means that only static field references in external classes will be retained when switching *Views*. *Android OS* might destroy any component at arbitrary times to retain memory if seen fit. Also, any functionality that needs implementation has to have explicit permission declaration in a *Manifest*-file. *ACF* uses also a well-defined package and folder-hierarchy, much alike *Spring* or *Java EE Web* - where static resources (layout, graphics and strings) are stored by convention in specific *XML*-files.

Coding in the Android paradigm is therefore vastly different from coding from a main-method and running the application on a desktop *Java Virtual Machine*.

Nevertheless, classic TDD is not applicable in Android development since it enforces template and inheritance over composition for View/Controller. The only classes testable in a classic TDD-sense are the POJO-classes of the domain model.

2.1.3 Domain Model

The main entry point for the business logic are the *Activities* and *Components* as defined in MVP and the constraints of *AFC*. The model business object for QuizWalk is constituted strictly by immutable objects in package "`model`", with one exception - a stateful Map-object field in the main class `QuizWalkGame`. All fields This allows convenient manipulation from the presentation layer, both in state changes and persistence of different games. Builder-pattern classes have also been created within the model for the relevant classes to simplify dynamic creation and recreation of the immutable objects from the presentation layer. The model does not permit null values and should not ever return null. Instead, the null-wrapper class `Optional<>` from *Google Guava* and other convenience methods are used to ensure that no invalid data will return a new object. Fast-failing mechanisms are also in place to ensure more efficient development process and debugging. No interfaces were extracted from the main model class since it can not be referenced from the presentation layer due to field reflection done by *ORMLite* and would therefore only be a template with no references. The essential methods are presented below of the main classes.

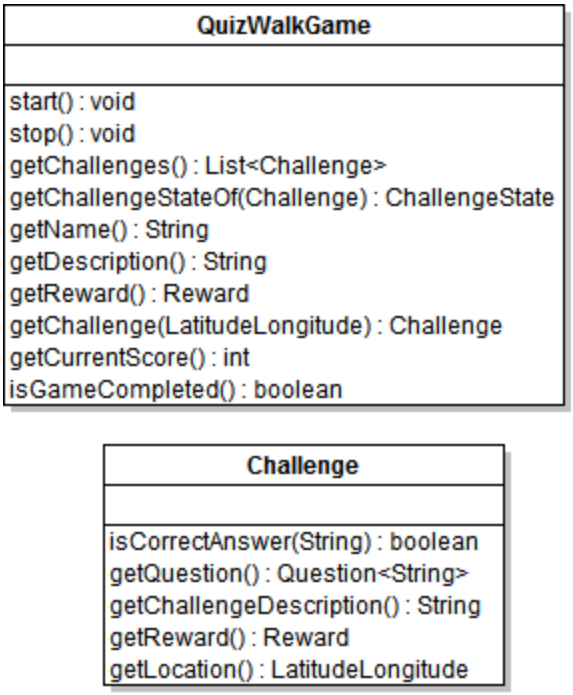


Fig 2: Essential public methods of the main classes of the model domain

.

2.1.4 Event Handling/Presentation layer

All GUI can either be coded into a *Component* or defined in XML-View files. Following best practices for Android-enabled devices, *Presentation* in MVP is loosely defined as an *Activity* Component. Event handling is instantiated by application-external components such as the Android OS or Google Maps API (For GPS-location change). All these classes are packaged into "`presentation`". For instance, if the user clicks on a location on the map view, Google Maps API will call its *custom* method defined in some *custom* listener referenced from the Activity.

To handle the constraints of not being able to directly pass object references to other application components a stateful singleton class is used which manages volatile states whilst application is in use.

2.1.5 Persistence

ACF offers an *SQL*-implementation class called *SQLite* that enables local persistence for the user device. To make the domain model as decoupled from Androids own implementation it has been modeled to avoid any dependency of the *SQLite* itself - favoring a higher-level handling of objects through *ORMLite*. With *ORMLite* classes and their fields are annotated which in turn maps these to the underlying relational database. Around this, helper and manager classes are

defined - The *Presentation* layer will only need to call high-level methods (e.g. `updateQuizWalk(QuizWalk)`) in order to persist data into local storage. A user can interact with the map to create their own `QuizWalkGame`-object and persist it to the database.
Also, the application is compatible with future implementations of a client/server model. This is done for example by implementing a dynamic *HTTP* server with *MySQL*. Data-interchange between client and server over the Internet with *JSON* would then be completely compliant.

2.1.6 Resources

The template concept in the *ACF* enables application static resources to be organized into a well-defined folder hierarchies. When *Android SDK* builds the project it generates a global identifier class file pointing to memory location. *Layouts*, *Strings* and graphics are all grouped this way. *Strings* specific for the model domain is however not dependent on Android SDK philosophy of templating. QuizWalk is also modeled for future internationalization in such way by externalizing resources.

2.2 Software decomposition

2.2.1 General

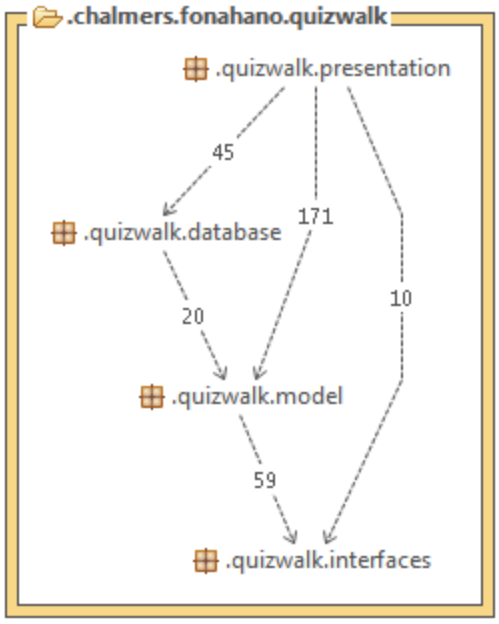


Fig 3: Package layering

- presentation: A pseudo View/Controller pattern suited for *ACF*. Inflates static *XML*-layouts.
- Resources: Graphics and .xml files describing Strings, Images and Layouts.
- model: Model part, with essentially no logic, in the MVP-pattern.
- database: Object-Relational-Mapping to the systems underlying *SQLite* database.
- interfaces: describes methods that the business model must implement

Decomposition into subsystems

2.2.3 Layering

(See Fig 3 and 1)
The presentation layer which represents both logic and view is the top layer - with references to all other layers. Database layer only has references to the model, while the model itself is the business implementation of the specified interfaces.

2.2.4 Dependency analysis

See Figure above (Fig 3 and 1). There are no circular dependencies.
The database layer depends on the local *SQLite* database through the *ORMLite* mapping. While the business model depends on *Guava* strictly for convenience methods. The presentation layer in turn completes the implementation by using Google Maps API to both show and enable interaction with GUI map.

2.2.4 Concurrency issues

The application itself runs on a single thread - Google Maps API and *Google Play Services* in turn run on multiple threads that QuizWalk itself is unaware of.

2.2.5 Persistent data management

As mentioned in chapter 2.1.5 the database handling is implemented by the only available *SQL*-standard database on the *ACF: SQLite*. In the philosophy of Object-Oriented programming an Object Relational Mapping library *ORMLite* is used by annotating fields that are to be persisted. Helper and managers classes are defined in the database package in order for high-level method calls from the *presentation* layer.

2.5 Access control and security

Running an application in *Dalvik-VM* on Android OS gives it the least access privilege possible. It

also isolates every application as far as possible in a *UNIX*-based OS. Theoretically, this makes it both difficult for the application to in a hidden manner be altered or alter any other process. However, as with any application information is passed in volatile memory and can be read if a black-hat has physical access. Also fields are declared final and objects are generally immutable. It fails fast if it senses invalid data. Passwords entered by users of QuizWalk are however hashed (*MD5*) and not saved raw.

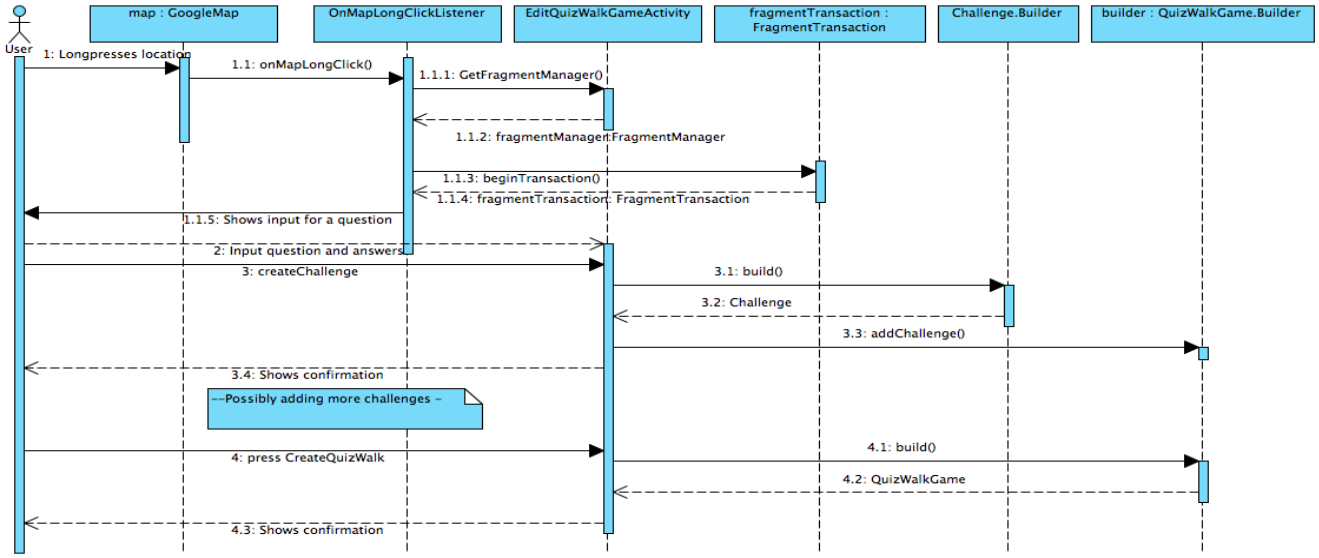
2.6 Boundary conditions

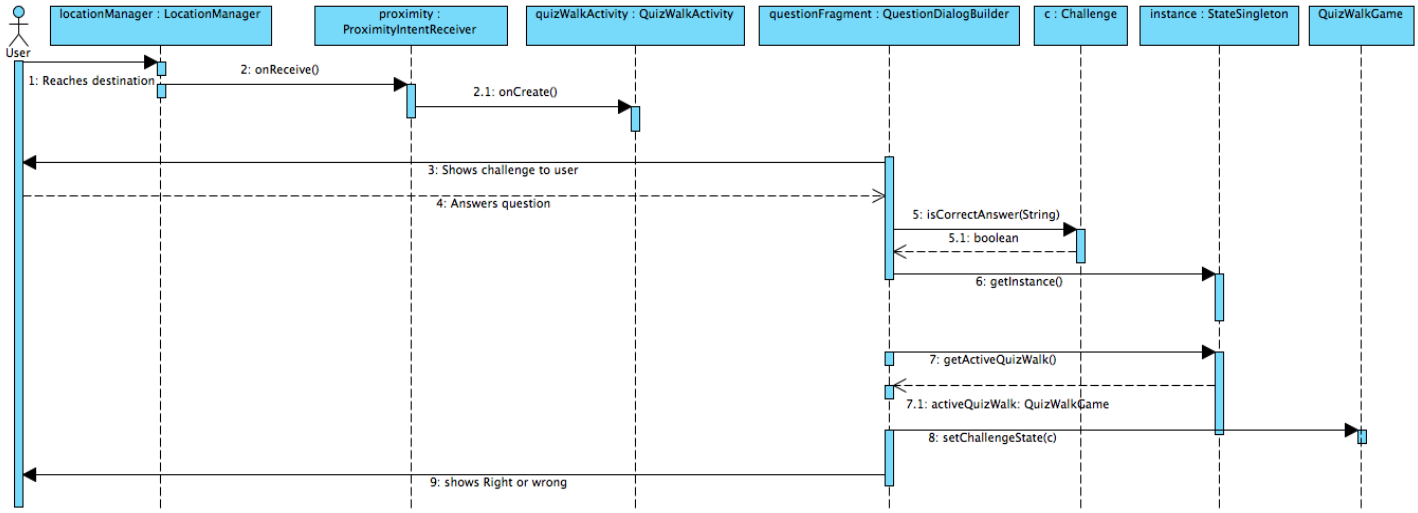
The application will start, initialize, work and stop as expected by a typical user on the Android device. If the device misses any of the required services (e.g. storage and GPS) the user will be prompted and application halted.

3 References

1. **MVP** http://en.wikipedia.org/wiki/Model_View_Presenter
2. **Android OS** http://www.android.com/about/
3. **Android SDK** http://developer.android.com/about/index.html
4. **JSON** http://en.wikipedia.org/wiki/JSON
5. **Google Maps API** (Android)
https://developers.google.com/maps/documentation/android/
6. **GSON** https://code.google.com/p/google-gson/
7. **ORMLite** http://ormlite.com/sqlite_java_android_orm.shtml
8. **Google Guava** https://code.google.com/p/guava-libraries/

4 APPENDIX


User
map : GoogleMap
OnMapLongClickListener
EditQuizWalkGameActivity
fragmentTransaction : FragmentTransaction
Challenge.Builder
builder : QuizWalkGame.Builder
1: Longpresses location
1.1: onMapLongClick()
1.1.1: GetFragmentManager()
1.1.2: fragmentManager:FragmentManager
1.1.3: beginTransaction()
1.1.4: fragmentTransaction: FragmentTransaction
1.1.5: Shows input for a question
2: Input question and answers
3: createChallenge
3.1: build()
3.2: Challenge
3.3: addChallenge()
3.4: Shows confirmation
--Possibly adding more challenges --
4: press CreateQuizWalk
4.1: build()
4.2: QuizWalkGame
4.3: Shows confirmation


User
locationManager : LocationManager
proximity : ProximityIntentReceiver
quizWalkActivity : QuizWalkActivity
questionFragment : QuestionDialogBuilder
c : Challenge
instance : StateSingleton
QuizWalkGame
1: Reaches destination
2: onReceive()
2.1: onCreate()
3: Shows challenge to user
4: Answers question
5: isCorrectAnswer(String)
5.1: boolean
6: getInstance()
7: getActiveQuizWalk()
7.1: activeQuizWalk: QuizWalkGame
8: setChallengeState(c)
9: shows Right or wrong