Developer manual

General idea and patterns

Architecture

The general idea behind the structure of the application is to try to keep the model code such as database calls as separate as possible from the controllers and view. Although the application is not using a strict MVC architecture the general idea is that the XML, `Overlay`, `View` and `Dialog` classes are representing the View, `Activity` and `Listener` classes are used as Controllers and the rest is basically Model code.

Design pattern

Callbacks

The main design pattern that is used is the `Callbacks` pattern which is used extensively in the Android core. This pattern is great and simple and allows each class to have it's own `Callbacks` interface which the using class can implement. This allows each class to contain its own calculations when an event is fired and the class implementing the callback doesn't need to worry about listening or calculating any information that is not relevant.

Listeners

We have strived to try and keep `Listener` classes external to the class in which they are used as much as needed to keep the using class as clean as possible. Any listener that is doing a lot of work can be located in the listener package. Some listeners have not been refactored out, either due to lack of time or that the listener just doesn't clutter the code much, yet. Any listener that is becoming more complex should immediately be refactored out to its own class to prevent cumbersome refactorisations in the future.

Future and refactorisations

Refactorisations

`RouteActivity` is now the main core of the application where a lot of work is being done. When the application development started we initially had two activities to handle the recording and usage of an old `Route`. Since they where very similar though, we refactored them into a single `Activity`. There is a possibility that this activity will continue to swell in future development so one should keep an eye on it and if needed the activity could be broken into an abstract base class which contains any common executing code and then two activities which inherit from this base class to implement the specifics of each screen.

Another area which might benefit from some extra attention is the package structure. It currently contains some packages with only one class and there is no structure in the packages to show which parts are which in the MVC architecture. When time is found, it is recommended to do such a refactorisation for a more simple and clean package structure.

Implement before release

The application in its current state is working well. There are however, some minor changes that would be needed for a successful launch when releasing the application to the public. These features are; implement a "first time start" helping screen that informs the user on how to use the application.

Another feature that is critical is to redesign the home page to also display a "Results" icon which would take the user directly to a list of routes and then on to a list of results for that specific route.

Experimental features

We have been experimenting with Dropbox syncronisation and we have a complete `DropboxHandler` class to handle uploads and downloads in a special "dropbox-experiment" branch. This class is located in the util package in that branch. Note that it does not perform any syncing so far but it could be used if the syncronisation or backup features are ever implemented. It is also missing an activity that can use it and it will most likely throw alot of errors since the Dropbox libraries are not included in the branch. The key and secret have also been removed since they are application specific and should be kept secret.

Traceability

To help in debugging, backtracking and maintaining a clean structure for development of the application we use GitHub issues tracker to track all changes we make to the application. An issue can be a complete feature or a small bug fix. All major issues are developed in their own separate branches which are named with the issue number first, then a small, easy to spell, description. For example: `XX-description` could be `47-stats` would be the branch name for issue 47 which is used to develop the statistics graphics branch.

Any minor bug fix should be fixed in the feature branch it belongs to. Sometimes though, a bug cannot be directly connected to a feature. We then use the branch "hotfixes" and make sure to add the issue number to the commit to make it easier to trace. A message could be something like: "`Fix: #123 message not displayed to the user when recording a message`".

This information about traceability is also available in the "Traceability" document.

Tests

The tests for the application are divided into two different categories, UI-tests and unit-tests. We use Robotium for the UI-tests to make sure the Views and navigation are working as expected. We try to use UI-tests for as many user interactions as possible in order to cover as much of the manual testing as possible. The UI-tests have a great advantage over unit-tests in that they do test a lot of code. One UI-test can test several activities, dialogs and even the database connection. Some code can however, be hard to test with UI-tests. We then resolve to unit-testing to make sure the methods and classes not covered by the robotium tests, deliver the expected results. Unit tests are also needed to test extreme input values.

The UI-tests can all be found in the `se.team05.tests.ui` package and the unit tests are located in the `se.team05.tests.<package-name-for-tested-class>` packages.

We aim to have a coverage of at least 90% since some code can be very hard to test in the system. It is hard to simulate incoming phone-calls, wake locks and the failure of creation of services. We therefore also use manual tests for the scenarios mentioned. These tests can be found in the manual test report document. We use Apache Ant builds together with Emma to get coverage reports that can be used in Eclipse. Note that the `coverage.ec` file we get from Emma is only supported in EclEmma version < 2. The recommended version to use is 1.5.3. The `coverage.ec` file is located in the root of the test project. The tests can be executed from the command line by running the `test.py` script in the root of the test project but not before running `setupAnt.py` at least once in the root of the main project. Note that Eclipse can mess up the project properties, any errors are usually resolved with "project->clean". Also note that you need to add some media/mp3 file to the sd-card of the device you are testing for the media tests to run properly.

Creating or importing the test project

The prefered way to manage tests in a android project is to set up them up in a separate project.This is because the tests are compiled into a separate application that performs testing

on the main application. Before compiling the test project in eclipse, it is important to make sure it is aware of the main project by including said project in the test project's build path. This can be found in eclipse by right-clicking the test project and choosing "Properties". Under "Java Build Path" there is a tab named "Projects", the main project must be listed there. If it is not, it is easy to just add it.
External libraries such as Robotium should be recognized automatically by eclipse and added to the build path. This can be verified by checking the "Libraries"-tab and under "Android Dependencies". The jar file should be listed there.

Build

The project can be built both from Eclipse and from the command line by using Ant.
There are build files located in both the main project directory and in the tests project root directory. When switching between builds you have to uninstall any previous installation of the application. Eclipse may also mess up the project properties
so "project->clean" can be used to fix a lot of issues.

Application flow

Startup

The application starts with the `MainActivity` where the user is presented with the primary navigation of the application. There are three images, add route, use an existing route and settings that can be clicked. Each of these clicks are listened to in the `MainActivityButtonListener` and when clicked the corresponding activity is launched.

Recording

When the user is recording a new route this is done in the `RouteActivity`, several options are available through the user interface. There are buttons to start and end the recording of a route which uses a simple click listener, the map view itself can also be tapped and it also has it's own listener to decide which latitude and longitude was tapped.

On tapping the map a popup dialog will appear which allows the user to add a `CheckPoint`. From here the user can select a name and radius for when the checkpoint is active. It is also possible for the user to record a command or instruction via the "Start recording" button. When the recording stops a new sound file is saved on the sd-card and the user can then select it by clicking the "Select" button. This button launches a new activity which queries the media store for information about music media on the device and lists all the known media. This list can be sorted in three different ways through an options menu which triggers new queries to the media store with different sort parameters.

When the user has chosen to stop recording another dialog which asks for name and description for the route is displayed. The results (time/speed/stretch etc) is shown to the user and said user is given an option to save the results or throw them away. The user can also choose to discard the route which will throw away all route information.

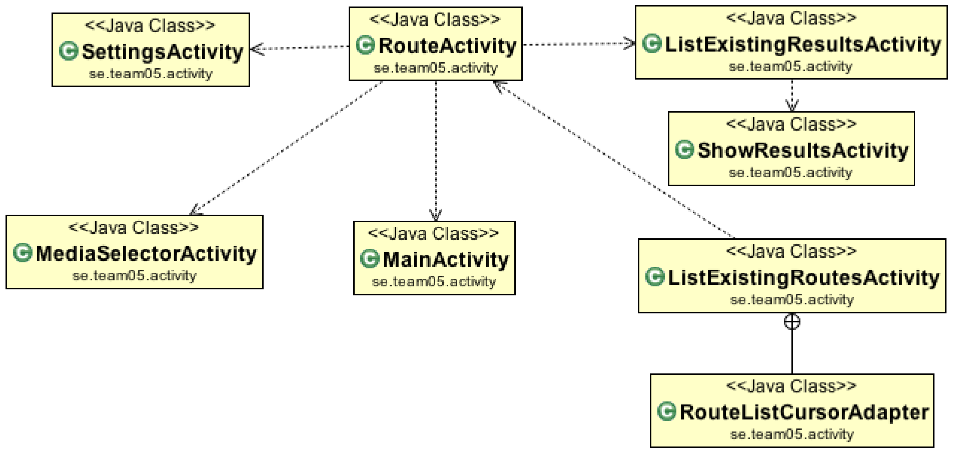
Using an existing route

When using an existing route the `RouteActivity` is also used. This time the recording functionality is disabled and the user is only prompted if they wish to save their results or not through a custom dialog.

Application structure

Activity package

This package contains all the activities.



ListExistingResultsActivity

An activity that will present the user with the option to view results of an old route. Gets results from database and presents them in a listview.

ListExistingRoutesActivity

An activity that will present the user with the option to choose and old route. Gets routes from database and presents them in a listview.

MainActivity

This is the launching point of the application. The main activity simply displays a few buttons which allow the user to choose what action to take with the application. The main activity uses the external listener `MainActivityButtonListener` to decide which `Activity` to launch next.

MediaSelectorActivity

This activity displays a list of the available media on the device. It allows selecting several items from the list and by selecting the "done" icon in the options menu, the activity will return the results to the calling activity. The list can be sorted via the options menu. The available sorting columns are artist, title and album. By default the list is sorted by artist name. The selection from the database consists of the `_ID`, `ARTIST`, `ALBUM`, `TITLE`, `DATA`, `DISPLAY_NAME` and `DURATION` columns and is also limited to contain only files that are marked as `IS_MUSIC`.

RouteActivity

This activity presents a map to the user that is used to display routes and checkpoints on top of the map as overlays. It also tracks the users movement and will draw the route as the user moves. This is accomplished by using Google's map API.

ShowResultsActivity

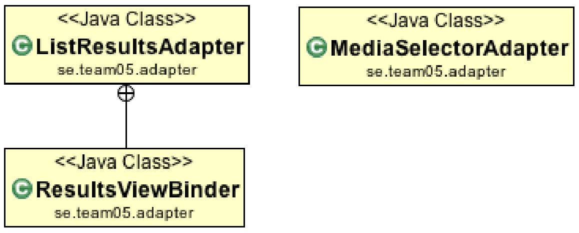
An activity that will present a result of an old route. Gets results from database and present them. The activity makes it possible to delete the results by pressing the Delete button, which activates the external listener ShowResultActivityButtonListener.

SettingsActivity

This class represents what the user has stored in his or her settings. It acts mainly as a visual representation of what is stored and allows the user to make changes. The actual storing is made in the `Settings` class.

Adapter package

This package contains the adapters used to populate list views.



ListResultsAdapter

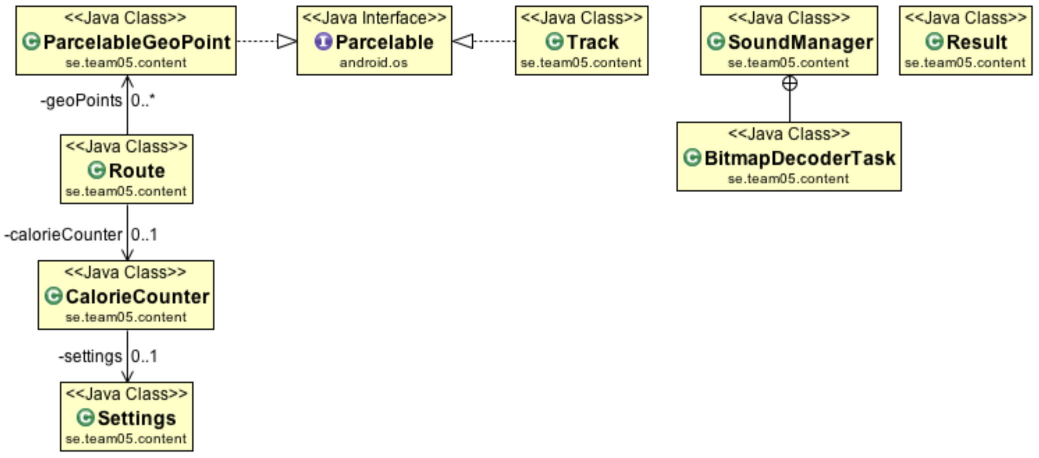
This adapter is used to bind the views in the result list to their appropriate column in the Cursor. It sets a new custom `ViewBinder` to do the actual binding.

MediaSelectorAdapter

This adapter is used by the media selector activity to display the list rows. It is needed to keep track of which checkboxes have been checked and which has not. The system is aggressive in trying to reuse views that are not currently being displayed which leads to strange behaviour with the checkboxes where they keep their "checked" state although they have not been checked for a specific item. The class is extending `SimpleCursorAdapter` for easy use of the cursor that can be obtained from a database or content resolver.

Content package

This package contains data objects and plain old java objects that are used to hold content.



CalorieCounter

A simple class for calculating calorie expenditure. This was made from a simple formula for calculating burned calories from walking or running and then calculating two factors, one for walking and one for running. Which factor to use is decided by the user in the settings activity. This factor is then multiplied with users weight and distance measured in metres.

ParcelableGeoPoint

This class is created to be able to save the geo points when a configuration changed has occured. The class implements the `Parcelable` interface so that it can be easily transferred via intents when the application is switching between activities.

Result

This class is the result class used when saving results after finishing a route. It contains several constructors in order to be be flexible during development of the android application.

Route

This class contains all data related to a single route plus all the geopoints and checkpoints associated with the route. It also provides several getters and setters for the various instance variables. It simply acts as a model in the mvc structure.

Settings

This class contains user settings. It works as a model in that it retains the data that it saves between sessions and also supplies other methods with said data. This is made possible by the use of the `SharedPreferences` library.

SoundManager

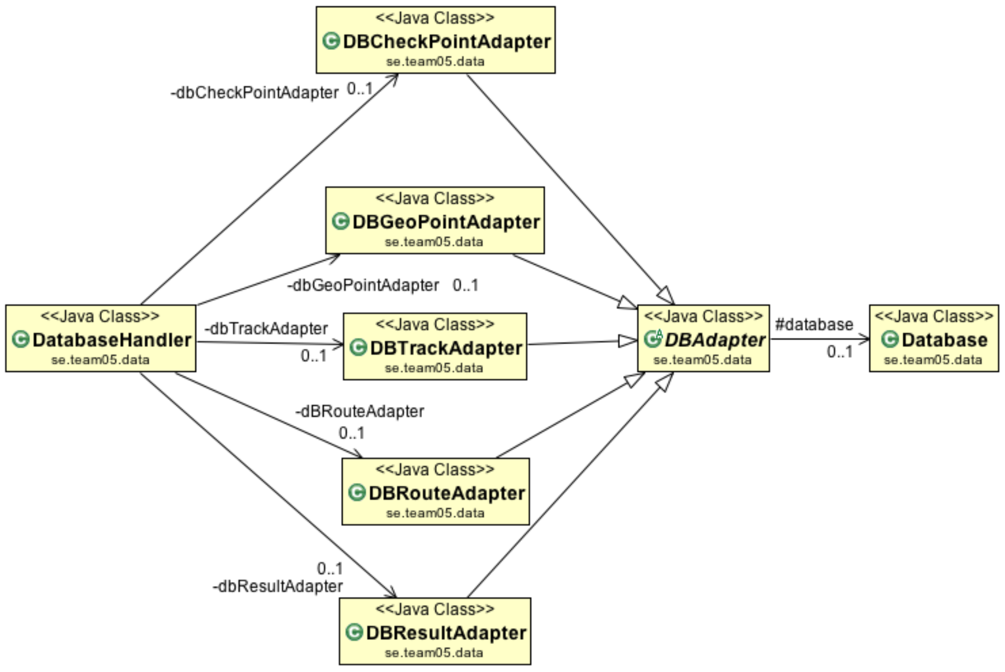
This class manages playing of audio from an audio pool and recording of new audios.. This class should only be used for audio files that are less than 1MB in size. For playing music the media player or a media service should be used instead. The audio files it records are stored in `/mnt/sdcard/Music/personal-trainer/` and each audio file also has an album art image in this directory which is created by the launcher icon on the first run.

Track

This is a classic plain old java object which stores information about a specific music track that the user has selected. The class implements the `Parcelable` interface so that it can be easily transferred via intents when the application is switching between activities.

Data package

This package contains all the database handling that the application uses.



Database

This class handles creation, updating and deletion of the database. It uses the adapter classes constants for table creation. This class also defines the database version number and its filename.

DatabaseHandler

This class handles the communication between the database, its adapters and the rest of the application. It keeps an instance of each table adapter that is used for quick and easy access to the database.

DBAdapter

This is an abstract class that works as a base for all the table adapter classes. It handles fetching the writable database, open and closing of the database. Any table adapter should extend this class to get said functionality for free.

DBCheckPointAdapter

This class is the checkpoint adapter class used to communicate with the "checkpoints" table in the SQLite database. It contains information about the columns, basic Create, Read, Delete operations and also the initial "create table" statement.

DBGeoPointAdapter

This class is the geopoint adapter class used to communicate with the "geopoints" table in the SQLite database. It contains information about the columns, basic Create, Read, Delete operations and also the initial "create table" statement.

DBResultAdapter

This class is the result adapter class used to communicate with the "result" table in the SQLite database. It contains information about the columns, and basic Create, Read, Delete operations and also the initial "create table" statement.

DBRouteAdapter

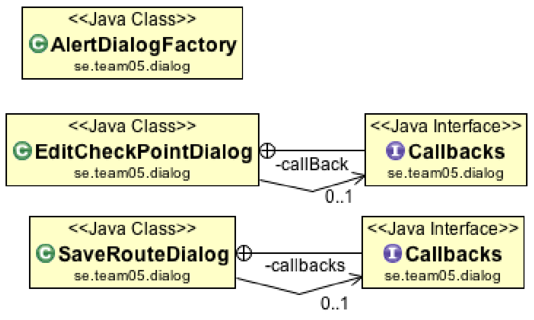
This class is the route table adapter class used to communicate with the "routes" table in the SQLite database. It contains information about the columns, basic Create, Read, Update, Delete operations and also the initial "create table" statement.

DBTrackAdapter

This class is the track adapter class used to communicate with the "tracks" table in the SQLite database. It contains information about the columns, basic Create, Read, Delete operations and also the initial "create table" statement.

Dialog package

This package contains the specialised Dialog classes and the alert dialog factory.



AlertDialogFactory

This factory class is used to create alert dialog instances for the save result dialog, confirm back press dialog and others. It always returns a created instance so remember to call show() on the alert dialog that is returned.

EditCheckPointDialog

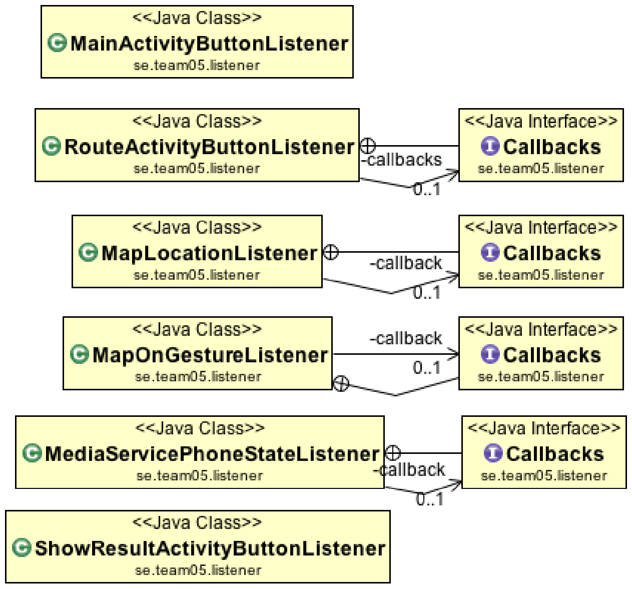
This is the dialog that pops up when a checkpoint is created or touched. The Dialog has settings such as name & radius and buttons for deleting and saving the checkpoint.

SaveRouteDialog

This Dialog class is shown to the user when the user has recorded a new route and wishes to finish recording it. It has inputs for the name and description of the route and it also has a checkbox which is connected to a boolean value which decides if the results that the user had (time, length) should also be saved with the route.

Listener package

This package contains the big listener classes that are used in the application. They usually use a Callbacks interface to communicate back to the caller.



MainActivityButtonListener

This is the listener class for the main activity and it takes a Context as a parameter which is used to launch the next `Activity`. This class is responsible for deciding which button was clicked and it then responds accordingly by launching the activity that the user wishes to see next. If no matching `View` can be found we just do a default return.

MapLocationListener

The `MapLocationListener` class is a `LocationListener` which uses callback to update the location. It also checks to see if the current location matches a checkpoint location in which case it starts the music service if tracks are set to play.

MapOnGestureListener

The `MapOnGestureListener` class is a `OnGestureListener` and a `OnDoubleTapListener` and listen after single tap's and double tap's to call the `onTap` method in callback with different event types.

MediaServicePhoneStateListener

This is a listener that is used to handle incoming phone calls and phone calls ending. It has a custom interface which is used to implement the Callbacks pattern.

RouteActivityButtonListener

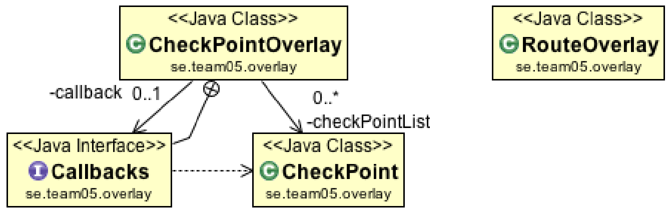
This listener is used by the route activity and it has callbacks that the route activity implements to perform the requested action. This interface uses the methods `onStartRouteClick, onStopRouteClick, onAddCheckPointClick` and `onShowResultClick`.

ShowResultActivityButtonListener

This is the listener class for the `ShowResultsActivity`. This class responds to button clicked and deletes a specified result.

Overlay package

This package contains the different map overlays.



CheckPoint

The `Checkpoint` class in which we can set different attributes and contains a geo point.

CheckPointOverlay

The `CheckPointOverLay` class contains a list of checkpoints and a default marker that will show up on map.

RouteOverlay

This `Overlay` uses a list of geopoints to draw a path on a map view. At the start and end point an oval is also drawn to mark the start and finish of the route.

Service package

MediaService

This `Service` class is used to play media from a device. It uses a media player which is loaded and prepared asynchronously and also shows a notification that the service is running as a foreground service to the user. It uses a custom phone state listener to listen for the interesting events from the system and then implements a custom

`Callbacks` interface to be able to receive information about the events. It requires a playlist to be passed in as a string array list extra with information about where the media it is supposed to play is located.

Util package

Utils

This utility class is used to perform system services such as acquiring and releasing the wake lock and also to start a timer that updates a callback through the `Callbacks` interface.

View package

EditRouteMapView

The `EditRouteMapView` class is a subclass from mapView with an overwritten `onTouchEvent` method with a gestureDetector which is used for detecting single and double taps.

TimeStretchChartView

This class uses the library achartengine and draws a time chart with the time passed on each result.