Application Design Document

MGMapViewer

Table of Contents

1 Introduction

This document is considered as the application design document for the Android app MGMapViewer. It tries to summarize information about this app, which are relevant in the development process. These information concern architectural aspects of the app and also patterns that were used to ensure a consistent code structure throughout the whole app.

Chapter 2 gives on overview over the top level components of the app. Chapter 3 contains a subchapter for each top level component that describes the most relevant aspects from development point of view. Finally chapter 4 introduces in more detail the common models and some techniques that were used in the whole app.

2 Architectural Overview

This chapter tries to give a rough overview about the app architecture of MGMapViewer.

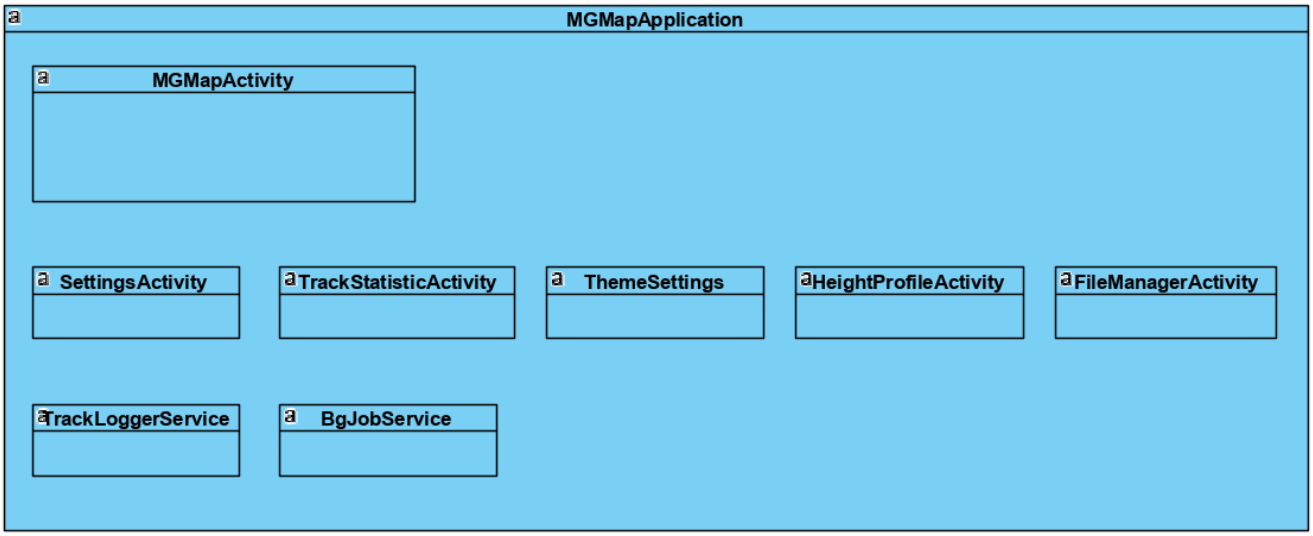

Figure 1: Architectural Overview of MGMapApplication

This figure reflects the main items of the app as listed in the Android Manifest file.

The application MGMapApplication provides the basis for this app. Especially it provides the ability to manage a set of TrackLog instances and to register observer on them. Secondly, it provides via the BgJobService the option to execute long running jobs in the background.

Beside the application that is given with the MGMapApplication class there are five activities:

- MGMapActivity: This is the main activity and by far the most important one. It's contains
  - all kinds of map visualisation
  - all track visualisations,
  - status information (dashboard, status line)
  - quick controls (menus, menu items)
- SettingsActivity: Provides multiple preference screens for settings, to trigger actions an
  - adjust application settings
  - trigger downloads (maps & software)
  - provide information and link to documentation
- TrackStatisticActivity: A table with some basic information about all tracks that are known to the application. It allows to trigger some actions like view, save, delete, share of a selected subset of tracks. A filter can be applied.
- ThemeSettings: Provides an extra preference screen for adjusting all settings that are part of the mapsforge themes.
- HeightProfileActivity: This activity provides height profiles for some tracks.
- FileManagerActivity: Allows to rename, view, share, receive via share, delete files in the private storage path of the app.

Beside the activities there are two services defined via the manifest:

- TrackLoggerServices: This service is the basis for three background task:
  - GNSS location service as needed for track recording
  - pressure sensor service as needed for altitude calculation
  - TextToSpeach service as needed for turning instructions
- BgJobService: This is a general purpose background job service, use e.g. for
  - Download jobs for map files and themes
  - Download jobs of tiles for TileStores
  - Down- and Upload jobs of gpx track to GDrive

3 Components of the MGMapViewer

3.1 MGMapApplication

The application context provides following data:

- `lastPositionsObservable`
  Provide current position for direct access and as observable.
- `availableTrackLogsObservable`
  Provide selected track and multiple available tracks for direct access and as observable. For the selected track there is a TrackLogRef used that enables to point on a particular segment of the track.
- `recordingTrackLogObservable`
  Provide recording track log for direct access and as observable.
- `markerTrackLogObservable`
  Provide marker track log for direct access and as observable.
- `routeTrackLogObservable`
  Provide route track log for direct access and as observable.

The onCreate callback starts a couple of Threads:

- Recover recordingTrackLog state from persistent storage, set GPS state depending on recovered recordingTrackLog state (finish after initialization)
- Synchronize .gpx files to .meta files, load meta file data to TrackLog objects, which provide fast access on most track properties without parsing the xml of the gpx. Finish after initialization.
- Start a Thread to handle new TrackLog points, runs permanently.
- Start logging supervision thread. The app is starting right at the beginning a logcat process for recording of log entries in logfiles. Since this separate process might be killed by Android, this thread checks the log process and if killed, it restarts it. This thread runs also permanently.

Beside providing access to the observables and the startup functionality there is on more thing: MGMapApplication provides access to the BgJobService. BgJob instances can be submitted to the job queue. If not yet running, the application starts the BgJobService. This is related to an Android notification with the number of not yet started BgJob instance waiting in the queue.

3.2 MGMapActivity

This is the main activity of the app. It provides map and track viewing capabilities. It’s running fix in portrait mode. The first subchapter will introduce the architecture of views, as they all together compose what we see in this activity. The second subchapter will describe the software structure of the MGMapActivity. Since most of the functionality of this app is related to this activity, the code is structured in multiple features, which are tried to be implemented rather independent, mostly in so called FeatureServices. The third subchapter will point out some of the Feature services in more detail.

3.2.1 View model of MGMapActivity

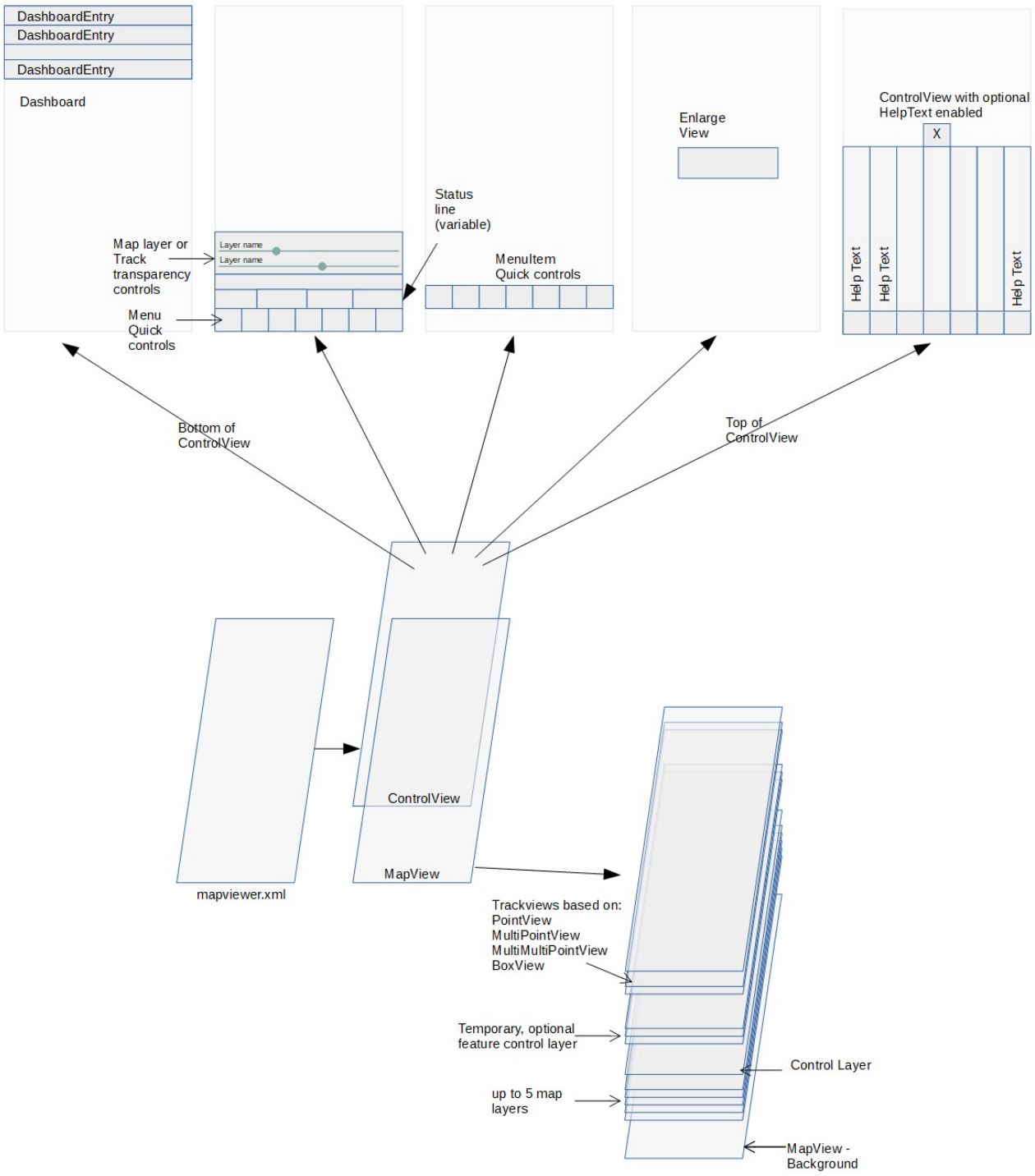

Figure 2: View Model

The views are splitted into two main groups:

1. MapView based view elements. The MapView is the central view object as provided by the mapsforge project. It allows to show multiple map layers. Independent on the fact that they show a whole map ore just a small circle of a point, they all have a strong relationship to a geographical position, so moving the viewed position will move all these objects together. There are also a few control layers with direct relationship to the corresponding view objects.

2. ControlView based view elements. These views provide additional information, either in textual form or as drawable icons. These view elements are also used to provide most of the control functionality of this activity. All these view elements have no relationship to the currently visual geographic position, so moving the map will not affect them.

3.2.2 Software structure of MGMapActivity

The MGMapActivity is using a MapViewerBase class as its parent. This class tries (together with the MapViewUtility) to encapsulate most MapView related topics. Again it's worth to mention that the viewing capabilities for maps are mostly provided by the mapsforge software[1] that is used in this activity. So rather at the beginning of the onCreate() method the MapView will be initialized and also the map layers will be created.

The additional functionality provided by MGMapActivity requires a none trivial amount of code. Therefore the code is structured in features, which operate rather independent on each other. Most of these features have a similar structure with following functionality

- provide some controls to trigger an action,
- trigger or do some action in background
- provide some status information (either as text or as drawable icon)
- use observes to refresh the status information on demand

Therefore a base class FeatureServices was introduced that provides common functionality, e.g. access to the MGMapActivity, MGMapApplication and some other classes. It enables also to bind these features to the lifecycle of the activity. So the onResume() and onPause() calls are forwarded to the feature services.

Each instance of a feature service includes by default an refreshObserver, which can be used to register it to relevant observables. Once an update is triggered by these observables, the refreshObserver triggers after a short timeout a doRefresh() method. The default implementation checks the activity state an if it is resumed it calls doRefreshResumed(). And also this method has a default implementation: it executes on the UI Thread the method doRefreshResumedUI(). For most feature services this is very convenient, since they have to do some updates only in RESUMED state on the UI.

All feature services are created in MGMapActivity onCreate() method. MGMapActivity manages a list of these feature services and provides a generic method to identify such a service if needed. Nevertheless, to realize mostly independent services this option is rarely used.

1 There are some functional enhancements as well as some fixes which were provided to the mapsforge project as pull requests.

Once the feature services are created, the ControlView is setup. The features should not now details about the control view layout. The control view class provides helper methods to setup all kinds of buttons, seek bars and other views for dashboard, status line and quick controls, but this code should not be mixed up with functional aspects of the features. Therefore an extra class ControlComposer is created. For most control components it call a helper method from control view to create it and assign it to its parent. The resulting component is given to an initXYZ() call of the feature service that shall provide a certain functionality to this view, so e.g.

```
initStatusLine(coView.createStatusLineETV(parent, 20), "height")
```

The "`coView.createStatusLineETV(parent, 20)`" call creates the status line view object, which is given with the info "height" to the initStatusLine method of the FSPosition feature service. This feature service will update this view in its doRefreshResumedUI method according to the current state.

Similar to the initStatusLine method there are

- initDashboard
- initLabeledSlider
- initQuickControl

methods to setup the whole bunch of control view elements. The ControlComposer is the central place where the controls and it's functionality are linked together. Once the control view setup is done, the onCreate method of the MGMapActivity is finished.

The onResume() method forwards this to all feature services to enable them to react on this event. Additionally it triggers (via changed method) a couple of the applications observables. So all observers of them can recalculate and visualize the current state.

Similar the onPause() method mainly propagates this event to the feature services, which may react on it.

Finally the onDestroy() method is doing some clean-up stuff. Be aware that there are some Settings, especially from the MapView, which require a recreate of the activity to become effective. So not only Android may destroy the activity to free some resources, also the SettingsActivity can trigger this lifecycle changes.

Beside the already explained stuff there are a few more aspects worth to be mentioned:

- MGMapActivity is registered for a few intents:
  - uri scheme `"mf-v4-map"` for map download
  - uri scheme `"mf-theme"` for theme download
  - uri scheme `"mgmap-install"` for download and install zip-Archives (e.g. used to install map sample configurations)
  - uri scheme `"geo"` to open geo location intents
  - uri scheme `"content"` with mime type `"application/gpx"` (and some othe types) to open tracks directly from other apps
  - intent type `"mgmap/showTrack"` to show tracks as selected in TrackStatisticActivity
  - intent type `"mgmap/markTrack"` to open the selected track in TrackStatisticActivity as marker track

- MGMapActivity starts/stops depending on the required GPS state the TrackLoggerService. It takes care about requesting the necessary permissions, if not yet available.
- MGMapActivity provides XmlRenderTheme as used for mapsforge maps
- MGMapViewer creates a controls layer in the MapView, which enables
  - on single tap: set a track as “selected”
  - on long tap: toggle gain/loss mode for
    - recordingTrackLog,
    - selectedTrackLog and
    - routeTrackLog.

3.2.3 Feature Services

The following table gives an overview on all feature services and tries list dependencies, especially in terms of preferences.

Feature Service	Function of Feature Service	Dependencies

`FSAlpha`	Provides transparency controls for map layers and for all kinds of tracks	`FSATL.STL_visibility`
		`FSATL.ATL_visibility`
		`FSMarker.MTL_visibility`
		`FSRecording.RTL_visibility`
		(for visibility of corresponding seek bars)

`FSAvailableTrackLogs`	Visualisation of available tracks and selected track (incl dashboard)	`FSMarker.MTL_visibility` (for “hide all” visibility)

`FSBeeline`	Visualisation of beeline between map center and current GNSS position;
	Statusline fields: beeline distance to center and zoom level
		`FSPosition.GpsOn`
		(determine current GPS position)
		`FSPosition.ZoomLevel`
		`MapViewPosition`
		(determine current map center position)

`FSControl`	Provides controls for all external triggered actions and activities; quick control menu handling incl. help	
`FSGraphDetails`	Developer Function: Show details of a GTileGraph; highlight close way	
`FSMarker`	Show MarkerTrackLog incl dashboard, provide controls for it	`FSATL.hideAll` (to hide MTL)
`FSPosition`	Visualize current position; center map on current position (depends on settings)	`MGMapApplication.Restart` (to set Center position to default “on”)
`FSRemainings`	Visualisation for statusline field remainings with remainings value from STL	`FSPosition.GpsOn` (determine current GPS position)
`FSRouting`	Calculation and visualisation (incl dashboard) of basic route depending on MarkerTrackLog	`FSMarker` (to inject FSMarker.LineRefProvider for control of route) `FSGrad.wayDetails` and

		FSBeeline.ZoomLevel (to switch visibility of relaxed nodes and Approaches) FSMarker.EditMarkerTrack (toggle direct routing only if switched on) FSPosition.GpsOn (precondition for Routing hints; base for RouteTrackLog remainings statistic in dashboard) FSMarker.autoMarkerSwitcher FSMarker.autoMarkerSettings (snap2way and alphaRoTL depend on both) FSMarker.alphaMTL (visualize MTL points as part of route, if low visibility from MTL) FSMarker.MTL_visibility (RoutingHints and MapMatching ability depends on it)
FSRecordingTrackLog	Manage and visualize (incl dashboard) the RecordingTrackLog	FSPosition.GpsOn (will be set on start/sop track record)
FSSearch	Provide Geocode search function and also reverse search	
FSTime	Provide statusline field: Time	

3.3 SettingsActivity

The settings activity provides a couple of preference screens, which are rather independent. The SettingsActivity allows to specify the desired preference screen class via intent. Currently this is used for following preference screens:

- MainPreferenceScreen
- DownloadPreferenceScreen

Changes on settings take effect via shared preferences. Typical usages are ListPreferences and SwitchPreferences, but it can also be done via OnClickListener. Depending on the particular setting the value of the setting can be obtained directly or there might be a listener registered for this particular preference. The app provides for this purpose an add on to the OnSharedPreferenceChangeListener (for details see chapter 4.3)

Furthermore there are multiple settings that are linked with an OnClickListener to a browse intent, opening a defined URL as the result of a click. This is used e.g. for all the download topics as well as for the documentation on github.io.

3.4 TrackStatisticActivity

The track statistic activity is the beside the MGMapViewer activity the most complex one. It provides a table with TrackStatisticEntry instances, where each known TrackLog is represented by one entry.

The TrackStatisticActivity contains a set of quick controls with the main difference, that these quick controls are rather trigger direct actions (in opposite to the menu quick controls of MGMapActivity). The creation and assignment of functionality works similar to the main activity.

Since the number of TrackLog objects might be high, a RecyclerView is used to visualize them. This prevents a significant delay during startup, even in case of several hundred TrackLog objects.

The other functionality is related to the quick controls and their enable-states. The quick controls allow a generic approach to view, save, delete, share and open them as marker track.

3.5 ThemeSettings activity

The ThemeSettings activity works almost exactly in the way as the mapsforge sample code suggest its usage. There is just a preference added to simply detect relevant changes in the settings. This is necessary to be able to recreate the MapView, which means to recreate MGMapActivity.

3.6 HeightProfileActivity

The HeightProfile activity is reusing a code package com.jjoe64.graphview that support visualisation of any kind of chart graph for Android. Here it is used for the height profile and (if switched on) for the ascent profile. These profiles can be shown for the SelectedTrackLog, the RecordingTrackLog and the RouteTrackLog.

For the ascent profile there is some smoothing function implemented. Nevertheless it often looks like a recording from a seismograph, if an earthquake occurs. So right now the usability is limited.

3.7 FileManagerActivity

The FileManagerActivity is created as a rection on the Google policy to restrict more and more the access to app directories. This started with Android 10 and now (with Android 14) even the development tools do not provide necessary access anymore.

Remarkable aspekt: Even in the “adb shell” the “`ls /sdcard/Android/data`” doesn’t show any app data. But a “cd <packageName>” (so e.g. “cd de.sof4mg.mgmap.rel” or cd “mg.mgmap”) is successful.

To enable an easy way to transfer files (maps, configuration, tracks, …) to and from the app directory, this FileManagerActivity is created. It’s using the “appDir” of the PersistencyManager as the root. This corresponds to the total path “`/sdcard/Android/data/<packageName>/files/MGMapViewer`”.

FileManagerActivity allows to navigate below this entry point. Again a set of quick controls enables multiple actions (browse, rename, share, receive via share, delete) on files. The browse action depends on a preference, whether to prefer the internal tiny editor or an external editor app for small text files.

The receive via share allows first to select the target directory and then to use the “save” to finally get the files in the desired place.

3.8 TrackLoggerService

The TrackLoggerService is the backbone of the recording capabilities. It has to run independent on the activities and their visibility. Once the recording is switched on, then this service shall not be interrupted for more then a few seconds. Android should not push it into background, executing only in long intervals and it should not terminate it for whatever reason!

There are some security and power consumption aspects related to this service – the following table tries to summarize them:

Aspect	From API-Level	Description
permission required: *ACCESS_FINE_LOCATION*	1	For permission is required to register a LocationListener on Androids LocationManager.
Notification required	26	The service has to provide a notification that it is running a foreground service (to make this very clear to the user)
Start Intent with `startForegroundService`	26	Android tries to distinguish between services which can run from time to time and "foreground" services which can run more often, but should not use too much time.
permission required: *FOREGROUND_SERVICE*	28	Additionally the app should claim to run foreground services already in the Manifest.
permission required: *ACCESS_BACKGROUND_LOCATION*	29	This new permission is required to distinguish for users between the allowing location access in foreground and in background.
permission required: *REQUEST_IGNORE_BATTERY_OPTIMIZATIONS*	(33)	If this service is used, then the app tries to triggered the intent action Settings.***ACTION_REQUEST_IGNORE_BATTERY_OPTIMIZATIONS*** to prevent battery optimization for this app.
permission required: *FOREGROUND_SERVICE_LOCATION*	34	The startForeground has to pass a type of service: ServiceInfo.***FOREGROUND_SERVICE_TYPE_LOCATION*** This requires to declare the permission upfront.

Beside the pure location information the TrackLoggerServices tries to enrich this information with height data. This is done with .hgt height data files, if available. But it is also done with pressure information, if the device supports a pressure sensor. From that pressure information it is possible to derive a much more precise delta height information than form .hgt data or even from the GNSS data.

There is a third task running in the scope of this service: If there is a RouteTrackLog available and if the setting "turning instructions" is switched on, then the output of those turning instructions is (via TextToSpeach) directly triggered with each new location.

3.9 BgJobService

The BgJobService provides a generic interface to run jobs in background. Initially it was introduced to manage the download of tiles to an offline tile store. Then this service extended, so the current list of usage scenarios is:

- Download of mapsforge maps
- Download of mapsforge theme
- Download jobs of TileStoreLoader
- Drop jobs for TileStoreLoader
- Download job for software update

There is a ThreadPool of up to 8 Threads executing these jobs.

4 Common Models and Techniques

This chapter gives a more detailed view on some important topics.

4.1 TrackLog Model

The following class diagram shows the relationship of the elements of mg.mgmap.model package:

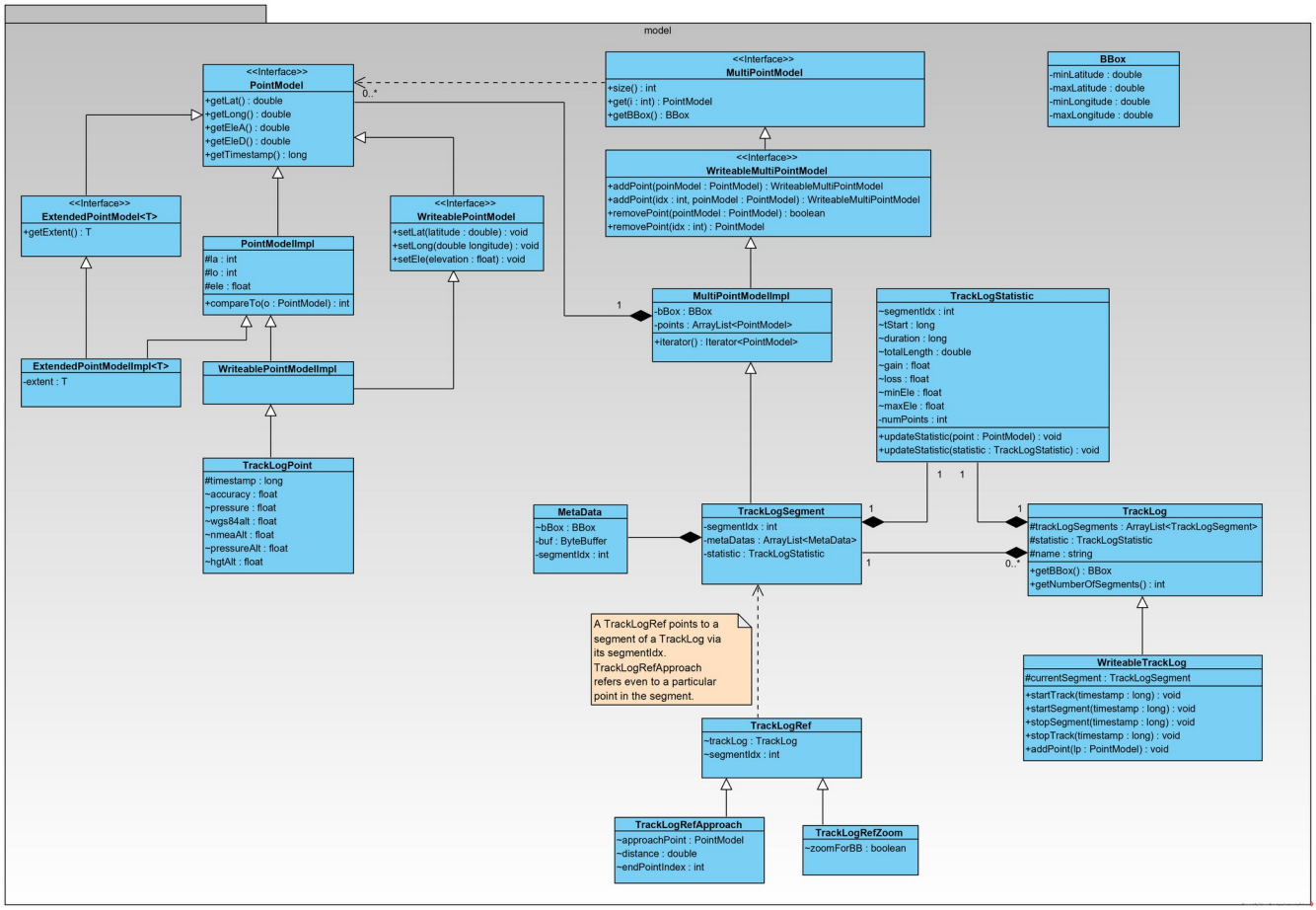

Figure 3: TrackLog model

It’s worth to mention that the core implementation of a point stores the latitude an d the longitude value as an integer value. This is done in microdegree, which provides a granularity of roughly 10cm. The core advantage is that processing inaccuracy is not really a problem, comparison is much easier.

This type of model is not only used for the internal storage of an track, but also for all view models. This is significant difference to the mapsforge examples, which use LatLong[] arrays where each LatLong instance keeps double values for latitude and longitude. So finally these model elements are passed directly to the views, there is no need to pass extra model objects to the views.

4.2 Graph

As described in the basic routing feature there is the option to find the shortest path between given points on the map. To do this there is the need to generate a graph from the given model. Obviously this is only possible for a vector map like mapsforge and not for pure graphical maps given via online or offline tile stores.

The PointModel and PointModelImpl classes (as seen in previous chapter) are not only the base for the TrackLog instances, but also for the graph modelling. The current realisation is based on tiles at zoom level 15. In terms of mapsforge map definition: All “ways” with the tag “highway” are taken into account. All of them are taken to setup the graph.

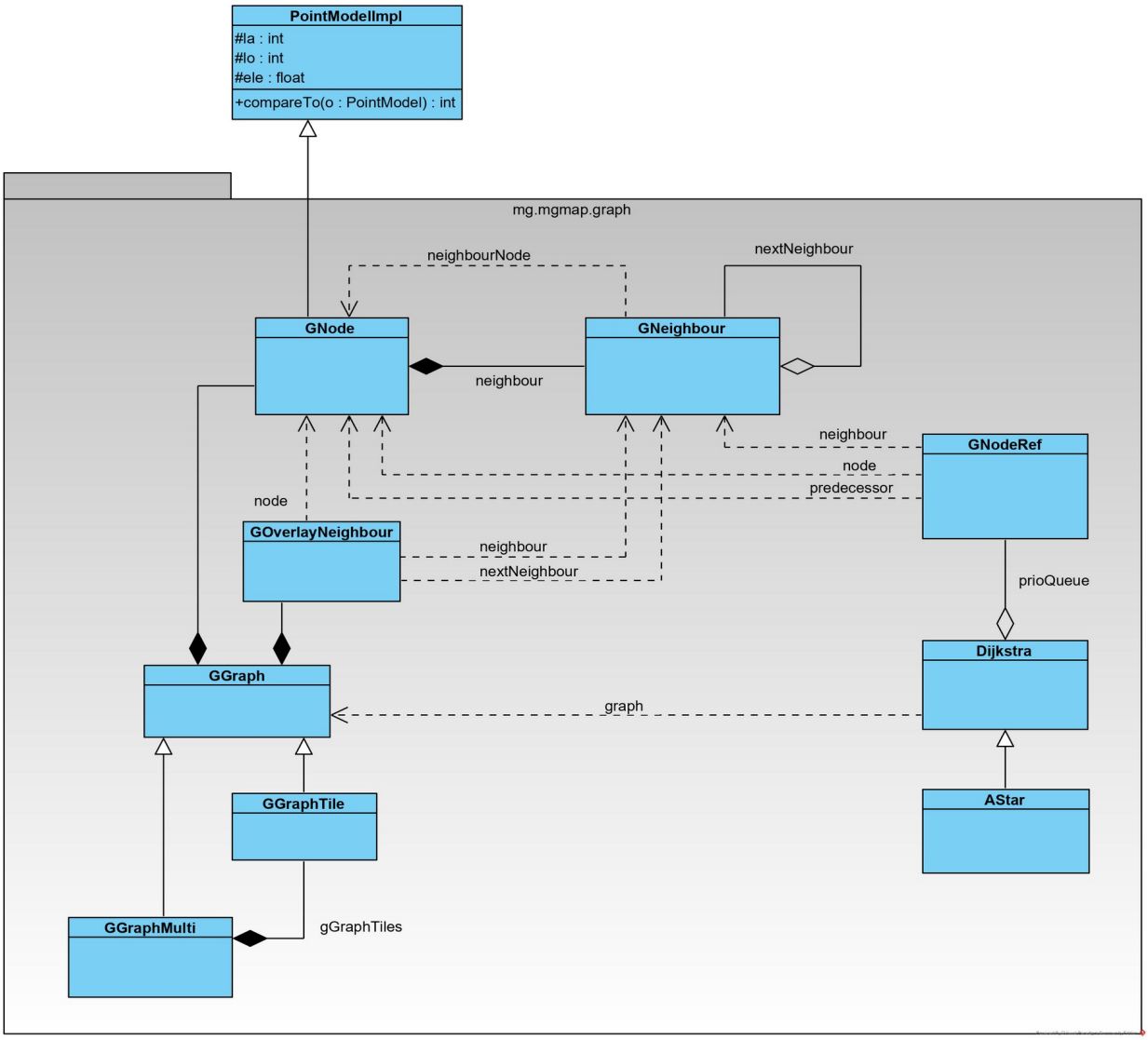

Figure 4: graph model

Since the ways stored in the map are not clipped at the tile border, they are overlapping at neighbour tiles. In fact the overlapping part has often even no common point, which makes it hard to setup a graph. So first step is to clip all ways at the tile borders. But even inside of tiles, there are sometimes “strange” data. E.g. if you look at the map, you see a crossing, where one path is ending on another one. If you zoom in to highest level, you might notice some inaccuracies. From a point of data, there are points with only a few microdegree distance. Without extra handling the graph considers two point as not connected, if they have only 1 microdegree distance (and if there is no defined way for this). In real life it’s hart to imagine that there are points with a distance of about 10cm and you cannot pass from one to the other. So in a first step there is a postprocessing of tile data to connect such kind of points.

Since route calculations often need more than one tile there is a procedure to setup a graph from multiple tiles, a GGraphMulti instance. The main effort is to connect multiple tiles at their borders. Again here we need a algorithm, which works even if the ways at the tile borders doesn’t fit exactly together.

Since the required tiles for a GGraphMulti often differ from one request to the next, the GGraphMulti is recalculated for each route. On the other hand some of the tiles are needed usually for multiple route calculations. Therefore the GGraphTile contains a LinkedHashMap that acts as a cache for the GGraphTile instances.

Since all results form the route calculation process based on GNode (so in fact on PointModel), they can easily passed to the view objects.

4.3 Preference Handling with PrefCache and Pref<>

Despite modelling of track information the app need to keep some information about current settings and states. Android provides to the option to store such information in shared preferences. This option is also used from the mapsforge sample software for its settings, but also to remember the theme settings. To obtain changes on the shared preferences the is an OnSharedPreferenceChangeListener. So this approach has two limitations:

- there is no option to register a change listener on a specific preference key
- there is no option to trigger a registered listener without changing the value.

E.g. for the fullscreen mode it is necessary after a search action to verify, whether the app is still in the mode which is currently stored in the preference. But with the normal preference handling I can trigger the observer only via a change of the value. Of cause I can find workarounds for this behaviour, but it would be nice to have a direct option for this.

Therefore the util package contains two classes: Pref<T> and PrefCache. PrefCache works as the name suggests as a cache for preferences. The scope of a PrefCache instance is a context, so each activity is using its own PrefCache. Once the activity is destroyed, the PrefCache will be garbage collected, no memory leaks will remain. The PrefCache will register an OnSharedPreferenceChangeListener and once the onSharedPreferenceChanged is call, this call will be passed to the corresponding Pref<>, if there is an instances with this preference key.

The class Pref<T> is a generic class that accepts Boolean, Integer, Float, String and Long for T. This limitation is related to the corresponding methods of shared preference. Pref<T> extends Observable, so it's easy to register any Observer on it. Furthermore it implements OnClickListener and OnLongClickListener with acts as default implementation with type Boolean. For those preferences both click listener toggle the state.

Instances of Pref<T> can be used for internal states (like e.g. fullscreen mode), which is relevant for multiple activities and which shall be unchanged even if the activity was destroyed for any reason. But this class can also be used for "local" preferences in a limited scope. If they are instantiated without explicit key, then they get a random UUID value as key and the are neither stored in the PrefCache nor in the SharedPreferences at all. They can be used e.g. inside a FeatureServices as long as it is running.

4.4 ExtendedTextView

The ExtendedTextView class is used frequently inside the app. It provides the ability to show an icon and or a text.

If an instance of ExtendedTextView is used to show text, then a Formatter will be passed that is able to format a given value in a proper String. So the typically a feature service has to set a new text as the result of some operation. With the Formatter the business logic of the feature service can simply use the setValue method without the need to care about representation.

Beside the visualisation of text, the ExtendedTextView allows to show an icon. This can be an additional icon (e.g. as with the length value in the dashboard), but it can also be an icon for its own (e.g. the quick controls). If an icon is shown, then this might be fix for this view, but it can also depend on up to two Pref<Boolean> instances, so it can show up to four states with four different icons.

And there is even one more thing: At least some of the views shall act as button, so they can configure one or two Pref<Boolean> action preferences. In this case the first on is toggled with a OnClickListener and the second one with an OnLongClickListener.

There are case, where one preference (e.g. `prefSearchOn`) is used as action preference and as state preference in an ExtendedTextView. So it changes its state due to the click and the view icon will be changed due to this. The business logic of the FSSearch is also trigger as an observer on this preference.

On the other hand there are more complex cases like starting a track recording. The quick control to start track recording needs an action preference to trigger the business logic. Due to this processing the GPS will be switched on and as the result of all the precessing the state of another preference will change, and this is related to the new icon for this view.