Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

HikeDroid – GPS Navigation for Hikers on Android Phones

Bachelor's Thesis

Damian Pfammatter
pdamian@student.ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Tobias Langner, Samuel Welten
Prof. Dr. Roger Wattenhofer

June 6, 2011

Acknowledgements

I am heartily thankful to my supervisors, *Tobias Langner* and *Samuel Welten*, whose encouragement, guidance and support enabled me to develop *HikeDroid* and this final report, presenting work and results.

I further offer my regards and blessings to all of those who supported me in any respect during the completion of this project.

Damian Pfammatter

Abstract

HikeDroid is an application for mobile devices using Google's *Android* platform. The application aims to support hikers with useful information about the surrounding area, terrain and hiking trails. To do so, the application marks the user's current position on the map and tracks the passed course. For this purpose, *HikeDroid* accesses the device's internal GPS receiver. In addition, *HikeDroid* tries to provide the user with a natural way to interact with the map. For that reason, *HikeDroid* supports two different touch gestures, *drag* and *pinch*. The drag gesture is used to move over the map. The pinch gesture provides an easy way to zoom in and out, in order to show more or less map details, respectively. The architecture of the application is developed with the requirement to be modular, flexible, easily understandable and extensible for future works.

This paper gives an overview about the general concepts, techniques and architectural decision used in *HikeDroid*. This paper is not intended to list all the implementation details used to realize the application.

Chapter 1 provides a short overview about Google's *Android* platform. In Chapter 2, *HikeDroid* is compared to similar existing applications. This chapter lists advantages and motivations for the use of HikeDroid. Chapter 3 states what goals the application tries to achieve. Chapter 4 lists some general problems and requirements that need to be considered during the development of a map application like *HikeDroid*. Chapter 5 presents some abstract models and theoretical backgrounds that are used in the application. Chapter 6 is more specific. It provides a more detailed look at the ideas of Chapter 5 and how they are in fact realized in *HikeDroid*. Chapter 7 lists some proposals, ideas and extensions for future work on the application. Chapter 8 concludes this paper.

Contents

Acknowledgements **i**

Abstract **ii**

1 Android **1**

2 Motivation and Related Work **2**

2.1 Motivation 2

2.2 Google Maps for Mobile 2

2.2.1 Overview 2

2.2.2 Comparison with HikeDroid 3

2.3 Geo Admin Web Interface 3

2.3.1 Overview 3

2.3.2 Comparison with HikeDroid 4

2.4 Swiss Map Mobile 4

2.4.1 Overview 4

2.4.2 Comparison with HikeDroid 5

3 Goals **7**

4 Problems **8**

4.1 Architecture 8

4.2 Map 8

4.2.1 Map Data 8

4.2.2 User Interaction 8

4.2.3 Tiles 9

4.2.4 Coordinate Systems 9

4.2.5 GPS Positioning 9

4.2.6 Layering 9

4.3 Cache 10
4.3.1 Interface 10
4.3.2 Trade-Off 10
4.3.3 Replacement Strategy 10
4.4 Asynchronism 10

5 Theoretical Background 11
5.1 Cache 11
5.1.1 Motivation 12
5.1.2 Interface 12
5.1.3 Hierarchy 12
5.2 Request Manager 13
5.2.1 Asynchrony 14
5.2.2 Field of Application 14
5.3 Map Adapter 15

6 Realization 18
6.1 Map 18
6.1.1 Surface View 18
6.1.2 Event Handling 18
6.1.3 Layers 19
6.1.4 Coordinate Systems 20
6.1.5 Drawer 21
6.2 Tile Map 22
6.3 Cache 22
6.3.1 Bitmap Cache 23
6.3.2 Byte Array Cache 23
6.3.3 SD Cache 23
6.3.4 Web Cache 24

7 Future Work 25
7.1 Suggestions for Improvement 25
7.1.1 Cache 25

7.1.2 Map Architecture 25
7.1.3 Offline Maps 26
7.1.4 Energy Saving Mode 26
7.1.5 Memory Leakage 26
7.2 Extensions 27
7.2.1 Data Sets 27
7.2.2 Hiking Information 27
7.2.3 Features 27

8 Conclusion 28

A UML Diagrams 1

B Reference Documents 4
B.1 Swiss Map Mobile, 2009 4
B.2 Swiss Map Mobile Sectors, 2009 4
B.3 Coordinate System Transformations, 2008 4

Chapter 1

Android

This chapter gives a short overview and description of the Android platform. Further information can be found on the Android website *http://www.android.com.*

Android is a software stack for mobile devices like smart phones, mobile phones, netbooks and tablets. It includes an operating system, middleware and key applications. Android[1] is based on the Linux kernel. It is developed and released by *Google Inc.*[2] and other members of the *Open Handset Alliance (OHA)*[3]. The Open Handset Alliance was found in November 2007 with the intent to improve open standards for mobile devices. It is a consortium of 80 hardware, software and telecom companies. Currently there are more than 200'000 applications available from Google's online application store *Android Market*[4]. A large community of Android developers exists. Android applications are primarily written in the *Java*[5] language using Java libraries developed by Google.

[1]Android Website, 2011. Android.com
Available at: http://www.android.com [Accessed June 6, 2011]

[2]Google Website, 2011. Google Inc.
Available at: http://www.google.com [Accessed June 6, 2011]

[3]Open Handset Alliance Website, 2011. OHA
Available at: http://www.openhandsetalliance.com [Accessed June 6, 2011]

[4]Android Market Website, 2011. Apps – Android Market
Available at: https://market.android.com [Accessed June 6, 2011]

[5]Java Website, 2011. Java.com
Available at: http://www.java.com [Accessed June 6, 2011]

Chapter 2

Motivation and Related Work

This chapter lists three typical examples of related works. For each example a short overview of its functionality is provided. Subsequently each example is compared to *HikeDroid*. This chapter focuses on what *HikeDroid* tries to achieve and what not, compared to other solutions.

2.1 Motivation

The vast majority of hikers have already been in this situation: One is out in the wild at a junction of hiking trails but does not know for sure at which particular point on the map one is presently located and further, which trail to take to get to the desired destination. The rise of omnipresent GPS devices in the form of modern smart phones opens completely new perspectives on this problem. How nice would it be whenever you take out your smart phone, it displays your current location and orientation on a high resolution hiking map straightaway? The goal of *HikeDroid* is to develop an application fulfilling this aim by accessing the *VECTOR25* hiking maps that have been released by the Swiss government at 1 July 2008.

2.2 Google Maps for Mobile

2.2.1 Overview

Google Maps for Mobile[1] is probably the best known and widest used map application for mobile devices. Figure 2.1a shows one of the latest features available in Google Maps for Mobile, namely 3 dimensional maps. Google Maps for Mobile is a free mapping application developed and distributed by Google through the Internet. It provides most of the web-based site's features and services. Google

[1]Google Website, 2011. Google Maps for Mobile
Available at: http://www.google.com/mobile/maps [Accessed June 6, 2011]

Maps for Mobile supports multiple map views like satellite, traffic, street or terrain. Google Maps for Mobile is not just a map application. It offers further functionalities like navigation assistance, journey planner and business locator.

2.2.2 Comparison with HikeDroid

Functionality As stated above, Google Maps for Mobile provides a wide amount of functionality. However, this has the drawback, that the concepts are often too general for a specific field of application. This is where *HikeDroid* tries to improve. *HikeDroid* adapts some functionality of Google Maps for Mobile and makes them more usable for hikers. The two major improvements are listed below.

Map Data *HikeDroid* is able to access high resolution Swiss maps which are full of useful details. They for example contain information about *streets*, *rivers*, *lakes*, *contour lines*, *towns*, *villages*, *buildings*, *waters* and *mountains*. Furthermore, *HikeDroid* is able to access Swiss hiking trail information. An overlay containing all the hiking trails can be shown over the general map data. By enabling the GPS locating service, hikers can see which trail they currently follow. Further information about hiking trails and other usable hiking information could be added by future works on *HikeDroid* (Section 7.2.2).

Offline Maps Google Maps for Mobile requires a permanent Internet connection to work, which is obviously not always possible while hiking. Furthermore, loading map data over a mobile Internet connection is often slow and expensive. However, *HikeDroid* allows users to load maps beforehand by using *Wi-Fi*. Users could therefore define sectors containing prospective routes and save for later use. *HikeDroid* does not yet implement full support for this idea. The idea should to be improved as proposed in Section 7.1.3. Google Maps for Mobile attempts to support offline maps as well in latest releases.

2.3 Geo Admin Web Interface

2.3.1 Overview

The *Swiss Federal Office for Topography (swisstopo)* provides a web interface[2] for freely accessing Swiss map data. The web interface is primarily intended to be integrated in a website. Different data sets are available for the following fields:

[2]Geo Admin Website, 2011. Öffentlicher Zugang zu Geoinformationen und Geodaten Available at: http://map.geo.admin.ch [Accessed June 6, 2011]

- basic data
- ground levels
- area and human population
- infrastructure and communication
- environment, biology and geology
- energy and economy

Recently a mobile version (BETA)[3] of the web interface got released. Figure 2.1b shows a screenshot.

2.3.2 Comparison with HikeDroid

Map Data *HikeDroid* basically accesses the same data as the web interface does. At the moment *HikeDroid* only supports the basic map data and information about hiking trails. As stated in Section 7.2.1, future work should make all data sets accessible in *HikeDroid*.

Zooming The mentioned web interface does not support seamless zoom. Furthermore it does not provide most of the desired functionality, for example to mark the device's current location. Altogether the interaction with the map is rather poor and not flexible enough to be applicable on mobile devices. *HikeDroid* therefore does not include or build on the provided web interface.

2.4 Swiss Map Mobile

2.4.1 Overview

The *Swiss Federal Office for Topography* provides an application for mobile devices to access its map data, called *Swiss Map Mobile*[4]. *Swiss Map Mobile* is a commercial application developed by the *Andreas Garzotto GmbH*[5]. Currently the application is available for *Windows Mobiles*, *Symbian S60 Mobiles* and

[3]Geo Admin Mobile, 2011. BETA mobile version
Available at: http://mobile.map.geo.admin.ch [Accessed June 6, 2011]

[4]Swiss Map Mobile Website, 2011. Swisstopo: Swiss Map Mobile
Available at: http://www.swisstopo.admin.ch/internet/swisstopo/de/home/products/maps/mobile.html [Accessed June 6, 2011]

[5]Andreas Garzotto GmbH Website, 2011. Die Hebammenpraxis für Projekte und Produktentwicklung
Available at: http://web.me.com/garzotto/ [Accessed June 6, 2011]

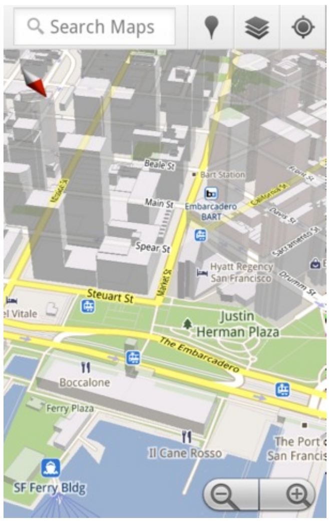
iPhones / iPads. A screenshot of the *iPhones* version is shown in Figure 2.1c. At the moment, there is no version available for *Android* mobiles. The *Swiss Map Mobile* application supports three different scales, *1:25'000*, *1:100'000* and *1:500'000*. The scale is manually chosen by the user. The application supports seamless zoom between each scale level. Further, the application is able to mark the current GPS position and to record way points as traces. A more detailed comparison of functionalities and platforms can be found in Appendix B.1.

2.4.2 Comparison with HikeDroid

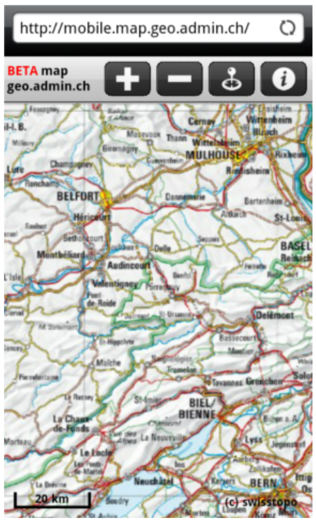
Cost As stated above, the *Swiss Map Mobile* application does not come for free. In order to download the application, users currently have to pay an initial cost of CHF 4.40[6]. However, this package does not yet contain any map material, but a limited amount of map data for testing. Further map data comes with additional costs. The Swiss map is partitioned into eight different sectors (Appendix B.2) each of which can be obtained for CHF 89.00. While *HikeDroid* tries to follow the idea of making all the map data freely available for users, especially for hikers, as it is already publicly available on the Internet.

Zooming *HikeDroid* and *Swiss Map Mobile* both provide seamless zooming functionality. But in comparison, *HikeDroid* switches automatically between different scale levels in order to provide the best resolution for the displayed map sector. *HikeDroid* supports scales of *1:25'000*, *1:50'000*, *1:100'000*, *1:200'000*, *1:500'000*, *1:1'000'000*, *1:2'500'000* and *1:5'000'000*.

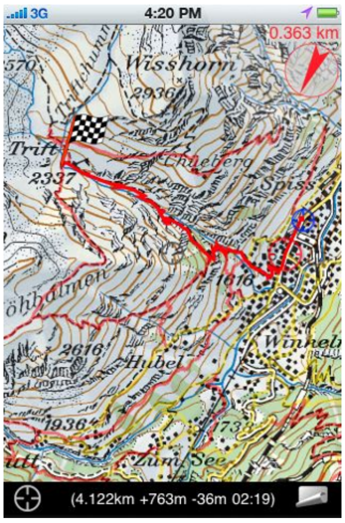
[6]Apple iTunes Website, 2011. Swiss Map Mobile
Available at: http://itunes.apple.com/ch/app/swiss-map-mobile/id311447284 [Accessed June 6, 2011]



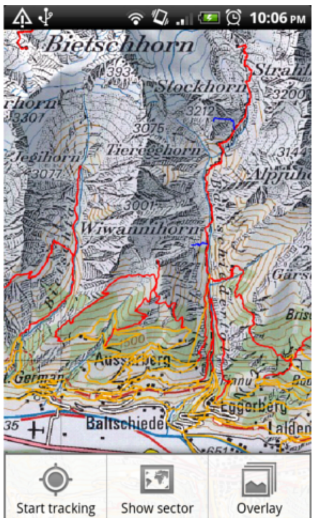
(a) Google Maps for Mobile



(b) GeoAdmin mobile version BETA



(c) Swiss Map Mobile



(d) HikeDroid

Figure 2.1: Screenshot Comparison of different Map Applications

Chapter 3
Goals

This chapter summarizes what the map application *HikeDroid* tries to achieve and what the *goals* are.

The goal is to develop an Android application that retrieves the *VECTOR25* hiking map data from the website *http://geo.admin.ch*. The map data has to be displayed correctly on the device's screen. Users should be allowed to interact with the map in a user friendly way. The application is primarily developed for *hikers*. The purpose for the application is to replace the need of having a conventional map of trails by hand while hiking. The potential of mobile devices should be exploited, for example by the included *GPS receiver*. The current location of the user is appropriately displayed on the map. The user can thereof conclude its current location and which trail to follow to get to the desired destination. Also, it is desirable that additional information is displayed on the map to further assist hikers. During the trip, the application should not require the establishment of a connection to the Internet. In other words, the application should allow the usage of *offline maps*. To further make the application usable for hikers, the application has to be *energy efficient*. Yet another important goal is the *extendibility* of the underlying framework. It is of particular importance to allow the implementation of advanced features later on.

Chapter 4

Problems

This chapter gives a short description of the initial position and enumerates the main problems that occurred when developing the map application.

4.1 Architecture

The underlying architecture should be flexible, easy to extend and open for future adjustments. To achieve this, the architecture used in *HikeDroid* consists of three main components. The first component models maps, its representation and its user interactions (Section 6.1). The second component models a cache architecture to allow fast map data accesses (Section 5.1, Section 6.3). The third component is the map adapter (Section 5.3) which connects the first two components.

4.2 Map

This section lists the problems that occur in conjunction with the map.

4.2.1 Map Data

The first problem that came up was the question of where and how to access the required map data. Also, to minimize the access costs was essential. It was not exactly clear what types of access costs exist and which ones the most relevant are. Considering mobile devices, time, memory and energy are crucial in deciding the costs.

4.2.2 User Interaction

The second problem that needs to be solved, is the fact that users should be capable to interact with the map. It should be possible to move over the map.

It should also be possible to zoom in or out in order to show more or less details, respectively.

4.2.3 Tiles

The retrievable *VECTOR25* map data represents a map as a collection of tiles. However, the first problem that occurred, is to correctly handle this tiles, which includes how to access and store them, as well as how to position them correctly on the screen. To make the tiles accessible without long delays turns out to be vitally important through out the overall performance of the application. This is where the idea of a multi-level cache hierarchy came up.

4.2.4 Coordinate Systems

Mainly to solve the previously mentioned problem of intuitively displaying the map data, one needs to consider different coordinate systems. There is at least the need to support the *Global Positioning System (GPS)* and the standard *Cartesian* coordinate system used to control the device's screen. It is likely that further coordinate systems have to be taken into account. It is for example common that maps for Switzerland and Liechtenstein rely on a coordinate system called the *Swiss Grid*. The problem one has to solve is the correct transformation between this coordinate systems.

4.2.5 GPS Positioning

One of the requirements is that the current GPS location can be correctly marked on the map. This requires the application to interact with the *GPS receiver* on the device. It further requires that a given GPS position can be found on the map and marked accordingly. This problem is related with the previously mentioned problem to handle different coordinate systems (Section 4.2.4).

4.2.6 Layering

So far, a map object only consists of a single layer which contains a visual representation of the basic topographic data. This layer is referred to as the *base layer*. However, the base layer does not always contain all the required information. For example, the base layer contains no information about hiking trails, which is an essential data set for hikers. To solve this problem in general, a map object has to support *overlays*. Informally, an overlay is a layer that is located on top of another layer. The handling of multiple overlays leads to further problems one has to take into account. First, the map needs to keep track of more data sets. Therefore, the process of storing, accessing and drawing the data gets more complex. Accordingly, an appropriate data structure has to be found. Second, users should be allowed to enable or disable a subset of layers. A user interface needs to be created for that purpose. A third problem that comes up, is the fact that not all layers are of exactly the same type. One may think of a special type of layer, which the application itself creates, for users to draw objects on.

4.3 Cache

This section lists the problems that occur in conjunction with the cache.

4.3.1 Interface

Different caches need to agree on a uniquely known way to access a cache. In that case, caches can easily talk to each other, which then results in building a cache hierarchy.

4.3.2 Trade-Off

Caches are a way to speed up the access of data. However, there is a trade-off between bandwidth and capacity. To bypass this problem, there are usually different caches arranged in a hierarchical structure. Typically caches in the top of the hierarchy are smaller, but serve accesses faster, where lower caches are larger and slower. Therefore one needs to find an adequate cache hierarchy to minimize the access cost.

4.3.3 Replacement Strategy

Because of the limited capacity, caches sometimes have to discard items. Therefore they need to follow a certain strategy defining which items have to be discarded, which was rather obscure at early stage.

4.4 Asynchronism

Is is essential for the application to handle tasks in an asynchronous way. In other words, some tasks should be allowed to get processed in the background without interrupting incoming calls. For example it is crucial for the *UI thread* to be small and simple in order to stay permanently responsive for user events.

Chapter 5

Theoretical Background

This chapter lists theoretical concepts and ideas that are used to realize the map application *HikeDroid*. However, all concepts and ideas listed here are independent of *HikeDroid* and not restricted to map applications.

Section 5.1 explains the concept of a *cache* and justifies the usage of caches in a map application. The idea of a *request manager* and it's field of application is introduced in Section 5.2. Section 5.3 introduces the concept of a *map adapter* that serves as a connecting piece between cache and map.

5.1 Cache

This section explains the idea and motivation behind the usage of a cache in map applications.

A cache is a component that stores a certain amount of data, usually some subsets of a lager amount of data, so that future requests for that data can be served faster. The case where the requested data is contained in the cache is called a *cache hit*, while cases don't contain requested data is called a cache miss. In the case of a cache hit, data can simply be read out, which is comparatively fast. In the case of a cache miss, data has to be fetched from its original storage location which is comparatively slow. In order to reduce costs and provide efficient ways to access data, caches are usually relatively small.

Definition 5.1 (Cache, Capacity, Bandwidth) *Let $I = \{i_1, i_2, ..., i_n\}$ be the set of all available data items. A* ***cache*** *C is a data structure that stores a set of at most m data items $D \subseteq I$, where m is called the* ***capacity*** *of C. The* ***bandwidth*** *$b(C)$ is defined as the number of cache accesses per time unit, that can be served by C.* ◇

5.1.1 Motivation

One of the most fundamental requirements for a map application is the fact that map data needs to be displayed as fast as possible, without long delays and response times. This requirement motivates the idea of using a cache to access the map data. To be cost efficient, the cache may consist of multiple cache levels building up a cache hierarchy. In response to a further requirement, the application should always remain responsive to user events. For the cache architecture this implies, that cache accesses should be non-blocking, meaning that the *UI Thread* does not have to wait for results, especially not in the case of a cache miss.

5.1.2 Interface

Each cache implements the following functionality:

- *contain:* check if a certain item is contained in the cache
- *get:* get a certain item from the cache
- *put:* put a certain item in the cache
- *remove:* remove a certain item from the cache

Forcing each cache to implement this functionality establishes a commonly known interface for accessing caches. This brings the advantage that instances are able to access caches without explicit knowledge of the underlying cache instance. Furthermore, caches may directly interact with each other leading to the possibility to create a multi-level hierarchy of caches.

5.1.3 Hierarchy

In order to realize the idea of a multi-level cache hierarchy, the basic functionality of a cache does not have to change. The access rules provided by the cache interface are preserved. However, to achieve the hierarchical structure each cache additionally needs to maintain the following two references:

- *next:* next, lower level cache
- *prev:* previous, higher level cache

The idea is, to build the cache hierarchy analogously to a *Linked List* data structure. The main advantage is that caches can be added and removed in a very flexible way. Furthermore, the caches can be easily reordered in the hierarchy.

Definition 5.2 (Level i Cache, Top Level Cache) *The i-th cache in the hierarchy is referred to as the **level i cache**. By definition, there are i higher levels of caches in the hierarchy. The level 0 cache has no previous cache in the hierarchy and is therefore called the **top level cache** (Figure 5.1).* ◇

Why is it useful to build a hierarchy of caches? To answer this question, we first have to define what it means by a cache being faster or slower, respectively larger or smaller compared to another cache.

Definition 5.3 (Relatively Fast, Slow Cache (Definition 5.1)) *A cache C_i is called **faster** than a cache C_j, if $b(C_i) > b(C_j)$. A cache C_i is called **slower** than a cache C_j, if $b(C_i) < b(C_j)$.* ◇

Definition 5.4 (Relatively Large, Small Cache (Definition 5.1)) *A cache C_i is called **larger** than a cache C_j, if $m_i > m_j$. A cache C_i is called **smaller** than a cache C_j, if $m_i < m_j$.* ◇

A hierarchical cache structure is typically used to solve the following problem: Normally there is a trade-off between *bandwidth* and *capacity.* A fast cache is typically of small size. A large cache is typically slow. To bypass this trade-off, caches are arranged in a cache hierarchy. The level i cache is typically smaller, but faster than a level j cache, with $0 \leq i < j$. The level j cache is typically larger, but slower than the level i cache.

5.2 Request Manager

This section explains the concept of a *request manager* and its field of application.

The concept of a *request manager* is simple and always follows the same pattern. Each request manager has an associated buffer queue into which requests are written. The request manager takes requests from the buffer and handles them. When there are no more requests in the buffer, the request manager waits. At the arrival of a new request the manager gets notified and resumes activity. In general, a request manager handles several types of requests. Each request type defines a sequence of steps that the request manager has to execute in order to handle the request.

Definition 5.5 (Request Manager) *A **request manager** RM holds a buffer queue BQ. Typically, BQ is implemented as stack data structure, meaning that it follows the last in first out (LIFO) principle (Definition 5.6). The request manager is a component that repeatedly executes the following steps:*

1. *Wait until $BQ \neq \{\}$*

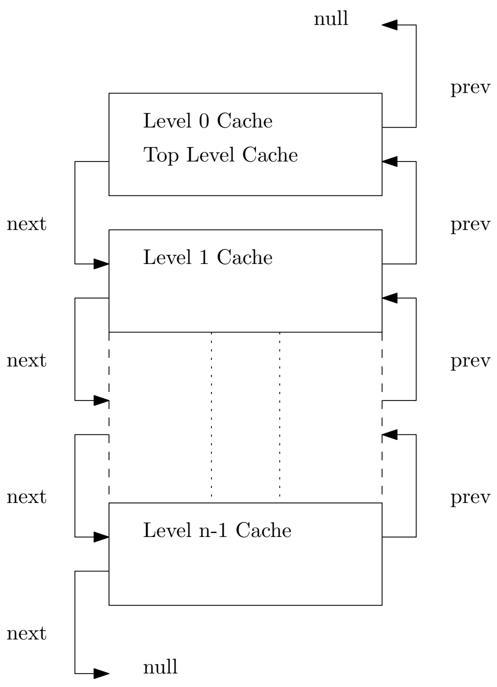


Figure 5.1: Hierarchy of n Caches

2. *Take request r from BQ using the LIFO strategy*
3. *Handle r*
4. *Go back to step 1* ◇

Definition 5.6 (Last In First Out (LIFO)) *The* ***Last In First Out (LIFO)*** *strategy refers to the way that items stored in a queue data structure are processed. By definition, the last data added to the structure must be the first data to be removed and processed.* ◇

5.2.1 Asynchrony

The concept of a request manager is mostly useful for the following reason: Request initiators do not have to wait until their requests are handled. This allows request initiators to proceed working while their requests are handled asynchronously.

5.2.2 Field of Application

The concept of request managers is designed for the application in caches (Example 1). However, it is important to notice that the concept itself is much more generally applicable. This means that other components, requiring a similar kind of functionality, are free to instantiate and use their own request managers to handle their own type of requests.

Example 1 (Request Manager) Figure 5.2 shows how the concept of request managers is used inside a cache. Each cache instance keeps a static number of request managers that handle incoming requests. Incoming requests are distributed equally along all managers. A cache initially chooses the number of available request managers by estimating its expected workload.

Remark 1 (Dynamic Request Manager Instantiation) The number of request managers used by a cache should be chosen dynamically, depending on the current workload. This is clearly a point which one may think to improve in future work (Section 7.1.1). □

The reason why it makes sense to use request managers to manage cache accesses, is to fulfill one of the requirements mentioned before. Namely the one that applications have to continuously stay responsive for user events. However, the question why the request managers are part of the caches still remains unanswered. In order to answer this question, we again consider the hierarchical cache setting of Figure 5.1. Mainly there are two advantages. First, caches at different levels in the hierarchy stay completely independent. The idea of allowing the reordering of caches in the hierarchy is preserved. Second, and even more important, the proposed setting allows caches themselves to issue new requests, most likely for other caches. For example, if an *item i* is missing at some *cache level l*, the cache at *level l* might issue a request to get *item i* from *cache level l+1*. The initiator cache does not have to wait for the request to be handled. It is possible that the request initiator gets notified when the request is handled. □

5.3 Map Adapter

This section briefly describes the concepts of a *map adapter*. Some further examples are provided to support the ideas and concepts.

A *map adapter* is a component that connects the map with the cache, as shown in Figure 5.3. The map adapter controls all interaction between these two components. It is possible, that the map adapter itself implements further functionality.

Example 2 (Map - Map Adapter - Cache) Consider the case where a map object wants to display a certain sector of the map. The map requests a visual representation of the sector from the map adapter. In order to handle this request, the map adapter requests some items from the cache. □

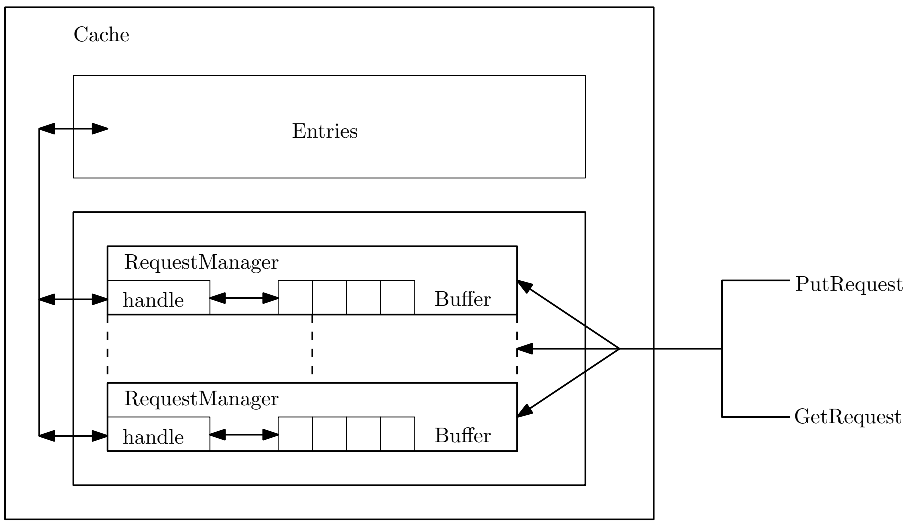


Figure 5.2: Cache with Request Managers

Example 3 (Cache - Map Adapter - Map) Consider the case, where a new item gets available in the cache. As a result, the cache notifies the map adapter. The map adapter may now want to update the map. □

Remark 2 (Further Functionality) In order to communicate with maps and caches, the map adapter implements further functionality or accesses other components (Example 4). □

Example 4 (Location Listener) The map adapter has access to another component in the system, the so called *location listener*. More precisely, the map adapter registers itself to the location listener in order to receive location updates. This means, that the location listener repeatedly provides the map adapter with the current location. As a result, the map adapter notifies the map. □

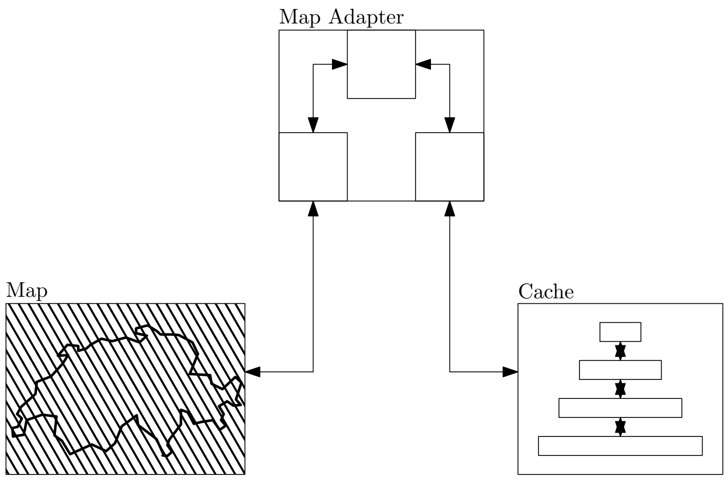


Figure 5.3: Map Adapter connecting Map and Cache

Chapter 6

Realization

In contrast to the theoretical Chapter 5, this chapter highlights more details of the specific realization of the map application *HikeDroid*. Section 6.1 shows how maps are represented and what their functionalities are. Section 6.3 points out how the previously introduced concept of a cache hierarchy (Section 5.1) is realized in *HikeDroid*. This section explains which types of caches are used at which level in the hierarchy and why this makes sense.

6.1 Map

A *map* is a rectangular area on the screen that shows some visual representation of objects, regions, themes and the relationships between them.

6.1.1 Surface View

A map is an interactive UI component extending the concept of a *Surface View*[1]. In principle, a *View*[2] handles the drawing of a rectangular area on the screen and responses to events. The *Surface* is a dedicated area embedded in the *View* into which the map is drawn. Making a map extend the *Surface View* allows users to include maps in the screen layout.

6.1.2 Event Handling

Maps implement a callback on the Surface (Section 6.1.1) in order to receive information about events. Maps primarily have to react to *Touch Screen Motion*

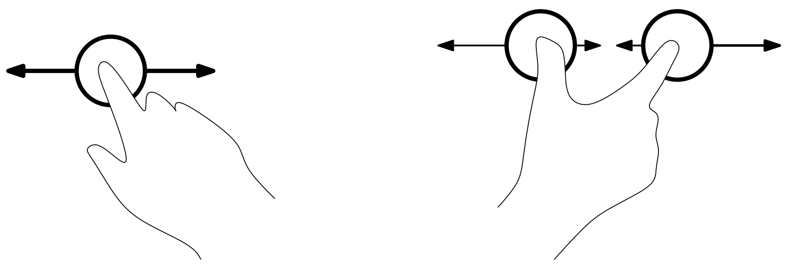
[1]Android Developers Website, 2011. SurfaceView
Available at: http://developer.android.com/reference/android/view/SurfaceView.html [Accessed June 6, 2011]

[2]Android Developers Website, 2011. View
Available at: http://developer.android.com/reference/android/view/View.html [Accessed June 6, 2011]

Events. There are two *Touch Screen Motion Events* that maps handle:

- *Drag Gestures* (Figure 6.1a)
- *Pinch Zoom Gestures* (Figure 6.1b)

The *Drag Gesture* is used to move over the map. The *Pinch Zoom Gesture* is used to zoom in and out. To be more explicit, the scale of the viewed area is changed in order to see more or less details.



(a) Drag Gesture (b) Pinch Zoom Gesture

Figure 6.1: Touch Screen Motion Events

6.1.3 Layers

A *layer* is a rectangular area containing specific geographic information (e.g. hiking trail information). A layer that lays on top of another layer is called an *overlay*.

A map in *HikeDroid* consists of a base layer and zero or more overlays (Figure 6.2). The base layer is special in the sense that users cannot disable it. It contains the basic cartographic data. Different to the base layer, users are allowed to enable and disable overlays. *HikeDroid* currently uses the following two overlays:

- *Hiking Trail Overlay*
- *GPS Data Overlay*

The *Hiking Trail Overlay* contains information about Swiss hiking trails. The *GPS Data Overlay* is a drawable layer. *HikeDroid* itself creates this layer by drawing recorded GPS points in it.

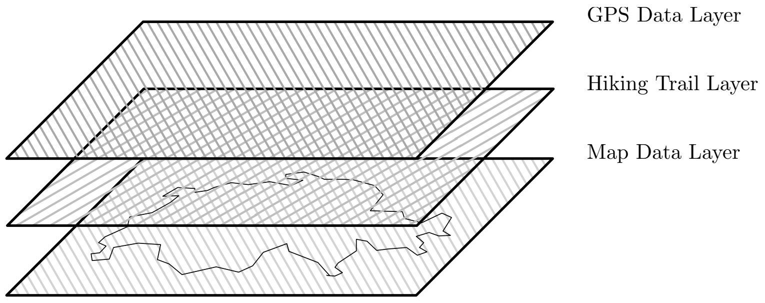


Figure 6.2: Multi Layering

6.1.4 Coordinate Systems

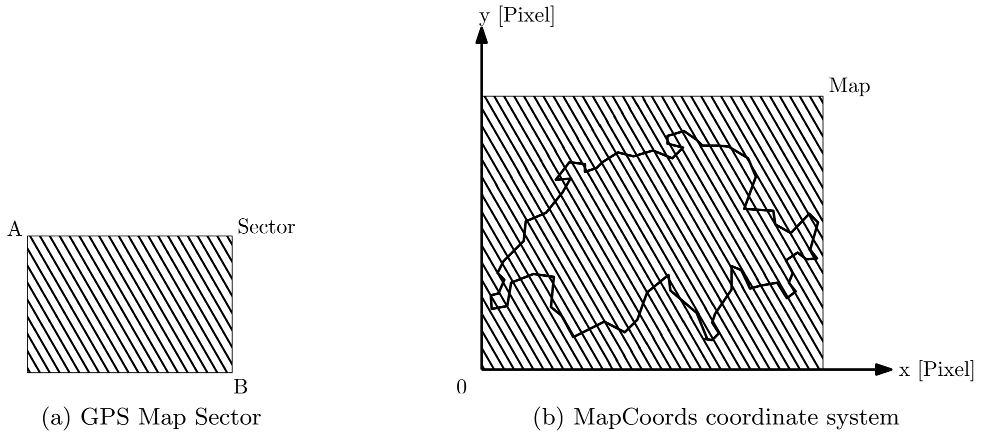
Global Positioning System (GPS) The *Global Positioning System (GPS)* is a global navigation satellite system that allows the location of positions around the world. Maps in *HikeDroid* provide an *Application Programming Interface (API)*, that takes GPS positions from API users and handles them correctly. More precisely, API users have to pass GPS positions to maps in order to use their functionality. To make this more understandable, consider the following two examples:

Example 5 (GPS Sector) A user can request a sector on the map by specifying two GPS positions A and B. A and B have to define a sector like the one in Figure 6.3a. The map internally handles the sector by transforming A and B to Swiss Grid and MapCoords positions. □

Example 6 (GPS Locating) The map API provides functionality to draw at a certain GPS position on the map. For example, this can be used to mark the current location of the device on the map. The device's GPS receiver can simply pass the current GPS position to the map. □

Swiss Grid (CH1903) A lot of maps for the countries Switzerland and Liechtenstein commonly use a coordinate system referred to as the *Swiss Grid*, or *CH1903* for short. Transformations between Swiss Grid and GPS coordinates are provided by the *Swiss Federal Office for Topography (swisstopo)* (Appendix B.3). The proposed transformations rely on an approximating solution that guarantees an exactness of at least 1 meter. The transformations are based on a well-defined fixed point, the *Department of Exact Science* at University Bern. It is located at position (600'000/200'000). *HikeDroid* uses the CH1903 coordinate system to map GPS coordinates to intern MapCoords.

MapCoords The units of length used to draw onto the device's screen, are pixels. Therefore it makes sense to introduce a reference coordinate system over maps that also uses pixels as the unit of length. This coordinate system is referred to as the *MapCoords*. MapCoords is a Cartesian coordinate system having its origin and orientation as shown in Figure 6.3b. MapCoords allows a simple mapping of a map sector to the device's screen.



(a) GPS Map Sector (b) MapCoords coordinate system

Figure 6.3: Map Sector and Coordinate System

6.1.5 Drawer

A map object has a special component that takes care of the drawing process. This component is called the map *drawer*. It is the map drawer's task to draw the requested part of the map to the device's screen. During its entire lifetime, the drawer simply executes the following two steps:

- Wait until getting notified
- Update, i.e. redraw the map

The drawer needs to be an autonomous component. More explicitly, the map drawer has to be implemented as a separate thread. Doing the drawing in a separate thread has several advantages. While drawing the map, the calling thread (e.g. the UI thread) remains responsive to events, for example to user events. Furthermore, this allows external entities to independently update the map. This is especially used by the map adapter (Section 5.3).

Example 7 (Non-blocking Drawings) Consider a case where the adapter requests some new sector of the map. It is very likely that the requested map data is not immediately accessible from the cache, meaning that not all the data is contained in the top level cache (Definition 5.2). The access time depends on the cache level in which the requested data is found. To satisfy the requirement of a non-blocking drawing process, some default loading representation will be drawn, while the data propagates upwards in the cache hierarchy. When the data is in the top level cache, the adapter updates the map to make the changes visible. □

6.2 Tile Map

A *tile map* is a special kind of map, which only differs from a general map by the fact that its rectangular area is partitioned into tiles. The tiles are arranged to form a two dimensional grid as shown in Figure 6.4. The number of tiles, out of which a certain map consists, varies between different zoom levels. This concept of representing a map is widely used. However, it is important to notice that other representations are possible as well. The general concept of a map can be extended appropriately.

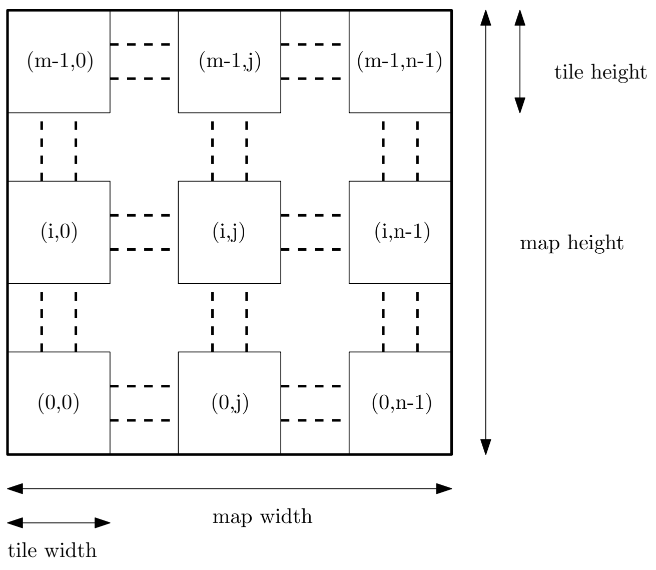


Figure 6.4: Tile Map

6.3 Cache

This section shows, how the theoretical concept of a cache hierarchy is actually realized. *HikeDroid* uses the cache hierarchy shown in Figure 6.5. In the following, the individual caches are briefly described.

Decoding The maps that *HikeDroid* accesses, are represented as a collection of tiles, as stated in Section 6.2. The tiles are stored in the image file format

JPEG. JPEG is a commonly used method of lossy compression for digital images. *HikeDroid* receives the tiles in JPEG format as a compressed stream of bytes. However, this byte stream first has to be decoded into a bitmap representation, which then can be drawn to the device's screen. This decoding is an expensive operation. The bitmap representation is no longer compressed, which means, compared to the JPEG format, more memory is used to store a tile in bitmap representation.

6.3.1 Bitmap Cache

HikeDroid uses an in-memory bitmap cache as *top level cache*. This cache stores a small set of items directly as bitmap objects, which can be directly drawn on the screen, without further transformations. Therefore, in the case of a *cache hit*, the requested item can immediately be given back, without doing further expensive calculations, like the decoding step described above. The bitmap cache therefore has a high bandwidth. The major drawback of storing items directly as bitmaps is the high memory consumption. The bitmap cache therefore has a small capacity.

6.3.2 Byte Array Cache

In *HikeDroid* the *level 1 cache* is a byte array cache. Like the *level 0 cache*, the byte array cache is also held in memory. Storing map data as byte arrays has the advantage of lower memory utilization. For a predefined amount of available memory, a higher number of items can be stored in this cache compared to the bitmap cache. More precisely, the byte array cache has a higher capacity compared to the bitmap cache. The drawback of storing items as byte arrays is the rather expensive and time consuming need of decoding items to bitmap objects. As stated above, bitmap objects are required by the drawing process. Compared to the bitmap cache, the byte array cache has a lower bandwidth.

6.3.3 SD Cache

As a *level 2 cache*, *HikeDroid* uses a SD cache. This cache no longer stores items in memory but on the external *Secure Digital (SD)* card. The major advantage of this cache level is the huge amount of available storage, compared to the two previous cache levels. The SD cache has a much higher capacity. Getting an item in bitmap format from this cache requires a similar transformation as the one proposed in the byte array cache. However, a much higher access time is required to read and write from the external SD storage. The SD cache has a much lower bandwidth compared to the previous cache levels.

6.3.4 Web Cache

Finally, *HikeDroid* uses as *level 3 cache* a web cache. Accessed items are loaded from the Internet, which requires a huge amount of time, compared to all the other levels. This cache has the lowest bandwidth. The web cache contains all available data. It therefore has the highest required capacity. In the case of *HikeDroid*, the map data is provided by the *Swiss Federal Geo Portal*[3].

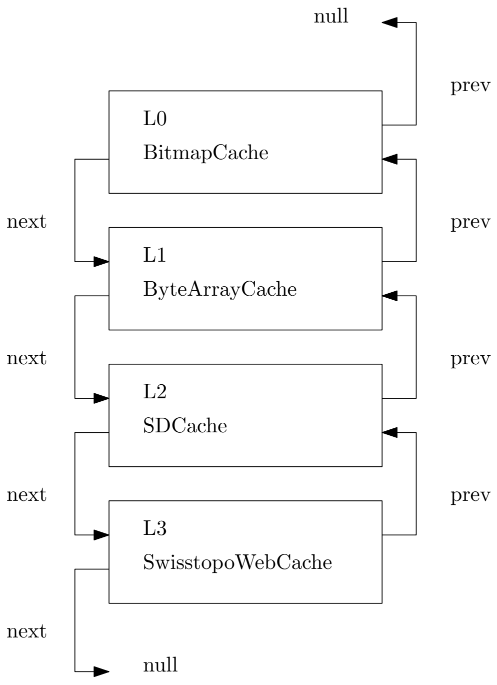


Figure 6.5: SwisstopoPixelMap Cache Hierarchy

[3]Geo Admin Website, 2011. Öffentlicher Zugang zu Geoinformationen und Geodaten Available at: http://map.geo.admin.ch [Accessed June 6, 2011]

Chapter 7

Future Work

By now, *HikeDroid* implements the basic architecture and functionality of a map application. All concepts are constructed to make them easily extensible by future work. New features can be built on top of it. The technical part of the application is now ready to be extended by interesting features and functionalities. One may think of various improvements and extensions for *HikeDroid*. This chapter provides some ideas and starting points for improving the application. There are also a few adjustments proposed how the current work could be enhanced.

7.1 Suggestions for Improvement

This section suggests some points in the current implementation that should be improved by future work.

7.1.1 Cache

Dynamic Request Manager Instantiation The cache itself should decide dynamically when to increase and decrease the number of request managers, depending on the current workload.

Memory Utilization Caches are currently able to store a predefined number of items. However, most caches should limit their capacity by the amount of available memory. A user could for example be allowed to define the amount of SD memory that *HikeDroid*'s SD cache can use.

7.1.2 Map Architecture

This is the part of the application that has to be adjusted the most. In the initial design phase, several issues were not sufficiently taken into account, mostly because they were unclear and unknown at this point in time. Almost all architectural problems that occur are based on the fact, that the concepts of tile maps, different map layers and different zoom levels do not properly fit together. This is the major point that needs to be improved by future work. Adequate adjustments to this part will solve most of the problems in the current implementation.

7.1.3 Offline Maps

At the moment, *HikeDroid* automatically stores the latest accessed map data on the device's SD card. Later accesses to this data can be handled without the need of an Internet connection. However, only data that was accessed before is stored. Moreover, not recently used map data may get overwritten. In fact, users should be provided with the possibility to specify a map sector which is then loaded and stored for future usage. The idea is that users can in advance load all the information needed for their hiking trip. During the trip, all used data should be accessible without the need of establishing a connection to the Internet.

7.1.4 Energy Saving Mode

Mobile devices have a limited amount of battery power available. Applications should therefore try to minimize their power consumption. *HikeDroid* should try to further reduce its power consumption, for example by providing different energy saving modes. Especially a permanently activated GPS receiver is power consuming. *HikeDroid* should provide an energy saving mode which turns the GPS receiver on and off in regular time intervals. For example during hiking, it seems enough to record GPS positions in regular time intervals of a few seconds.

7.1.5 Memory Leakage

The current implementation of *HikeDroid* seems to suffer from memory leaks. Terminating and restarting the application's main activity (for example by switching the screen between portrait and landscape mode) for several times leads the application to run out of memory. This error arises most likely because of the mistake to keep a long-lived reference to the *Context*[1].

[1]Android Developers Website, 2011. Context
Available at: http://developer.android.com/reference/android/content/Context.html [Accessed June 6, 2011]

7.2 Extensions

This section proposes some further ideas of how to enhance the current implementation.

7.2.1 Data Sets

At the moment, *HikeDroid* is not yet able to use the full information provided by the *Swiss Federal Office for Topography*[2]. It only allows users to display hiking trail information. As stated in Section 2.3, further data sets are available. *HikeDroid* should be extended to allow users the access of this information.

7.2.2 Hiking Information

One may think of various extensions providing users with more information about hiking trails. For example, information about the altitude characteristics of a track can be shown. Further, the user could be provided with an estimated amount of time required for a certain hiking trail. *HikeDroid* will therefore most likely have to access vector data of the trails, which is currently not supported.

7.2.3 Features

HikeDroid is designed in a modular way and allows easy extension for its features. There are almost no limitations in extending *HikeDroid*'s features. Elevation profiles for hiking trails, sharing points of interest and notifying the user of incoming thunderstorms are only a few ideas that one may think to add.

[2]Swisstopo Website, 2011. The Federal Geo-Information center
Available at: http://www.swisstopo.admin.ch [Accessed June 6, 2011]

Chapter 8

Conclusion

The Android application *HikeDroid* makes the *VECTOR25* map data freely accessible by mobile devices. *VECTOR25* maps contain a bunch of useful details and informations for hikers. *HikeDroid* provides a convenient way to access this data. Users are allowed to interact with the map. They can move to their desired position, zoom in and out in order to see more or less details, respectively. Multiple scale levels are used to provide the best resolution. *HikeDroid* takes into account, that hikers do not always have the possibility to establish a connection to the Internet. Therefore, a map sector can be loaded in advance at home and accessed offline during the trip. *HikeDroid* further tries to minimize the energy consumption. Additionally the receipt of the devices GPS location is supported. The current location can be directly marked on the map. Furthermore, a walk can be tracked. Users can select different layers of information. For example Swiss hiking trails can be enabled. *HikeDroid* is based on a flexible and extensible architecture. The modular design allows future work to extend the application with additional features.

List of Figures

2.1 Screenshot Comparison of different Map Applications 6

5.1 Hierarchy of n Caches 14
5.2 Cache with Request Managers 16
5.3 Map Adapter connecting Map and Cache 17

6.1 Touch Screen Motion Events 19
6.2 Multi Layering 20
6.3 Map Sector and Coordinate System 21
6.4 Tile Map 22
6.5 SwisstopoPixelMap Cache Hierarchy 24

A.1 Map Adapter UML Class Diagram 1
A.2 Request Manager UML Class Diagram 2
A.3 Cache UML Class Diagram 2
A.4 Map UML Class Diagram 3

Appendix A

UML Diagrams

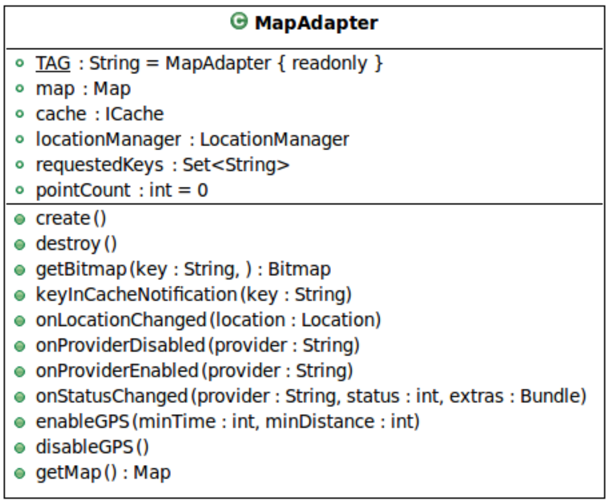

MapAdapter
TAG : String = MapAdapter { readonly }
map : Map
cache : ICache
locationManager : LocationManager
requestedKeys : Set<String>
pointCount : int = 0
create ()
destroy ()
getBitmap (key : String,) : Bitmap
keyInCacheNotification (key : String)
onLocationChanged (location : Location)
onProviderDisabled (provider : String)
onProviderEnabled (provider : String)
onStatusChanged (provider : String, status : int, extras : Bundle)
enableGPS (minTime : int, minDistance : int)
disableGPS ()
getMap () : Map

Figure A.1: Map Adapter UML Class Diagram

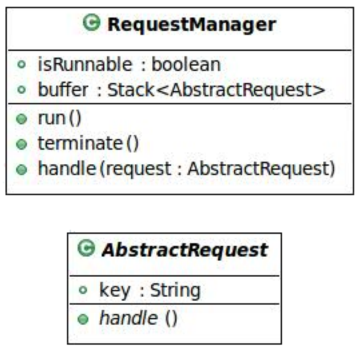


Figure A.2: Request Manager UML Class Diagram

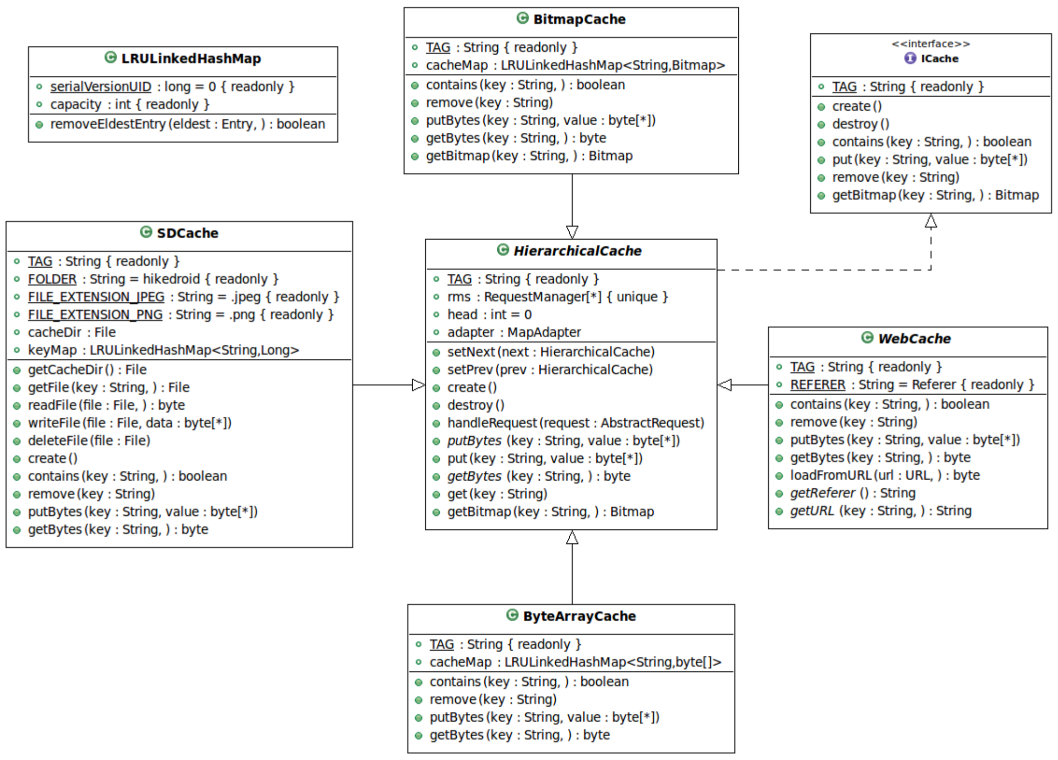

LRULinkedHashMap
serialVersionUID : long = 0 { readonly }
capacity : int { readonly }
removeEldestEntry (eldest : Entry,) : boolean
BitmapCache
TAG : String { readonly }
cacheMap : LRULinkedHashMap<String,Bitmap>
contains (key : String,) : boolean
remove (key : String)
putBytes (key : String, value : byte[*])
getBytes (key : String,) : byte
getBitmap (key : String,) : Bitmap
<<interface>>
ICache
TAG : String { readonly }
create ()
destroy ()
contains (key : String,) : boolean
put (key : String, value : byte[*])
remove (key : String)
getBitmap (key : String,) : Bitmap
SDCache
TAG : String { readonly }
FOLDER : String = hikedroid { readonly }
FILE_EXTENSION_JPEG : String = .jpeg { readonly }
FILE_EXTENSION_PNG : String = .png { readonly }
cacheDir : File
keyMap : LRULinkedHashMap<String,Long>
getCacheDir () : File
getFile (key : String,) : File
readFile (file : File,) : byte
writeFile (file : File, data : byte[*])
deleteFile (file : File)
create ()
contains (key : String,) : boolean
remove (key : String)
putBytes (key : String, value : byte[*])
getBytes (key : String,) : byte
HierarchicalCache
TAG : String { readonly }
rms : RequestManager[*] { unique }
head : int = 0
adapter : MapAdapter
setNext (next : HierarchicalCache)
setPrev (prev : HierarchicalCache)
create ()
destroy ()
handleRequest (request : AbstractRequest)
putBytes (key : String, value : byte[*])
put (key : String, value : byte[*])
getBytes (key : String,) : byte
get (key : String)
getBitmap (key : String,) : Bitmap
WebCache
TAG : String { readonly }
REFERER : String = Referer { readonly }
contains (key : String,) : boolean
remove (key : String)
putBytes (key : String, value : byte[*])
getBytes (key : String,) : byte
loadFromURL (url : URL,) : byte
getReferer () : String
getURL (key : String,) : String
ByteArrayCache
TAG : String { readonly }
cacheMap : LRULinkedHashMap<String,byte[]>
contains (key : String,) : boolean
remove (key : String)
putBytes (key : String, value : byte[*])
getBytes (key : String,) : byte

Figure A.3: Cache UML Class Diagram

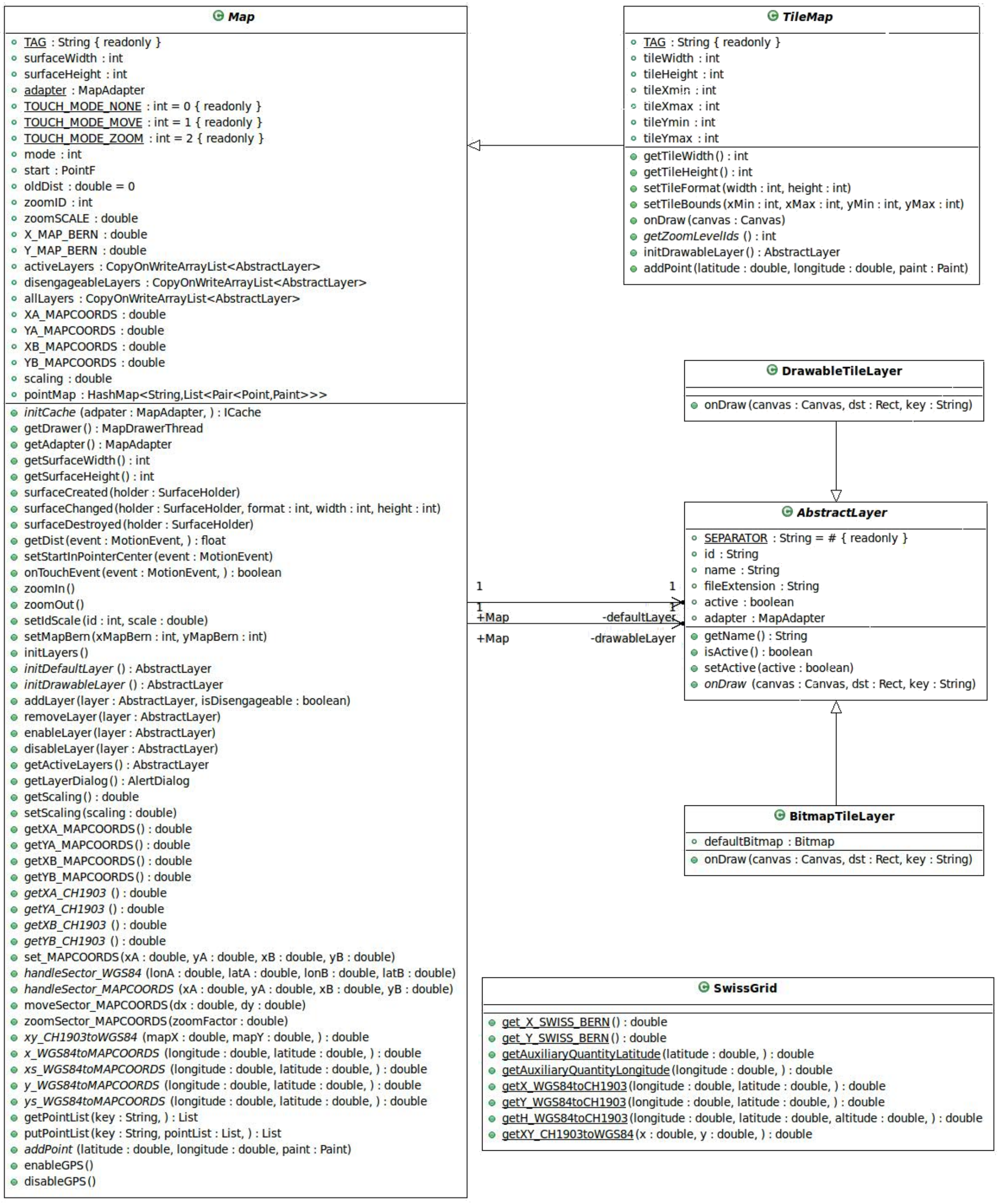

Map
TAG : String { readonly }
surfaceWidth : int
surfaceHeight : int
adapter : MapAdapter
TOUCH_MODE_NONE : int = 0 { readonly }
TOUCH_MODE_MOVE : int = 1 { readonly }
TOUCH_MODE_ZOOM : int = 2 { readonly }
mode : int
start : PointF
oldDist : double = 0
zoomID : int
zoomSCALE : double
X_MAP_BERN : double
Y_MAP_BERN : double
activeLayers : CopyOnWriteArrayList<AbstractLayer>
disengageableLayers : CopyOnWriteArrayList<AbstractLayer>
allLayers : CopyOnWriteArrayList<AbstractLayer>
XA_MAPCOORDS : double
YA_MAPCOORDS : double
XB_MAPCOORDS : double
YB_MAPCOORDS : double
scaling : double
pointMap : HashMap<String,List<Pair<Point,Paint>>>
initCache (adapter : MapAdapter,) : ICache
getDrawer () : MapDrawerThread
getAdapter () : MapAdapter
getSurfaceWidth () : int
getSurfaceHeight () : int
surfaceCreated (holder : SurfaceHolder)
surfaceChanged (holder : SurfaceHolder, format : int, width : int, height : int)
surfaceDestroyed (holder : SurfaceHolder)
getDist (event : MotionEvent,) : float
setStartInPointerCenter (event : MotionEvent)
onTouchEvent (event : MotionEvent,) : boolean
zoomIn ()
zoomOut ()
setIdScale (id : int, scale : double)
setMapBern (xMapBern : int, yMapBern : int)
initLayers ()
initDefaultLayer () : AbstractLayer
initDrawableLayer () : AbstractLayer
addLayer (layer : AbstractLayer, isDisengageable : boolean)
removeLayer (layer : AbstractLayer)
enableLayer (layer : AbstractLayer)
disableLayer (layer : AbstractLayer)
getActiveLayers () : AbstractLayer
getLayerDialog () : AlertDialog
getScaling () : double
setScaling (scaling : double)
getXA_MAPCOORDS () : double
getYA_MAPCOORDS () : double
getXB_MAPCOORDS () : double
getYB_MAPCOORDS () : double
getXA_CH1903 () : double
getYA_CH1903 () : double
getXB_CH1903 () : double
getYB_CH1903 () : double
set_MAPCOORDS (xA : double, yA : double, xB : double, yB : double)
handleSector_WGS84 (lonA : double, latA : double, lonB : double, latB : double)
handleSector_MAPCOORDS (xA : double, yA : double, xB : double, yB : double)
moveSector_MAPCOORDS (dx : double, dy : double)
zoomSector_MAPCOORDS (zoomFactor : double)
xy_CH1903toWGS84 (mapX : double, mapY : double,) : double
x_WGS84toMAPCOORDS (longitude : double, latitude : double,) : double
xs_WGS84toMAPCOORDS (longitude : double, latitude : double,) : double
y_WGS84toMAPCOORDS (longitude : double, latitude : double,) : double
ys_WGS84toMAPCOORDS (longitude : double, latitude : double,) : double
getPointList (key : String,) : List
putPointList (key : String, pointList : List,) : List
addPoint (latitude : double, longitude : double, paint : Paint)
enableGPS ()
disableGPS ()
TileMap
TAG : String { readonly }
tileWidth : int
tileHeight : int
tileXmin : int
tileXmax : int
tileYmin : int
tileYmax : int
getTileWidth () : int
getTileHeight () : int
setTileFormat (width : int, height : int)
setTileBounds (xMin : int, xMax : int, yMin : int, yMax : int)
onDraw (canvas : Canvas)
getZoomLevelIds () : int
initDrawableLayer () : AbstractLayer
addPoint (latitude : double, longitude : double, paint : Paint)
DrawableTileLayer
onDraw (canvas : Canvas, dst : Rect, key : String)
AbstractLayer
SEPARATOR : String = # { readonly }
id : String
name : String
fileExtension : String
active : boolean
adapter : MapAdapter
getName () : String
isActive () : boolean
setActive (active : boolean)
onDraw (canvas : Canvas, dst : Rect, key : String)
1
1
1
1
+Map
-defaultLayer
+Map
-drawableLayer
BitmapTileLayer
defaultBitmap : Bitmap
onDraw (canvas : Canvas, dst : Rect, key : String)
SwissGrid
get_X_SWISS_BERN () : double
get_Y_SWISS_BERN () : double
getAuxiliaryQuantityLatitude (latitude : double,) : double
getAuxiliaryQuantityLongitude (longitude : double,) : double
getX_WGS84toCH1903 (longitude : double, latitude : double,) : double
getY_WGS84toCH1903 (longitude : double, latitude : double,) : double
getH_WGS84toCH1903 (longitude : double, latitude : double, altitude : double,) : double
getXY_CH1903toWGS84 (x : double, y : double,) : double

Figure A.4: Map UML Class Diagram

Appendix B

Reference Documents

The following reference documents are also contained on the CD that comes with this report.

B.1 Swiss Map Mobile, 2009

Comparison of the functions/platforms
Available at: http://www.swisstopo.admin.ch/internet/swisstopo/de/home/products/maps/mobile.parsysrelated1.56233.downloadList.28768.DownloadFile.tmp/smm2009vergleichfunktionen.pdf [Accessed June 6, 2011]

B.2 Swiss Map Mobile Sectors, 2009

Status of map content
Available at: http://www.swisstopo.admin.ch/internet/swisstopo/de/home/products/maps/mobile.parsysrelated1.56233.downloadList.82447.DownloadFile.tmp/nachfuehrungsstandswissmapmobile2009all.pdf [Accessed June 6, 2011]

B.3 Coordinate System Transformations, 2008

Formeln und Konstanten für die Berechnung der Schweizerischen schiefachsigen Zylinderprojektion und der Transformation zwischen Koordinatensystemen
Available at: http://www.swisstopo.admin.ch/internet/swisstopo/de/home/topics/survey/sys/refsys.parsysrelated1.23611.downloadList.12097.DownloadFile.tmp/refsysd.pdf [Accessed June 6, 2011]