Development of an Android App for One-Time Password Generation & Management

Bachelor Thesis
Sommersemester 2010

Bearbeitet von:
Michael Barth
(Matrikelnummer: 26206)

Betreuer:
Prof. Dr. Christoph Karg

Hochschule Aalen

Hochschule für Technik und Wirtschaft Aalen
Fakultät Elektronik und Informatik
Studiengang Informatik

Abstract

This work covers the development of an Application for the Android platform for One-Time Password Management and Creation, including all fundamentals that are necessary to do this.

One-Time Passwords are introduced in general. Their benefits and drawbacks are discussed and their usage is illustrated with a practical example. Concluding, a specific OTP implementation (OTPW) is introduced.

Following is an introduction on the topic of Random Number Generation in the context of cryptography. An overview over attacks on Pseudo Random Number Generators (PRNGs) is given, as well as some design guidelines to prevent them.

The Android platform is introduced in detail to establish a basic understanding of the target platform, describing its architecture and application framework.

An extensive introduction to development for Android is given in the next chapter, including installation and setup, general guidelines on developing for mobile devices and practical examples of the most important components with source code.

The application developed over the course of this thesis will then be described in detail, including its architecture, design decisions and an elaboration on the implementation details.

A conclusion, including an evaluation of the Android platform and the application, summarises this work.

This work was done within the scope of the *Hochschule für Technik und Wirtschaft Aalen*, Germany, as a bachelor thesis over the course of the 8th semester.

Contents

Ehrenwörtliche Erklärung (Declaration of Honour)

Ich versichere hiermit, dass ich die vorliegende Bachelorarbeit gemäß § 27 Abs. 1 SPO 28 der Hochschule Aalen, selbstständig und unter ausschließlicher Verwendung der angegebenen Quellen und Hilfsmittel erstellt habe.

Die Arbeit wurde bisher keiner anderen Prüfungsbehörde in gleicher oder ähnlicher Form vorgelegt und auch nicht veröffentlicht.

Michael Barth, 1. Oktober 2010

List of Figures

List of Tables

Listings

1 Introduction

1.1 Motivation

The smartphone market has become very popular with the release of the Apple iPhone in January 2007, which sparked the creation of apps – small application software that can be downloaded and installed on the iPhone. Aside from the introduction of a touchscreen as the main input device, apps were one of the most intriguing new features the iPhone introduced to the smartphone market, allowing each user to enhance his phone with new functionality on a whim. Today it is said there is an app for everything.

Come 2008, Google made its successful entrance into the market with the Android open-source operating system for smartphones [1]. The smartphone market has grown significantly since then and is said to grow even more in the future [2]. With Android the first open platform for smartphones appeared and became rather popular with the open-source community as Apple has a very firm and restrictive grip on the iPhone, which it enforced even more with version 4 of the iPhone OS.

The less restrictive terms of use as well as the booming popularity of smartphones in general, which become more and more powerful by the year, are strong incentives to take a closer look at the Android platform – including a real-life example of app development. To conclude this an evaluation of the platform itself and the app development process will be done.

1.2 Outline

Chapters two till four will introduce the reader to the fundamentals for this work. Chapters five and six contain the main part of this work: A deeper look into Android application development and the documenatation of the developed app. In the last chapter, a conclusion is drawn including an evaluation of the results.

In the first chapter One-Time Passwords (OTPs) are introduced: How they work, the general idea behind them and how they can improve security over the internet.

Following is a chapter which introduces the concept of random number generation on deterministic machines like the computer and explains why this is so difficult, especially in the context of cryptography.

In the third chapter the reader will be introduced to the Android platform, providing some fundamental facts on the platform itself and its application framework.

The fifth chapter will delve into application development for Android, including installation and setup of an development environment.

The sixth chapter documents the developed OTP Management app, explaining design decisions and key implementation details.

Lastly, in the seventh chapter, a conclusion on the experiences and results of this work will be given.

1.3 Goal of this work

The goal of this work is both to explore and use the Android platform and its possibilities in app development. To get a real "experience in the field", an app for the storage, management and creation of One-Time Passwords will be developed during the course of this work. With the insights gained through this pragmatic approach an overview over the platform is given concluded by an evaluation of the platform itself as well as the app development during this work.

1.4 Approach

First an extensive introduction and orientation period was taken, to get a grip of the platform by reading the *android developers dev guide* [3] and *Professional Android 2 Application Development* [4] by Reto Meier. After that, I started programming the actual app. A month into programming I started documenting the basics for this work while continuing to further developing the app.

1.5 Utilised tools and resources

The Android application was created utilising the following tools and resources:

- eclipse IDE 3.5.2
- Android Development Tools 0.9.7 (ADT)
- Android SDK, r6
- USB Driver for Windows, r3
- Mercurial SCM 1.6.2
- Windows 7 Professional

This documentation was created using the following tools and resources:

- Kile 2.0.83
- Ubuntu 9.10
- Microsoft Visio 2007
- Enterprise Architect 7.5
- Windows 7 Professional

2 One-Time Passwords

We will examine One-Time Passwords in this chapter. To get a basic understanding, we will first discuss what they are, what their purpose is and how a One-Time Password system can be designed in general. After the basics, we will take a closer look at a specific open-source One-Time Password package for Linux: OTPW [7].

So what is a One-Time Password? Put simply, a One-Time Password (OTP in short) is a password that can be used only a single time (only one time, therefore the name). After it has been used, it becomes invalid (or depleted) and cannot be used again. This helps to prevent some shortcomings of *static passwords* – passwords that do not change typically.

2.1 Why One-Time Passwords?

Static passwords don't change unless it becomes necessary to change them – when it expires or the user forgets his password, for example. This fact can be easily exploited by an attacker in a replay attack, in which the attacker can acquire and misuse the authentication information of a user by eavesdropping on the underlying communication channel between user and system over a network.

2.1.1 Replay attacks

A replay attack utilises the knowledge that the authentication information will typically stay the same. In order to exploit this, all the attacker needs to do is to intercept a valid data package (using a *password sniffer* or *keylogger*) containing the authentication information of the user – most commonly a username/password pair. Without having to decrypt or "crack" the password he can do a number of malicious things once he obtained such a package, like using it to authenticate himself at a later time or repeating the package.

Example

Suppose Alice wants to order some products from Bob's webshop. To prove her identity, Alice logs into Bob's webshop by entering her username and password (this is called authentication*). Eve is eavesdropping on the communication channel between Alice and Bob and intercepts this transmission. Eve can now use the intercepted package to authenticate herself as Alice. Eve could also intercept the transmission containing the* *order of Alice and repeatedly sent it to Bob. Later, Alice will be surprised to find she has been charged several times for the same order.*

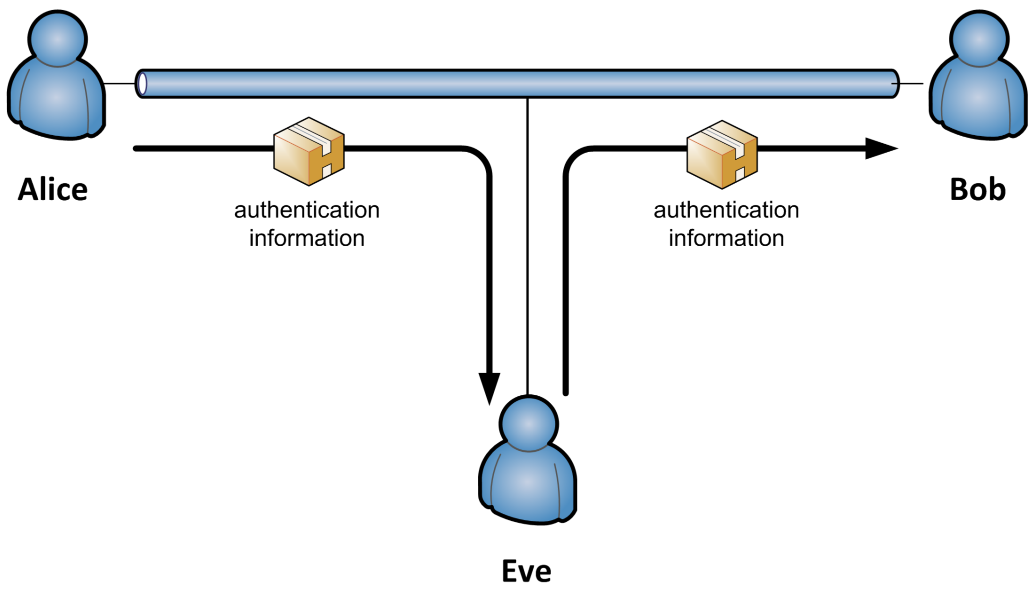


Figure 2.1: Alice tries to authenticate herself with Bob, but Eve intercepts the package and executes a replay attack

This practically sidesteps the security promise of a password in a rather easy way for the attacker. Regardless how strong a password is – that means how hard it is to decrypt or "crack" – the attacker can use the intercepted package to authenticate himself as another person and enter secure areas or execute actions he should not be allowed to. There is no way for the system to discern a valid data package sent by the true user from a valid data package sent by the attacker.

As can be seen from the example, a simple attack like this can already cause serious damage. One-Time Passwords are one of the solutions conceived to prevent this.

Password Sniffer/Packet Sniffer

A packet sniffer is a software that can intercept and analyse traffic being sent over a network or part of a network. It captures, decodes and analyses the packages being sent over the network.

Password sniffers are packet managers specialised to find authentication information (usernames and passwords) in packages. Most password sniffers work passively, meaning they don't generate any network traffic to avoid being detected.

Keylogger

Keylogger typically refers to a software that tracks every keystroke the user inputs on his keyboard. Usually, the software is installed without the knowledge of the user to spy authentication information. A keylogger either stores its collected data locally from where it can be collected by the attacker or sends the log directly to the attacker.

There are also various hardware keyloggers, which work on the same basic principle of capturing keystrokes for an attacker to exploit.

2.2 The Idea behind One-Time Passwords

The remedy here is easy: When the attacker utilises the knowledge that authentication information typically stays the same for each log-in, change that fact. An One-Time Password is a password that can be used exactly once. In other words, it will be different each time. This already solves the security vulnerability caused by a replay attack.

Typically, an OTP is used in conjunction with static authentication information. When the user wants to log in, the system asks the user for his username, password and in addition an One-Time Password. The username and password are static, they don't change for each and every attempt to log in and are therefore easy to remember. In contrast, the OTP will be different every time.

When the attacker tries to send a package repeatedly, the system will notice that the One-Time Password is not correct (it stays the same) and can issue a warning. In case the attacker manages to intercept and send his package first, then the user will notice something is wrong when his attempt to log in fails.

One-time passwords are one way to remedy the vulnerability of static passwords to replay attacks.

2.2.1 Limits of One-Time Passwords

With the benefits of One-Time Passwords in mind, it is also important to note their limits and drawbacks.

- OTPs offer no protection to **interception in general**. Other security protocols have to be established to detect when an attacker intercepts a message, preventing it from ever reaching its recipient.

- Also, an attacker may profit from simply **delaying a message** for a set amount of time. The delaying itself can not be prevented by OTPs, but it's possible to prevent the delayed message from being accepted at a later point by using *time-synchronised* One-Time Passwords. How this works will be further elaborated in section 2.3.1.

- OTPs **do not protect the privacy of data** [9].

- OTPs – like many other security systems – are vulnerable to **social engineering**. Attackers have been able to trick people into providing one or more One-Time Password(s) they've used in the past [8]. With those it is possible to generate the next OTPs that will come up, as will be described in section 2.3.1.

- OTPs can not prevent active attacks like **session hijacking**, in which an attacker obtains the active user session after he has successfully authenticated himself but before he logs out [9].
- OTPs are **dependent on the non-invertability of the secure hash function** used [9]. At the time of this writing, the suggested MD4 and MD5 have already been proven "severly compromised" (see [10] and [11]). The use of SHA-1 is also discouraged, as it has also been reported being broken in 2005 [12].

One-time passwords should never be the sole answer to security, but one layer in a layered security scheme that helps prevent some attacks – but by far not all.

2.2.2 Vulnerabilities

An attacker can try a *race attack* by listening to most of the One-Time Password (possibly with a keylogger) and guessing the remainder. By guessing the last word/the last few characters before the legitimate user has finished the attacker can try multiple guesses in order to race the user to complete authentication before him [9]. This can work as it is much easier to guess only the last few parts than the whole password.

The RFC 2289 suggests preventing a user from starting multiple simultaneous authentication sessions to block out an attacker as a possible solution. Although a timeout would be necessary to prevent a Denial of Service attack [9].

2.3 Design

Altough One-Time Password systems can be designed in several different ways, they all follow a basic scheme. This basic scheme, as well as some of the possibilities when designing an OTP system, will be elaborated in this section.

Basically, authentication with One-Time Passwords follows these steps:

1. The user wants to authenticate himself with a service to access restricted information or services.
2. In order to authenticate, the user provides authentication information.
   a) In addition to the usual static authentication information, the user is asked to provide a specific OTP which he may keep on a list or generate on the fly using a special device.
3. The system verifies the authentication information and grants or denies access to the user.

The static authentication information prevents access by an unauthorised person in case the One-Time Passwords are lost or stolen, since the One-Time Password alone is useless. The static authentication information can be deposited in several ways, storing it (encrypted) in a database being the most common one. One-time passwords can also be stored or generated on the fly for verification. This depends on the specific generation method used, as will be explained in the *Password Generation* section.

2.3.1 Password Generation

Randomness[1] plays a major part in the generation of One-Time Passwords. Otherwise it would be easy to deduce future One-Time Passwords by observing previous ones. Specific generation methods may vary greatly in their implementation details, but the password generation is generally based on one of the approaches following below [8]:

- time-synchronisation
- the previous password in combination with a mathematical algorithm
- a challenge in combination with a mathematical algorithm

How those approaches differ from one another is elaborated in the following sections.

Based on Time-Synchronisation

One-time passwords can be generated based on the current time, these are called time-synchronised One-Time Passwords. The server clock must be synchronised with the clock of the tool used to generate the passwords. Time skew can be a problem here since the generated passwords will be based on the time (solely or amongst other things like the previous password or a secret key). Devices running on different times will generate different – and therefore invalid – One-Time Passwords.

Also, time-synchronised OTPs must be entered within a certain period of time, after which the One-Time Password expires. This can be beneficial to prevent delayed messages from being accepted.

Based on the Previous Password

A *one-way function* is used to generate each new password based on the beforgoing password. Assuming a one-way function $f()$ and initial seed s, passwords are generated in the fashion

$$f(s), f(f(s)), f(f(f(s))), ...$$

[1]See chapter 3 for more details on problems and approaches to achieve this on deterministic machines like the computer.

as many times as neceassary. The passwords are then used in the reverse order, starting with the last password [8].

This is called the *Lamport scheme* after its inventor, Leslie Lamport. The Lamport scheme is safe in theory since a one-way function is defined to be easy to compute but extremely difficult to inverse (meaning calculating its inverse function f^{-1}). When an attacker gets his hands on a single OTP p_1 he can theoretically calculate the next One-Time Password from it – which was actually the beforegoing one in the generation process or, in other words, the input used to generate p_1. Calculating the input of a function is theoretically possible by calculating its inverse function. Therefore:

$$f^{-1}(p_1) = p_0$$

Whereas p_0 is going to be the next valid One-Time Password. The security of the Lamport scheme stems from the idea of computing the inverse of a one-time function being a computationally infeasible task, which translates to: By the time the attacker has managed to successfully compute the inverse it will no longer be of use to him.

In the majority of cases, *cryptographic hash functions* are used as a one-way function to calculate the passwords. The quality of the one-way function is of utter importance to the security of One-Time Passwords generated with the Lamport scheme.

Unfortunately, there is no perfect one-way function, and as Bruce Schneier in *Practical Cryptography* [13] states:

> *"There are very few good hash functions out there. At this moment you are really stuck with the SHA family, or possibly MD5. (...) Unfortunately, all of these hash functions have some properties that disqualify them according to our security definition."*

This has proven to be true two years later when the SHA-1 hash function was reported as being flawed [12].

All summed up, it can be said that the Lamport scheme is good in theory, but suffers from the absence of a cryptographically robust one-way function to this date.

Based on a Challenge

Challenge-based systems issue a challenge to which the user must respond. The response is then used to alter the One-Time Passwords generated in a deterministic way, meaning the result is reproducable given the response is known and used to generate the OTP in exactly the same way. Examples are providing a personal identification number (PIN) after which the One-Time Password will be generated based on the challenge response or the server displaying a random number which the user has to enter on his OTP generation device.

This approach requires the user to be able to generate One-Time Passwords on the fly when needed. Challenge-based systems can be used in combination with the Lamport scheme as detailed in section 2.3.1.

2.3.2 Storing passwords

As One-Time Passwords are randomly generated and lots of them are needed, they are hard to memorise. There are different approaches how to handle this, some require storing the OTPs, others generate them on the fly with a special hardware device.

Printed List

The simplest possible solution involves printing a generated list of One-Time Passwords unto a sheet of paper. This has the advantage of being cheap, portable and easy to perform, as almost anyone should have access to a printer. Also, keeping the passwords on a sheet of paper prevent all dangers through hacking attempts on the user's computer.

On the other hand, a sheet of paper may be forgotten, lost or stolen. Also, a sheet of paper can't produce new passwords when the list runs out of unused ones.

Special Hardware Device

Special hardware devices are not cheap, but offer good security and the ability to generate One-Time Passwords on the fly when needed. They offer almost all benefits like the printed list on a sheet of paper, like being portable and being safe from hacking attempts.

But they can also be forgotten, lost or stolen.

Keeping them on a Computer

Keeping the list on a computer allows the generation of OTPs on the fly without a problem. The chance of losing a computer is marginally small, likewise the chance of it being stolen.

On the downside, it's prone to hacking, rather unwieldy – even a laptop – and anything but cheap.

Mobile Phones

There are two possible solutions to handle One-Time Passwords on a mobile phone: Either with an application software running on the phone or by SMS delivery. First, we'll talk about the general benefits and drawbacks of mobile phones, then we will look into the specific benefits and drawbacks of the two presented methods.

Mobile phones are portable and almost everyone carries one with him all the time, so the chance of forgetting it is small. Also, both methods allow the generation of new OTPs on the fly.

Mobile phones in general are prone to being lost or stolen.

Delivery via SMS has the benefit of being cheap to acquire: Since most people already own a mobile phone capable of sending and receiving SMS, the hardware isn't a cost factor in most cases.

On the other hand, depending on the frequency at which One-Time Passwords are needed, the costs of sending SMS can add up.

Managing OTPs via an application software on the mobile phone offers the benefit of being cheap to operate: No costs for requesting a new One-Time Password.

But devices that allow such applications to run may be more expensive then typical mobile phones and not everyone owns a mobile phone capable of this, so there may be acquisition costs.

All in all, mobile phones offer a good compromise between cost and efficiency. A modern smartphone can practically replace a special hardware device completely.

2.4 OTPW – A One-Time Password Login Package

OTPW is a One-Time Password package for authentication in Unix-like operating systems (covering amongst others Unix, Linux, OpenBSD, NetBSD, and FreeBSD) developed by Markus Kuhn. It comes with an OTP generator (`otpw-gen`) and two verification routines (`otpw_prepare()` and `otpw_verify()`) that can easily be added to login programs on POSIX systems. A wrapper for the *Pluggable Authentication Method* (PAM)[2] interface is also available. [7]

The purpose of the OTPW package is stated in the introduction as follows:

> *"(...) traveling computer users often want to connect to their home system via untrusted terminals at conference hotels, other universities, and airports, where trusted encryption software is not available. Even Secure Shell does not protect against keyboard eavesdropping software on the untrusted terminal. A loss of confidentiality is often acceptable in these situations for the session content, but not for reusable login passwords. One-time-password schemes avoid the transmission of authentication secrets that are of any value after they have been used."*
>
> Quoted from [7]

This work is based on version 1.3 of OTPW released on the 30th September 2003.

2.4.1 Installation

The installation of OTPW is easy and described on the official homepage [7]. For typical usage it suffices to install the `otpw-bin` package which should be available for most popular Linux distributions. It is also possible to download and compile the sourcecode of OTPW, for those distributions where the comfort of an already compiled version is missing.

On Ubuntu 9.10 – the operating system used for this work – it is available via the `apt-get` package manager and can be installed executing the following command in the terminal:

[2] "*The [PAM] library is a generalized API for authentication-related services which allows a system administrator to add new authentication methods simply by installing new PAM modules, and to modify authentication policies by editing configuration files.*" [14]

```
$ sudo apt-get install otpw-bin
```

The above has to be entered without the `$` character, which just serves as an terminal prompt symbol to indicate those commands should be inputted there. The PAM wrapper is available under the package name `libpam-otpw` and can be installed likewise:

```
$ sudo apt-get install libpam-otpw
```

Please refer to the official OTPW homepage for more detailed instructions on installing, configuring or compiling OTPW or the PAM wrapper.

2.4.2 Usage

The usage of OTPW can be roughly splitted into two activities: Generating the One-Time Password list and using them to log in. A section is dedicated to each activity following below.

Generating passwords

Generating One-Time Passwords with OTPW is easy. Just execute the `otpw-gen` executable from the command line in the following manner to print a OTP list:

```
$ otpw-gen | lpr
```

If OTPW is already active on the system (which will usually mean the generator has been run before) it will ask the following:

```
Overwrite existing password list '~/.otpw' (Y/n)?
```

This will practically invalidate the OTP list that has been generated before and replace it with the password list that will be generated this time. This makes sense since there is no use in keeping old password lists valid when new ones have been generated and are available.

Next, the user will be asked to enter a prefix password two times to avoid typing errors:

```
Enter new prefix password:
Reenter prefix password:
```

This can be thought of as the static authentication password the user will have to memorise along with his username. The prefix password ensures losing the One-Time Password list won't result in a security problem, as the OTPs alone won't suffice to authenticate.

As soon as the user has entered and confirmed the prefix password, `otpw-gen` will generate 280 OTPs in 5 columns of 56 passwords per column and a total of 60 lines by default. Unless told otherwise, One-Time Passwords generated by OTPW will use random characters rather than 4-letter words like defined in RFC2289. This behaviour can be changed by providing the `-p1` option parameter.

There are several other options available:

```
Options:

-h <int>        number of output lines (default 60)
-w <int>        max width of output lines (default 79)
-s <int>        number of output pages (default 1)
-e <int>        minimum entropy of each One-Time Password [bits]
                (low security: <30,
                 default: 48,
                 high security: >60)
-p0             passwords from modified base64 encoding (default)
-p1             passwords from English 4-letter words
-f <filename>   destination file for hashes (default: ~/.otpw)
-d              output debugging information
```

When executed like above, it will also send the result to the default printer (indicated by the '`| lpr`'). Alternatively, it is also possible to stream the output into a file:

```
$ sudo touch otp_list
$ otpw-gen > otp_list
```

Though possible, it is discouraged to store the password list on a computer. This just offers more potential targets to an attacker. For this work, it is necessary to transfer the `otp_list` file to the Android mobile device. It is recommended to delete the file from the computer afterwards.

Logging in

When logging into a system that uses OTPW, the password prompt will provide a three digit number (index) that indicates which OTP the authentication server expects from the user. This will look similar to this:

```
username:
password 019:
```

To successfully log in, the user has to provide his usename and then the prefix password immediately followed by the One-Time Password identified by the three digit number on the password line. Suppose the username is 'mbarth', the prefix password 'prefix' and the One-Time Password 019 is 'ABcd1234', then the authentication information has to be entered the following fashion:

```
username: mbarth
password 019: prefixABcd1234
```

Assuming all information was entered correctly the user will be authenticated and granted access.

If an attacker starts a *race attack* against the user – by trying to authenticate several hundred milliseconds before the legitimate user can – and the system detects concurrent login attempts, then it will – instead of granting access – ask for three additional OTPs:

```
username: mbarth
password 024/007/113: prefixQ=XK4I7wIZdBbqyHA5z9japt
```

This will deplete only the initial OTP. The three additional OTPs, that had to be provided to prevent the race attack, will **not be depleted**.

> *This way, the three-password method ensures that an attacker cannot disable the OTPW mechanism by locking all passwords. The triple challenge ensures that many ten thousand network connections would be necessary to perform a race attack on the same password triple, which is not practical.*
>
> Quoted from [7]

On the other hand it should be noted that this leaves three furthermore valid One-Time Passwords in the hand of the attacker. The attacker can exploit this mechanism to gather more valid OTPs. Although he does not necessarily know the index of these OTPs, it still provides him with a pool of valid One-Time Passwords which he can further exploit (for authentification trials, e.g.).

2.4.3 Design

The main difference to note on the design of OTPW is that it is not based on the *Lamport scheme* as introduced in section 2.3.1. Also, OTPW does not store the encrypted One-Time Passwords on the host to keep it free from secrets and therefore minimise potential targets for attackers.

Instead, a hash of the prefix password concatenated with each One-Time Password of the current list is kept in a hidden file in the users' home directory. The details on this file will be elaborated in section 2.4.3 *Server configuration*.

OTPW will not deplete more than one OTP at a time, to prevent attackers from intentionally exhausting a One-Time Password list.

Generating passwords

OTPW generates its passwords using the `RIPEMD-160` hash function which is – at the date of this writing – still considered safe. The random number generator used by OTPW is seeded by hashing together the output of several shell commands to provide unpredicability. This produces a 160 bit random state of which the first 72 bit are encoded by a modified `base64` scheme to produce an ASCII string from the random state. The remaining 88 bits are kept undisclosed. This produces a single One-Time Password.

For the next password, a new random state is generated by concatenating the last random state with the current high-resolution system time and then hashing it again with `RIPEMD-160`, eventually `base64` encoding the first 72 bit again. This procedure is repeated until enough passwords have been generated [7].

The `base64` scheme is modified to replace ambiguous characters like `0O` and `1lI` that are difficult to distinguish in some fonts.

Storing passwords

The primary intent of OTPW is to print the One-Time Password list on a sheet of paper. This is apparent when looking at the output of the `otpw-gen` command, which is optimised for printing (therefore the 5 column layout instead of just writing one password per line like it is done in the `.otpw` file introduced in section 2.4.3 *Server configuration*).

Through the usage of a prefix password, the danger of losing an OTP list or theft causes no security breach in itself, as an attacker still won't be able to authenticate himself with only the password list. It's another story if the attacker managed to listen on previous logins was able to deduce the prefix password from these attempts. Hence, it is recommended to replace the lost list – and therefore invalidate it – as soon as possible nevertheless.

Server configuration

So how does the server know which OTPs are valid? When the user generates a new OTP list, the OTP generator silently creates a hidden file in the users' home directory that stores the first 72 bit of a `RIPEMD-160` hash of the prefix password concatenated with each of the One-Time Passwords. This file is called `.otpw` and looks similar to this:

Listing 2.1: Example of the .otpw file

```
OTPW1                     // file format identifier
10 3 12 8                 // info of list properties
023vf+Uvbg7AqjC           // OTP hashed with prefix pw
024aPEXvP3pgUYa
---------------           // Used & erased OTP
---------------
025Mm+mJWbStzL5
008r3b5efMVT%IU
002AQ63iP8ymMWq
005:PcEr:LoieK0
---------------
0135:Gw==tjv=rA
```

The first line identifies the file format. The second line contains several infos:

- number of originally generated password entries (10)

- digits per password number (3)
- characters per base-64 encoded hash value (12)
- characters per One-Time Password (8)

After that the hashed combination of the prefix password with the OTPs follows. Lines containing used passwords are erased with hyphens to prevent any reuse [7].

3 Random Number Generation

Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin.

— John von Neumann (1951)

Randomness – in the form of *Random Number Generation* (RNG in short) – plays an important part in many areas of application in the IT sector. Fields like algorithmic probability, information theory, computer simulations, computer games, pattern recognition, statistics or cryptography rely on randomness by some means or other. But, like von Neumann jokingly indicated in the quote above, generating random numbers on a deterministic machine is kind of paradox and anything but easy.

This chapters aims to provide a basic introduction into what randomness is and how it is applied in cryptography. *Pseudo Random Number Generatators* (PRNG) are introduced, an overview over generation methods is given, as well as their problems, and last but not least, what to take into consideration when using PRNGs in cryphography is discussed.

3.1 Definition of Randomness

Depending on the field and context, the word *random* holds different meanings. For this work we will rely on the definition of randomness used in statistics:

> *"A numeric sequence is said to be statistically random when it contains no recognizable patterns or regularities; sequences such as the results of an ideal die roll, or the digits of π exhibit statistical randomness.*
>
> *Statistical randomness does not necessarily imply "true" randomness, i.e., objective unpredictability. Pseudorandomness is sufficient for many uses."*
>
> Quoted from [15]

Predictability – or to be more precise the lack thereof – is a key aspect of randomness. The measure for randomness is *entropy*. Simply put, entropy is a measurement for disorder. This term is borrowed from information theory where it is known as the *Shannon entropy*, which in turn borrowed it from the laws of thermodynamics. We're going to work with the definition of the Shannon entropy here.

3.1.1 Shannon entropy

In information theory, entropy is a measure describing the uncertainty of a random value. The Shannon entropy measures the amount of information contained in a message, usually in bits. This is typically used to determine the absolute limit possible with lossless compression [16].

For a random variable X with n outcomes $\{x_i : i = 1, \ldots, n\}$, the Shannon entropy is defined as

$$H(X) = -\sum_{i=1}^{n} \Big(p(x_i) \cdot log_b p(x_i) \Big)$$

where $p(x_i)$ is the probability of the occurrence of x_i [16].

In cryptography, this measure has another use. A 32-bit word that is completely random has an entropy of 32 bits. When the 32-bit word takes on only four different values, each with a probability of 25%, then the word has an entropy of 2 bits. In conclusion, entropy does not measure how many bits there are in a value, but rather how uncertain you are about the actual value [13].

Taking the values from the example and inserting them into the formula, we get the following:

$$\begin{aligned} H(X) &= -\sum_{i=1}^{4} \Big(p(x_i) \cdot log_2 p(x_i) \Big) \\ &= -4 \cdot \Big(0.25 \cdot log_2(0.25) \Big) \\ &= 2 \end{aligned}$$

since each value x_i has the same probability of occurrence.

The more you know about a value, the smaller its entropy is. Therefore it's best to strive for a high entropy for good random numbers to make them unpredictable, especially in cryptography.

3.2 Real Random vs Pseudo Random

So how do we get random numbers? There are two general approaches: Using random data from real sources that are considered to be random, called "real" random, or generating them using computational algorithms to produce long sequences of seemingly random numbers, called pseudo random.

3.2.1 Real Random

Real random consists of measuring physical phenomena that are expected to be random but which may need to be compensated for possible biases in the measurement process. This may be based on simple methods like flipping a coin, rolling a die, turning a roulette wheel. More advanced techniques are based on atomic or subatomic phenomena like radioactive decay, thermal noise, radio noise or the laws of quantum physics [17].

Typical computers offer some real random sources, too. Mouse movement or the exact timing of key strokes are well-known examples. Even random fluctuations in harddisk access time – the timing of actual movements of a hard disk read/write head – caused by turbulence inside the enclosure may be used.

Lastly, there are dedicated built-in random number generators in some modern computers. This is an improvement over separate RNGs, as it makes attacks on the RNG device harder [13].

Problems

Real random sources typically suffer from systematic biases that cause the random numbers to be not uniformly distributed. This can be compensated by using a cryptographic hash function on those random values to obtain uniformly distributed bits [17].

Also, those sources can be somewhat suspect as there may be situations in which an attacker can perform measurements on the random source or even influence it. This may very well be the case for mouse movements or key strokes, which may not be truly random in the first place. A user may have a roughly predictable typing speed or be suggested to move his mouse in a particular pattern by the attacker. In cryptography, this is unfavorable, as one has to always keep in mind that an intelligent adversary will try to break the weakest link in a security system – which could, of course, be the Random Number Generator.

Last but not least, good real random sources aren't always available. When the user is not typing or moving the mouse, you won't be able to receive random numbers from those methods. This can be a serious problem for a RNG running on a Web server which doesn't even have a keyboard and mouse. Physical random number generators are fairly intricate things (like a hardware generator based on quantum mechanics) – much more likely to break than the usual parts of a PC. When this happens, you're out of luck. Furthermore, it is very difficult to measure the actual amount of entropy you'll get from a real random source [13].

3.2.2 Pseudo Random

All the problems inherent in real random sources (as discussed in section 3.2.1) led to the invention of pseudo random generation methods. A pseudo random generation method is a *"[deterministic] algorithm for generating a sequence of numbers that approximates the properties of random numbers."* [18]

A Pseudo Random Number Generator (PRNG) is not truly random. Instead, it collects randomness from a number of typically low-entropy input sources and tries to generate an output that is practically indistinguishable from real random sources [20]. The sequence produced by the PRNG is determined by a random initial value (more often than not called a *seed*) with which the PRNG produces a sequence of seemingly random data. The seed is also called the PRNG's *state* [18]. Note that a specific seed will always produce an identical sequence of pseudo random data. In other words, a PRNG is not random at all.

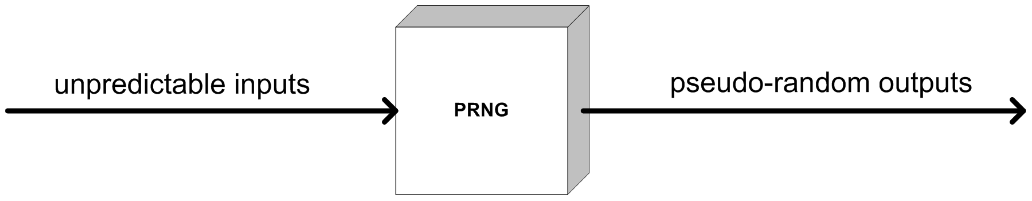


Figure 3.1: Black box diagram of a pseudo random number generator

Pseudo Random Number Generators are important for simulations, cryptography, and lots of other applications. There are several different approaches to generating pseudo random numbers, each developed for different fields of application and, as a result, offering different qualities and drawbacks.

Problems

Of course, PRNGs are not perfect. The output from many PRNGs exhibit artifacts such as shorter sequences of random numbers than expected from some seeds (called *weak seeds*), non-uniformly distributed numbers for large amounts of generated numbers, correlation of successive values, poor dimensional distribution or differently distributed distances between certain values. There are several statistical tests to detect such artifacts, such as the Diehard tests by George Marsaglia [15].

TOUR OF ACCOUNTING
OVER HERE WE HAVE OUR RANDOM NUMBER GENERATOR.
NINE NINE NINE NINE NINE NINE
ARE YOU SURE THAT'S RANDOM?
THAT'S THE PROBLEM WITH RANDOMNESS: YOU CAN NEVER BE SURE.
© 2001 United Feature Syndicate, Inc.

Figure 3.2: Dilbert illustrating the problem with verifying good RNGs. From www.dilbert.com

There is a bunch of other possible problems with PRNGs in cryptography which will be looked upon in section 3.3.

3.3 Application in Cryptography

As already mentioned, the application of random data is manifold, even in the sector of IT alone. For this work, we will focus the discussion on cryptographically secure pseudo random number generators.

For security applications, unpredictability is of utter importance. Generating good random data is a vital part of many cryptographic operations. Random data has to be unpredictable to the attacker to ensure the security of a cryptographic system. But good random data is also very difficult to attain.

To get a sense of understanding, we first need to define the background of cryptography. Otherwise we won't be able to tell the difference between good random data and weak random data in the context of cryptography.

3.3.1 The Context of Cryptography

Cryptography initially started out as the art of encryption. Today, cryptography has become much more, covering authentication, digital signatures, safe communication, and many other security aspects.

But cryptography by itself is pretty useless, it has to be part of a system. Like a lock in the physical world is pretty useless by itself, but when it becomes part of a bigger system like a door that belongs to a building it suddenly becomes an important part in keeping thieves and burglars out.

Cryptography is like the lock in the example: It has to provide access to some people but not to others. As can be imagined, this is very tricky. Many parts of security systems just keep everybody out. Cryptography has to distinguish between legal and illegal access and let the right people in and escpecially keep the wrong out. This is much more difficult than keeping everybody out.

Since detecting an attacker who broke the cryptography is very unlikely, as he appears like a valid person with the key to the lock, it is much more important to get it right. When an attacker manages to break a security systems by literally breaking through the wall, this will at least leave obvious traces of the break-in [13].

The Weakest Link Property

Bruce Schneier in *Practical Cryptography* [13] vividly points out, there is an important rule to security systems:

> "**A security system is only as strong as its weakest link.**
>
> *Look at it everyday, and try to understand the implications. The weakest link property is the main reason why security systems are so fiendishly hard to geht right*"

This is due to the adversarial setting, which will be elaborated in section 3.3.1. To make it short: The adversarial setting says an attacker is smart and will always strive to attack the weakest link of a security system. This leads to many implications.

First, every security system consists of a large number of different parts. Assuming the attacker is only going to attack the weakest link, it can be said that *"strengthening anything but the weakest link is useless."* [13]

While this is true in theory, it is not always the case as the attacker may not know what the weakest link is. Nevertheless, this thought should not be confused with gained security. How the attacker is going to attack the system solely depends on the individual attacker, of which the designer of the security systems knows nothing and should assume nothing.

Hence it is advisable to strengthen every link, but especially the weakest. To do this, it is necessary to know what the weakest link is in the first place. The answer to this requires a lot of analysis work and experience.

In summary it can be said that to break into a secured system an attacker has to find but only one weak link. In contrast, to make a system completely secure, every single link has to be strengthened. It's obvious that the attacker has the advantage here.

The Adversarial Setting

Security systems are unique in that they have an adversarial setting. Engineers in general face many problems when constructing complex systems that affect the design of the system, but those problems are predictable.

> *"[In security] our opponents are intelligent, clever, malicious, and devious; they'll do things nobody had ever thought of before. They don't play by the rules, and they are completely unpredictable."*
>
> Quoted from [13]

This is a very difficult setting. A security engineer has to work with an attacker in mind about whom he knows nothing. He does not know who she is, what she is capable of, what her aims are, when she is going to attack, what she is going to attack, how she is going to attack, or how resourceful she is. But one thing is sure: The attack will occur long after the security system has been designed, possibly using tools and utilising more powerful computers than have been available at the time the system was designed [13].

When designing a system for security, one has to always keep this harsh setting in mind.

3.3.2 Considerations for Cryptographically Secure PRNGs

A Pseudo Random Number Generator is its own kind of cryptographic primitive, which has to be understood and analysed for security as all other primitives of cryptography. A PRNG *can* be the single point of failure for a cryptographic system, making the careful selection of good algorithms and protocols irrelevant. Also, many attacks are made easier than they need to be by the inappropriate selection of a badly designed PRNG [20].

All this is reason enough to take a closer look at what special requirements there are for cryptographically strong PRNGs and how such PRNGs can be designed. To get a basic understanding, we will first take a look at what kind of attacks there are. Knowing the attacks is essential to understanding and fixing the weaknesses of PRNGs in the first place.

Attacks on PRNGs

Bruce Schneier et al described some general classes of attacks on PRNGs in their paper *Cryptanalytic Attacks on Pseudorandom Number Generators* [20] as follows:

- **Direct Cryptanalytic Attacks**. An attacker is able to directly distinguish between random outputs and PRNG outputs. This attack is applicable to most, but not all PRNGs. Cases where the PRNG output is never directly seen from the outside are not vulnerable to this attack.
- **Input-Based Attacks**. When an attacker is able to use knowledge of or control the input to an PRNG directly to cryptanalyse it, then this is called an input-based attack. This attack can be further divided into *known-input*, *relayed-input* and *chosen-input* attacks, but this is beyond the scope of this work (refer to the paper [20] for more details on those attacks).
- **State Compromise Extension Attacks**. A State Compromise Extension Attack builds upon the previous successful (partial) acquisition of the internal state S from a PRNG. Supposing the attacker learns – for whatever reason – the internal state S at any point in time, then an attacker may be able to recover unknown PRNG outputs – or distinguish them from random outputs – from before S was compromised or recover outputs from after the PRNG has collected sequences of input that the attacker cannot guess.

  This is most likely when a PRNG is started in an insecure state due to insufficient starting entropy. More details on methods to achieve this are given in the paper [20].

The paper continues with examples on abstract attacks against an idealised PRNG and demonstrates those attacks on four real-world PRNGs. Furthermore, guidelines on using vulnerable PRNGs are given that strive to offer ways of protecting the vulernable PRGNs against those attacks.

Requirements

Cryptographically secure pseudo random number generators (CSPRNG) have to fulfill all requirements typical PRNGs have to, but the reverse is not true. In addition to passing the regular statistical tests, a CSPRNG has to withstand serious attacks of intelligent adversaries. In order to do this, a CSPRNG should meet the following criteria [19]:

- A CSPRNG should satisfy the “next-bit test”, which is defined as follows: *“Given the first k bits of a random sequence, there is no polynomial-time algorithm that can predict the (k+1)th bit with probability of success better than 50%.”* [19]

- A CSPRNG should withstand a *State Compromise Extension Attack*: In the case that some or all of the internal State of an CSPRNG has been revealed or guessed correctly, it should be impossible to reconstruct previous random numbers. Also, it should be impossible to predict future conditions of the CSPRNG's state.

Most PRNGs do not satisfy this requirements and, as a result, are not suitable as CSPRNGs or for security-critical applications.

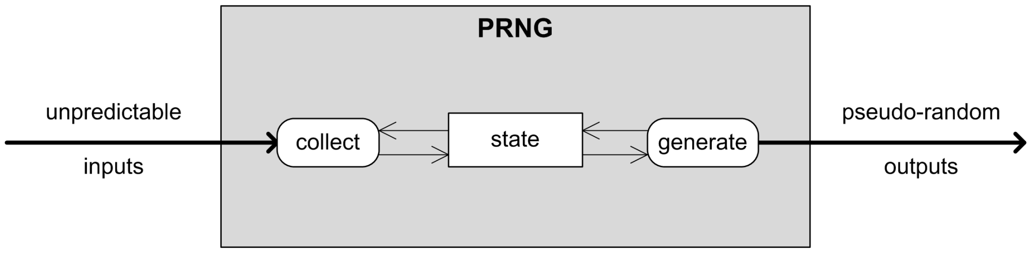


Figure 3.3: White box diagram of a typical PRNG showing its inner workings.

For example, if the internal state of a PRNG that does not satisfy the second criteria is ever revealed, it can never recover to a secure state. Though recovering from such an attack is difficult, it can be done. By pooling enough random data using some source of real random data at an unpredictable time, it is possible to "shake the attacker off" by mixing this source of additional entropy into the internal state of the PRNG. If the mixed in entropy is big enough, the attacker won't be able to predict the new internal state and the PRNG has effectively recovered to a secure state [13]. This is called "catastrophic reseeding".

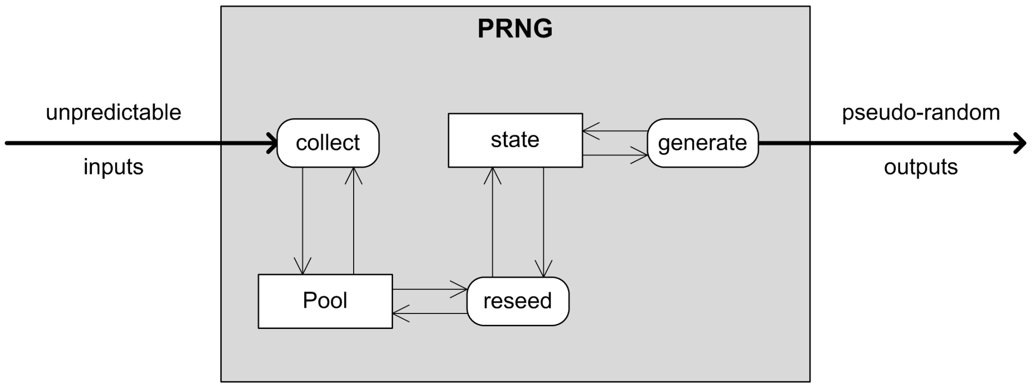


Figure 3.4: PRNG using "catastrophic reseeding" to update its internal state

This concept is illustrated in Figure 3.4, where the inner workings of a PRNG can be seen that utilisies a collection pool of entropy like in the idea described above.

Design Guidelines

The paper of Schneier et al [20] continues to provide guidelines on the design of cryptographically secure PRNGs:

1. **Base the PRNG on something strong**. The PRNG should use some other cryptographic primitive that is believed – or ideally be proven – to be strong.

2. **The internal PRNG state has to change over time**. This is to prevent a single compromised state from being unrecoverable.

3. **Do "catastrophic reseeding" of the PRNG**. This is done by separating the part that generates the random output from the entropy pool. Make sure to change the internal state only when the entropy in the pool is big enough, that is to say the collected entropy has to be great enough to resist an iterative guessing attack.

4. **Resist backtracking**. This means that an attacker that compromised the internal state at time $t + 1$ should not be able to guess the output at time t. Passing the internal state through a one-way function every few outputs limits the possible scope of backtracking.

5. **Resist Chosen-Input Attacks**. The input to an PRNG should be combined with its internal state in such a way that neither an attacker knowing the internal state but not the input nor an attacker knowing the input but not the internal state can guess the next (new) internal state.

6. **Recover from Compromises Quickly**. *"The PRNG should take advantage of every bit of entropy in the inputs it receives. An attacker wanting to learn the effect on the PRNG state of a sequence of inputs should have to guess the entire input sequence."* [20]

Designs of CSPRNGs can also be based on mathematical problems thought to be hard, like the *Blum-Micali algorithm* CSPRNG, which is based on the difficulty of computing discrete logarithms [19].

4 The Android Platform

The following chapter will describe the Android platform as a whole, offering an introduction to its fundamentals, as well as its basic and more advanced features. This also includes an overview over the Application framework, which forms an essential part of the Android platform.

4.1 Platform Fundamentals

As of late, the Android platform has become popular. But what exactly is Android and what does it consist of?

> *"The Android platform is an open-source software stack that includes the operating system, middleware, and key mobile applications along with a set of API libraries for writing mobile applications that can shape the look, feel, and function of mobile handsets."* [4]

Based on a heavily modified Linux kernel, Android offers an open platform with few barriers for the committed developer who wants to utilise the power of *mobile devices* that come packed with lots of gimmicks like cameras, GPS, touchscreens, various multimedia capabilities and different sensors today.

Initially, Android was developed by *Android Inc.*, a company that was later purchased by Google which in turn founded the *Open Handset Alliance*, a business alliance comprised of 65 companies as of July 2010, for developing open standards for mobile devices. This, of course, includes maintaining and continuing the development of the Android platform.

But what makes Android so special? Before Android there were and still are many other mobile device platforms which are largely closed environments built on highly fragmented, proprietary operating systems that require proprietary development tools. The mobile devices that run those proprietary operating systems often prioritise native applications over those written by third parties [4].

Android, in contrast, allows third parties to develop their applications with the same APIs and lets those applications execute with the same privileges as native applications. Furthermore, Android has an excellent documentation, a thriving developer community, and no development or distribution costs at all, which means anyone with a computer and some dedication can start developing their own apps.

4.1.1 Architecture

To get started we will first take a look at the architecture of Android. Figure 4.1 shows the overall system architecture of the Android software stack.

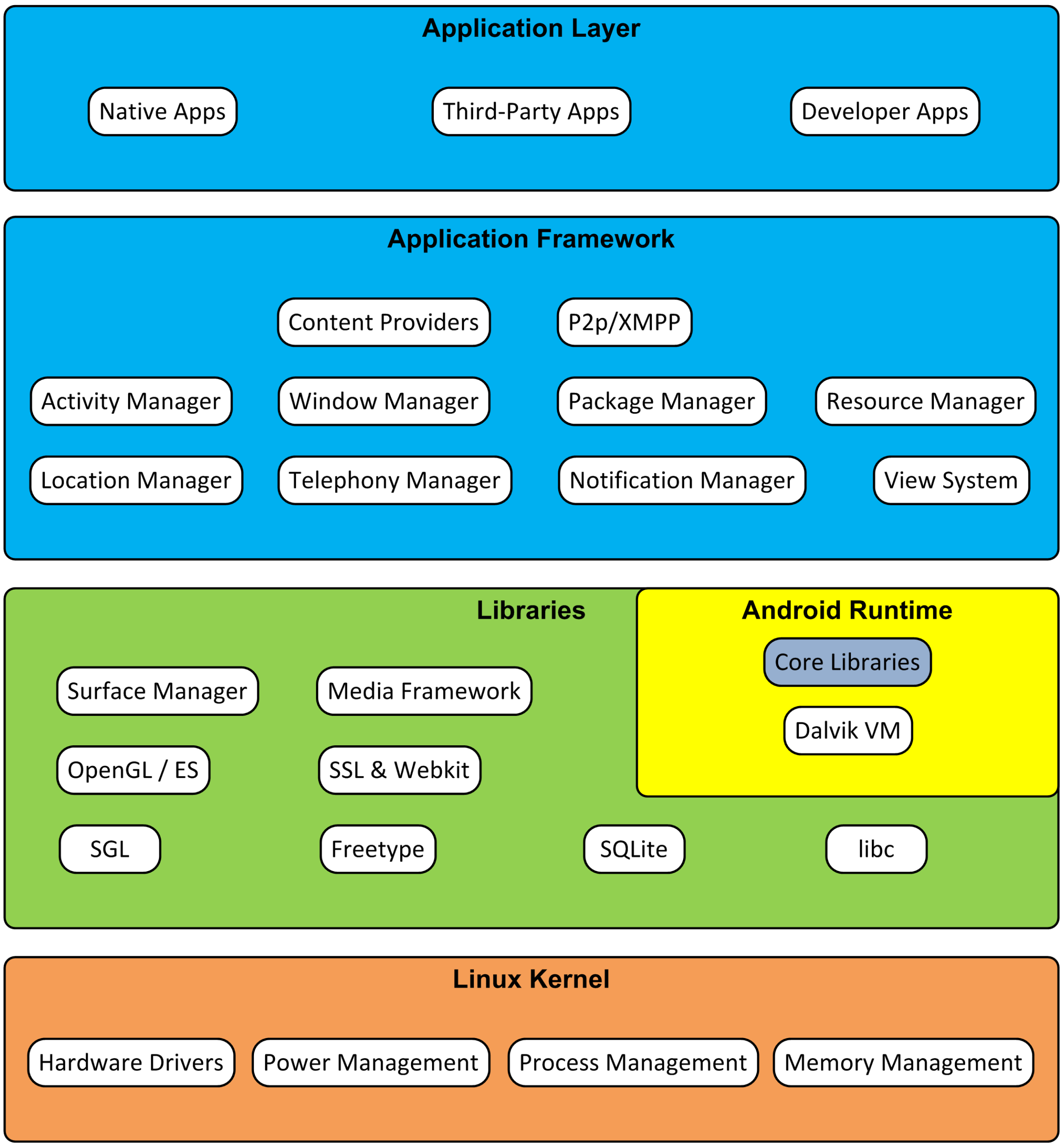


Figure 4.1: Overview over the Android software stack

At the bottom resides the modified linux Kernel which handles hardware drivers, process and memory management, security, network, power management and all other functions closely tied to the underlying hardware. This layer provides an abstraction between the hardware and the software layers on top of it.

On top of the kernel lies a set of various libraries written in C/C++. These libraries expose several capabilities that the developer can utilise. Some of the libraries are [3]:

- **System C Library** - a BSD-derived implementation of the standard C system library (libc), tuned for embedded Linux-based devices.
- **Media Libraries** - those libraries support the playback and recording of many popular audio, video and image formats like MPEG4, H.264, MP3, AAC, JPG, and PNG.
- **Surface Manager** - handles display management, compositing the 2D and 3D graphic layers.
- **SGL** - stands for ”Scalable Graphics Library” and provides the 2D graphics subsystem for Android. Works with the Surface Manager and Window Manager to implement the overall Android graphics pipeline.
- **3D Libraries** - an implementation of OpenGL ES 1.0 which is based on OpenGL 1.3. OpenGL ES 2.0 – based on OpenGL 2.0 – is supported since Android 2.0 and the NDK r3. The libraries use either hardware 3D acceleration (where available) or the included, highly optimized 3D software rasterizer.
- **LibWebCore** - a web browser engine based on Webkit that powers both the browser and any embeddable web view. Also supports SSL for Internet Security [4].
- **FreeType** - a bitmap and vector based font rendering library.
- **SQLite** - for lightweight relational database support available to all applications.

The Android Runtime resides in the same layer as the native libraries. It consists of core libraries and the Dalvik Virtual Machine, which is used to run every Application on Android. We will take a closer look at the Android Runtime later, as it is one of the most important subsystems.

The next layer is the Application Framework. It provides the classes and components used to create Android applications. This includes access to the device's hardware. The Application Framework was also used to implement the native Android apps, it therefore offers all the capabilities that are available on the mobile device in the form of framework APIs. We will further examine this layer later, too.

At long last and on top of all the layers life the Android applications themself, both native and third-party. The Application layer runs within the Android Runtime, utilising the components provided by the Application Framework. It is therefore guaranteed that third party applications will be run with the same priority and can be developed using the same possibilities as the native apps that come bundled with Android.

We will now take a closer look at the components that make Android unique.

4.1.2 Kernel

The Android kernel is based on the Linux kernel, to be more specific on Linux kernel 2.6. It was tweaked by Google outside the main Linux kernel tree. Though it is based on Linux, the fact that the Android Kernel neither does have a native X Window System implementation nor that it implements the full set of standard GNU libraries makes it difficult to port existing GNU/Linux applications or libraries to Android [5].

Table 4.1 provides an overview over all released Android versions and the Linux kernel version they are based on to the date of this writing, as well as the date they were released. The upcoming Gingerbread release is also included, although it's not yet clear what Linux kernel version it will be based on or when it will be released, so this information might change.

Android Version	**Kernel Version**	**Release date**
1.1	?	9.02.2009
1.5 (Cupcake)	2.6.27	30.04.2009
1.6 (Donut)	2.6.29	15.09.2009
2.0 (Eclair)	2.6.29	26.10.2009
2.1 (Eclair)	2.6.29	12.01.2010
2.2 (Froyo)	2.6.32	20.05.2010
Gingerbread	2.6.33 or 34	scheduled for Q4 2010

Table 4.1: Version and kernel history of Android

The modifications to the kernel for Android are maintained as a set of patches to the standard Linux kernel, which Google periodically upgrades to the latest released Linux kernel version. These patches have been described to fall into roughly four categories [6]:

- bugfixes (which are easy to resubmit into the main kernel development tree)
- generic standalone Android-related drivers
- generic Android-related drivers that add new infrastructure
- support for new SoCs, boards and devices

4.1.3 Android Runtime

The Android Runtime is a very defining component at the core of the Android software stack. It consists of core libraries and the Dalvik Virtual Machine, both of which will be described in the following sections. The Android Runtime is what powers applications and provides the basis for the Application Framework.

Core Libraries

Although Android apps are written in Java, the Dalvik VM ist not a regular Java Virtual Machine. The Android core libraries offer most of the functionality available in the Java programming language plus some Android-specific functionality. [3]

While the Java API is nearly fully supported by those core libraries, one should keep in mind that there may be some deviations from the standard Java Virtual Machine like an alternative implementation of functions with different performance characteristics.

The Dalvik Virtual Machine

Android runs every application in its own process with their own instance of the Dalvik Virtual Machine. Therefore, the Dalvik VM is a register-based virtual machine that has been optimised to efficiently run multiple instances of itself simultaneously. Register-based machines offer the merit of fewer instructions to load and manipulate data than stack machines, at the cost of longer instructions. This means less instructions for context switches between the insulated processes of the applications. Dalvik relies on the Linux kernel for process management, threading and low-level memory management.

Dalvik is not a Java Virtual Machine, as it does not operate on Java bytecode. It executes files in the Dalvik Executable (.dex) format. The .dex format is optimised to cause a minimal memory footprint. Java classes compiled by the Java compiler can be converted into the .dex format by the ”dx” tool that is included in the Android Development Tools (ADT) [3]. More detailed information on the .dex file format can be found at `dalvikvm.com` [22], which contains the results of reverse engineering on the .dex format done by *Michael Pavone*.

4.1.4 Application Framework

The Application Framework provides the foundation to application building. It consists of classes for application creation, hardware access, as well as user interface- and resource-management.

The application architecture was designed with reuse in mind: every application can publish its capabilities to let other applications take advantage of those (under some security rules enforcement by the framework, of course). This means every stock applications' capability may be replaced by a custom third-party app.

All applications are built using the following underlying set of components [3]:

- A set of **Views** that handle presentation, including lists, grids, buttons, text boxes, etc.
- Several **Content Providers** that allow apps to access data of other apps (like Contacts, Music, Images, etc.) for the purpose of data sharing
- A **Resource Manager** which offers easy access to complimentary resources like images, localised strings, or (screen-dependent) layouts

- A **Notification Manager** that allows applications to display notifications in the status bar in order to gain the attention of the user
- An **Activity Manager** that handles the application lifetime cycle, enabling a smooth user experience despite the somewhat limited resources on mobile devices

The aforementioned components will be described in greater detail in chapter 5.

4.1.5 Security Model

Android's security model is based on applications runnning in their own insulated processes with their own instance of the Dalvik VM and a permission system. The first is a sandbox environment that prevents apps from doing any harm to other applications or the system itself. The latter limits the capabilities of applications to rather basic features. When an application needs to use more enhanced (and possibly risky) features, it must state this in its manifest file (see section 5.4.1).

The stated permissions are shown to the user before the installation of the application so he can check if the application asks for anything suspect. This assumes the user is prudent and knowing. Unfortunately, the permissions are rather general and don't offer any insight into how the application actually uses the permissions.

Problems with the Permission System

It's not hard to conceive a SMS messaging app that seems genuine in asking for the `BROADCAST_SMS` permission, but instead of only sending messages to the recipient like the user expects the app to, it sends them to the creator of the app as well or is misused as a relay node for the creator's own messages. This is but a simple example how users can be tricked into thinking an app is safe when it isn't. This isn't far from reality, as security researcher Jon Oberheide already tried this on the Android Market which caused Google to act:

> *"Recently, we became aware of two free applications built by a security researcher for research purposes. These applications intentionally misrepresented their purpose in order to encourage user downloads, but they were not designed to be used maliciously, and did not have permission to access private data — or system resources beyond permission.INTERNET."* [25]

The current stance of Google has been to remove the app from the Android Market and even remove it remotely from all devices [25], which has caused concern in the community and revealed the fact that Google may even install apps without the user's permission, which unfolds a security risk in itself.

> *"(...) the remote install functionality is of more concern from a security perspective. (...) if an attacker is able to MITM [the] SSL GTalkService*

connection for a particular device, it may be possible to spoof these INSTALL_ASSET messages to deliver a malicious application payload. If Google's GTalkService servers were compromised, the malicious impact would obviously be a bit more widespread." [24]

4.2 Application Fundamentals

The following section is an abbreviated summary of the *Application Fundamentals* section on *Android developer* [3] which serves as a basic introduction to further the understanding of this work. For more details please review the original page on *Android developer.*

4.2.1 Context and Packaging

Android applications run in a rather isolated context. When an application is started it begins to live in its own world in many respects:

- by default, every application runs in its own Linux process
- each process has its own Dalvik VM, as mentioned in section 4.1.3
- each application gets a unique Linux user ID by default. Meaning the applications' files are only visible and accessible by this user

Though it is possible for two applications to share the same user ID, this is not the norm. To preserve system resources, applications with the same user ID may also run in the same process and share the same Dalvik VM.

Android Packages

Android applications written in the Java programming languages are compiled and then – along with all data and resources required by the application – bundled into an Android package (.apk) file by the *Android Asset Packaging Tool* (aapt). As an apk file, the application can be distributed and installed on different devices – it's the file that get's downloaded from the *Android Market* when you install a new application. One apk file contains exactly one application.

Installing apk files on a Device

There are three possible ways to install an Android Package on a device:

1. by publishing and downloading it from the *Android Market*
2. by using the Android Debug Bridge (adb)
3. by copying it on the SD card and using another app to install it (many file managers offer this feature)

Using the Android Debug Bridge will be described in section 5.3.

4.2.2 Components

Android is designed in a component-based way, which is a central feature: Any one application may use any other application or part of another application, provided the other application permits it. Instead of programming your own Contacts list view you can reuse the one provided by the Android framework. Or you could write your own Contacts list view and permit other applications to reuse your implementation (without having to link against your application or incorporating the code from your application).

Android applications do not have a static entry point like a `main()` function since this would not work well with the component-based approach. Instead, to start an application – or a component of an application – you have to send a message (called *Intent* in Android) to the framework, which will locate, instantiate and start the corresponding component based on some criteria that may be packaged into the message.

Android ensures that, whenever there is a request to start a specific component, that component is running (launching it if necessary) and an instance of that component is available (instantiating it if necessary) to be used by the requester.

Android has four types of basic components that will be described in the following sections.

Activities

An Activity represents a graphical user interface catered to allow the execution of a single action by the user. Although activities work together to present a seamless user experience, each Activity is independent of the others. An Activity must be derived from the base class `Activity` or a subclass of it.

An application may consist of multiple Activities or only a single one, depending on the complexity of the task and the design of the application. One of the Activities is typically marked as the entry Activity which will be presented when the application is launched. Navigating through Activities is done by starting another Activity from the current one by sending an *Intent* to the framework. This will build an Activity stack (called a *Task*, see section 4.2.4), pushing and popping Activities in the way the user navigates through the app.

Activities are given a window to draw in that fills the screen by default, but there may be smaller screens floating on top of others, in which case the Activity below will still be visible in the background. Pop-up dialogs are a good example of this design.

The visual content of the windows is provided by a hierarchy of *Views* – objects derived from the `View` base class that manage the rectangular space they're assigned to. Parent views hold and organise the layout of their children, passing any draw call to them as well. Views are at the heart of user interaction and Android already comes with several view components that can be used, like buttons, text fields, check boxes and more.

Services

A service is an application without a graphical user interface. It runs silently in the background either to do some work or idling while waiting for requests by other components. Every service needs to be inherited from the `Service` base class.

Services should be used to execute computationally intensive tasks to keep the user interface responsive or run tasks that don't need the user's attention like playing music (the GUI to control the music playback, on the other hand, should be a regular visible Activity).

Services can either be used by simply telling the framework to start them or by binding to them, which means getting a reference to use the services' interface. The framework will take care of launching the service if it isn't already running.

Services, like every other component, are started in the applications main thread by default. In order to avoid blocking the user interface, services should be started in another thread. See section 4.2.5 for more details on Threads and Processes.

Broadcast Receivers

A Broadcast Receiver – as the name already implies – is a component that listens to and receives broadcasted announcements (mostly by the system) and reacts to those according to its programming. Broadcast receivers must inherit from the `BroadcastReceiver` base class.

Applications can have any number of Broadcast Receivers which respond to any announcement the application considers important. A Broadcast Receiver does not display an user interface, but may use any other component to respond to an announcement, like starting an Activity or using the *Notification Manager*.

Broadcast Receivers are not meant to deal with tasks themself, but rather to listen to events and start an Activity or Service that will deal with the event in their stead. This practice is enforced by the Android runtime: Broadcast Receivers that don't return from their `onReceive()` method within 10 seconds will cause an *Activity Not Responding* warning dialog (see section 5.2.2 for more details).

Content Providers

A Content Provider makes an application's data or a certain set of the application's data available to other applications, thus allowing data to be shared amongst all applications. It must extend the `ContentProvider` base class.

Other applications use a `ContentResolver` object to access the data offered by any one of the Content Providers, utilising Android's interprocess communication mechanisms to do so. There are already some built-in Content Providers, for example to access the contacts or media data of the device.

4.2.3 Launching and Shutdown

Activities, Services and Broadcast Receivers are activated by an asynchronous message called *Intent*, which is an object of the `Intent` class. For Activities and Services, the Intent object names the action being requested and the data to be acted on by the receiver. For Broadcast Receivers, it only names the event being announced.

Launching an Activity

An Activity can be launched (or given something new to do) from another Activity using Intents in two ways:

- It can be simply started with the `Context.startActivity()` method, which immediately presents the new Activity on top of the current one
- It can be started for a result with `Activity.startActivityForResult()` which immediately presents the new Activity, like `startActivity()` does, but also gives feedback to the initiating Activity after the launched Activity has finished. A return value may be passed back in an Intent object which can be extracted in the automatically called `onActivityResult()` method of the initiating Activity.

Either way, the newly started Activity can fetch the Intent that launched it by calling its `getIntent()` method. This can be used to pass parameters to the new Activity. Also, when an Activity is launched by an Intent, its `onNewIntent()` method gets called by the framework. This hook allows to react to newly received Intents on already instantiated Activities.

Launching a Service

A Service is started by calling `Context.startService()`. Android will call the Service's `onStart()` method in correspondence to that, passing it the Intent object that started the Service.

To exert control over the Service, an Activity has to be bound to the Service with the `bindService()` method, which establishes an ongoing connection between the Activity and the Service in the form of a reference. The `onBind()` method is called on the bound Service in correspondence to this, with the Intent object as one of its parameters. `bindService()` may also start the Service if it's not already running.

Initiating a Broadcast

A broadcast can be initiated using an Intent object by calling one of the following asynchronous `Context` methods:

- `sendBroadcast()`, which simply sends the Intent to all interested Broadcast Receivers
- `sendOrderedBroadcast()` delivers the Intent one at a time to allow prioritisation

- `sendStickyBroadcast()` which performs a regular `sendBroadcast()` with the Intent “staying around” after the broadcast is complete, so that others can instantly retrieve that last Intent when calling `registerReceiver()` rather than waiting for the next broadcast to be initiated

Subsequently, Android will call the `onReceive()` method on all interested Broadcast Receivers, passing the broadcasted Intent alongside the call.

Shutting Down Components

Memory and process management is handled exclusively by the Android runtime. Because of this, Activities and Services may remain running for quite some time before they are shut down. Although Android provides methods to indicate that it should properly shut down an Activity or Service for when it fulfilled its task, this will give no guarantee of when or even if it will actually happen.

An Activity can be closed by calling the `finish()` method. Activities that were started for a result must return a resulting intent object by calling `setResult()` before closing it. Also, an Activity can close its *Sub-Activities* – denoting Activities it started using the `startActivityForResult()` – by calling `finishActivity()`.

Services, on the other hand, are not closed (since they are not visible) but stopped by calling its `stopSelf()` method or by calling `Context.stopService()`.

Closing does not mean shutting down the component, but it can lead to the runtime just doing so. Components might be shut down by the Android system when they are no longer needed or the need to free resources arises – which will cause a component to be shut down prematurely. This concept will be introduced in section 4.2.6.

The Manifest File

Up to now we just talked about starting Activities or Activities launching other Activities. But how is this done in the first place? How does Android know about all those Activities and applications? The manifest file is the answer to that.

The manifest file tells the Android system about the existence of the application and lets you define other metadata about the application, which includes an entry for every component the application contains, such as Activities, so the framework knows what components there are. The component entries are used to specify additional settings and properties for the components, which include if an Activity should be started when the application gets selected in the *Application Launcher* – the screen listing all applications that can be launched on the device.

This manifest file will also be automatically packaged into the Android Package (apk) file. It is a structured XML file that is always named `AndroidManifest.xml` and must be located at the root of the hierarchy.

We will talk about the manifest file in greater detail in chapter 5.

Intent Filters

Intent filters are part of the Android manifest file and define to what kind of Intents the components of an application listen to. This filters can be as specific as to say "*activate Activity/Service XYZ*" or as coarse as "*open a web view window*".

Intent filters allow you to specify certain criteria like a type of action, category or even data that the component can act on. Android will try to locate the best application to handle the Intent by comparing the `Intent` object with the Intent filters and selecting the best fit.

When there are multiple applications installed that fit the criteria Android automatically presents a list and lets the user select his preferred application to use. This even allows to replace built-in Android applications that come bundled with the system, like the eMail or phone app.

4.2.4 Activities and Tasks

As already mentioned, Activities can launch other Activities, even Activities in other applications can be started by simply putting together an Intent object with the required information and calling `startActivity()`. When you want to show a location, you can start a map screen Activity – borrowed from the Google Maps app, for example.

To the user, it will seem like the map screen belongs to your application even though it's defined in another application and runs in that other application's process. When the user is done with the map screen, he can hit the back button and your application will reappear.

Tasks

Android manages this user experience by keeping both Activities in the same *Task*. A Task is a group of related Activities that form what the user experiences as an application, even though some Activities might be borrowed from other applications.

A Task is arranged as a stack, with the Activity that started the task at the root. Typically, this is the Activity that will be started from the Application Launcher. The currently running Activitiy is always at the top of the stack. When a new Activitiy is started, it will be pushed onto the top of the stack. When the back button is pressed, the Activity at the top is popped and the one below will become the active Activity.

The stack contains objects of the Activities, so the task may contain multiple objects of the same Activity class. Also, Activities in the stack will never be rearranged, only pushed and popped.

A Task is an atomic group: All Activities in a task move together as a unit. The Task (including all Activities) can be brought to foreground or sent to background by the home key.

The default behaviour of a Task can be freely adjusted.

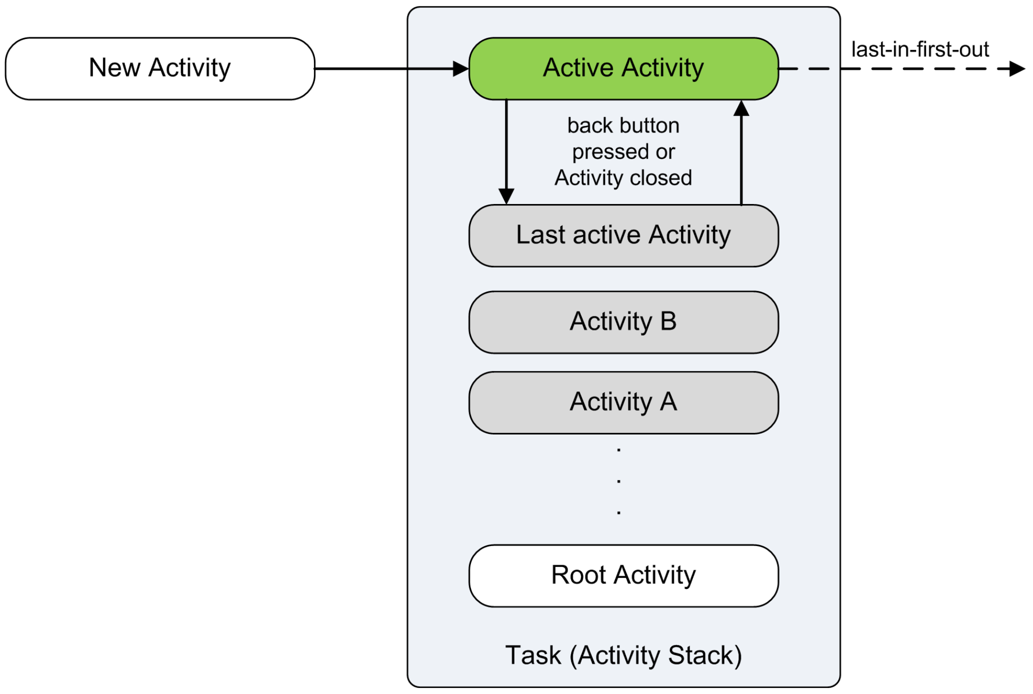


Figure 4.2: Illustration of an Android Task (or Activity Stack). Figure adapted from [4]

Task Affinity

By default, Activities of the same application have an affinity for each other: they belong to the same Task. The affinity can also be set individually so Activities defined in different applications can share an affinity with the current application or Activities defined in the current application can have a different affinity.

The affinity can be set in the Intent object that starts the Activity. By passing the `FLAG_ACTIVITY_NEW_TASK` the Activity will be started either in a new Task or – if there's already a Task with the same affinity – it will be launched into this Task. The affinity of a task is determined by the affinity of its root Activity.

When an Activity allows reparenting (the `allowTaskReparenting` attribute is set to `true`), it can move from the Task it started in to the Task it has an affinity for when that task comes into the foreground.

Launch Modes

There are four different launch modes available, that can be set in the `activity` entry of the manifest files. Those are:

```
standard
singleTop
singleTask
singleInstance
```

Unsurprisingly, `standard` is the default launch mode. The modes differ in four aspects:

- Which **task** will hold the Activity that responds to the Intent?
- Can there be multiple **instances** of the Activity?
- Can the instance have **other Activities** in its task?
- Will a **new instance** of the class be launched to handle a new Intent?

Table 4.2 provides an overview over the launch modes and the aspects of difference:

	Task	**Instances**	**Other activities**	**New instance**
standard	same[1]	multiple	yes	yes
singleTop	same[1]	multiple	yes	no[2]
singleTask	new	single	yes	-
singleInstance	new	single	no	-

Table 4.2: Launch modes and their differences

[1] Unless the `FLAG_ACTIVITY_NEW_TASK` flag is specified in the Intent object, which forces the Activity to be launched in a new task.
[2] If the `singleTop` Activity is at the top of the stack when a new Intent arrives, it will be reused. If there already is an instance of a `singleTop` Activity in the stack but it does not reside at the top, a new instance of the `singleTop` Activity will be created and pushed onto the stack to handle the new Intent.

4.2.5 Processes and Threads

When an application is run for the first time, Android starts a new Linux process with a single thread of execution for the application to be run in. All components of the application are run in this thread by default. It is possible for components to run in other processes or spawn additional threads.

Processes

Activities, Services, Broadcast Receivers and Content Providers can be set up to run in different processes in the manifest file. It is possible to set them up so each runs in their own process or that some of them share a process while others do not. They can even be set so that components of different applications run within the same process – provided they share the same linux user ID and are signed by the same authorities.

All components are instantiated and run in the main thread of a process. This means they should not perform computationally intensive tasks since this would cause the application and any other component of the application – including the user interface – to block. This violates one of the design guidelines for Android (*Designing for Responsiveness*, see 5.2.2) and should be avoided.

There is no telling when Android will shut down a process. It will happen when memory is low and required by another process that has a more immediate purpose to the user. In this case, Android decides which process to kill by weighting the relative importance of the processes to the user, meaning a process that hasn't been used for some time is likely to be shut down before any more recently used processes. The life cycle of processes will be elaborated in greater detail in section 4.2.6 *Lifecycle*.

Threads

Threads should be used for computationally intensive tasks that can be run in the background to keep the user interface and main thread of the application responsive. As a general rule, anything that may not be completed quickly should be run in a background thread.

Threads may be spawned and used just like in the Java programming language, using `Thread` objects. On top of that, Android provides a number of convenience classes for managing threads like the `AsyncTask` class.

Remote Procedure Calls

Although Android does have a lightweight mechanism for remote procedure calls, it is not of interest for most applications (which also applies to this work) and therefore will not be further discussed here.

4.2.6 Lifecycle

Every application component does have a lifecycle, beginning with object instantiation and ending when those instances are destroyed by the Garbage Collector. In between, they may sometimes be active or inactive.

But unlike in other systems, Android applications and components do have a limited control over their own lifetime. Instead, applications must listen and react to changes in their lifetime state, taking special care to be prepared for premature termination as Android manages its resources aggressively in order to retain responsiveness [4].

This sounds more drastic than it is in most cases. As a general rule of thumb, it can be said that:

> *"All Android applications will remain running and in memory until the system needs resources for other applications."*
>
> — Reto Meier, *Professional Android 2 Application Developtment* [4]

In the following sections, the lifecycles of Activities, Services, Broadcast Receivers and eventually processes will be looked upon and described.

Activity Lifecycle

An Activitiy may run through three basic states in its lifetime:

- **active or running** when it's in the foreground
- **paused** when it is partially covered by another Activity on top of it
- **stopped** when it was sent to the background or is not visible (when it is completely covered by an Activity on top of it)

Activities that are paused or stopped may be either asked to finish by the system (calling its `finish()` method) or the process might simply be killed. When the user returns to this Activity, it has to be restarted and restored to its previous state. It is therefore advisable to always save the state of an Activity when it loses focus.

There are several method hooks that get called by the system in the lifetime of an Activity, which may be overridden to react to certain lifetime state changes. These hooks are:

```
void onCreate(Bundle savedInstanceState)
void onStart()
void onRestart()
void onResume()
void onPause()
void onStop()
void onDestroy()
```

All Activities *must* implement the `onCreate()` method to do the initial setup when the Activity is instantiated. The other hooks are optional.

These seven hooks define the entire lifetime of an Activity, which consists of three nested lifecycle loops:

- The **entire lifetime** of an Activity starts with the call to `onCreate()` and ends with the call to `onDestroy()`.
- The **visible lifetime** takes place between the call to `onStart()` and `onStop()`. Between those two calls, the Activity is visible on the screen, though it may not have the focus (when it's covered by another Activity on top of it that does not consume the whole screen).
- The **foreground lifetime** between the calls to `onResume()`and `onPause()`. During this time the Activity has the focus and the user can interact with it. An Activity can frequently transition between `onPause()` and `onResume()`, which also happens when the device goes to sleep.

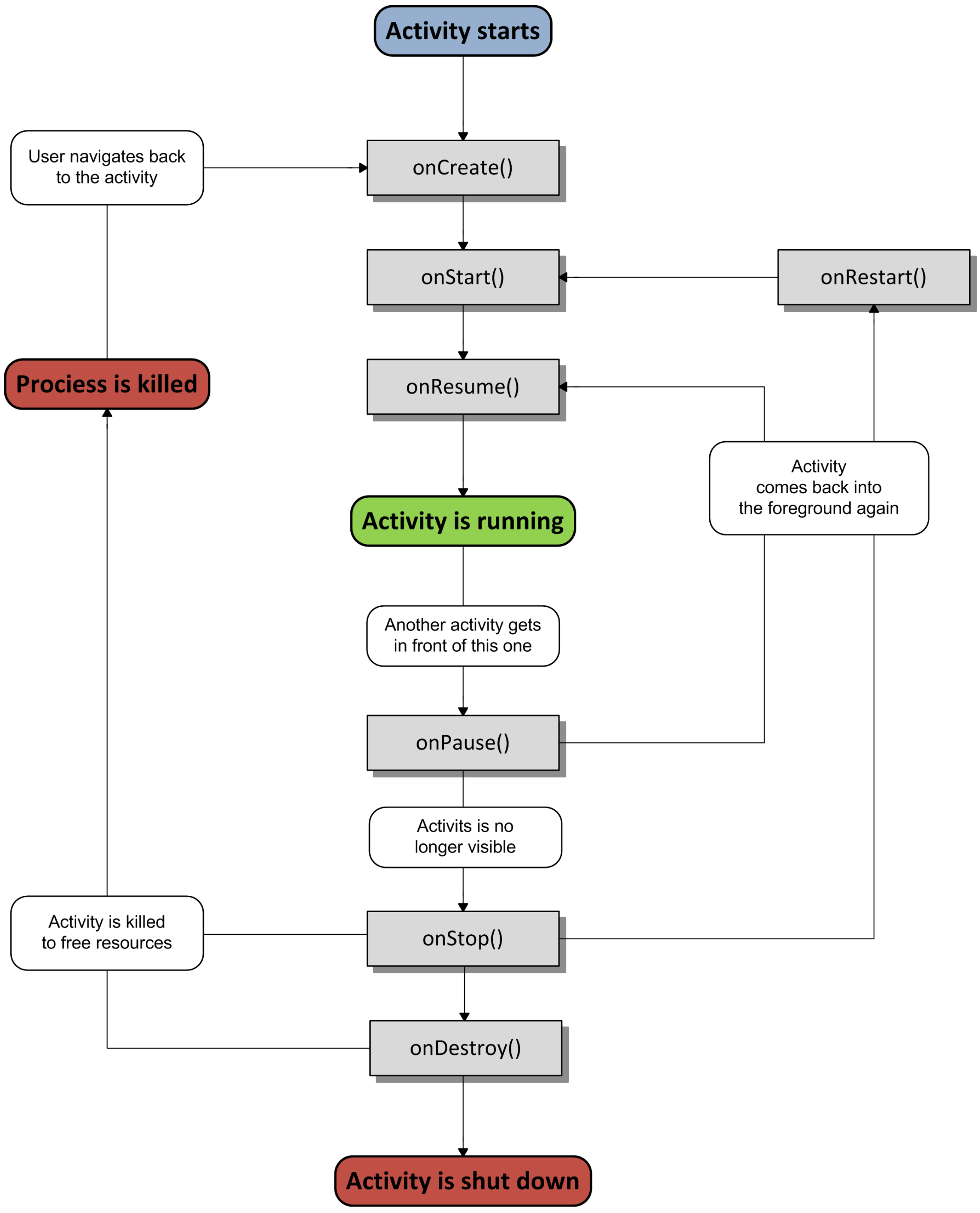

Activity starts
User navigates back to the activity
onCreate()
onStart()
onRestart()
Prociess is killed
onResume()
Activity comes back into the foreground again
Activity is running
Another activity gets in front of this one
onPause()
Activits is no longer visible
Activity is killed to free resources
onStop()
onDestroy()
Activity is shut down

Figure 4.3: Activity lifecycle from instantiation to its destruction

Figure 4.3 illustrates the lifecycle of an Activity in a more well-arranged way.

It is important to note that both `onStop()` and `onDestroy()` are **not guaranteed to be called** when an Activity is killed, you should always use `onPause()` to store data that should be persisted.

Service Lifecycle

A service can be used in two ways:

The first way is to start it by calling `Context.startService()` and stop it by calling `Context.stopService()`, in addition it can stop itself with the methods `Service.stopSelf()` or `Service.stopSelfResult()`. In this way, the Service will run in the background until it fulfills its purpose and then be closed by either the initiating component or the Service itself.

Or it can be bound to a component using `Context.bindService()`, which establishes a connection through which the component can control and access the Service to do its bidding. When the component is finished with the Service, it simply needs to call `Context.unbindService()` to close the connection. If the Service isn't running when `Context.bindService()` is called it will be started. Also, it's possible that several components connect to one and the same Service using this method.

At last, it is also possible to bind a component to an already running Service that was started using `Context.startService()`.

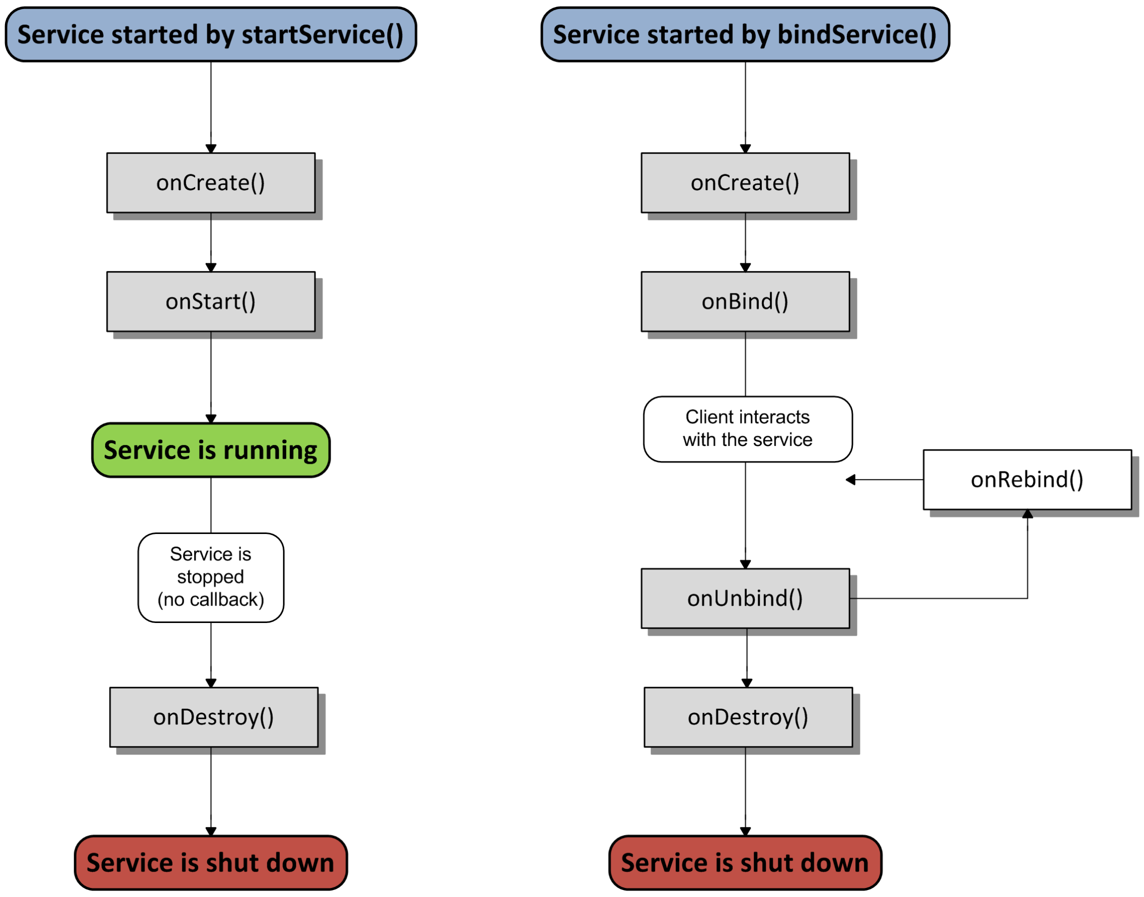


Figure 4.4: Service lifecycle from instantiation to its destruction

A Service has a couple of method hooks like Activities that lets the developer react to lifetime state changes. These methods are:

```
void onCreate()
void onStart(Intent intent)
void onDestroy()
```

As a Service is not visible to the user since it runs in the background, there are only two nested lifecycle loops:

- The **entire lifetime** of a Service starts with the call to `onCreate()` and ends with the call to `onDestroy()`, just like with Activities.
- The **active lifetime** starts with the call to `onStart()`. There is no corresponding `onStop()` for when the Service stops.

The `onCreate()` and `onDestroy()` methods are always called, whether the Service was started using `startService()` or `bindService()`, but `onStart()` is only called in the first case. For bindable Services, there is an additional interface which adds the following method hooks:

```
IBinder onBind(Intent intent)
boolean onUnbind(Intent intent)
void onRebind(Intent intent)
```

When they are called is pretty self-explanatory: `onBind()` is called when a component binds itself to a Service, `onUnbind()` when it unbinds and `onRebind()` is called when new clients have connected to a Service after it has been notified from within its `onUnbind()` method that all connections have been closed.

Figure 4.4 illustrates the lifecycle of a Service. Although there are two separate life cycles – one where it has not been bound and the other where the service has been bound to a component – it should be noted that it's potentially possible to bind to Services that have been started with `startService()` like in the first lifecycle.

Broadcast Receiver Lifecycle

Broadcast Receivers only have a single hook method

```
void onReceive(Context curContext, Intent broadcastMsg)
```

which is called when a broadcasted message has arrived. The Broadcast Receiver is only considered active while executing its `onReceive()` method to process the broadcast. When it's done, it is inactive again.

A process with an active Broadcast Receiver cannot be killed, but an inactive one can. When a computationally intensive task has to be executed in response to a received broadcast, it is advised to start a Service to perform the processing. The general advise of spawning a thread and returning from the `onReceive()` method may lead to unexpected behaviour, as the processing is still running when the Broadcast Receiver's process is considered inactive, which could lead to the problem of the process being killed before the processing is done.

Process Lifecycle

The general philosophy of the Android system is to keep a process running as long as possible, but eventually the need to free resources will arise when memory runs low. To determine which process to kill, Android maintains an importancy hierarchy of all processes, with the process having the lowest importance being killed first. When this does not free enough resources, the next lowest process will be killed continuing until enough resources have been freed.

The importance of a process is determined by the components it runs and their states. There are five levels of importancy in this hierarchy, which will be introduced in the following list, sorted by importancy starting with the highest:

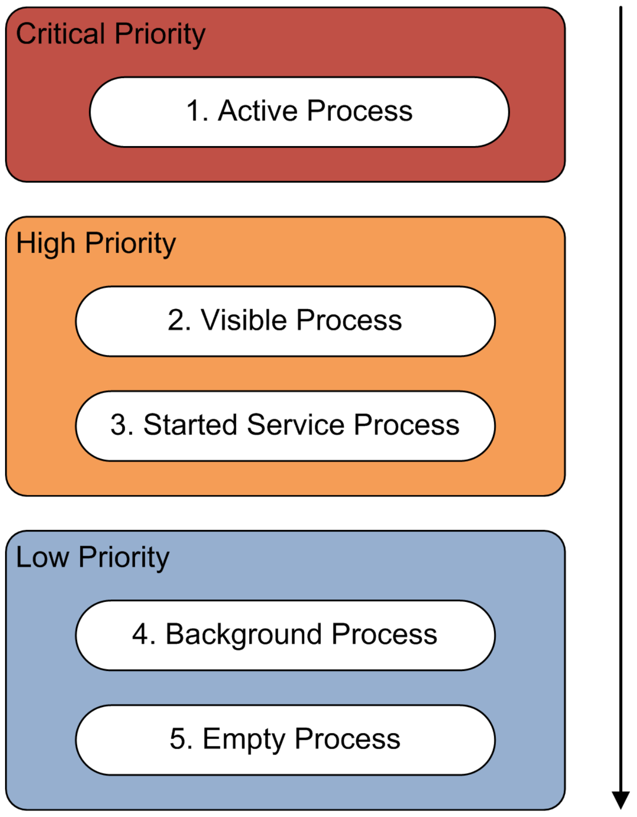


Figure 4.5: Overview over the process priorities. Figure adapted from [4]

1. A **active/foreground process** is one that the user is currently interacting with in some way. These are the processes Android tries to keep responsive by reclaiming resources. A process is considered in the foreground when any of the following criteria matches:

    - Activities the user is interacting with (the Activity's `onResume()` has been called)
    - Services that are bound to an Activity the user is interacting with

   - Services that are executing one of the following methods: `onCreate()`, `onStart()` or `onDestroy()`
   - Broadcast Receivers that are running their `onReceive()` method

   Only a few foreground processes will run at any given time and they will be killed only as a last resort when system resources are so low that they cannot continue to run smoothly or at all.

2. A **visible process** does not have a component in the foreground but is still visible to the user somehow. This can happen when:
   - It hosts an Activity that it is still visible (its `onPause()` method has been called) to the user. For example an Activity that has been partially covered by the foreground Activity.
   - It hosts a Service that is bound to a visible Activity

   Visibile processes are considered extremely important and only killed when it is required to keep the foreground processes running.

3. A **service process** is one that hosts a Service which has been started with `startService()` and does not fall in one of the two higher categories. Android tries to keep those alive since they typically serve some purpose the user cares about, like playing music in the background.

4. A **background process** is one that is currently not visible (its `onStop()` method has been called). To have a finer level of control, they are kept in a *least recently used* list which is checked additionally to ensure processes with the least possible impact are killed first. Background processes may be killed at any time to free up resources for higher prioritised processes.

5. An **empty process** contains no active components and is kept around just for the purpose of caching – to speed up starting times of applications.

Android ranks the processes based on the highest-ranked component it contains. So a process containing a Service and a visible Activity is ranked as visible process rather than a service process. Additionally, a processes' ranking may be increased by dependencies between processes: A process that serves another one can never be ranked lower than the process it is serving.

5 Developing for Android

In this chapter an introduction into Android app development is given, starting with how to install and set everything up. Following, some consideration is given to the development environment of mobile devices in regard to limitations by the hardware and general design guidelines. Afterwards, we will take a closer look at the Android Development Tools (in short ADT). Lastly, an extensive overview over the most important and most used Android components is given with examples on how to use them.

5.1 Getting Started

Android applications are written in Java and by the most part the standard Java API is supported by the Dalvik Virtual Machine. The Android SDK comes with all tools and APIs necessary to write Android apps. All that's left to do is to set it up properly before you can start delving into the depths of app development.

For developers that already have experience with Java, the transition is smooth and easy. Developing for Android is, by the most parts, like developing regular Java applications, though you have to keep a more limited hardware platform in mind.

If you don't have any experience in Java, but other object-oriented languages like C#, you shouldn't have any major difficulties translating this knowledge into Java and Android development.

5.1.1 What You Need

To get started with Android development, you just need the Android SDK, the Java Development Kit (JDK) and an Editor or – preferably – an Integrated Development Environment (IDE) for Java like Eclipse – to make development easier and more comfortable. When you're using Eclipse, you also need the Android Development Tools (ADT), a plug-in for Eclipse that adds support for Android development to the IDE.

The Android SDK, JDK, Eclipse and ADT are available for Windows, MacOS and Linux, due to Java's platform-independent nature. It makes no difference which platform is used for development – all tools run on every platform and since the Android apps are run in a virtual machine, there is no advantage from using any particular operating system [4].

The Java Development Kit

The JDK can be downloaded from Oracle (formerly Sun) at the following address:

`http://java.sun.com/javase/downloads/widget/jdk6.jsp`

For this work, the JDK 6 with Update 20 was used, but more recent versions should work as well.

The Android SDK

The latest Android SDK can be downloaded from the following address:

`http://developer.android.com/sdk/index.html`

For this work the Android SDK r06 was used.

Eclipse

The latest version of Eclipse can be downloaded from the following address:

`http://www.eclipse.org/downloads/`

For this work Eclipse Classic 3.5.2 was used in combination with ADT 0.9.7. ADT can be downloaded here:

`http://developer.android.com/sdk/eclipse-adt.html`

5.1.2 Installation and Setup

The following order of installation is recommended:

1. Java Development Kit
2. Eclipse
3. Android SDK
4. Android Development Tools

These steps will be described in more detail in the following sections.

Installing the JDK and Eclipse

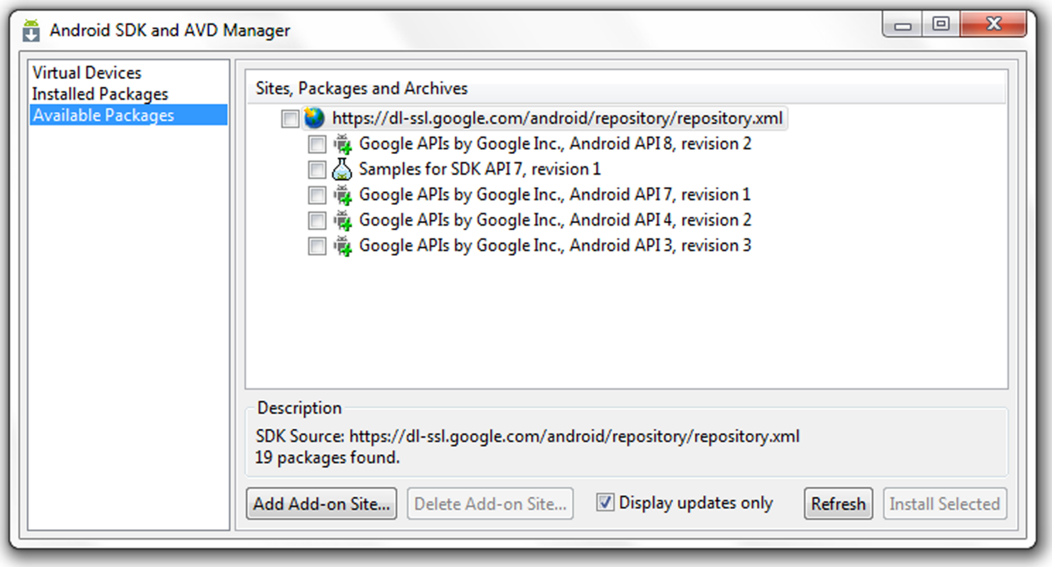
Installing the JDK is easy: Just download and run it. The installer will then guide you through the installation process.

Eclipse doesn't even have to be installed. Just download und unzip it into the folder you want it in. If you prefer an installer, there is an installer version available for Eclipse, too.

Installing the Android SDK

The Android SDK comes bundled in a ZIP package. To install the SDK, undertake the following steps:

1. Unzip the package into your desired output folder.
2. Start the *Android SDK and AVD Manager*.
   - On **Windows**, run the "`SDK Setup.exe`".
   - **MacOS** and **Linux** users need to execute the "`android`" executable in the `tools\` subfolder.
3. Click the *Available Packages* option on the left panel (see figure 5.1).
4. Select the SDK Platform versions you wish to install. It doesn't hurt to include the Documentation as well.
5. Click *Install Selected*.



https://dl-ssl.google.com/android/repository/repository.xml
Google APIs by Google Inc., Android API 8, revision 2
Samples for SDK API 7, revision 1
Google APIs by Google Inc., Android API 7, revision 1
Google APIs by Google Inc., Android API 4, revision 2
Google APIs by Google Inc., Android API 3, revision 3

Description
SDK Source: https://dl-ssl.google.com/android/repository/repository.xml
19 packages found.

Figure 5.1: The Android SDK and AVD Manager

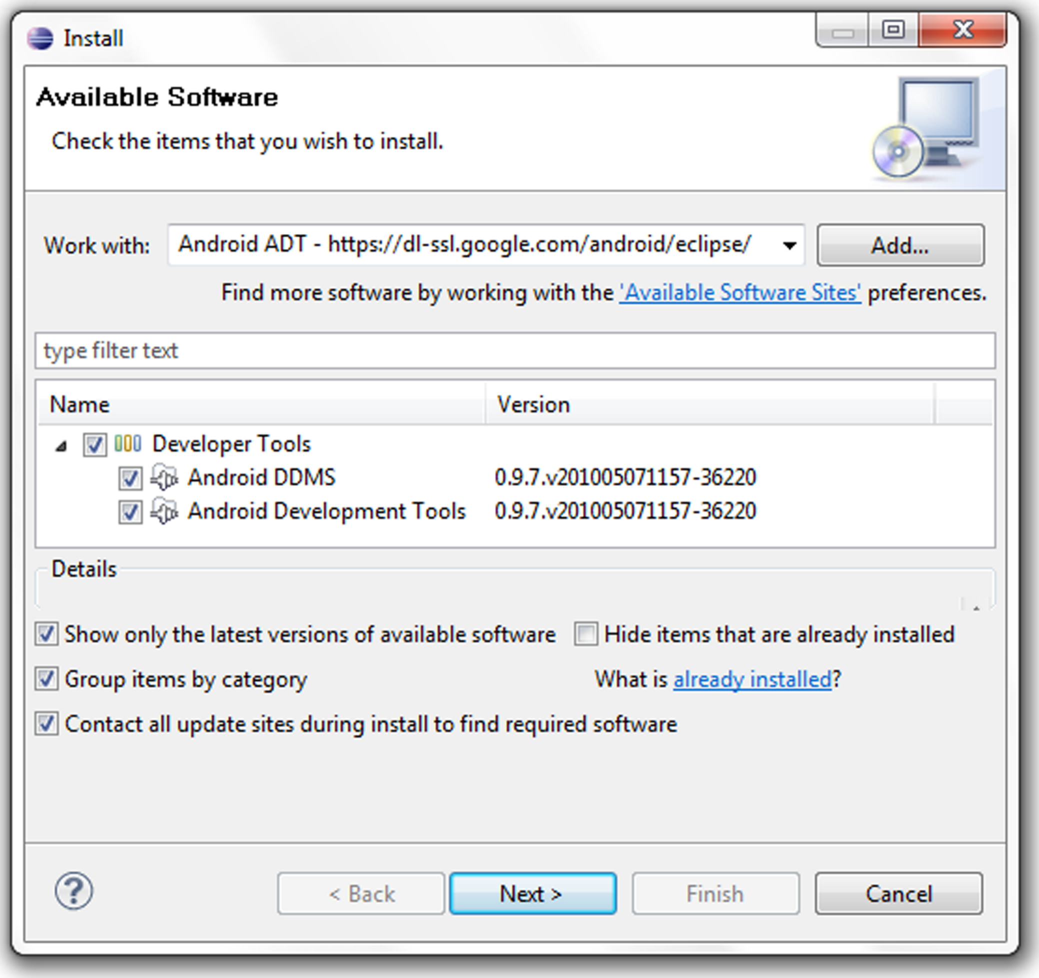
This will download and install the SDK into your SDK folder.

Installing the Android Development Tools

The Android Development Tools offer some compelling tools for Android development, integrating the development tools, an emulator and a .class-to-.dex converter [4] into the Eclipse IDE.

Install ADT following those steps:

1. In Eclipse, select `Help > Install New Software...`.
2. Click `Add...` on the right side to add the following site:
   `https://dl-ssl.google.com/android/eclipse/`



Install

Available Software

Check the items that you wish to install.

Work with: Android ADT - https://dl-ssl.google.com/android/eclipse/ [Add...]

Find more software by working with the 'Available Software Sites' preferences.

type filter text

Name	Version
☑ Developer Tools	
☑ Android DDMS	0.9.7.v201005071157-36220
☑ Android Development Tools	0.9.7.v201005071157-36220

Details

☑ Show only the latest versions of available software ☐ Hide items that are already installed

☑ Group items by category What is already installed?

☑ Contact all update sites during install to find required software

Figure 5.2: The window to install new plug-ins in Eclipse

3. Eclipse will now search the site for suitable Eclipse plug-ins.
   Check `Developer Tools` and then click `Next`.
4. An `Install Details` screen is shown. Ensure that both, *Android DDMS* and the *Android Development Tools*, are selected and click `Next` once again.
5. Read and accept the terms of the license agreement. Select `Finish` to start the installation process.
6. After the installation is complete, restart Eclipse.
7. Go to `Window > Preferences`.

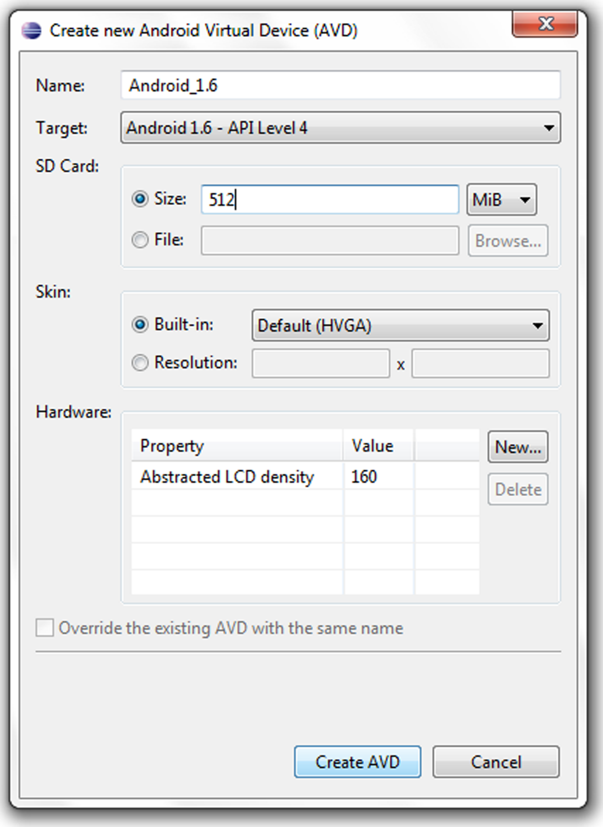
8. Select the `Android` entry on the left side and enter the location to the Android SDK. Confirm it by clicking `OK`. You have successfully installed and set-up ADT in Eclipse.

Setting up a Virtual Device

The last step before we can start developing an app is to set-up an *Android Virtual Device* (AVD in short). This is recommended for a seamless experience, since an emulator is needed to run, debug and test apps on. Though it is possible to use a real device for this, it's easier and more comfortable with an AVD.

Setting an AVD up involves the following steps:

1. Start the *Android SDK and AVD Manager*, either from Eclipse or like it was described in section 5.1.2 in step 2.
2. Select `Virtual Devices` in the left pane.
3. Click `New...` to open the creation dialog.


Create new Android Virtual Device (AVD)

Name: Android_1.6
Target: Android 1.6 - API Level 4
SD Card:
Size: 512 MiB
File:
Skin:
Built-in: Default (HVGA)
Resolution: x
Hardware:

Property	Value
Abstracted LCD density	160

Override the existing AVD with the same name

Figure 5.3: The Android Virtual Device creation dialog

4. Configure the virtual device to your liking. This involves:

- Entering a **name** for the device. This is for simple identification, so choose a meaningful name.
- Selecting a **target platform** for the AVD. This determines which Android version will be run on the emulator.
- Specifying the size of the emulated **SD card**. Note that this size is statically reserved for the AVD, which is stored on the same drive as your OS by default. This means when you specify 512 MB for the SD card, then the emulator file will take up 512 MB additional disk space. This can really add up when you have several AVDs, in case you're wondering where your disk space went like the author of this work did.
- Selecting or specifying a screen resolution, called **skin**. This is to allow the emulation of different devices. If unsure, select *Default (HVGA)*.
- Adding or removing **hardware properties** from a list of several available properties. By default, a pixel density of 160 is specified here.

5. Click `Create AVD` and it is done.

These are the steps necessary to set-up a basic AVD which should suffice for most applications. Delving into the more specific settings available is beyond the scope of this work.

5.1.3 Getting the first Application to Run

We will start by creating and running a simple Hello World application to get a basic understanding of the process of creating and running applications. Through the course of this chapter, I will assume you are working with Eclipse and ADT, therefore all step-by-step guides will be based on this assumption.

Although for the most part of this chapter the choice of editor/IDE will not matter, there are some sections (escpecially at the beginning where an introduction to the tools is given) which are only applicable to Eclipse with ADT.

Creating a new Android Project

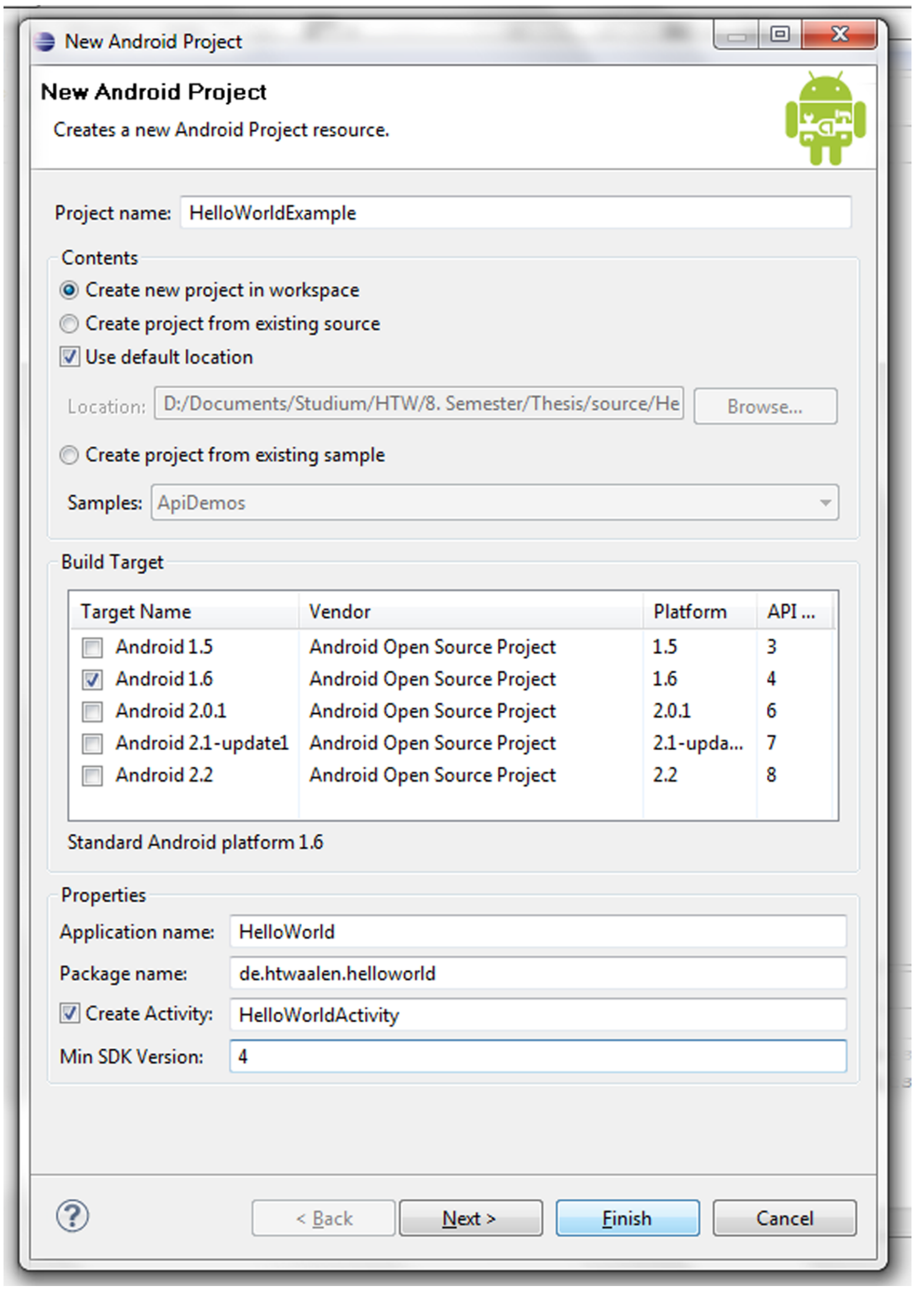
Creating a new Android Project can be done in three steps with the ADT plug-in in Eclipse.

1. Select `File > New > Android Project`. If Android Project is not listed there directly, select `File > New > Project...`, look for a folder called `Android`, select `Android Project` from there and click `Next`.

2. In the following screen, fill out all necessary information like project name, target platform, application name, package name and Activity name. See figure 5.4 as an example.

3. Click `Finish`.

That's it.


New Android Project

New Android Project

Creates a new Android Project resource.

Project name: HelloWorldExample

Contents

(●) Create new project in workspace

( ) Create project from existing source

[✓] Use default location

Location: D:/Documents/Studium/HTW/8. Semester/Thesis/source/He

( ) Create project from existing sample

Samples: ApiDemos

Build Target

	Target Name	Vendor	Platform	API ...
☐	Android 1.5	Android Open Source Project	1.5	3
✓	Android 1.6	Android Open Source Project	1.6	4
☐	Android 2.0.1	Android Open Source Project	2.0.1	6
☐	Android 2.1-update1	Android Open Source Project	2.1-upda...	7
☐	Android 2.2	Android Open Source Project	2.2	8

Standard Android platform 1.6

Properties

Application name: HelloWorld

Package name: de.htwaalen.helloworld

[✓] Create Activity: HelloWorldActivity

Min SDK Version: 4

Figure 5.4: Android project creation screen in Eclipse

Creating a Run Configuration

The next step is to create a run configuration so we can start-up an emulator to run the application on.

1. In our *HelloWorldExample* project, select `Run > Run configurations...`.
2. Right-click on the `Android Application` entry on the pane to the left, then select `New`.

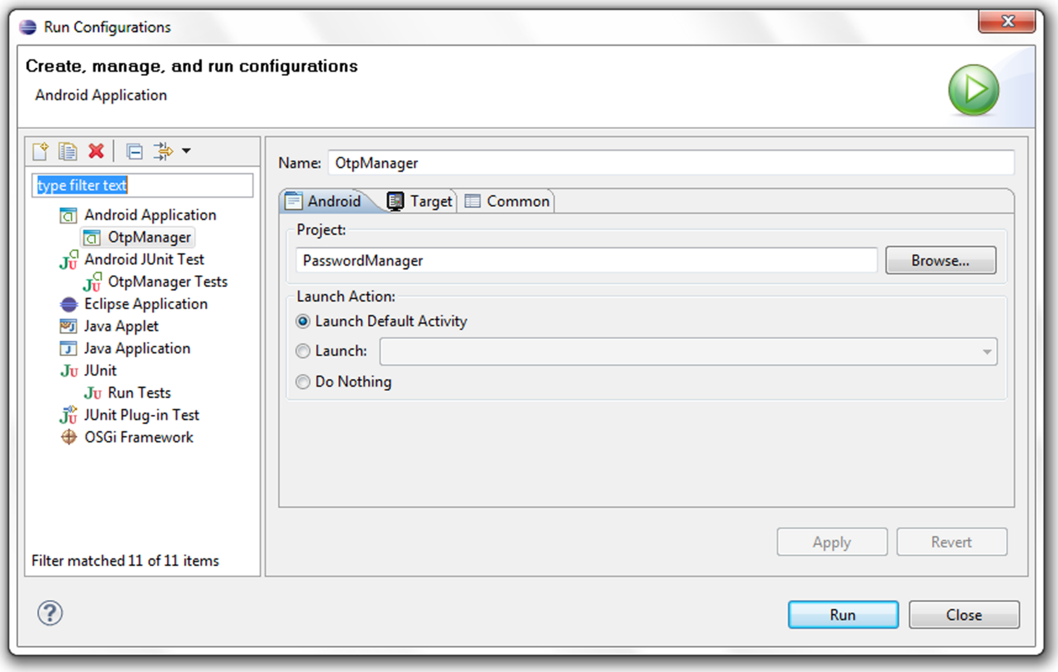
3. Enter a meaningful name for this run configuration, like "*Hello World*".

4. In the *Project* section, click `Browse` and select the recently created *Hello-WorldExample* project. You can also set the overall *network speed* and average *network latency* to simulate a real network, wipe the user data or disable the boot animation.

5. Select the `Target`-tab and select the AVD you prefer for deployment.

   Note: When you want to start your application on a real device, select `Manual` as *Deployment Target Selection Mode*. This will show a selection screen when running/debugging your application. When you have a real device – that has the USB Debugging option enabled – connected to your computer via USB it will be shown in the selection dialog to run/debug the application on.

   To debug a real device you need to set your application in the `ApplicationManifest.xml` as *debuggable*. See section 5.4.1 *The Application Manifest File* for more details on how to do this.



Run Configurations

Create, manage, and run configurations

Android Application

Name: OtpManager

Project: PasswordManager

Launch Action: [✓ Launch Default Activity] Launch: Do Nothing

Filter matched 11 of 11 items

Figure 5.5: Creating a run configuration to start an AVD

6. You may add the run configuration to your favourites menu. Switch to the `Common`-tab and check `Run` in the *Display in favorites menu* pane. This is optional, though.

7. Click `Apply` and then `Close`.

You can create a *debug configuration* much in the same way as described for the run configuration, with the only difference that you should add it to the `Debug` favourites menu, of course.

Running/Debugging the Application

After creating a run configuration (see section 5.1.3) you can easily run an application on an emulator by selecting the recently created run configuration in the favourite run menu. This will take care of...

- Ensuring the java classes have been compiled into `.class` files
- Ensuring the `.class` files have been converted into `.dex` (Dalvik Executable) files by the `dx` tool.
- Packaging the applications dex and all its resource files into an `.apk` (Android Package) file by the `aapt` tool.
- Starting the emulator (this can take a while, so don't worry when your screen looks like figure 5.6 for a long time).

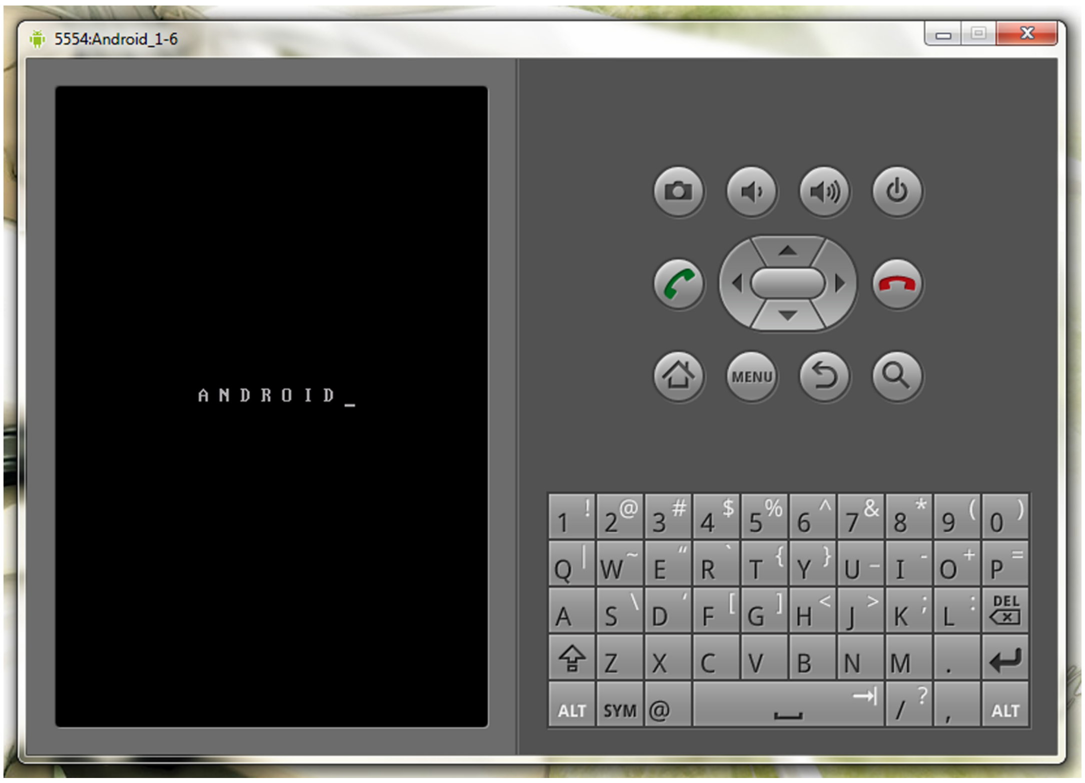


Figure 5.6: Starting screen of the AVD. Startup can take a while

- Uploading the `.apk` file to the emulator.

- Installing the application on the emulator
- Running the application on the emulator as soon as it has booted successfully.

The *HelloWorldExample* project already contains everything needed to compile and run it, thanks to the ADT plug-in. We will take a closer look at the building blocks of an application project in section 5.4.

This concludes the set-up section, which told you how to install and configure an environment that allows the development of Android applications.

5.2 Developing for Mobile Devices

Although by using Java the ambitious Android developer does not perceive much of the underlying hardware, it should still be kept in mind when developing apps. There are several factors to consider when developing for mobiles devices, which will be elaborated in the following sections.

5.2.1 Hardware Limitations and Considerations

Compared to Desktop Computers, mobile devices are lacking in several departments hardware-wise [4]:

- They have lower processing power
- More limited RAM
- Rather limited permanent storage capacity
- Small screens with rather low resolution
- Limited battery life

Some points may not be true for all devices, especially the newer ones, but when designing an application one should keep in mind that a lot of people still use older devices. In order to reach a large user base, the following guidelines should be followed.

Efficiency

Considering the above limitations, the first thing that comes to mind is being efficient. Although this can be applied to every software – and typically is – it should be emphasized when developing for mobile devices. Since users generally prefer small size and long battery life over raw processing power, manufacturers have made this a priority in their research and development of new devices. This means the processing power of mobile devices won't increase as much over time as one may expect coming from a desktop computer background [4].

Also, mobile devices cannot be upgraded like desktop computers. Optimising the code to run quickly and responsively on the current generation of mobile devices is therefore key to a good application.

On the other hand, it is important to apply optimisations judiciously. In most applications, there is only a small part of the application that takes up most of the processing time and that should be optimised. Or, as Donald E. Knuth put it: *"We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil."*

Limited Storage Capacity

Compared to desktop computers, mobile devices offer relatively sparse storage capacity. SD cards nowadays have sizes in the Gigabytes (ranging from 4 to 32 GB), whereas desktop computers offer terabytes of permanent storage space. On top of that, Android before Version 2.2 didn't allow the installation of apps on the external storage (SD card). On the older versions, apps can only be installed on the internal storage of a mobile device, which is much more limited than the external storage and cannot be interchanged to upgrade it [4].

This further tightens the limitation of storage capacity, as the internal storage space for typical Android devices range from 256 MB (e.g. on the Google G1/HTC Dream) to 512 MB (e.g. on the Google Nexus One) at the time of this writing, which can be practically halfed as the system partition resides also on the internal storage. As a result, it is advisable to keep the space requirements of the application low.

Design for Small Screens

The small screens of mobile devices challenge the developer to find a good compromise between offering as much information as possible and keeping the user interface well-arranged and easy to use. This is even more difficult for touch screen devices since touch screen input requires components like buttons or text fields to be big enough to be comfortably selected with fingers – which are considerably less delicate than a stylus.

For this reason optimising the user interface for small screens is important. But with the increasing appearance of Android tablets it may be well worth considering making the UI scalable to larger screens as well.

The User's Environment

For most people, mobile phones are first and foremost phones, secondly SMS and eMail messengers, thirdly a camera and fourthly an MP3 player. This should have an impact on your application's design. A good application should be polite and well-behaved, not forcing the user's attention, along other virtues [4]:

- **Well behaviour** includes suspending the application when it's not in the foreground. It makes no sense to update the UI when the application cannot be seen, it will only consume battery life. Android offers several hooks for suchs events, like `OnPause()` and `OnResume()`.

- An app should **switch quickly and seamlessly from the background to foreground**. There are many cases which may require the user to focus his attention on another application, like an incoming phone call. Making sure the application can be sent to and resumed from the background quickly enhances the user experience with the application. This also includes saving the state of the application in case Android kills it to free resources for other Activities in the meantime. When the user resumes to your application, he should find all in the state he left it. This can be achieved by using `OnSaveInstanceState()` and `OnRestoreInstanceState()`.

- **Being polite** means the application should never force the users attention or interrupt his focus on another Activity. The use of Notifications and Toasts (see section 5.5.4) allows to inform and remind the user that his attention is needed. Alerts like LED flashes, vibrating the phone or using sounds are also available, but should be used judiciously to avoid annoying the user.

- The application should **be and always remain responsive**. Responsiveness is one of the most important features an application for a mobile device should have. The annoyance of waiting on a frozen software is even more annoying on a mobile device, which most users expect to use casually on the way.

5.2.2 Android Design Guidelines

So, how do we apply all of this to Android? The previous section has given but only a few hints. In this section, we will take a closer look on how to design Android applications around performance, responsiveness and seamlessness.

Designing for Performance

The points listed here are an abridged version taken from the *Designing for Performance* section of the Android Developer Guide [3].

- **Avoid creating objects**. It is always advisable to avoid creating objects you don't have to, since object creation is never free. A typical pitfall is creating objects in loops, e.g. like `String` manipulation. Since concatenating a String effectively creates a new `String` object, a `StringBuilder` should be used instead.

- **Prefer static over virtual**. Calls to static methods are about 15-20% faster, so if you don't need to access object fields you should make the method static.

- **Avoid internal getters/setters**. Since virtual method calls are more expensive than instance field lookups, prefer using instance field lookups over getters and setters inside a class. Of course, this should not be used as an excuse to violate the OOP principle of *information hiding* by making instance fields public in the name of optimisation.

- **Use static final for constants**. Looking up a static final field is faster for various reasons. (see the source for more details)
- **Use foreach wherever possible**. The `foreach` loop is optimised and faster in almost any case. Also, it's shorter to write and therefore cleaner.
- **Avoid enums where you only need ints**. Enums take up more space and are slower than int constants (`static final`, of course), so this is a typical tradeoff between clean code and performance. The Android guidelines advise to use enums for APIs and try to avoid them where performance matters.
- **Use package scope with inner classes**. Inner classes can access private fields and methods of their outer classes. This is in principle illegal, so the VM creates static package accessor methods to circumvent this. This may have an invisible impact on the performace, as dicussed above with avoiding internal setters/getters. The solution is to declare fields and methods that are used within inner classes to have package access.
- **Use floating-point judiciously**. Floating-point is approximately 2x slower than integer.

Designing for Responsiveness

Applications can be fast, but still unresponsive. Hence responsiveness is a topic of its own. Android takes responsiveness really seriously. The *Activity Manager* and the *Window Manager* constantly monitor all applications for their responsiveness. If an application does not respond to an input event for 5 seconds or a broadcast receiver does not finish executing within 10 seconds, an *Application Not Responding* (ANR) dialog will be shown [3].

The ANR dialog is very intrusive. It is modal, steals focus, and won't go away until the user selects an option offered by the dialog (like *Force Close*) or the application starts responding again. A good application should never show an ANR [4].

By default, an application is run in a single main thread. Hence, tasks that are computationally expensive should always be started in a separate thread to prevent the main thread from blocking, which will cause an ANR after 5 seconds. Applications or Broadcast Receivers can also start a background service to execute expensive tasks.

Designing for Seamlessness

Seamlessness is a somewhat nebulous concept. The general goal of seamlessness is a consistent user experience which involves several qualities as listed below (taken from [3] and [4]):

- Speed and responsiveness should not degrade the longer a device is on
- Knowing the Android system silently kills processes, an app should always present a consistent user interface state
- Persist your application's data between sessions, so it can seamlessly return to its last state
- Present a consistent and intuitive user interface that blends in well with the system
- Don't interrupt the user
- Don't assume a touchscreen or keyboard
- Conserve battery life
- Don't overload a single activity screen

This list is not exhaustive, there can be much more to a seamless user experience. But it should give a good idea of what seamlessness is all about.

5.3 Android Development Tools

The Android SDK comes with many tools to support development. Many of those tools work in the background of the Eclipse ADT plug-in to make the developer's life easier (like the `dx` or `aapt` tool). A complete list along with descriptons of all tools can be found on the official homepage in the Guide section under *Tools* [3].

We will examine the more important tools that the developer will work with in the following sections.

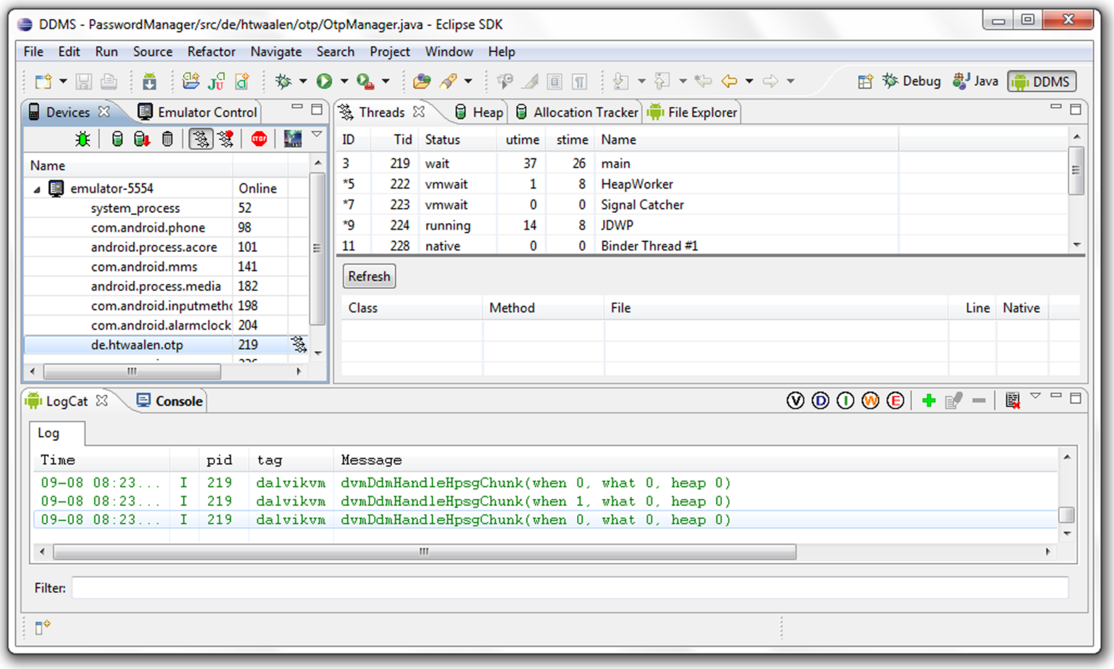
5.3.1 Android SDK and AVD Manager

As the name already implies, the main purpose of this tool is to manage the SDK components and Android Virtual Devices. The SDK Manager let's you download all available Android Platform APIs for app development, as well as additional APIs like the Google API that gives the developer access to Google libraries such as Google Maps. When a new revision of any of the APIs is released, you can use the SDK Manager to update your SDK.

Aside from that, you can create (as described in section 5.1.2) or delete AVDs.

5.3.2 Dalvik Debug Monitor Service

The Dalvik Debug Monitor Service (DDMS in short) is a really useful and handy tool. The DDMS allows you to take a look at what's happening "under the hood" of an Android device – be it real or emulated. It shows a list of all detected and connected Android devices, allows to exert control over emulator features like simulated telephony actions or GPS, lets you look at active processes, view the stack and heap, watch and pause active threads, view the log (which also reports access times in ms for Activities), take screenshots, and explore the file system of any connected device [4].


DDMS - PasswordManager/src/de/htwaalen/otp/OtpManager.java - Eclipse SDK
File Edit Run Source Refactor Navigate Search Project Window Help
Debug
Java
DDMS
Devices
Emulator Control
Threads
Heap
Allocation Tracker
File Explorer
Refresh
LogCat
Console
Log
Filter:

Name	
emulator-5554	Online
system_process	52
com.android.phone	98
android.process.acore	101
com.android.mms	141
android.process.media	182
com.android.inputmeth	198
com.android.alarmclock	204
de.htwaalen.otp	219

ID	Tid	Status	utime	stime	Name
3	219	wait	37	26	main
*5	222	vmwait	1	8	HeapWorker
*7	223	vmwait	0	0	Signal Catcher
*9	224	running	14	8	JDWP
11	228	native	0	0	Binder Thread #1

Class	Method	File	Line	Native

```
Time              pid  tag       Message
09-08 08:23...  I  219  dalvikvm  dvmDdmHandleHpsgChunk(when 0, what 0, heap 0)
09-08 08:23...  I  219  dalvikvm  dvmDdmHandleHpsgChunk(when 1, what 0, heap 0)
09-08 08:23...  I  219  dalvikvm  dvmDdmHandleHpsgChunk(when 0, what 0, heap 0)
```

Figure 5.7: The DDMS perspective in Eclipse

The DDMS is available as its own perspective in Eclipse and uses the Android Debug Bridge to communicate with the Android emulator or device.

5.3.3 Android Debug Bridge

The Android Debug Bridge (ADB in short) is a command line tool that lets you connect with an Android emulator or device. It's a client-server application that is made up of three components: a daemon running on the emulator, a service running on the developers computer, and client applications (like the DDMS) that communicate with the daemon through the service.

The ADB lets you push and pull files, install applications, and run shell commands on the device. This is especially useful as it allows you to query or modify the system directly at its base. ADB provides an `ash` shell (also known as *Almquist shell*) with the available shell commands being stored in `/system/bin/` on the device.

Useful ADB commands

Following is a list of useful ADB commands. For a complete list, please refer to the official documentation at [3].

- `devices`: Lists all connected devices.
- `install [-r] <path-to-apk>`: Installs the given Android Package on the device. When the `-r` parameter is specified, the app will be reinstalled.
- `push <local> <remote>`: Copies the specified file from the local machine to the target device.
- `pull <remote> <local>`: Copies the specified file from the target device to the local machine.
- `shell`: Starts a remote shell in the command line.
- `shell <command>`: Issues the shell command to the target device.

Connecting a Real Device

To connect a real device with the ADB, nothing special must be done except connecting the device via USB with the developers machine. Windows users need to install a special USB driver before doing so, which can be downloaded from the Android developers website [3].

After connecting the device to the computer, the command `adb devices` should print a list containing the connected device. When several devices are connected, you need to specify a target for the `adb` commands. This can be done using the `-s` parameter:

```
adb -s <serial-number> <command>
```

Whereas serial number is the name of the device as listed when the `devices` command is executed, like `emulator-5554` for an emulator.

5.4 Creating Applications

Applications for Android consist of components like Activities, Services, Content Providers or Broadcast Receivers which are loosely coupled through the use of intents and bound together by an application manifest that describes each component and how they interact. It also includes application metadata like hardware and platform requirements.

5.4.1 The Application Manifest File

Every Android application contains a manifest file, called `AndroidManifest.xml`, which resides in the project root folder. The manifest is an XML file that includes nodes describing the properties and structure of the application's components as well as nodes describing their interaction through *Intent Filters* and *Permissions* the application needs.

The manifest file of the *HelloWorldExample* project can be seen in listing 5.1. It consists of a root `manifest` node with a `package` attribute set to the project's Java package. The `xmlns:android` attribute includes several system attributes that are used within the file [4].

Listing 5.1: The manifest for the HelloWorldExample project

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest
    xmlns:android="http://schemas.android.com/apk/res/android"
    package="de.htwaalen.helloworld"
    android:versionCode="1"
    android:versionName="1.0">

    <application
            android:icon="@drawable/icon"
            android:label="@string/app_name">

        <activity android:name=".HelloWorldActivity"
                  android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN"
                    />
                <category android:name="android.intent.category.
                   LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

    <uses-sdk android:minSdkVersion="4" />

</manifest>
```

The `versionCode` and `versionName` attributes define the version of the app. The first is an integer used internally for version comparison, the second is the more pretty version name (like 1.0, 2.2, etc.) that's displayed to the user.

The first child node of the manifest is `application`, which defines properties for the application (like title and icon as seen in the listing) and defines the application's structure. The application node acts as a container for all application components it consists of. A manifest file can only contain one application node [4].

Application Structure

An `activity` node is required for every Activity of the application. Trying to start an Activity that is not defined here will result in a runtime exception being thrown. The `activity` node in listing 5.1 defines the Activity's Java class name (`android:name` attribute) and a title (`android:label` attribute), but other properties may be specified as well. There are similar tags for the other components: `service` for Services, `provider` for Content Providers and `receiver` for Broadcast Receivers.

A main launch Activity must be included in every application. This is specified by the `intent-filter` node inside the activity by the following two nodes:

```
<action android:name="android.intent.action.MAIN" />
<category android:name="android.intent.category.LAUNCHER" />
```

The first node defines this Activity as the entry point for the application and the second node adds this application to the Application Launcher so it can be started by the user. Applications that don't add themself to the Application Launcher can only be launched by other applications. It is possible to add multiple `action` tags to the Intent Filter, each defining an Intent the Activity could handle.

Application Interaction

The last node is `uses-sdk`, which determines the minimum Android SDK version required to run this application through the `android:minSdkVersion` attribute. A target or maximum SDK version could also be specified.

The interaction of this application with the system, components or other applications is defined by many other `uses` tags like [4]:

- `uses-configuration`: This tag is used to specify which input mechanisms are required by the application. Any combination of input mechanisms can be specified here.

  Example:

  ```
  <uses-configuration android:reqTouchScreen=["finger"]
                      android:reqNavigation=["trackball"]
                      android:reqHardKeyboard=["true"] />
  ```

- `uses-feature`: When the application requires a specific feature, this can be defined here. This will prevent your application from being installed on a device which does not have the required hardware features, like GPS or a camera. There is a wide variety of Android devices with differing hardware configuration.

  Example:

  ```
  <uses-feature android:name="android.hardware.camera" />
  ```

- `uses-permission`: Android's security model enforces that every application has to explicitly declare which features it needs to run properly. The required permissions will be displayed to the user before the installation of the application.

  Example:

```
<uses-permission android:name="android.permission
                               .ACCESS_LOCATION" />
```

There are more tags that can be added to the manifest file to further customise the application (like self-made permissions), but this is beyond the scope of this work. Refer to the official Android developers reference for more details [3].

5.4.2 Activities

Activities are the visible components of the application, the presentation layer. Every screen of an application is an Activity or, to be more precise, a class extending the `Activity` base class. Activities use Views to form a graphical user interface that is displayed to the user.

Listing 5.2 shows the `HelloWorldActivity`, which is the main Activity of the *HelloWorldExample* project.

Listing 5.2: The main activity of the HelloWorldExample project

```java
package de.htwaalen.helloworld;

import android.app.Activity;
import android.os.Bundle;

public class HelloWorldActivity extends Activity {

    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }

}
```

The `onCreate` method of the base class is overridden. This method is used to initialise the component when its first created and to set up the GUI of the Activity (by the call of `setContentView(R.layout.main)`). What's passed in as a parameter is a resource ID to a layout file. `R` is an automatically generated class that holds a reference ID to every resource in the `res\` folder of the project. The Eclipse ADT tools automatically generate and update this class. Section 5.4.3 will discuss resource management in greater detail.

There are many other virtual method hooks inherited from the `Activity` base class that can be overridden. The most prominent being the lifecycle methods as discussed in section 4.2.6, which can be overridden to react to lifecycle state changes of the Activity. Furthermore, there are additional methods to set up *Options* and *Context Menus*, as will be shown in section 5.5.3.

Subclasses of Activity

There are several convenience subclasses of Activity that aim to make some typical uses of Activities easier. Below is an incomplete list with some example subclasses:

- `AliasActivitiy`: Stub Activity that launches another Activity and then closes itself. Allows to implement an alias-like mechanism.
- `LauncherActivitiy`: Displays a list of all Activities that can be run for a given Intent (in other words, Activities that listen to the Intent or the category of Intent given in their Intent Filter).
- `ListActivitiy`: If your Activity is basically a list-like component, extend this class instead of Activity, as it already comes with a built-in `ListView` and exposes the event handlers for when the user selects an item.
- `PreferenceActivity`: When you want to create a settings/preference screen for your application, this is the right base class to inherit from. It already comes with built-in support for `SharedPreferences`, an interface for accessing and modifying preference data. Android offers a sophisticated framework for preference handling.

5.4.3 Resources

Android emphasises the concept of externalising resources such as images or strings from the application, so they can be changed independently. This also allows to easily provide alternative resources for different device configurations like different screen sizes or languages.

All resources are expected to be placed under the `res\` folder, which the ADT project creation wizard automatically creates along with the following folders:

- `drawable-hdpi`
- `drawable-mdpi`
- `drawable-ldpi`
- `layout`
- `values`

The drawable folders are for images and graphics, separated into three folders for high (hdpi), medium (mdpi) and low (ldpi) DPI displays. The layout folder contains XML files that define the layout (using Views) for the applications' Activities. The values folder is for simple values such as string, array or style values.

Resource Types

The following table gives an overview over the resource types of Android [4]:

Type	**Folder**	**Description**
Animations	`anim\`	Android supports two types of Animation: *Tweened animations* to rotate, move, stretch, and fade a `View` and frame-by-frame animations. Frame-by-frame animations are a sequence of drawables, hence the animation's XML file is saved in the drawables folder instead.
Drawables	`drawable\`	Drawable resources include bitmaps and *NinePatch* (stretchable PNG) images. Though GIF and JPEG files are also supported, the use of PNG is recommended.
Layouts	`layout\`	Contains XML files that enables the decoupling of the presentation layer by designing UI layouts in XML rather than in code. The layout files are "inflated" within an Activity using the `setContentView()` method. Using XML files for layouts is a best-practice.
Menus	`menu\`	Contains XML files that describe options and context menus content and layout. This further decouples the presentation of an app and is a best-practice.
Simple Values	`values\`	Supported simple values include strings, colors, dimensions, and string or integer arrays. All values are stored within XML files.
Styles & Themes	`values\`	Style resources externalise styling information about the app.

Table 5.1: List of all resource types supported by Android

Using Resources

Resources are accessed by using the static `R` class. This is a generated class that contains static subclasses for each resource type and is based upon the `res\` folder's content. The `R` class is automatically generated when the project is compiled or – in case of the ADT plug-in – automatically whenever you make a change to the `res\` folder.

Resource access is done by simply calling the static fields in the static class, e.g. like this:

```
R.string.app_name
R.drawable.app_icon
R.layout.main
...
```

This returns the ID of the resource, which can be used with the Android API. In case the resource is needed directly, there is a `Resources` class that can be used to attain it in the context of an Activity:

Listing 5.3: Attaining resources using the Resource class

```java
Resources res = getResources(); // called in an Activity

String appName = res.getString(R.string.app_name);
Drawable appIcon = res.getIcon(R.drawable.app_icon);
XmlResourceParser layout = res.getLayout(R.layout.main);
```

Resources for Different Hardware and Languages

Android comes with a dynamic resource selection mechanism that allows to easily adjust an application for different hardware devices or languages. Alternative resource values can be specified by creating a parallel directory structure inside the `res\` folder. By appending conditions separated by a hyphen `-` the alternative resources are used when those conditions are met.

To localise strings, it suffices to create a hierarchy like this:

```
res\
    values\
        strings.xml
    values-de\
        strings.xml
    values-fr\
        strings.xml
```

On german devices, the `strings.xml` located in `values-de\` will be used. On french devices, the `strings.xml` located in `values-fr\` will be used. On all other devices, the `strings.xml` located in `values\` will be used.

It is also possible to use more than one condition by simply appending them, separated by hyphens and given that they are not of the same category or contradicting each other. This system applies to all resource types and folders. There are conditions for [4]:

- Mobile Country Code and Mobile Network Code
- Language and Region
- Screen Size

- Screen Width/Length
- Screen Orientation
- Screen Pixel Density (as used in `drawable-hpdi` for example)
- Touchscreen Type
- Keyboard Availability
- Keyboard Input Type
- UI Navigation Type

As a result, it is possible to create completely different layouts for other screen sizes, use different graphics for different pixel densities, etc.

5.4.4 Intents

Intents are a central concept of Android and used as a message-passing mechanism to decouple the components and applications from each other. Intents can be used to:

- Declare the intention that an Activity or Service should be started to perform an action
- Explicitly start a particular Activity or Service
- Broadcast an event

Intents can be sent to any application installed on a device and are therefore capable of crossing an application's boundary. This allows to reuse components of other applications in your own application. Intents are also used to start and transition between Activites.

Activities can be started in two ways: Explicitly or implicitly.

Listing 5.4: Explicitly starting an Activity

```java
Intent intent = new Intent(this, CallActivity.class);
startActivity(intent);
```

In listing 5.4 the class of the Activity is explicitly given to the Intent to start it. Whereas in listing 5.5 only a type of action is given to the Intent (to show an Activity that lets the user make a phone call). The Android system determines which Activities are capable of handling this action and start the Activity or, in case there are several Activities able to handle the action, a selector is shown where the user can select his favored application.

Listing 5.5: Implicitly starting an Activity

```java
Intent intent = new Intent(Intent.ACTION_CALL);
startActivity(intent);
```

This system allows the user to use alternative applications for every proprietary application the device is delivered with.

Sub-Activities and Returning Results

An Activity started by `startActivity` is not related to the Activitiy that started it in any way and does not return anything when finished. Sub-Activities, on the other hand, provide feedback to their parent Activity when they finish and can return results. Sub-Activities are started with `startActivityForResult()` and need a request code, which is a simple integer value to identify the Activity that just returned in case there is more than one.

Listing 5.6: Starting a Sub-Activity that may return a result

```java
private static final int RETURN_CODE = 1;

Intent intent = new Intent(this, MyActivity.class);
startActivityForResult(intent, RETURN_CODE);
```

Sub-Activities are regular Activities started in another way. The hook method `onActivityResult()` will be called when a Sub-Activity finishes. Reading the results can then be done in the following manner:

Listing 5.7: Fetching a result from a finished Sub-Activity

```java
private static final int RETURN_CODE = 1;

@Override
public void onActivityResult(int requestCode,
                             int resultCode,
                             Intent data)
{
    super.onActivityResult(requestCode, resultCode, data);

    switch(requestCode) {
        case RETURN_CODE: {
            if(resultCode == Activity.RESULT_OK) {
                String s = data.getStringExtra("data");
                int i = data.getIntExtra("otherData", 0);
                // and so on...
            }
            break;
        }
    }

}
```

The results are returned within the `Intent` object and can be retrieved using the `get[Type]Extra`-methods. Most primitive types (like `boolean`, `byte`, `char`, ...) and array versions of those primitive types are supported. The name used to access these values can be chosen in the Sub-Activity, where the values are inserted into the Intent object like this:

Listing 5.8: How to return data from a Sub-Activity

```java
Intent data = new Intent();
data.putExtra("data", "String value");
data.putExtra("otherData", 42);

setResult(RESULT_OK, data);
finish();
```

The `setResult()` and `finish()` methods are inherited from `Activity`. `finish()` closes the Activity.

5.5 User Interface

The User Interface (UI) in Android is composed of three basic types, which namely are [4]:

- **Views**. This is the base class for all visual interface elements (sometimes called *controls*). All UI controls, including the layout classes, are derived from `View`.
- **View Groups**. `ViewGroup` is derived from `View` and can hold multiple View objects. This is used to create compound controls made up of interconnected child views. The layout managers, that help laying out the controls, are extended from this class.
- **Activities**. These represent the canvas views can be drawn upon. Android Activities are the equivalent of Forms. To display an UI, assign a layout (or view) to an Activity.

The complete Android UI is built based upon those three basic components. Android, of course, provides several, more sophisticated controls like buttons, text fields, checkboxes, etc. ready to be used. It is also possible to modify or create new controls by extending those basic classes.

5.5.1 Layouts

Layout managers, or most often just called *layouts*, are extended from the `ViewGroup` class. View groups can be used in two ways: to create a compound control or to manage child views, the latter is the case for layouts. Layouts can be nested in other layouts, thus allowing arbitrarily deep and complex interfaces, though it is recommended to keep the nesting level as shallow as possible, not only to reduce complexity but also for performance reasons.

There are several types of layout managers, which handle the layout differently, like `LinearLayout`, which aligns each child view either in a vertical or horizontal line, or `RelativeLayout`, which lets you define the position of each view element relative to the others. A complete list of available layouts can be found on the official Android developers website [3].

It is recommended to define layouts in XML.

Listing 5.9: Structure of a simple layout file

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/
    android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">

    <TextView
        android:id="@+id/someLabel"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/some_text"
    />
    <EditText
        android:id="@+id/someTextField"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
    />

</LinearLayout>
```

A simple example is given in listing 5.9, which defines a `LinearLayout` containing two controls: A simple label and a text field, that allows the user to input text. The label is defined to take up only as much space as needed (both vertically and horizontally by declaring `wrap_content` on both properties), whereas the text field should fill up the remaining horizontal space (`fill_parent`).

Both controls can be accessed in code using the IDs defined in the first property of each control. The `@+id/` prefix merely indicates that the following sequence defines a new ID. The label references a string resource called `some_text` defined in `strings.xml` indicated by the line `android:text="@string/some_text"`.

5.5.2 Accessing Views

As can be seen in the `HelloWorldActivity`, layouts are layed out (or "inflated", as Android calls it) by calling the `setContentView()` method of an Activity, passing in the resource ID of a layout. To access a view element defined in the layout file, take a look at the following code snipped:

Listing 5.10: How to access a view element defined in a layout file

```
private TextView myTextView;

@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);  // inflate layout

    // Get reference to view element defined in layout
    myTextView = (TextView)findViewById(R.id.myTextView);
}
```

It is also possible to create and inflate layouts in code, though this is discouraged as externalising the layout offers more flexibility and independence.

Listing 5.11: Creating a view in code

```
private TextView myTextView;

@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    myTextView = new TextView(this);
    setContentView(TextView);

    myTextView.setText("Hello World!");
}
```

5.5.3 Menus

Android supports two types of menus: An options menu that's displayed when the user presses the "menu" button available on most devices, and a context menu that's displayed when the user holds his selection on certain elements, like list elements in a `ListView`.

Both types of menus can and should be defined in XML files. While it is possible to create them in code, just like layouts, defining them in XML files is the recommended practice.

Following is a sample of a menu XML file that can be either registered as an options menu or a context menu, the XML looks the same for both.

Listing 5.12: A simple menu defined in XML

```
<?xml version="1.0" encoding="utf-8"?>
<menu
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:name="@string/menu_title"
>
    <item
        android:id="@+id/firstItem"
        android:title="@string/first_item"
        android:icon="@drawable/ic_menu_add">
    </item>
    <item
        android:id="@+id/secondItem"
        android:title="@string/second_item"
        android:icon="@drawable/ic_menu_second"
    >
```

```
        <!-- Submenu -->
        <menu>
            <item
                android:id="@+id/firstSubItem"
                android:title="@string/first_sub_item">
            </item>
            <item
                android:id="@+id/secondSubItem"
                android:title="@string/second_sub_item">
            </item>
        </menu>
    </item>
</menu>
```

As can be seen, it's also possible to nest a menu within another to create a submenu. The submenu will be displayed after the item enclosing the submenu has been selected. You can only go one level deep with submenus.

Creating Menus

Creating an option or context menu can be done by overriding the appropriate method of the `Activity` base class: `onCreateOptionsMenu` for an option menu and `onCreateContextMenu` for a context menu. The implementation details are the same for both, though:

Listing 5.13: Creating an options menu

```java
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    super.onCreateOptionsMenu(menu);

    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.options, menu);

    return true;
}
```

Handling Menu Selections

Handling menu item selections is done the same way: Override the corresponding method and implement the details like below:

Listing 5.14: Handling an options menu item selection

```java
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    super.onOptionsItemSelected(item);

    switch (item.getItemId()) {
        case R.id.firstItem: {
            // Do something...
            return true;
        }
        case R.id.secondItem: {
            // Do something else...
            return true;
        }
    }

    return false;
}
```

There is a corresponding `onContextItemSelected()` method for context menus.

5.5.4 Notifications and Toasts

Notifications and Toasts can be utilised to inform the user of something in a subtle way. This is the preferred way for most events that require the user's attention.

Toasts

Toasts are small dialog boxes that remain visible for only a few seconds before fading out. They neither interrupt the user nor steal focus, so they are perfect to inform the user about events without forcing anything.

Toasts can easily be created and displayed as shown here:

Listing 5.15: Creating and displaying a Toast

```
Context context = getApplicationContext();
String message = "Live long and prosper!";
int duration = Toast.LENGTH_LONG;

Toast toast = Toast.makeText(context, message, duration);
toast.show();
```

Listing 5.15 displays a Toast for about 5 seconds before it's fading out. By default, toasts are displayed at the bottom of the screen, but this can be customised by calling `setGravity()`.

Notifications

Notifications, in contrast, are by far more powerful than Toasts. Only a short introduction of what notifications are capable of is given here. Notifications are handled by the `NotificationManager` and have the ability to:

- Add new entries to the status bar
- Display additional information and launch an Intent on the extended status bar window
- Flash the lights/LEDs of the device
- Vibrate the device

- Play sound alerts like ringtones or other media

Using notifications is the preferred way for invisible components like Services, Broadcast Receivers or inactive Activities to inform the user about something that may require the user's attention [4].

Creating and displaying notifications is more complicated than displaying Toasts, but notifications are an important concept of Android and were introduced here because of this fact. A detailed explanation of their use, however, is beyond the scope of this work, but can be looked up on the Android developers website [3].

6 OTP Manager App

After the basic and advanced foundations have been covered, we can finally take a look at the Android application that was developed over the course of this work.

The requirements and foregoing considerations will first be elicited and the underlying use cases will be laid out. Afterwards the design, including the architecture of the application, will be shown and elaborated in detail. Then, the implementation details of key features will be presented in their own section. Lastly, a section on usability considerations concludes this chapter.

6.1 Requirements and Foregoing Considerations

The goal of this work was twofold:

First, an Android application with field utility should be developed. Utilising the power and ubiquity of smartphones, the idea of a smart app to help managing and working with One-Time Passwords was devised. The OTPW package was chosen as backing OTP system.

The second goal was to explore the Android platform itself and document the results. At the date of this writing, it is clear that Android will play a major role in the future of the smartphone market and in many people's lives. The release of Android heralded a new era of open development possibilities on mobile devices. This work aims to contribute to this paradigm shift away from restrictive, proprietary mobile devices to a free, open environment that encourages diversity and choice.

6.1.1 Requirements

To get a basic idea of what I was up to, I fleshed out a list of requirements first:

- Management of One-Time Password lists
  - Organising OTP lists in browsable collections
  - Search function to assist user in finding what he is looking for
  - Filter to assist user in browsing the lists
  - Sorting options to assist user in browsing the lists
  - Marking used OTPs
- Creating new One-Time Password lists that are compatible with OTPW
- Importing existing OTPW password lists via file browser

This list combines the basic features with additional convenience features any good OTP management application should feature in my opinion. The main feature is, of course, the management aspect. The application should be easy to use and empower the user with a convenient, handy tool to work with One-Time Passwords.

The primary use case in figure 6.1 illustrates those requirements in a more clearly arranged way.

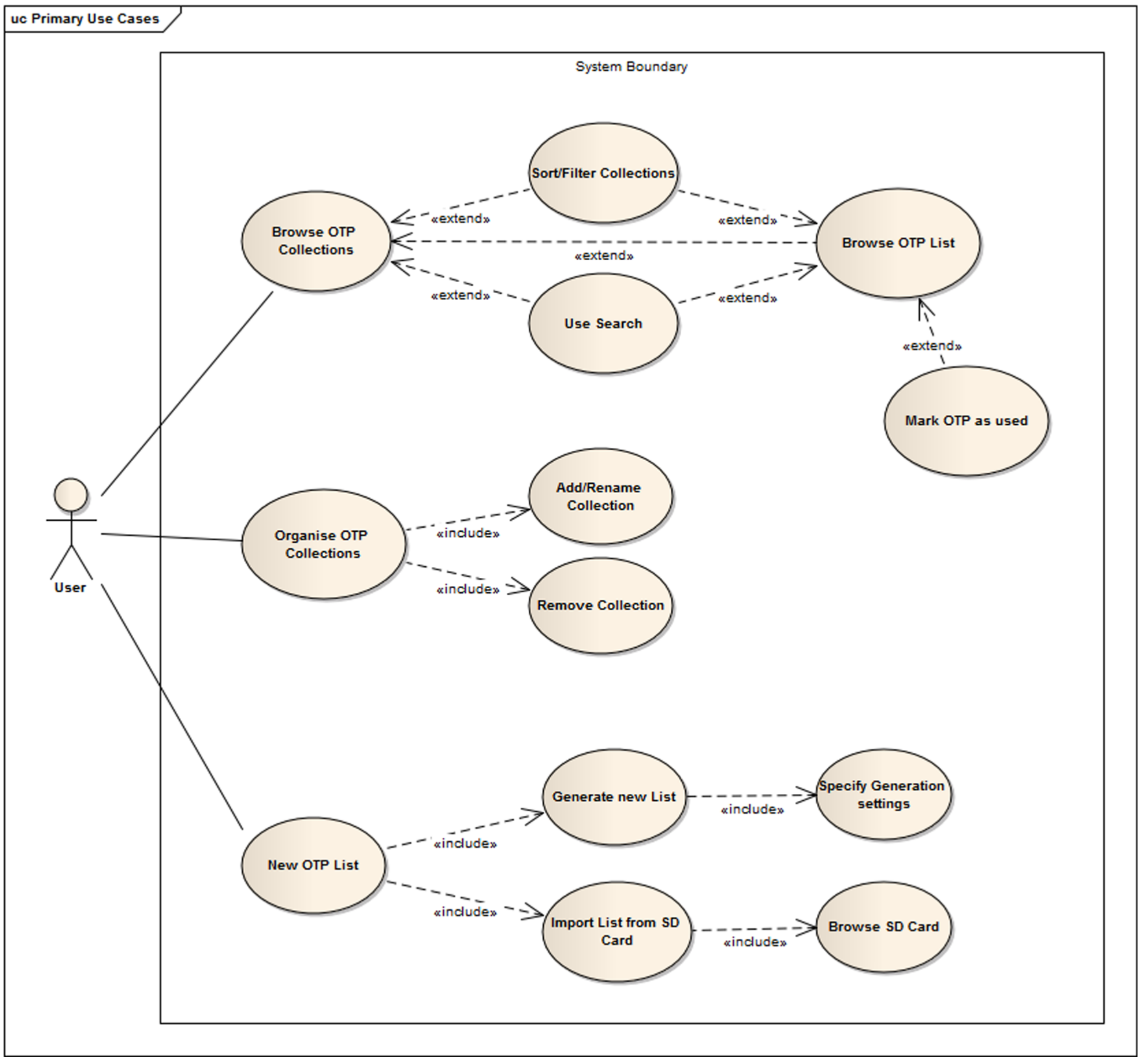


Figure 6.1: Primary use cases of the OTPManager App

6.1.2 Foregoing considerations

By laying out the use cases in the primary use case, it was obvious that the ability to easily browse and find the OTP lists as well as generating new lists would be the two defining components that should receive the most focus and attention.

Browsing

Browsing needs to be fast, easy and effective to satisfy the user. In order to achieve this, it had to be backed by a data source and structure that supported all these qualities. Also, a decision had to be made whether to implement the browsing capability in a general way to possibly support other types of OTP packages or systems in the future, or to tailor it specifically to OTPW.

Furthermore, having thought about using a database as data source, I considered to abstract the access to the data source to make the application independent of it (with a pattern like *Database Access Objects*). After much consideration, I decided against it since this would introduce a layer of indirection that could affect performance and increase the complexity of the design, all for the unlikely prospect of another database system on the Android platform.

In the end, the idea of using a database as data source was dropped completely in favor of a file-system-based solution which fit more nicely with the OTPW generator that could produce files. This made it easy to import OTP lists from OTPW as copying the files onto the SD Card was all that was necessary to do. Additionally, this solution was fast, easy and matched perfectly to the directory/file-like metaphor taken to organise the OTP lists in the first place.

Generating One-Time Passwords

An unknown factor was the One-Time Password generation. Having no experience with developing on mobile devices, I was unsure if the computational power of such a device was sufficient to generate adequately safe random OTP lists. As it turned out, this concern was unfounded: The processing power of my HTC Magic, featuring a *528 MHz Qualcomm MSM7201A ARM11 processor*, was more than enough to handle such a task in a timely manner.

Generating good random data is key to generating good OTP passwords. In contrast to ordinary computers, most mobile devices offer several excellent sources for true randomness: sensors. Most devices come packed with at least acceleration and orientation sensors that can be queried for random input data, many do also offer light, temperature or magnetic field sensors.

As was pointed out in chapter 3, true randomness isn't necessarily cryptographically safe. So in order to utilise the random data of the sensors in the most safe way, I had to think of a way to do so.

6.2 Design

With the requirements analysis and foregoing considerations finished, the next step was to design the application's architecture keeping the qualities identified in the analysis process in mind. Although an architecture does not necessarily provide or guarantee those qualities, it certainly can get in the way when the wrong design decisions are made for the foundation.

Before being able to sketch the actual architecture, final decisions had to be made for some of the factors identified by the analysis process that would define the key qualities I wanted to achieve with the architecture.

6.2.1 Design Decisions

Although the initial idea was to devise an app specifically for OTPW, I came to the conclusion that it wouldn't cost much to design the architecture in a more general way. As long as the components that change were well encapsulated, it would be possible to safely extend or change the supported OTP list formats later. As encapsulating the things that change is generally a good practice in object-oriented systems, I deemed it worth the effort.

After a few tests, the Android platform turned out to be powerful and fast, so performance was almost never a problem. As a result, I aimed for an extensible and clear design.

To keep the responsibility centralised, the main Activity of the application controls most of the workflow. Other Activities were used as delegates to fulfill certain specific tasks, but what should be launched when is determined by the central Activity to keep responsibility clear and the flow easy to follow. This also minimalised dependencies.

To loosely couple the components with the main Activity and keep responsibilities local, a general guideline was that each component should take care of its work itself. The main Activity only acts as a controller that coordinates everything, but does know as little as possible about how to do it. This is all encapsulated in the components that will do the real work.

6.2.2 Filesystem-driven Approach

Simply put, it can be said that the OTP Manager is a file browser application with some special functions tailored to handling One-Time Password lists. Of course, there is more to it (like the OTP list generation facility), but it helps to convey the general idea of the management component.

There are two types of components: *Collections* and *One-Time Password lists*. Both could be described as directories and files (by which they are actually represented in the backing data source, but they are not limited by their representation). Collections can contain both: other collections – arbitrarily deep nested – or OTP lists. OTP lists contain pairs of an index, necessary to identify the OTP, and the One-Time Password.

The OTP lists are read using a regular expression that matches the OTPW list format by default to extract every index-password pair. Already used passwords are tracked by a marker within the OTP list file. Sorting and filtering is done in memory utilising the capabilities of Java and the core libraries (no need to reinvent the wheel here).

6.2.3 Final Architecture

Figure 6.2 gives an overview over the final system architecture. Methods and fields were intentionally left out to improve clarity. Note that the `de.htwaalen.otp` package contains all Activities, whereas the subpackages contain the business logic.

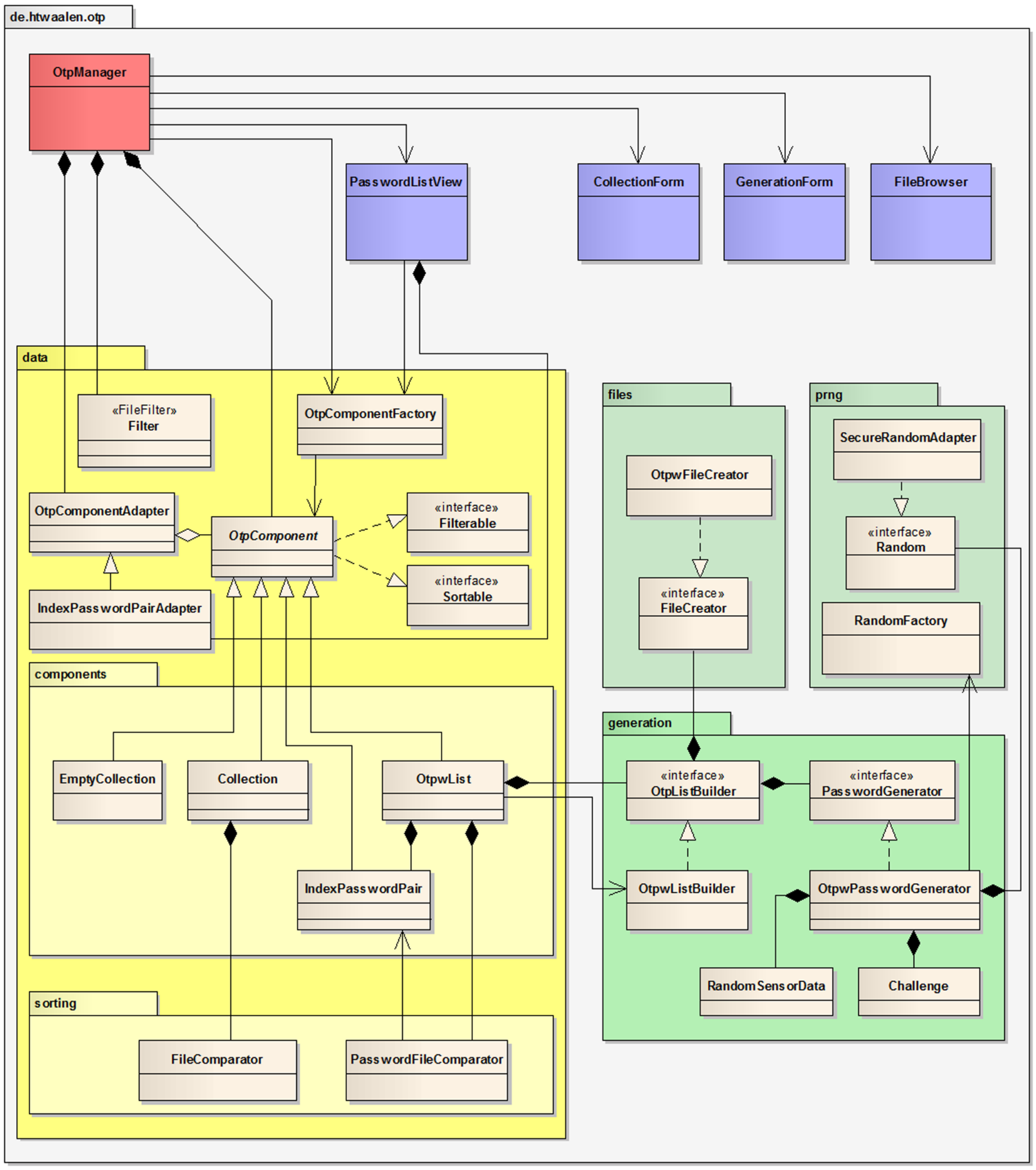


Figure 6.2: The overall architecture of the OTPManager App

As already mentioned, the `OtpManager` (depicted in red) is the main Activity that contains most of the program flow, while the other Activities (depicted in blue) are used by the main Activity to fulfill little tasks like showing a user input form to allow the input of a collection name (`CollectionForm` Activity). The `OtpManager` Activity is also responsible for displaying and browsing the collections, whereas the `PasswordListView` Activity displays the actual OTP list's content. The `FileBrowser` Activity is used to import OTP lists from the SD card.

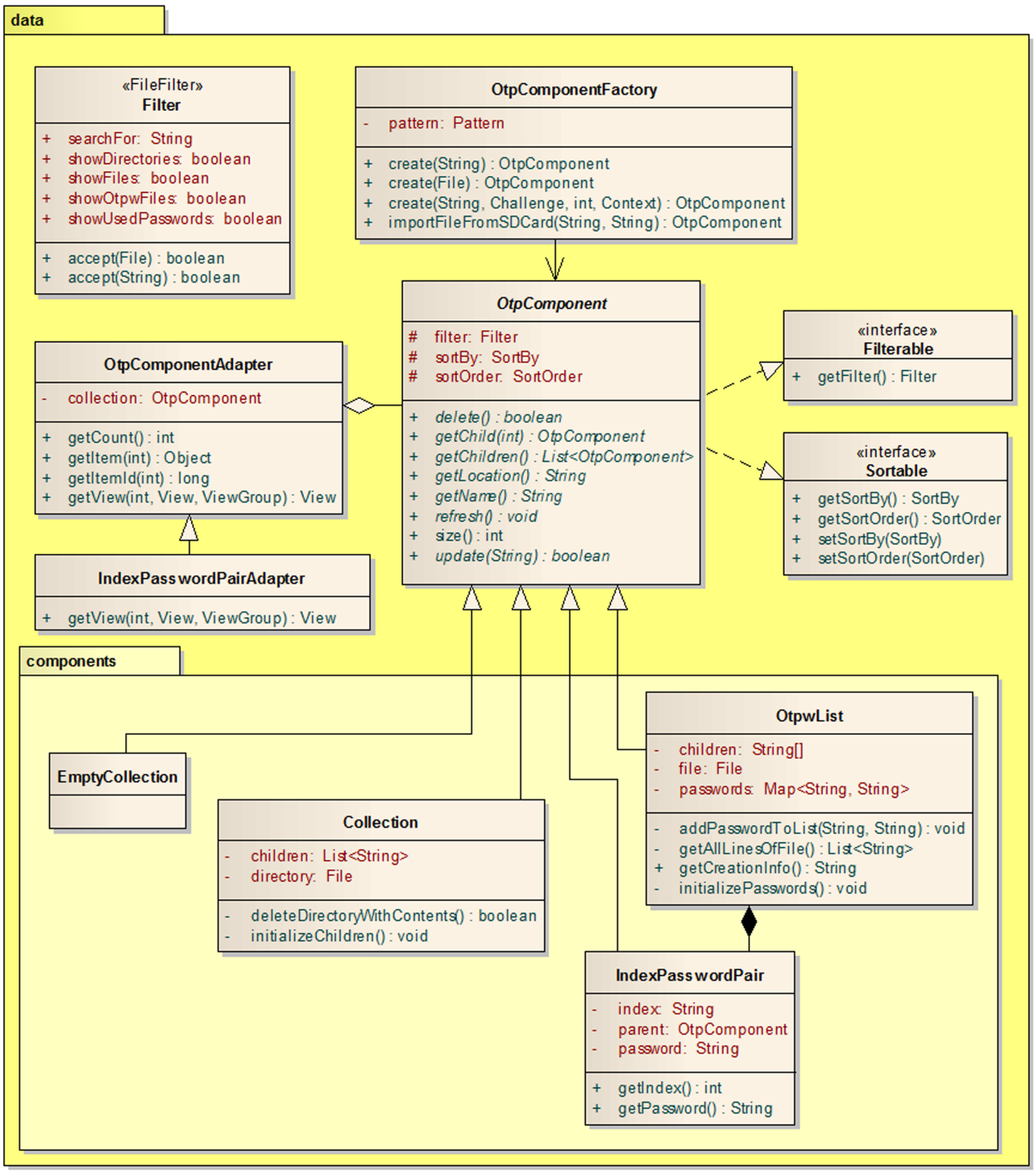


Figure 6.3: Detailed class diagram of the data package

The `data` package (depicted in yellow, see figure 6.3 for a detailed view) contains all components that are used by the `OtpManager` Activity to delegate requests of the user to the actual business logic – which is mostly hidden from the main Activity in the `components` package. This intermediary layer in the data package ensures that the main Activity is loosely coupled with the actual business logic, making it easily extendable.

The classes in the `components` package realise the *Composite pattern* (see [21], p. 163), with `OtpComponent` (an abstract base class) defining their interface to the outside world and implementing the common functionality for all components. To decouple the exact object types from the main Activity, the *Factory Method pattern* (see [21], p. 107) was used. For clarity, all factory methods were bundled into their own static factory class called `OtpComponentFactory`.

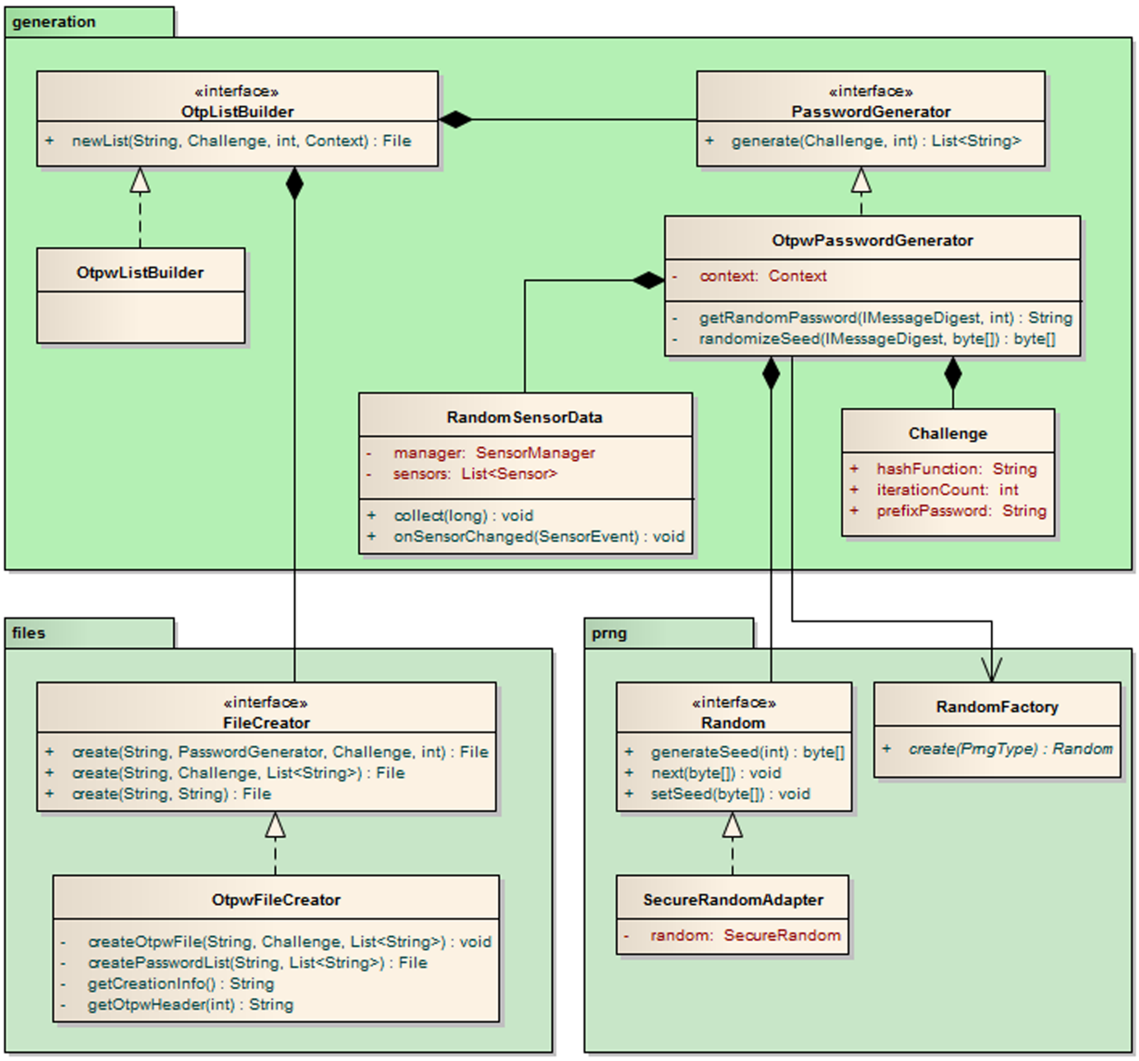

generation
«interface» OtpListBuilder
+ newList(String, Challenge, int, Context) : File
«interface» PasswordGenerator
+ generate(Challenge, int) : List<String>
OtpwListBuilder
OtpwPasswordGenerator
- context: Context
- getRandomPassword(IMessageDigest, int) : String
- randomizeSeed(IMessageDigest, byte[]) : byte[]
RandomSensorData
- manager: SensorManager
- sensors: List<Sensor>
+ collect(long) : void
+ onSensorChanged(SensorEvent) : void
Challenge
+ hashFunction: String
+ iterationCount: int
+ prefixPassword: String
files
«interface» FileCreator
+ create(String, PasswordGenerator, Challenge, int) : File
+ create(String, Challenge, List<String>) : File
+ create(String, String) : File
OtpwFileCreator
- createOtpwFile(String, Challenge, List<String>) : void
- createPasswordList(String, List<String>) : File
- getCreationInfo() : String
- getOtpwHeader(int) : String
prng
«interface» Random
+ generateSeed(int) : byte[]
+ next(byte[]) : void
+ setSeed(byte[]) : void
RandomFactory
+ create(PrngType) : Random
SecureRandomAdapter
- random: SecureRandom

Figure 6.4: Detailed class diagram of the generation package

The OTPW list generation capabilities are encapsulated and implemented in the `generation` package (depicted in green, see figure 6.4 for a detailed view). The `OtpListBuilder` is an interface that encapsulates the OTP list generation details, which can be viewed as a kind of *Strategy pattern* (see [21], p. 315) as each implementation of the interface represents some kind of a OTP list building strategy.

Concrete implementations of the `OtpListBuilder` class use the `PasswordGenerator` and `FileCreator` interfaces to delegate the task of generating passwords and creating a password list file. Currently, there is only one implementation for OTPW list generation and creation that use the corresponding OTPW `PasswordGenerator` and `FileCreator` implementations.

Lastly, the `prng` package is a gathering place for a `Random` interface and all PRNG implementations, which consists of only the `SecureRandomAdapter` at this time that adapts (see *Adapter pattern* in [21], p. 139) the `SecureRandom` PRNG to the interface defined in this package.

6.3 Implementation

In the following section, I will explain and highlight some of the implementation details. Development started with the realisation of OTP collection browsing, including sorting and filtering. After this was working properly, the generation of OTP lists was implemented following the implementation details described on the OTPW website [7]. Importing OTP lists and searching in OTP list files was next, including adding icons to make the UI visually more appealing.

The last step was to refactor and polish the app as well as implement convenience features like *Mark as used*.

6.3.1 Pseudo Random Number Generation

To utilise the random data provided by the sensors in a safe way, they were mixed with other random sources (like the current time) and used as seed for a cryptographically secure PRNG. This way, it is possible to use true randomness and still guarantee cryptographically strong random data.

Collecting Random Data from the Sensors

First, random sensor data had to be collected. In order to benefit from all available sensors, the `RandomSensorData` class was written to collect all values from the given sensors in an arbitrary order. If no specific list of sensors to query is given to the constructor, the class defaults to querying all sensors available on the device.

In Android, to collect sensor data you have to implement a `SensorEventListener` and listen for sensor events which contain the sensor data at that time. This was a problem since I needed the sensor data only and right at the time of generation. I didn't want to listen to all sensors all the time, which would have wasted the battery and processing power of the device.

To resolve this, I wrote the `collect` method (see listing 6.1) which registers itself to a given list of sensors (or all available, as already mentioned) and listens for a set amount of time. The `sleep` is necessary since sensor events are not received at the time of registration but later when the sensor has something to report.

Listing 6.1: The RandomSensorData class to collect sensors values

```java
public class RandomSensorData implements SensorEventListener
{
    private SensorManager manager;
    private List<Sensor> sensors;

    public List<Float> values = new ArrayList<Float>();
    // ...

    public void collect(long timeSpanInMs) {
        // When no sensors are given, use all available
        if(sensors ==  null || sensors.isEmpty()) {
            sensors = manager.getSensorList(Sensor.TYPE_ALL);
        }

        // Register as listener to all sensors in the list
        for (Sensor sensor : sensors) {
            manager.registerListener(this, sensor, SensorManager.
                SENSOR_DELAY_FASTEST);
        }
        SystemClock.sleep(timeSpanInMs);  // stay ahwile and
            listen
        manager.unregisterListener(this); // stop listening
    }

    /* Called when new sensor data is available */
    public void onSensorChanged(SensorEvent event) {
        for (float v : event.values) {
            values.add(v);  // simply add all values
        }
    }

}
```

`onSensorChanged` receives a `SensorEvent` object that contains all related sensor data. Since it does not matter which data is received from what sensor, all reported values are simply added to a list which will be mixed into the seed of the PRNG later.

Randomising the CSPRNG Seed

Listing 6.2 demonstrates the usage of the `RandomSensorData` class.

First, the hash function is initialised to the value to hash, the current seed – which is a randomly generated string with a length of 1 to 16 characters. Then, random data is collected from the sensors for 1000 ms, which was deemed a long enough timespan to produce a sensible amount of sensor values verified through tests.

Listing 6.2: The PRNG seed is further randomised before it is used

```java
private byte[] randomizeSeed(IMessageDigest md, byte[] seed)
{
    // Initialize hash function with seed
    md.update(seed, 0, seed.length);

    // Collect random data from sensors
    SensorManager manager = (SensorManager) context.
       getSystemService(Context.SENSOR_SERVICE);
    RandomSensorData data = new RandomSensorData(manager);
    data.collect(1000); // collect for 1000 ms

    byte[] random = md.digest(); // hash initial seed
    int mixedSize = random.length + (Float.SIZE/Byte.SIZE);
    byte[] mixedState = new byte[mixedSize];
    ByteBuffer buffer = ByteBuffer.wrap(mixedState);

    // Iterate through all collected values, mix them with
    // the current seed and hash the result
    for (float value : data.values) {
        buffer.clear();
        buffer.put(random);
        buffer.asFloatBuffer().put(value);

        md.update(buffer.array(), 0, buffer.array().length);
        random = md.digest();
    }

    return random;
}
```

This random data is then inserted into the `ByteBuffer` alongside the hashed value of the initial seed and then hashed again. This is done for every single sensor value that was collected. After all sensor values haven been mixed and hashed, the result is returned as randomised seed to initialise the CSPRNG.

6.3.2 OTPW Implementation

This section describes how generating One-Time Passwords according to the OTPW design guidelines was realised. The OTP generation in the `OtpwPasswordGenerator` is pretty straightforward (see listing 6.3).

At the beginning of the method, a CSPRNG object is created and initialised, using the randomised seed from the previous section. Then, for as many passwords as requested, the method `getRandomPassword` is called in a loop and the returned OTP is added to a list, which will eventually be returned to the caller.

Listing 6.3: The OTPW password list generation method

```java
public List<String> generate(Challenge c, int numPasswords)
{
    Random rng = RandomFactory.create(PrngType.JavaSecureRandom);
    byte[] seed = rng.generateSeed(16);

    IMessageDigest md = HashFactory.getInstance(c.hashFunction);
    seed = randomizeSeed(md, seed);
    md.update(seed, 0, seed.length);

    List<String> passwords = new ArrayList<String>();
    for (int i = 0; i < numPasswords; i++) {
        passwords.add(getRandomPassword(md, c.iterationCount));
    }

    return passwords;
}
```

Usually, the `generate` method is called from within a `FileCreator` implementation, which uses the returned OTP list and takes care of writing the passwords to a file.

Listing 6.4: Implementation of the OtpwFileCreator class

```java
public File create(String saveTo, int numOfPasswords,
        PasswordGenerator generator, Challenge challenge)
{
    List<String> passwords = generator.generate(challenge,
                                          numOfPasswords);
    return create(saveTo, challenge, passwords);
}
```

The file's format depends on the `FileCreator` implementation. Obviously, the `OtpwFileCreator` writes an OTP list to a file conforming to the format of the OTPW package. It also creates a hidden `*.otpw` file (whereas `*` corresponds to the name of the OTP list). As mentioned in section 2.4.3, this file is necessary to configure the server to accept the generated OTPs.

Generating the One-Time Passwords

The last section described the creation of OTP lists, omitting the implementation details of `getRandomPassword` for the actual One-Time Password creation.

Listing 6.5 reveals the implementation details of the `getRandomPassword` method.

Listing 6.5: The OTPW single password generation method

```java
private String getRandomPassword(IMessageDigest md,
                                 int iterations)
{
    byte[] random = md.digest();
    int mixedSize = random.length + (Long.SIZE/Byte.SIZE);
    byte[] mixedState = new byte[mixedSize];
    ByteBuffer buffer = ByteBuffer.wrap(mixedState);

    // hash OTP with current system time several times
    for (int i = 0; i < iterations; i++) {
        buffer.clear();
        buffer.put(random);
        buffer.putLong(System.nanoTime());
        md.update(buffer.array(), 0, buffer.array().length);
        random = md.digest();
    }

    // Use first 72 bit of hash
    String pw = ModifiedBase64.encode(random, 0, 9, false)
                .substring(0, 8);

    return pw;
}
```

This implementation mimics the OTPW design as described on the OTPW website [7], which is also described in section 2.4.3 of this work. Again, a `ByteBuffer` is used to mix the random bytes with the current system time. This whole iteration is repeated many times, hashing and further mixing the result time again and again until the iteration count selected by the user has been reached.

Then, according to the OTPW design guidelines, only the first 72 bits of the hashed value are encoded using the modified base64 encoding – which simply replaces some ambiguous characters for readability. The resulting password is then trimmed to 8 characters and returned.

Generating the Server Configuration File

The server configuration file contains the hashed values of the prefix password concatenated with each One-Time Password to safely verify the given login information without exposing the prefix password. It also keeps track of already used OTPs by removing those values from the file. Since it is an integral part of the OTPW package's design and cannot be generated from an existing list afterwards easily, the app had to take care of creating a corresponding file for each OTP list, too.

The lines containing the hashed passwords need to be shuffled as the server asks for them in the order they are written to the file. If this would be in an order sorted by the index, the inquiry would become predictable. The function that creates the hashed values and shuffles the line can be seen in listing 6.6 below.

Listing 6.6: The generation method for the .otpw server configuration file

```
private List<String> getShuffledPasswordLines(Challenge c,
                                              List<String> passwords)
{
    List<String> shuffledPWs = new ArrayList<String>();
    int index = 0;
    IMessageDigest md = HashFactory.getInstance("ripemd-160");
    DecimalFormat df = new DecimalFormat("000");

    for (String otp : passwords) {
        // Create hash of prefix pw concatenated with OTP
        String pw = challenge.prefixPassword + otp;
        byte[] seed = pw.getBytes();
        md.update(seed, 0, seed.length);

        String hash =
            ModifiedBase64.encode(md.digest(), 0, 9, false)
            .substring(0, 12);
```

```java

        // Add to list with leading index
        shuffledPWs.add(df.format(index++) + hash);
    }

    Collections.shuffle(shuffledPWs, new SecureRandom());
    return shuffledPWs;
}
```

Those lines are then written to the file after the header information that specify the file format (see section 2.4.3).

It is important to note that only the first 72 bits of the hashed value are encoded by the modified base64 encoding here, too, but that 12 characters of the resulting hash are stored in the file in contrast to the 8 letter One-Time Passwords that OTPW uses.

6.3.3 Performance

Performance had been a concern since the start, but as it turned out Android was able to churn out 100 OTPs, with 50 hash iterations per password, in a couple of seconds without any problem. This was far more faster than anticipated and fast enough for typical usage, so no optimisation was necessary at the generation process.

But to keep the user interface responsive for those few seconds in accordance to the Android design guidelines, an `AsyncTask` was implemented to move the generation into its own thread. `AsyncTask` is an auxilliary base class that offers a simple and convenient mechanism to perform background operations. It encapsulates thread handling and offers event handlers that run synchronised with the GUI thread to allow updating the UI elements for progress reports.

This allows the user to continue using the app while an OTP list is being generated in the background. When generation finishes, the OTP list view is updated and the user will be informed by a Toast.

Listing 6.7: AsyncTask to run the OTP generation process in its own thread

```java
private class GenerateOtpListTask extends AsyncTask<Void, Integer
    , Void>
{
    private String saveTo;
    private Challenge challenge;
    private int numOfPasswords;
    // ...

    @Override
    protected Void doInBackground(Void... params) {
        // Pass request to generate a new OTP list to the
        // factory, which will in turn pass it to the OTP
        // list component that will take care of it
        OtpComponentFactory.create(saveTo, challenge,
                numOfPasswords, getApplicationContext());
    }

    /* Inform the user about what's going on. */
    @Override
    protected void onPreExecute() {
        super.onPreExecute();

        Toast t = Toast.makeText(getApplicationContext(),
                R.string.otp_generation_started,
                Toast.LENGTH_LONG);
        t.show();
    }

    /* Update list and inform user */
    @Override
    protected void onPostExecute(Void result) {
        super.onPostExecute(result);

        refreshList();
        Toast t = Toast.makeText(getApplicationContext(),
                R.string.otp_generation_finished,
                Toast.LENGTH_SHORT);
        t.show();
    }
}
```

Listing 6.7 shows `GenerateOtpListTask`, an inner class of the OtpManager and a subclass of `AsyncTask`. As the `OtpManager` class is the only instance that needs to use it an inner class was used to hide the implementation.

As can be surmised, `onPreExecute` and `onPostExecute` will be called right before `doInBackground` runs respectively right after it has finished. The `AsyncTask` is run by calling `execute` on an instanced object. The pre- and post-methods in this case inform the user about the start/end of the generation process.

The OTP list generation itself is encapsulated by the `OtpComponent` hidden by the factory. All that needs to be done is to call the factory, pass all information necessary for the generation process to it, and let the component take care of proper creation and generation itself. The `OtpManager` does not and should not know what exactly is being created or how, for easy extensibility.

Browsing Performance

As described in section 6.4, the browsing performance received special attention. Figure 6.5 visualises the average access time on OTP lists of typical sizes.

As can be seen, the access time shows linear growth (with an outlier on lists of size 400, but with a control sample of only 10 values this can be expected). Also, the average access time is below 1 second for lists of sizes from 100 to 500 OTPs, which is acceptable.

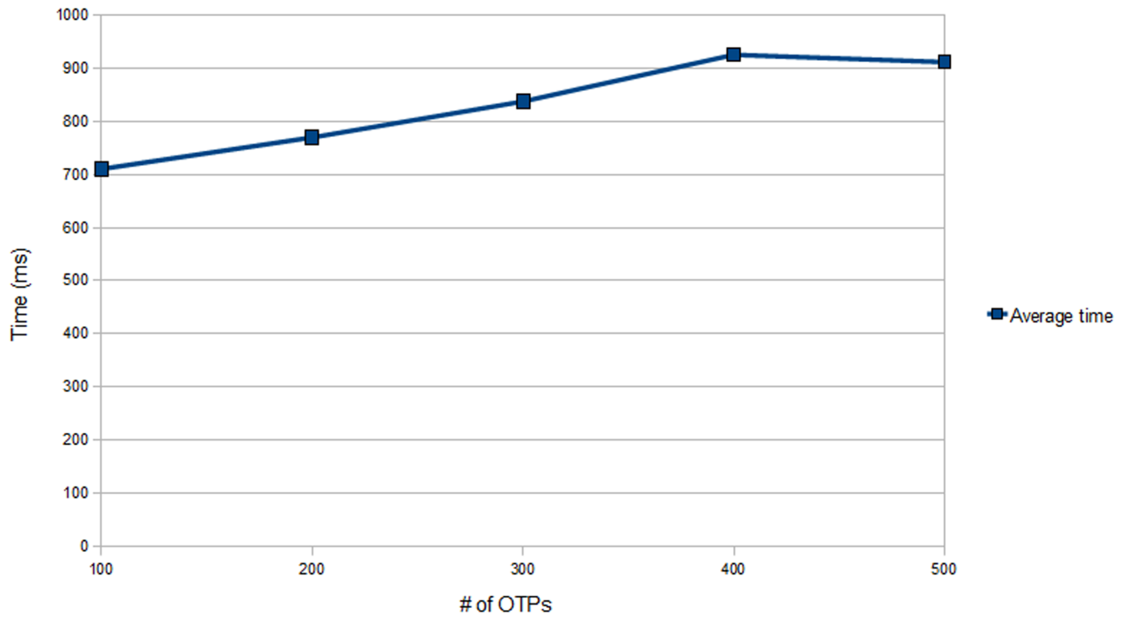


Figure 6.5: Average access time for OTP password lists of differing sizes

6.3.4 Mark as used-feature

To retain an overview over already used passwords, the '*mark as used*'-feature was implemented to display already used passwords striked-through (see figure 6.7) and to allow the sorting and filtering by usage state (used/unused).

To keep track of the usage state in a persistent manner, an implementation operating on the OTP lists themself has been chosen. One-Time Passwords are stored in the following manner:

```
[3-digit-index] [OTP]
000 g46DSGtg
001 Jggf53L2
002 83rfSA35
...
```

The first three digits represent the OTPs' index and, separated by a single whitespace, the OTP itself follows. To mark an OTP as used, this format was extended to this:

```
[usage-marker][3-digit-index] [OTP]
!000 g46DSGtg
!001 Jggf53L2
002 83rfSA35
...
```

The exclamation mark denotes an already used OTP, meaning the OTPs with index `000` and `001` have already been used, whereas `002` is still unused. This is simple and could be implemented in a fast way.

6.4 Usability Considerations

The One-Time Password lists should be organisable in any way the user wants them to be organised. There is no user like the other, everyone has his own, preferred way of organising things, so the application should support this. Hence, an arbitrarily nestable collection structure was created, allowing collections to contain other collections or lists or both.

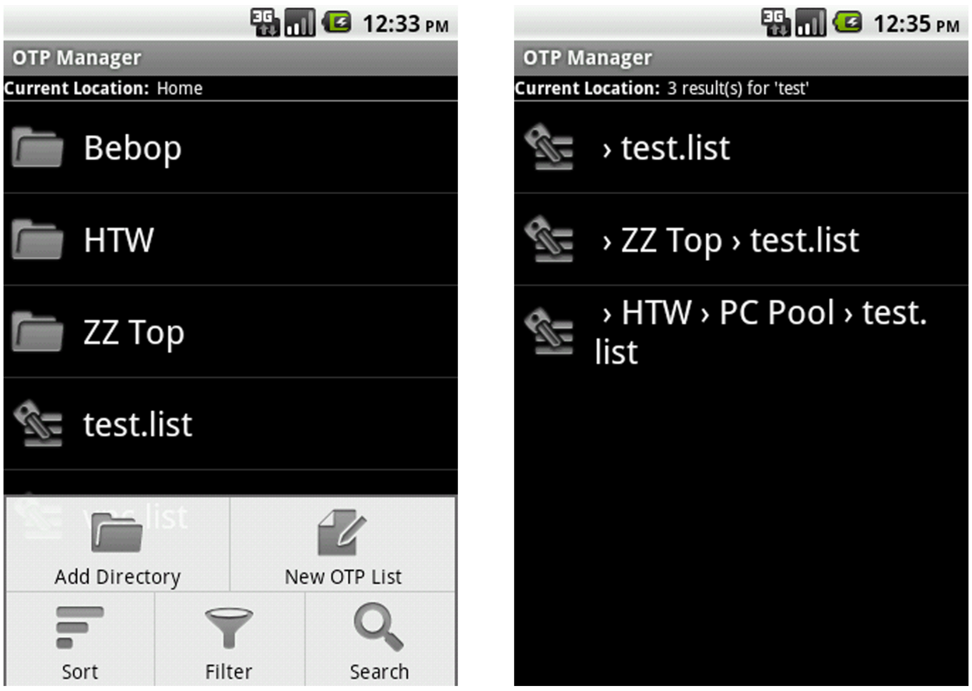
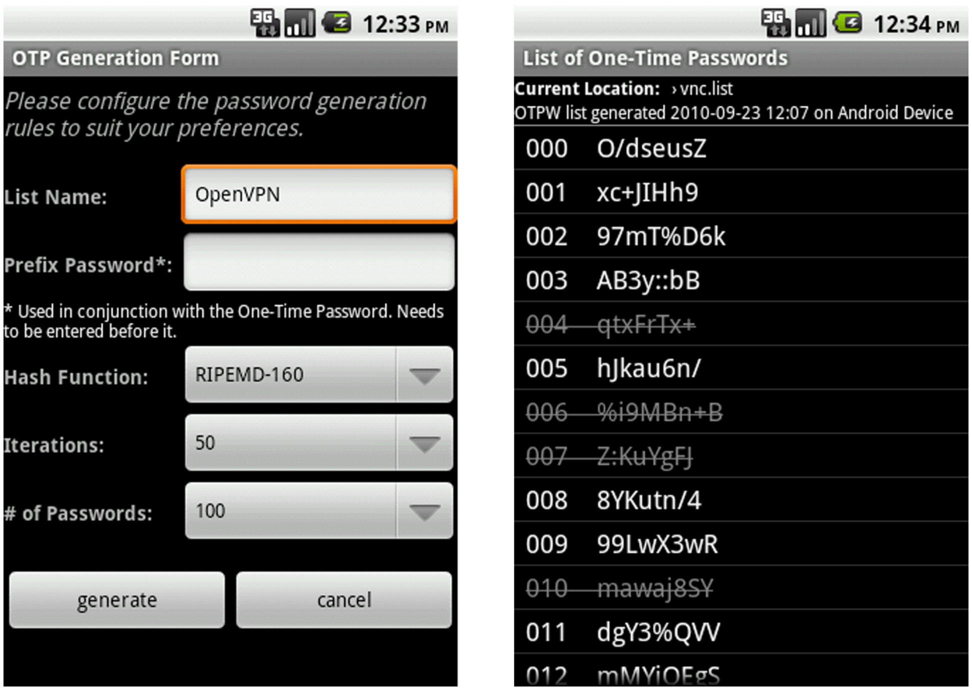


Figure 6.6: Left: The browsing screen. Right: The search results screen.

Furthermore, the lists should be browseable in an efficient and comfortable manner. For this sake searching, sorting and filtering features were implemented. Ease of use is about being efficient, and this tools help the user in finding what she is looking for as fast as possible.

Another important factor was performance. Browsing the lists needs to be fast. When every selection takes two seconds to process, the user would be annoyed sooner or later. As a result, browsing collections is fast and effortless (taking around 400-500 ms as reported by the DDMS). Displaying a One-Time Password list takes a little bit longer than browsing collections (see section 6.3.3 for details), as there usually are a few hundred OTPs to be read. Since the user has already found what he is looking for when opening an OTP list, this is considered acceptable.



12:33 PM

OTP Generation Form

Please configure the password generation rules to suit your preferences.

List Name:	OpenVPN
Prefix Password*:	

* Used in conjunction with the One-Time Password. Needs to be entered before it.

Hash Function:	RIPEMD-160
Iterations:	50
# of Passwords:	100

12:34 PM

List of One-Time Passwords

Current Location: › vnc.list

OTPW list generated 2010-09-23 12:07 on Android Device

```
000  O/dseusZ
001  xc+JIHh9
002  97mT%D6k
003  AB3y::bB
```
~~004 qtxFrTx+~~

005 hJkau6n/

~~006 %i9MBn+B~~

~~007 Z:KuYgFJ~~

008 8YKutn/4

009 99LwX3wR

~~010 mawaj8SY~~

011 dgY3%QVV

012 mMYiOEgS

Figure 6.7: Left: The OTP list generation screen. Right: The OTP list overview screen.

To alleviate orientation when browsing, a bread-crumb-like path of the current location is displayed above the list view that displays the content of the currently selected collection (see figure 6.6). Also, the buttons in the options menu have been arranged in a sensible, logical way grouping creational features in the upper row and supporting options on the lower row, since those features are expected to be required more often and the lower row is more easily accessible with the thumb when holding the smartphone in a single hand.

7 Conclusion

The following chapter summarises the results of this work, evaluating the Android platform and the application itself.

7.1 Evaluation

7.1.1 Android Platform and Development

The Android platform proved to be flexible, powerful and easy to use. It is well-engineered, offers good perfomance and a solid base of components that can be reused or extended to accommodate differing needs. The documentation is exhaustive and outstanding, also there are many tutorials and books to get started, easing the familiarisation process significantly.

Development is smooth and fast, supported by the sophisticated tools delivered with the SDK that enable fast development cycles, offer powerful debugging capabilities and allow the creation of diverse apps that can take full advantage of the underlying hardware.

On top of that, Android offers a wide variety of devices by different manufacturers giving the user the power of choice and the possibility to find a device that suits her needs better.

The security model is robust, insulating each app to limit possible damage of attacks while allowing them to cooperate or share data and components betweeen them. The only real drawback is the permission system which expects from the user to make prudent decisions on granting rather general permissions for applications that do not elaborate on how those general permissions are actually used.

7.1.2 The application

The development of the OTP Management app went smooth, allowing me to focus on the important aspects of the application – not having to struggle with the platform or the application framework itself most of the time.

The design guidelines by Google on the Android developers website [3] are well thought-out and help creating an user friendly app. The documentation helped me alot to solve problems or get to understand the usage of certain components. Thanks to being able to reuse many platform components – and being able to customise those few that had to be adjusted – it was relatively easy to create an application that fit seemlessly into the system.

The performance and processing power was surprisingly good and the knowledge that it's possible to use the *Native Development Kit* (NDK) for cases where it's still not fast enough certify the maturity and power of the platform.

7.2 Outlook

The platform independent nature, open philosophy and support by the Open Handset Alliance warrant the success of Android. As could be seen in chapters 4 and 5, the platform is elaborate, well-engineered and its popularity is rising [23].

Another strong point is that development for Android is free of charge and utilises the Java programming language, meaning there are barely any obstacles to start developing for it. Contrast this to Apple's iOS SDK, which requires the acquisition of a license approved by Apple and is only available for Mac OS X, then it's predictable that the Android Market will surpass the iPhone Store in terms of app abundance some day.

Concerning the permission system's issue which assumes prudent, knowing users it can only be said that the future has to tell how this system will work out in the end. At the time of this writing there have only been few cases of malware for Android, but it can be expected that with the rising popularity of the platform the number of malware occurrences will also rise.

Google's current stance of remotely removing malicious apps at least allows them to contain the damage when it could not be prevented.

A List of Acronyms

App	*Application software.* In this work the term is exclusively used to address a computer program that can be installed and used on a smartphone like the Apple iPhone or any Android-based smartphone.
AVD	*Android Virtual Device.* A virtual machine that emulates a fully operational Android device. An AVD offers options to model the VM after actual devices for testing purposes.
CSPRNG	*Cryptographically Secure Pseudo Random Number Generator.* A CSPRNG is a PRNG that satisfies additional properties – in the context of cryptograhpy – to secure it against attacks that aim to break the unpredictability of its output.
IDE	*Integrated Development Environment.* An IDE provides extensive tools and factilities to programmers for software development. Most common is the provision of a source code editor, compiler, build automation tool and debugger, though additional tools may be provided as well.
PRNG	*Pseudo Random Number Generator.* Refers to an algorithm that produces seemingly random sequences, which is not truly random but determined by an initial value called *seed*.
OTP	*One-Time Password.* A password that can only be used for a single login. After it's used it becomes invalid and cannot be used again.
RNG	*Random Number Generator.* A Random Number Generator is a device (computational or physical) that generates sequences of random numbers.
SoC	*System-on-a-chip.* Refers to integrating all components of a computer or other electronic system into a single integrated circuit (chip).
VM	*Virtual Machine.* A virtual machine is a software that simulates a machine (like a computer) and executes programs like they were running on the simulated machine.

Bibliography

[1] **Android Apps boomen**
heise, 11th July 2010
http://preview.tinyurl.com/2whgxdz

[2] **Extensive Smartphone & Chip Market Study**
3G, 5th February 2009
http://preview.tinyurl.com/27zmy4u

[3] **Android Developers – Dev Guide**
Google, as at July 2010
http://preview.tinyurl.com/y8on4u6

[4] **Professional Android 2 Application Development**
Reto Meier
Wrox, 1st edition, 1st March 2010

[5] **Android (operating system)**
Wikipedia, as at August 2010
http://en.wikipedia.org/wiki/Android_(operating_system)

[6] **Some clarification on "the Android Kernel"**
LWN.net, as at August 2010
http://lwn.net/Articles/373374/

[7] **OTPW – A one-time password login package**
Markus Kuhn, Version 1.3, 30th September 2003
http://www.cl.cam.ac.uk/ mgk25/otpw.html

[8] **One-time password**
Wikipedia, as at August 2010
http://en.wikipedia.org/wiki/One-time_password

[9] **A One-Time Password System**
N. Haller et al, February 1998
http://www.faqs.org/rfcs/rfc2289.html

[10] **MD4**
Wikipedia, as at August 2010
http://en.wikipedia.org/wiki/MD4#Security

[11] **MD5**
Wikipedia, as at August 2010
http://en.wikipedia.org/wiki/MD5#Security

[12] **SHA-1 Broken**
Bruce Schneier, February 15, 2005
http://www.schneier.com/blog/archives/2005/02/sha1_broken.html

[13] **Practical Cryptography**
Bruce Schneier & Niels Ferguson
Wiley Publishing Inc., 2003

[14] **Chapter 18. Pluggable Authentication Modules (PAM)**
The NetBSD Project, as at August 2010
http://www.netbsd.org/docs/guide/en/chap-pam.html

[15] **Statistical randomness**
Wikipedia, as at August 2010
http://en.wikipedia.org/wiki/Statistical_randomness

[16] **Entropy (information theory)**
Wikipedia, as at August 2010
http://en.wikipedia.org/wiki/Entropy_(information_theory)

[17] **Random Number Generation**
Wikipedia, as at August 2010
http://en.wikipedia.org/wiki/Random_number_generation

[18] **Pseudo Random Number Generator**
Wikipedia, as at August 2010
http://en.wikipedia.org/wiki/Pseudorandom_number_generator

[19] **Cryptographically Secure Pseudo Random Number Generator**
Wikipedia, as at August 2010
http://preview.tinyurl.com/rh66l

[20] **Cryptanalytic Attacks on Pseudorandom Number Generators**
J. Kelsey, B. Schneier, D. Wagner, and C. Hall
Fast Software Encryption, Fifth International Workshop Proceedings (March 1998), Springer-Verlag, 1998, pp. 168-188. or http://www.schneier.com/paper-prngs.html

[21] **Design Patterns – Elements of Reusable Object-Oriented Software**
Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
Addison-Wesley, 1st edition, November 10, 1994

[22] **Dalvik Virtual Machine**
dalvikvm.com, as at August 2010
http://www.dalvikvm.com

[23] **Report from comScore Confirms Android, Mobile Internet Growth in US**
Greg Sterling, Internet2Go, September 16, 2010
http://preview.tinyurl.com/2d852ez

[24] **Android Also Gives Google Remote App Installation Power**
Dennis Fisher, threatpost, June 25, 2010
http://preview.tinyurl.com/26rv66u

[25] **Exercising Our Remote Application Removal Feature**
Tim Bray, Android developers blog, June 23, 2010
http://preview.tinyurl.com/2dofgzs