UNIKASSEL
VERSITÄT

Android
Sensor Data Collection Framework

Project Report of

Katy Hilgenberg

At the Department of Electrical Engineering/Computer Science
Institute for Knowledge and Data Engineering

Advisor: Dr. Martin Atzmüller

Duration: 01. April 2011 - 31. October 2011

Contents

1. Introduction

1.1 Background

The Knowledge & Data Engineering Group at the University of Kassel is involved in the VENUS research cluster as part of the program for excellence in research and development of the State of Hesse. *"The goal of VENUS is to explore the design process of future networked, ubiquitous systems, which are characterized by situation awareness and self-adaptive behavior"*[Ven11]. A branch in this research is concerned with the analysis of usage data in sensor based systems.

Nowadays mobile phones are widely-used communication and social interaction media. Modern smartphones do provide a large range of different sensor types, like GPS, Accelerometer, Gyroscope, GSM, WLAN or Bluetooth. The sensor information can be used for localisation purpose or position tracking, as well as for the analysis of social interactions.
Beside Apples iOS and BlackBerry[1], Android[2], an operating system for mobile devices published under the Apache License, has become one of the most popular platforms for smartphones recently. The current Android API[3] development is focused on the optimisation for tablet computers, another device type with a high increase in popularity.

In this context, the task of collecting and providing sensor data of common mobile Android devices, becomes a matter of particular interest.

1.2 Objective

The aim of this project was the design and implementation of a device independent and configurable framework application, to collect, store and transfer available sensor data from mobile Android devices.

Essential requirements have been

- adaptivity to variable equipped systems, with different sensor types available and dynamic sensor state changes at runtime (e.g. unavailability in flight mode),
- easy to extend for new sensor types,
- temporary persistent local storage of collected sensor data,
- an adaptive, efficient and system resources preserving transmission service, able to deal with unavailability of transmission possibilities using different transmission strategies,
- various configuration options.

A high value was set to a proper software design, such as considering aspects of extensibility, modularity and robustness or following concepts like abstraction, delegation and information hiding. In particular, the usage of known design patterns to solve common problems was intended.

[1] developed by Research In Motion
[2] Android is developed by the Open Handset Alliance led by Google
[3] API is a common abbreviation for Application Programming Interface

1.3 Approach

As I was new to Android development, it was necessary to familiarize myself with the platform, the concepts and the development environment first. This was done accompanying to the requirements analysis.

The first project step was the analysis of the objective with respect to the required framework features. Together with the project advisor, several requirements have been defined. Afterwards, a rough design concept for the framework was developed and a basic time schedule done. The subsequent development process was split in three sub-phases, each planned as an iterative process of design refinement and implementation, having one of the core features (sampling, persistent storage and transfer) as topic.

The project phases:

- 11.04.2011 - 30.04.2011:
  Requirement analysis and basic concept.
- 01.05.2011 - 31.05.2011:
  Development of the application structure and the core functionality, including sensor abstraction, scanning and sampling, basic structures for configuration management and the data representation layer.
- 01.05.2011 - 30.06.2011:
  Development of the persistence layer together with the asynchronous sample collection.
- 01.07.2011 - 15.08.2011:
  Development of the data transfer functionality.
- 15.08.2011 - 31.10.2011:
  Bug fixing and finalization, documentation and project report.

1.4 Resources

While there are various resources for Android development, I would like to point out those, which have been relevant for the framework development.
A good introduction to Android development is given by the first edition of the book *Android: Grundlagen und Programmierung* from Arno Becker and Marcus Pant [BP09]. The *Android Developer Site* [And11] provides a detailed API documentation as well as many useful developer guides and code examples. It is the first choice for Android related topics and questions. A valuable source for special issues is *stackoverflow* [Sta11], a free question and answer platform for programmers.

For the software modelling part, I did rely on the standard work in *Design Pattern* [GHJJ96] and *UML@Work*[HKKR05]. The UML diagrams have been constructed using the open source Editor *UMLet*[4].

[4]the UMLet project site: http://www.umlet.com/

2. Requirements

The requirement analysis was based on the initial objective definition, accompanied by the analysis of the Android-API with regard to the limitations of the platform. The results are subdivided into functional and non-functional requirements and will be listed in the following sections.

2.1 Functional Requirements

Functional requirements are ordered with respect to the core features of the framework.

2.1.1 Sensor Data Sampling

Support of Different Sensor Types

There are several sensor types to support which do not only differ in the kind of sensor information, but also in the way of access. In general, different sensor types do provide data in varying update frequencies and are accessible by different scanning methods (e.g. active cyclic polling or passive observation).

The initial amount of supported sensors was limited to five representative sensor types:

- GSM[1],
- WLAN[2] (respectively Wifi[3]),
- Bluetooth,
- GPS[4],
- Accelerometer.

All types together do cover most of the different types of sensor access offered by the Android API. Based on the distinctive sensor types, a proper degree of abstraction must be found for sensors, related scanners and specific sensor data.

Proper Degree of Abstraction

The internal sensor representation has to guarantee a homogeneous handling of different sensor types, with respect to the aspects of sampling and configuration.

Expandability

Future framework extensions for new sensor types must not effect the generalized collection, storage or transmission tasks. Furthermore this task is reduced to the extension of available basic types or the implementation of interfaces.

[1] Global System for Mobile Communications (cellular phone technology)
[2] Wireless Local Area Network
[3] Wireless Fidelity (IEEE 802.11b wireless networking)
[4] Global Positioning System

Modularity

The module for sensor data collection is the only device dependent part in the whole framework. It has to provide an independent interface which allows the observation of taken samples without any knowledge about internal issues.
A common solution would be the usage of an observer pattern to deliver the samples by callback methods to registered observers. This does allow the implementation of other forms of sample processing, like broadcasting.

Sensor Availability State Awareness

The awareness of sensor availability does imply two aspects. On the one hand it is necessary to identify supported sensor devices which are available in the system in general, even if they are temporarily not accessible (e.g. in flight mode). On the other hand the availability state for access has to be observed at runtime. Thus, sampling can be limited to available sensors, to preserve system resources.

Sensor Device Configuration Options

For each supported sensor device the following settings are available:

- the enabled state (does indicate whether device scanning is activated for a supported sensor device or not),
- the scan frequency in milliseconds,
- a sample transmission priority (to be used by transmission strategies)

As there are several platform limitations for the scan frequency, any undersized configuration value will just be overridden by internal limits.

2.1.2 Sensor Data

Sensor Data Representation

It is necessary to find a uniform internal data representation for the varying sensor data. All types of sensor samples do have at least three properties in common:

- a sensor device identifier,
- a time stamp in UTC[5] (when the sample was taken)
- a transfer priority (inherited from the associated device)

In addition, there is the sensor specific data part, which is dependent on the concrete sensor device type. Beside the device module, sample processing must be independent from knowledge about specific sensor data.

Serializability

The internal sample data representation has to be serializable for the purpose of transmission. An XML[6] representation will be the first choice here, as it is a well supported standard for the exchange of structured data.

[5] international acronym for *Coordinated Universal Time*
[6] common abbreviation for *Extensible Markup Language*

Sensor Specific Data

For the five representative sensor types the following sensor specific information will be supported:

- GSM:
  signal strength and cell identifier.
- WLAN:
  signal strength, SSID, BSSID, frequency, level and capabilities.
- Bluetooth:
  signal strength, hardware adress, Bluetooth name and Bluetooth device class.
- GPS:
  longitude, latitude, altitude, speed and accuracy.
- Accelerometer:
  acceleration in x, y, and z direction.

2.1.3 Persistent Storage

Sensor data samples may be provided in a high frequency for further processing. The sample transmission task depends on available connections and has to select samples based on specific criteria. Hence it is required, that the collected samples will be stored locally on the device in the meantime.

Database Scheme

Android does provide several options for persistent storage of application data. Most do rely on files, but SQLite[7] databases are also supported. For transfer purpose, samples have to be selected ordered by priority and time stamp. For that reason, a database is the most efficient solution.
In general it is preferable to find a simple scheme with a uniform representation for the sensor samples. This can be done by storing the more sensor specific data in a serialized style, while the common properties, like time stamp and priority, can be represented as fields as usual.

Concurrent Access

Storing the samples in the database and removing stored samples for transmission purpose are asynchronous tasks. The way, the database access will be implemented, has to deal with the lack of support of concurrent access by SQLite databases.

Caching of Collected Data

As the database may be temporarily locked due to concurrent access, it seems reasonable to cache collected data in memory first. In general, sample processing must not slow down the sampling process.

Database Limit Overrun Strategy

Due to the intended high configuration potential of the framework and all its service parts, there is no way to guarantee a continuous pick-up service of stored samples by the transmission module. If the frequency of the incoming collected samples is higher as the delivery rate of the transfer service, the database size will grow until the size limit is reached. Thus, a strategy is needed, to respond to a size limit overrun.

[7] see http://www.sqlite.org/

There are several possible actions:

- queue the incoming samples and just wait a while (to sort out momentary transfer bottlenecks),
- delete older samples in the database (optionally priority dependent),
- notify the user about the problem (should be done in case of actions with consequences, like deletion),
- maybe stopping the whole service.

In general, an ordered combination of some of the actions would be the best strategy.

Persistence Layer Configuration Options

The following settings will be available:

- a possibility to disable persistent storage,
- a database size limit,
- a possibility to select a strategy, for the case that the database size limit has exceeded,
- necessary configuration options for the different strategy actions (e.g. the count of samples to delete).

2.1.4 Data Transfer

The transfer of collected samples will be done asynchronously. The core features for transmission have been defined by efficiency and and by saving resources.

Resource Preservation

To save resources, first a collection of samples with a configurable size is selected from the database and serialized into an XML file. In a second processing step, the files for transmission will be compressed into an archive.

Compression

The common archive types zip and jar will be supported.

Sample Selection

By definition samples will be selected by age and priority for transmission. More precisely, the samples with the oldest time stamp and highest priority will be removed first from the database.

Reliability

The archive is temporarily stored in the local file system for further processing. In case of transmission errors or lost connections the archive it will be processed in the next execution cycle. Samples, removed from the database, will not get lost that way.

Connection Strategy

By the restriction to Android API level 7[8], just two selectable options to connect to the internet do remain. It is possible to choose between a mobile service (GPRS[9] or UMTS[10]) or WLAN (Wifi).

Related connection strategies are

- use any available connection,
- use an available WLAN connection,
- use an available mobile connection or
- use an ordered combination of both options, favouring an available WLAN connection.

Protocol

It was agreed, that in the beginning just the HTTP protocol with basic authentication is supported. There will not be a configuration option for the transmission protocol. The protocol is extracted from the configured URL. An invalid or unsupported scheme in the URL will raise an error. The design has to take into account the support of other protocols.

User Notifications

As the transmission part is strong dependent from the configuration settings and the availability of an internet connection, the user should get proper notifications in case of errors (e.g. authentication or URL scheme errors).

Data Transfer Configuration Options

The following settings are intended:

- a possibility for the user to disable the sample transfer,
- an option to configure remote host settings: URL and authentication data, including user name and password
- additional transfer settings:
  - the transfer frequency in seconds,
  - a lower limit for the count of samples to transfer,
  - an upper limit for the count of samples to transfer,
  - a possibility to select the archive type,
  - a possibility to select the connection strategy.

2.1.5 Configuration Properties

Runtime Configuration

A configuration activity is needed, to allow the user to change all available configuration settings at runtime.

[8] the restriction to API level 7 is explained in the section *Target Platform*
[9] General Packet Radio Service
[10] Universal Mobile Telecommunications System

Default Configuration

A possibility to define default values for all the available configuration options is required. For the moment, the preferred solution is a configuration file. It would be nice to be able to limit the range of supported sensor devices as well.

2.1.6 Other Functional Requirements

Logging

From the developer point of view, some kind of logging is essential. Both for debugging purpose and to allow inspection of the service activities by the device owner. Logging must be done asynchronously too, to avoid the slowdown of other service parts. To display the log information, a visible control activity is needed. Having the option to configure several log level is recommended.

Application Structure

The Android application will consist of

- a service covering the discussed framework features,
- a control activity to explicitly start or stop the service and to display log informations,
- a preference activity for configuration purpose.

2.2 Non-Functional Requirements

2.2.1 Target Platform

As the Android device simulator does lack the possibility to simulate various sensors, e.g. Bluetooth, it is necessary to rely on tests running on real devices. Thus, the Android target platform was restricted to API Level 7 (Android Version 2.1 Eclair), the highest available OS Version for the first generation G1 smartphone, which was provided as test device by the institute.

2.2.2 Development Environment

The reference environment for development is based on

- Eclipse Helios (Versions 3.6.2) as IDE,
- Android Development Tool-kit for Eclipse Version 10.0.1,
- Android SDK[11] Version 10.

2.2.3 Source Documentation

A sufficient source code documentation is a fundamental part of development and may reduce the necessary familiarisation time for new developers. Therefore, a full Javadoc-style source documentation on class and package level is intended, to prepare a substantial HTML-documentation of the framework and its components.

2.2.4 Unit Test Coverage

Another important quality aspect is a large code coverage by unit tests. With regard to the limited development time, the effort for unit tests is limited to a feasible extent, with focus on the core functionality. To keep the size of the deployable application package small, a separate test project is required. Tests should be organized according to the frameworks package structure.

[11] Software Development Kit

3. Design

As mentioned in the introduction, the conceptual work in this project was fluent. The initial draft was a rough concept, including a basic application design, functional core modules and some other directly identifiable elements of the framework. During the development phases, the details of the affected modules have been worked out and the composition was improved.
This chapter does introduce the general framework composition, with some insights in the line of action as well as some arguments for special design decisions and concepts. The subsections will adress selected topics of the modelling process, without any claim to completeness. While some are more general, others are related to specific problems.
From now on, I will sometimes refer to the framework using its codename *"SDCFramework"*, with SDC as the abbreviation of Sensor Data Collection.

3.1 Modelling the Framework

The major part of writing a software framework is the provision of generic functionality, which can be used to easily create an application specific implementation. Some relevant aspects are

- functional separation,
- definition of interfaces and control flow,
- configuration options for adjustment,
- extensibility by specialisation, without modifying the core framework code,
- a defined default behaviour.

There are several approaches to implement a framework. However, the basic task is to construct decoupled generic modules consisting of components, which can be combined as needed, and to provide configurable structures to put components or modules together.
Thus, the two initial steps have been the decision for a concept of the android application and a functional composition of the framework service. Subsequently the initial package structure was defined.

3.2 The Android Application Concept

A key concern of this project was the implementation of a sensor sample providing Android application for mobile phones, implemented in the style of a extensive configurable modular framework. Therefore, the composition of the application was an important initial step.
An Android application does consist of one or more application components[1]. The available components are services, activities, content providers and broadcast receivers.
Broadcast receivers are used to respond to system-wide broadcasts. They do allow the capture of so-called *Intents*[2], wrapping actions or data from other activities or services.

[1] see http://developer.android.com/guide/topics/fundamentals.html
[2] Intents are passive data structures to hold action commands or data as content, for details refer to http://developer.android.com/reference/android/content/Intent.html

Broadcasting is a wide-used concept in Android. You will find many components in the framework using broadcast receivers. For example, the sample scanner types for Wifi and Bluetooth sensors.
Content providers do offer a possibility to manage shared application data. This concept was used for the first framework extension, to realize a data sink as sample source for a virtual sensor device.

Obviously the service is the right component for the core functionality of the framework. The major disadvantage of an activity is the dependence on the activity life cycle[3], especially on the foreground lifetime. A running activity, loosing the focus, will be at least paused by the system. As soon as it is not visible any more, it can be stopped or even worse, just be killed. But the framework has to guarantee the continuous provision with samples, independent of a visible user interface. In contrast to activities, the service life cycle does depend from the kind of start. There are two different concepts of interest for the framework service. At first a service can be started to run sticky in the background, until it is explicitly stopped. Furthermore, a service does allow bounded connections from other clients, influencing its life cycle. If a service was created through binding, it will be destroyed as soon as there is no more client connected[4]. It is intended to support both concepts. The binding concept does allow the framework service to act as sample provider for other applications[5] running on the same device. Whereas the explicit service start is used to implement a stand-a-lone version of the framework, using a control activity to start and stop the service.
The Android concept to store configuration data is based on the usage of so-called shared preferences[6]. To manage the visual parts of persistent stored preference data, the Android API provides a special activity type, the *PreferenceActivity*. There was no reason not to follow the Android concept. Hence, a separate activity to edit shared preferences is used to provide access to the service settings at runtime. It is attached to the option menu of the control activity.
The control activity can be regarded as graphical user interface of the service. On the one hand it does allow the supervision of the service, including the explicit start and stop and the access to the configuration settings, and on the other hand it can be used to monitor the service activities with the help of the displayed log information.
A permanent notification, to remind the user about the running service, is installed in the systems notification area, as long as the service is running. The selection of this notification will start the control activity, to grant a quick access to the user interface.

In summary, the framework application is composed of two activities and a service component[7]. Given that both activities are simple components, the focus of further considerations is on the service organisation.

3.3 The Service Organisation

Most of the required separate service modules are feature-related. They have already been identified in the requirements analysis phase. This section will explain the identification process for the final service modules from a rather functional point of view.

[3] see http://developer.android.com/reference/android/app/Activity.html#ActivityLifecycle
[4] as long as it is not started explicitly.
[5] the *SDCFrameworkDemo* project is an example for an application using the *SDCService* as sample provider.
[6] shared preferences do cover the file based storage and the access to configuration settings in the form of key-value pairs, see http://developer.android.com/guide/topics/data/data-storage.html#pref
[7] for details see 4.2.8 The Framework Application

3.3.1 The Core Modules

The required features of sampling, persistent storage and data transfer, did define the first three core modules of the service:

- a **device management module** providing sensor samples as well,
- a **module for collection and persistent storage** of provided sensor samples and
- a **module covering the transmission part**, including selection of relevant stored samples, packaging and compression as well as the final transfer to a remote host.

Let me provide some background information about the decomposition. Regarding the storage and the transfer feature, the planning of independent modules is reasonable and does follow the framework concept of a functional separation with a defined control flow. The same is also true for a separate device module.
A very early decision was, to use the *Observer* pattern[8] to completely decouple the device dependent parts from other functional modules. Thus the device management was planned as separate module with a central access component, observable for samples and hiding the details of the internal device representation and the device specific sensor data.[9]. The *Observer* pattern does provide a common solution to be notified about sample data, respectively sample events, without a direct access to the source component. It does basically allow, to loosely attach the event observers of one module to the observable event sources of another one. In this context, it was necessary to place the sampling feature in the device module as well. An alternative solution would have been a further separation of device management and sampling, but I found no reasonable argument to do so. It turned out, that the way to scan for sensor samples does strongly depend on the sensor type. In the final architecture, a sensor device is associated with an appropriate scanner component, which is responsible for the sampling task. Device and scanner, both are closely related to each other, so why separate it?

In some kind, we can regard the composition of the three core modules as layer architecture.

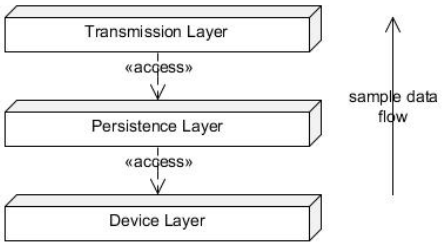

Figure 3.1: Module Layers and Sample Data Flow.

The persistence layer, residing above the device layer, is acting as intermediate layer for the transmission layer. All layers do just depend on its direct predecessor, while the sample data flow is directed upwards. The access to the device layer is based on observation, while the exchange between the persistent and the transmission layer takes place via a detour over the database.

[8] see *Generic Implementation of the Observer Pattern in the Architecture Overview*
[9] of course, an appropriate sample abstraction was part of this strategy as well

3.3.2 Other Service Modules and Components

In addition to the core modules, three more service components have been identified :

- a **logging component**,
- a **broadcasting module** and
- a **preference management module**, covering the aspects to default settings and runtime configuration.

While the logging part of the service was regarded as an utility component, the preference management was planned as separate modules as well. First of all, to separate it from other modules and to hide specific details about the storage of configuration settings.
You will find an additional module for broadcasting listed, which can not directly be deduced from the requirements. The idea, to enhance the service by the feature to broadcast samples and log-data in form of Android *Intents*, did arise while planning the observable feature for the device module. The observer concept does allow to attach more than one sample observer to the device layer. Against the background, that the service is a sample provider, broadcasting is an efficient solution to distribute samples to other applications. In this way, an activity can receive provided sensor samples by simply using a broadcast receiver. Another application area for the broadcast feature is the provision of log information. With an observable *Logger* component, log events will be published in kind of *Intents* by the service, while the service control activity can use a broadcast receiver to capture the log information for display purpose.

3.3.3 The Interaction of the Service Components

The figure below displays the interaction of the identified service components, respectively the modules. It does illustrate the intended data flow and the planned access scheme, with regard to sensor samples, log information and preference changes.

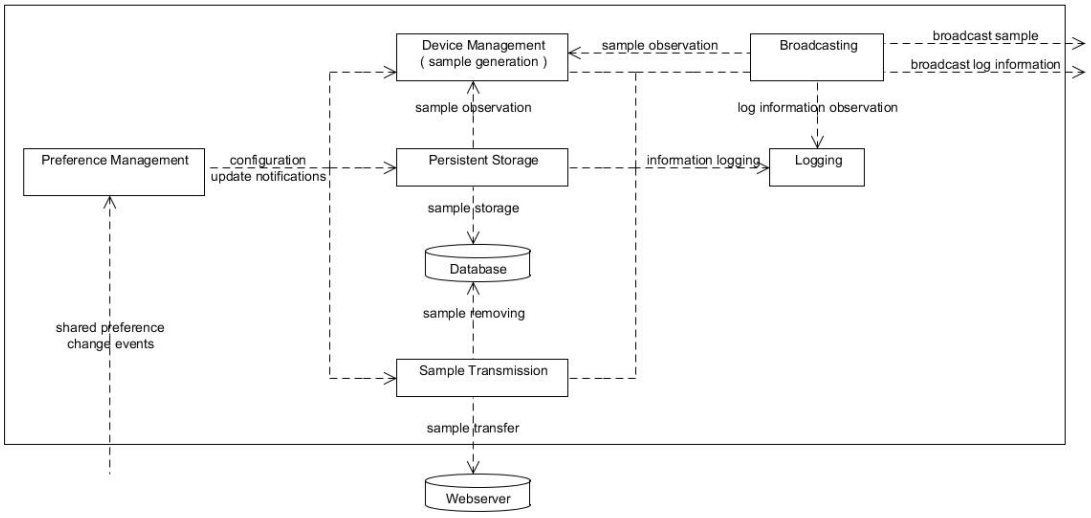

Figure 3.2: Illustration of the Interaction of the Service Components.

It was intended to allow the separate deactivation of the persistent storage and transmission feature. According to the layered architecture, the deactivation of the storage module does indirect affect sample transmission, since no more samples will be collected and stored in the database. This may be an interesting option to allow the transfer service to empty the database, if the database size has exceeded.[10]

[10] not covered by a database full strategy yet

3.4 Package Design

The package design is based on the concept of module isolation. All the module-related classes do reside in a separate package. Each module package has a sub-package, containing all the interfaces and related types for external access. The sub-packages have been named *facade*, even if this concept does not really fit the *Facade* design pattern. In some packages, there are management components, singletons and abstract types, which are still imported into other modules. At least, this is a first approach of a structural separation between the module interface and internal components.
A utility package does hold useful tool classes, components like the *Logger* or generic types like the *Observer* implementation[11]. Other packages are intended to group related classes together, for example the application or service components. You will find the final package structure, as well as some insights in package details, in the chapter *Architecture Overview*.

3.5 Serializable Data

Initially, there have been two data types in the framework, requiring serializability as feature, the sensor samples and the default framework configuration parameters.
While planning the transmission module, the question did arise, how to determine the origin of the transmitted samples at server-side. The Android class *Build*[12] does provide static access to specific system information, like the current build, the device name or its manufacturer. A third serializable data type for this specific device information was introduced at this point. Its XML representation is transmitted together with the serialized sensor samples, in a separate file.
XML as preferred format for data exchange, has already been defined in the requirements phase. The Android API level 7 does support SAX[13] and in parts[14] *DOM*[15].
SAX parsing is notification based and requires to keep track of the object data, while the parser runs through the document. In contrast, a DOM parser does read the full document into an object-oriented memory representation, which can be queried for the structured content. A SAX based solution may be faster and does require less memory, while DOM is more flexible, allowing the manipulation of the underlying XML data as well.
In general there have been two options. Either to write own helper classes, utilizing the serialization support provided by the Android API, or to use a third-party library for XML parsing. Due to the large extent of this project, I did prefer the third-party library approach.

Criteria for a third-party XML parsing library have been:

- a sufficient parsing speed and a fair size,
- minimal efforts in attaching it to the framework and realizing the data serialization feature,
- suitability for mobile platforms, especially Android.

There are many common libraries available for XML serialization, but the majority is too heavy-weighted for a mobile platform or does depend on features not supported by the Android API, at least not by the selected API level.

[11] see *Components of the Package Util* in the *Architecture Overview*
[12] see http://developer.android.com/reference/android/os/Build.html
[13] *Simple API for XML* see http://www.saxproject.org/
[14] with API Level 8 the *DOM* support was improved by adding the *Java XML Transformer API*, see http://developer.android.com/reference/javax/xml/transform/dom/package-summary.html
[15] the W3C *Document Object Model* see http://www.w3.org/DOM/

Based on some web research about Android and XML serialization, two libraries have been short-listed for evaluation: The *XStream*[16] library and the *Simple*[17] XML Framework. It turned out quickly, that *XStream* was not fully supported on Android[18], while the *Simple* framework was actually optimized for Android in its recent version 2.5.3[19].
So what is the advantage in using the *Simple* framework? *"Simple is a high performance XML serialization and configuration framework for Java. Its goal is to provide an XML framework that enables rapid development of XML configuration and communication systems. This framework aids the development of XML systems with minimal effort and reduced errors"*[Sim11]. A rapid prototyping did reveal, that this framework does really keeps the promise made by its name. Making a type serializable, is simply done by placing annotations in the class file (to define the XML type schema). A provided *Persister* component, with generic methods to read from and write to files or streams, is used for the serialization and deserialization task. To maximize the runtime performance, it is necessary to use just one persister instance repeatedly[20]. *Simple* is a robust and easy to learn framework for XML serialization. To speak from my own experience, it runs stable and fast enough on the Android platform.
All classes and types related to serializable data are located in the package *data* or below. A static class does act as a proxy for the global *Persister* instance. It is used by the serializable types to create its own XML representation. The pure Java types are hold in a separate subpackage, to allow the deserialization of sensor data from XML files in pure Java projects using the *Simple* library. A detailed description of the *data* package is given in the section *Serializable Data* of the architecture overview.

3.6 Independent Sample Data Representation

From the perspective of module isolation, it is quite important to keep the persistent storage and transmission modules irrespective of device specific details. Hence, the sample representation has to be a proper abstraction of sensor samples, so that it can be used by the sample processing modules without knowledge about the device specific sensor data.
There are three properties, all samples do have in common: the device identifier, a time stamp and the sample priority. Beside these common attributes, a sample holds device specific sensor data. The sample representation in the framework follows this structure. Beside the common properties, it has a reference to an interface type for device specific sensor data. At runtime, the interface grants access to the XML representation of the concrete type linked to the data property.
The database scheme for persistent storage does reflect the sample representation. All samples are stored in a single table. The table does consist of six columns. One column for a unique row identifier, three columns to hold the common sample properties and the last two columns, to store the XML representation of the device specific sample data together with the name of the concrete type. The name of the type must be stored to be able to recreate an object from the XML representation of the device specific data.[21]
However, the final concept does allow the introduction of new sensor types with device specific data, without affecting the persistent storage or transmission module.

[16] see http://xstream.codehaus.org/
[17] written by Niall Gallagher and published under the Apache Licence, see http://simple.sourceforge.net/
[18] much later I did found a special patched *XStream Version 1.2.2*, published by Markus Junginger at http://jars.de/java/android-xml-serialization-with-xstream, but the final decision for *Simple* XML was already made and I did not further investigate it.
[19] the recent version, when planning the serialization part of the framework in April 2011.
[20] due to the dynamic way to build the class schemata using Java reflection, compare http://stackoverflow.com/questions/6074353/xml-with-simplexml-library-performance-on-android.
[21] the *Persister* class in the *Simple* framework can read the contents of an XML stream and convert it into an object of a schema-compatible type.

3.7 The Configuration Concept

The configuration settings will be stored as shared preferences[22] for the framework application. To hide the internal knowledge about the kind of persistent storage, a central access component for preferences and configuration settings is provided.
In addition, the representation of configuration settings is split in two types. Regarding a specific setting, the related preference type does cover the Android representation with a unique identifier as key and a default value, while the configuration type is used to hold the current configuration value. A preference type can read the current configuration value from a *SharedPreferences* instance and convert it to the corresponding configuration type. Configuration types are also used to distribute configuration updates in the framework.

As central management component, the application preference manager is responsible for:

- the access to shared preferences, including
  - the determination of the current value for given preference type,
  - the listening for shared preference changes,
  - the notification of observers about shared preference changes in the kind of configuration change events.
- the access to configuration values, including
  - the observer registration for configuration change events,
  - the direct access to current configuration settings (without knowledge about the related preference type).

While the access to preferences is first and foremost offered for the preference activity, the access to configuration values is provided for the service modules.
The application preference manger is observable for different types of configuration change events. Notifications will be given in the form of an event object, which does hold an instance of the related configuration type with the updated value.[23]
A central observation for configuration changes is installed in the service manager, which is the supervisor for all other service components and responsible to distribute the incoming configuration updates to the affected modules.
Beside the handling of runtime configuration changes, a possibility to preconfigure default values is required. This is realized by the provision of an XML-encoded configuration file in the projects asset folder. A special configuration management component[24] is responsible to read the default configuration from the file, and to update the related preference defaults in the application preference manager.

3.8 Aspects of Parallel Processing

In the beginning of the project, the three core features of the framework have been identified as parallel tasks. The need for an asynchronous execution of sampling and persistent storage, did result from the requirement to provide sensor samples in a configurable frequency. Thus, the task for persistent storage of collected samples is not allowed to slow down the sampling process in any way. The transfer service is independent from the other tasks by its function, and has to be executed in an own configurable frequency.

[22] see http://developer.android.com/guide/topics/data/data-storage.html#pref
[23] for configuration types and events compare 4.2.3 *Configuration Settings*
[24] the *SDCConfigurationManager* located in the package *preferences*

Regarding the sampling task, in principle we are talking about more than one parallel task. First each sensor device has its own sample frequency and, in addition, there are quite different ways to obtain sensor samples. Finally, each device needs its own scanner for the periodic sampling task.
The logging of errors, warnings or other information, is a parallel task as well, justified by the criterion, not to slow down other threads using the logging component.
As abstract base class for thread-based components, a special worker thread type[25] is available in the package *util*. The worker thread implementation is based on a proven concept, already used in other Java projects. It does provide an abstract implementation for a long-living daemon thread, supporting a special working state. If the thread is working, it does execute an abstract work method in its main loop. As soon as the work is stopped, the thread will not terminate, but is sent to sleep and can be restarted again. Also, a basic exception handling does prevent undocumented crashes.
The initial intention was, to use a worker thread instance for any kind of parallel task. But while it is necessary to have an extended control over the asynchronous logging, transfer and storage tasks, this does not generally apply for all the sampling tasks.
For some specific sensor types, sampling does just require the device scanner to take a sample in a given frequency from its corresponding device. This can be reduced to a simple timer task. In contrast, there are devices like Bluetooth and Wifi, requiring the observation of a cyclic triggered device scan process. For the latter a thread-based solution was favoured.
An ideal solution for the simple timer tasks would allow asynchronous task execution, without having an extra thread spawned for this purpose. From this point of view, there was no benefit in using the common classes *Timer* and *TimerTask*, as each timer has its own thread[26]. But the Android API does also provide a so-called *Handler* class. Amongst other things, a handler can be used for the delayed execution of scheduled runnables.
A handler is associated with the message queue of the thread to which it belongs. Thus a runnable, posted to a handler for delayed processing, is finally executed using the threads message queue. In summary, the usage of a handler is a quite efficient and resource preserving solution to realize a simple timer tasks. For the *SDCFramework*, the handlers will be associated with the main thread of the service.
As a result of these considerations, the *SDCFramework* does support two different concepts for asynchronous sampling. The related component for thread-based sampling is the *SampleReceivingDeviceScanner*, which is using a worker thread to observe repeatedly triggered device scans. The *Handler*-based solution is composed of two components, the *SampleTakingDeviceScanner* and the runnable *SampleTakingTask*[27].

3.9 Usage of Design Pattern

One of the hardest problems in object-oriented software design is the choice of a proper model. In general there is a large number of applicable solutions to model a specific software problem.
Design patterns[28] are generic reusable template solutions to solve many common problems and they are well known as proven paradigms for software design. Thus, the usage of design patterns was favoured. For frequently used patterns, like the *Observer* pattern, a generic implementation is provided.

[25] for details see section *The Worker Thread* in 4.2.1 *Components of the Package Util*
[26] see http://developer.android.com/reference/java/util/Timer.html
[27] for details of device scanning see 4.2.4 *Sensor Devices and Scanner*
[28] a good reference for design pattern is [GHJJ96]

The following list will give some examples for the usage of specific patterns in the framework architecture:

- **Observer**
  Used to decouple functional modules in regard to data exchange or to realize event handling in general (e.g. for sample collection, to observe preference changes, log events or network connections)[29].
- **Strategy**
  Used to implement different strategies to react on a database limit overrun[30] or to establish transfer connections[31],
- **Chain of Responsibility**
  Used to implement ordered chains of different strategies[32],
- **Visitor**
  Used to apply configuration updates to sensor devices [33] ,
- **Abstract Factory and Builder**
  Used to create sensor devices [33] and to build strategy chains based on the current configuration settings.
- **Singleton**
  Used to model global unique component instances, e.g. the *Logger*[34] or the *SensorDeviceAvailabilityTester* [33] .
- **Facade**
  Inspiration for the idea to separate of types and interfaces from the implementing module components.

[29] see 4.2.1.2 *Generic Implementation of the Observer Pattern*
[30] compare 4.2.5 *Sample Collection and Persistent Storage*
[31] compare 4.2.6 *Sample Transmission*
[32] see 4.2.1.6 *Generic Chain of Responsibility Implementation*
[33] compare 4.2.4 *Sensor Devices and Scanner*
[34] see 4.2.1.4 *The Asynchronous Logging Concept*

4. Architecture Overview

In contrast to the design chapter, the architecture overview is intended as introduction for developers addressing concrete implementation details of packages and components. After a short illustration of the final package structure, we will take a selective look in the details of some packages.
A complete documentation of all available packages and the contained components would go beyond the scope of this project report. Those who are interested in more details are invited to have a deeper look into the source code and the related Javadoc documentation[1], that does contain some class dependency graphs as well.

4.1 The Package Structure

The final package structure of the *SDCFramework* project is shown in the following below. The notes do describe the general package contents.

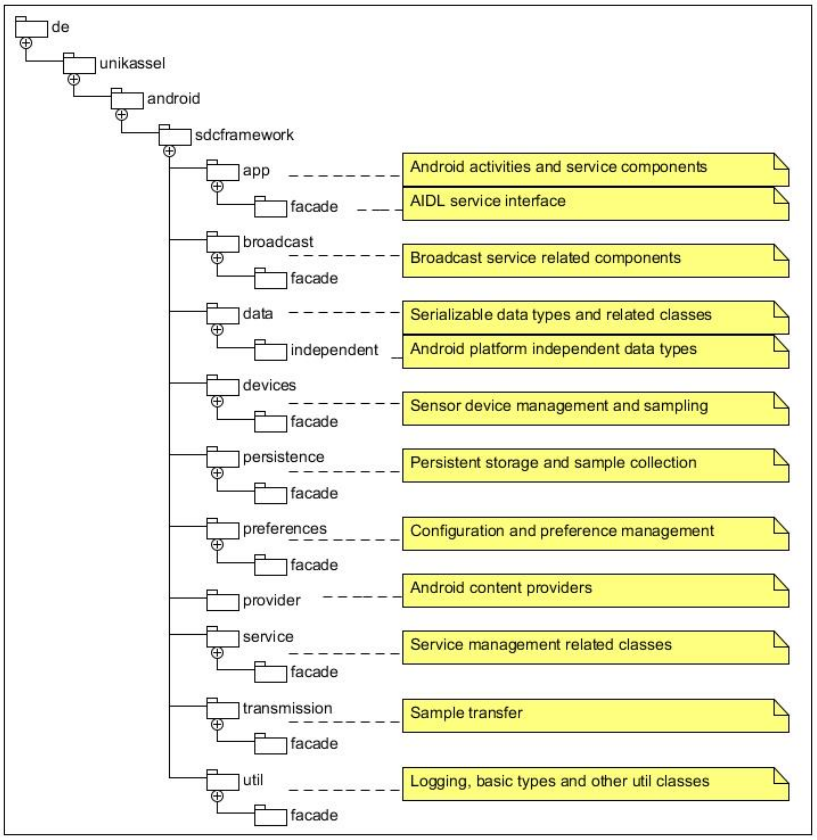

Figure 4.1: The final package structure.

While all the Android application related classes are located in the package *app*, the functional modules of the service component are implemented in the packages *broadcast*, *devices*, *persistence*, *transmission* and *service*[2]. The remaining packages do cover more general aspects of the framework, like serialization or preference management.

[1] the HTML documentation can be generated by executing an ant script in the projects *doc* folder
[2] for background information see chapter 3.4 *Package Design*.

4.2 Package Insights

This section will give a selective introduction to the content of relevant packages. The focus is on the service modules and the related components in other packages. For a better understanding, the subsections are built upon each other, starting with the basic concepts and components.

The following aspects will be covered:

- the most relevant components located in the package *util*,
- the contents of the package *data*,
- configuration related classes in the package *preferences*,
- a detailed look into the the Android service, affecting the packages *devices*, *persistence*, *transmission*, *service* and *app*.

To illustrate the package details and the contained components, I will primarily rely on UML class diagrams. They are intended to display the relevant types and classes for a given topic, together with the most important relations and dependencies in this context. In other words, the level of detail in the diagrams depends on the current point of view.

4.2.1 Components of the Package Util

Beside the Logger component and several utility classes[3], the package *util* does contain basic types, sometimes implemented generic, which are frequently used to model components in other packages.

4.2.1.1 The Worker Thread

The worker thread concept does serve the needs of asynchronous tasks in the framework.

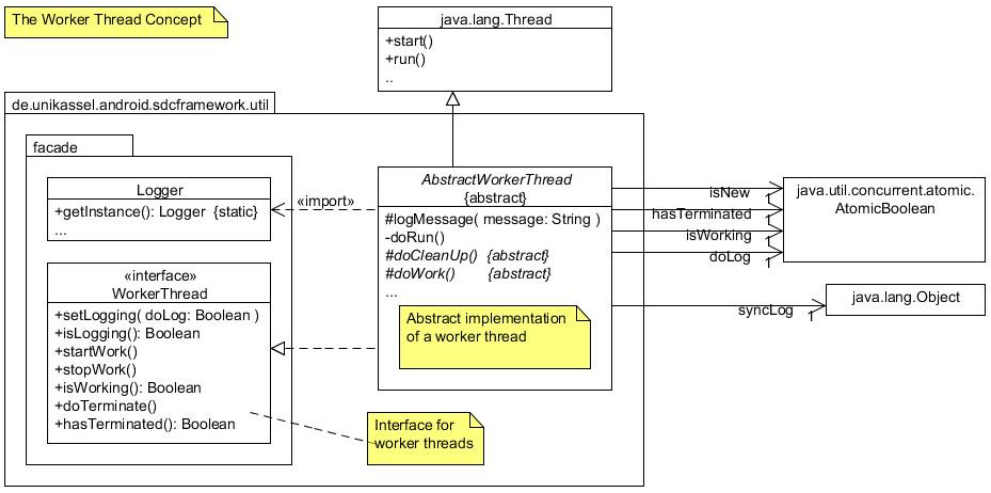

Figure 4.2: The base class for framework threads

Beside a standard behaviour for the thread start and termination, the abstract worker thread implementation does provide the feature to stop and restart the thread as often as necessary. Runtime exceptions will be catched and automatically sent to the logger[4]. Also

[3] e.g. for compression and file access
[4] the logging concept is explained in the section *The Asynchronous Logging Concept*

the running state is logged in case of changes. The abstract *doWork()*-method is called periodically in the running loop, as long as the thread is in working state. Stopping the thread does mean sending it to sleep.
Asynchronous tasks can easily be implemented by extending the *AbstractWorkerThread* class. In the majority of cases the implementation of the abstract methods should do it.

4.2.1.2 Generic Implementation of the Observer Pattern

As already mentioned, the *Observer pattern*[5] is the most frequently used design pattern in the framework. A general reason to use the observer pattern is object decoupling. An observable object should be able to notify other objects (the observers) about changes (in our context so-called observable events), without having explicit knowledge about the concrete observer types.

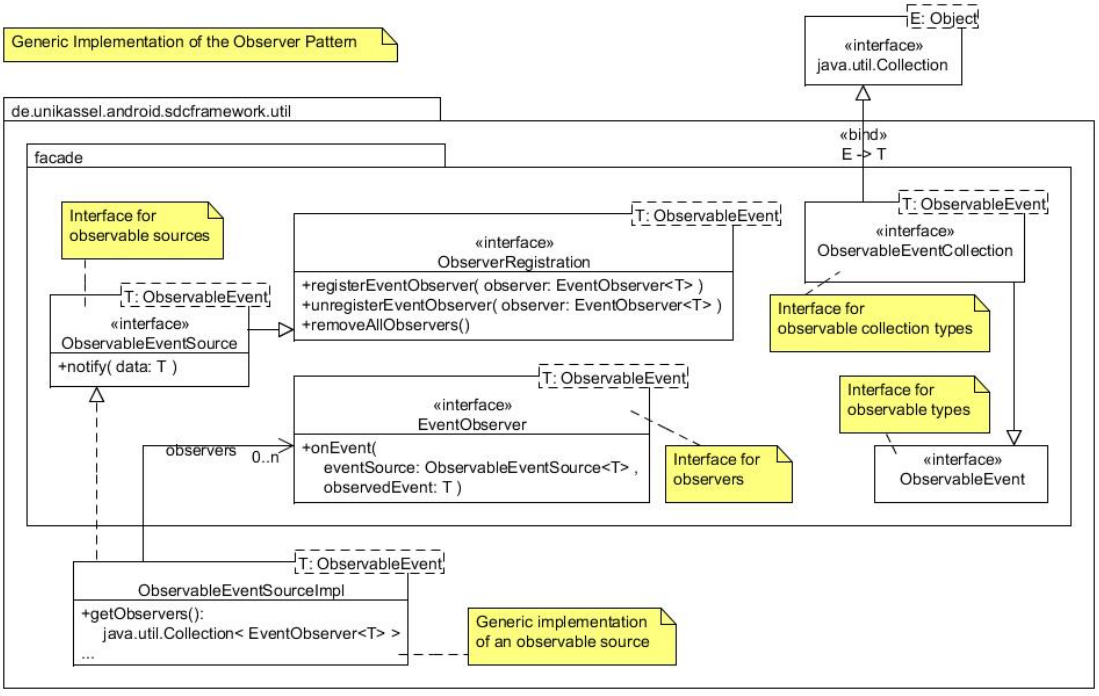

Figure 4.3: The generic observer pattern implementation

A generic implementation was recommended to build other components upon it. Any observable event type has to implement the *ObservableEvent* interface. For the modelling of observable collection types, there is a special interface called *ObservableEventCollection*. While observers have to implement the *EventObserver* interface for the observed event type, observable components can simply extend the generic implementation *ObservableEventSourceImpl* and bind it to the concrete event type.

4.2.1.3 Event Collection and Dispatching

The motivation for the event collection and dispatching concept was the need of asynchronous event processing. Say we have an observable event source running in a thread, which does produce data in the form of events. Whenever an event is generated, it does notify its observers by calling the *notify*-method. This method is invoking the *onEvent* method of all registered observers. Any event processing, done by a registered observer in

[5]the Observer pattern is explained in [GHJJ96, p.287]

its event handler, is executed in the context of the notifying thread, potentially slowing down the thread execution. To avoid the loss of performance, it is necessary to reduce the observer-sided event processing time to a minimum.
A reasonable approach to do so, is simply caching the events in a queue and running another thread to process the queued events asynchronously. This task is modelled with two components, the event collector, covering the caching part, and the event dispatcher, covering the part of asynchronous processing of queued events. Both components are observable event sources itself, but for different types.

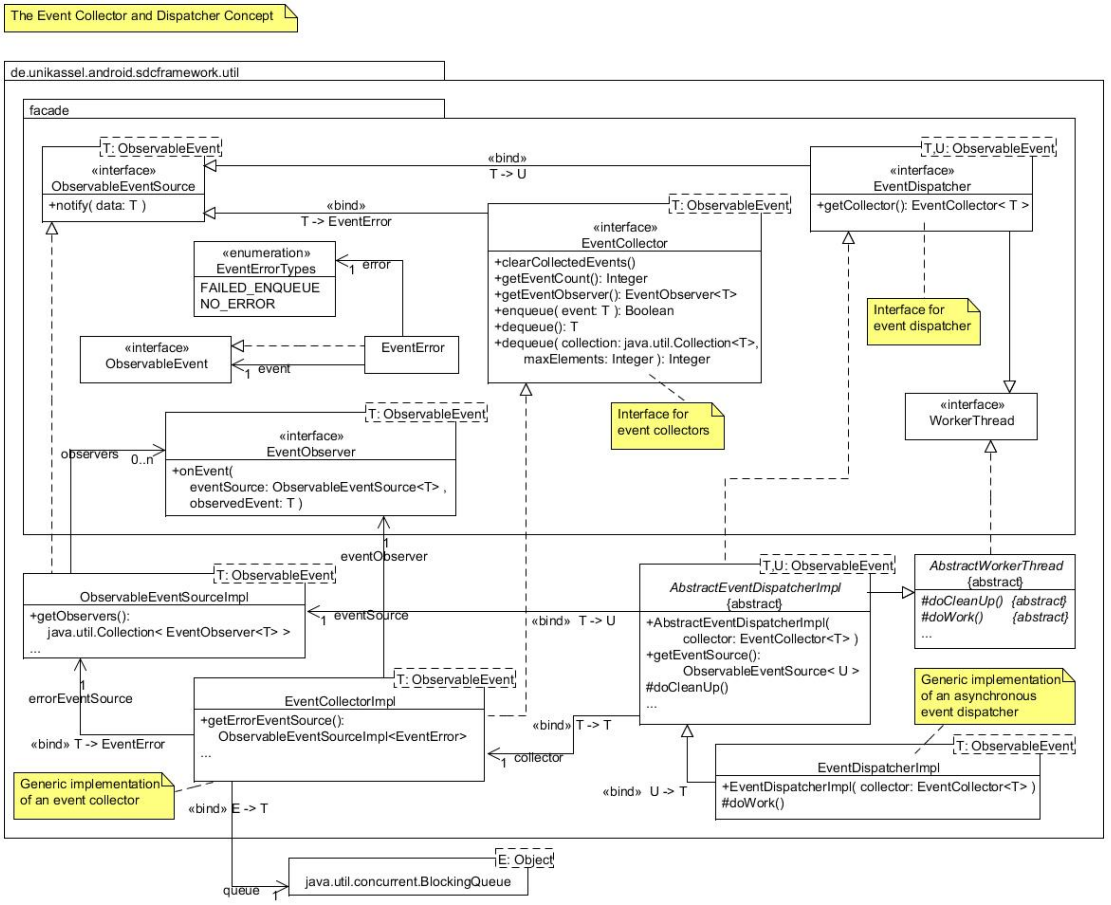

Figure 4.4: The generic event collector and dispatcher components

The generic event collector implementation (*EventCollectorImpl*) has an observer to observe the original event source for the generic event type. Observed events will be stored in a thread safe blocking queue. The access to the queued events is covered by the implemented *EventCollector* interface methods. As the caching is done in memory, errors may occur in event processing. To handle this event errors, the event collector is observable for the *EventError* type. The implementation of the *ObservableEventSource* interface for the *EventError* type, is done by delegation to an internal bounded-instance of the *ObservableEventSourceImpl*.
The generic event dispatcher implementation (*EventDispatcherImpl*) is finally an extension of the *AbstractWorkerThread*. It is observable for *ObservableEvent* types. The implementation of the *ObservableEventSource* interface for the generic event type is done by delegation. An instance of the *EventCollectorImpl*, bounded to the generic event type as well, is used as event queue source. The implementation of the abstract *doWork*-method does dequeue events from the event collector to notify registered observers about it.

You may wonder, why the intermediate class *AbstractEventDispatcherImpl* was not mentioned. This is just an abstraction layer, to allow the modelling of event dispatcher components with different types for the collected and the dispatched events.

4.2.1.4 The Asynchronous Logging Concept

Now as we are familiar with the worker thread, the generic observer and the dispatcher concept, it is time to explain the implementation of asynchronous logging.
Log information is represented by the *LogEvent* type in general. This type does implement two interfaces. While we know the *ObservableEvent* interface will make this type observable, the *BroadcastableEvent* interface does allow its conversion to a broadcastable Android Intent. A special attribute of the *LogEvent* type is the log level. It is used to limit the extent of broadcasted log events, by simply ignoring events with a level below the configured global log level.

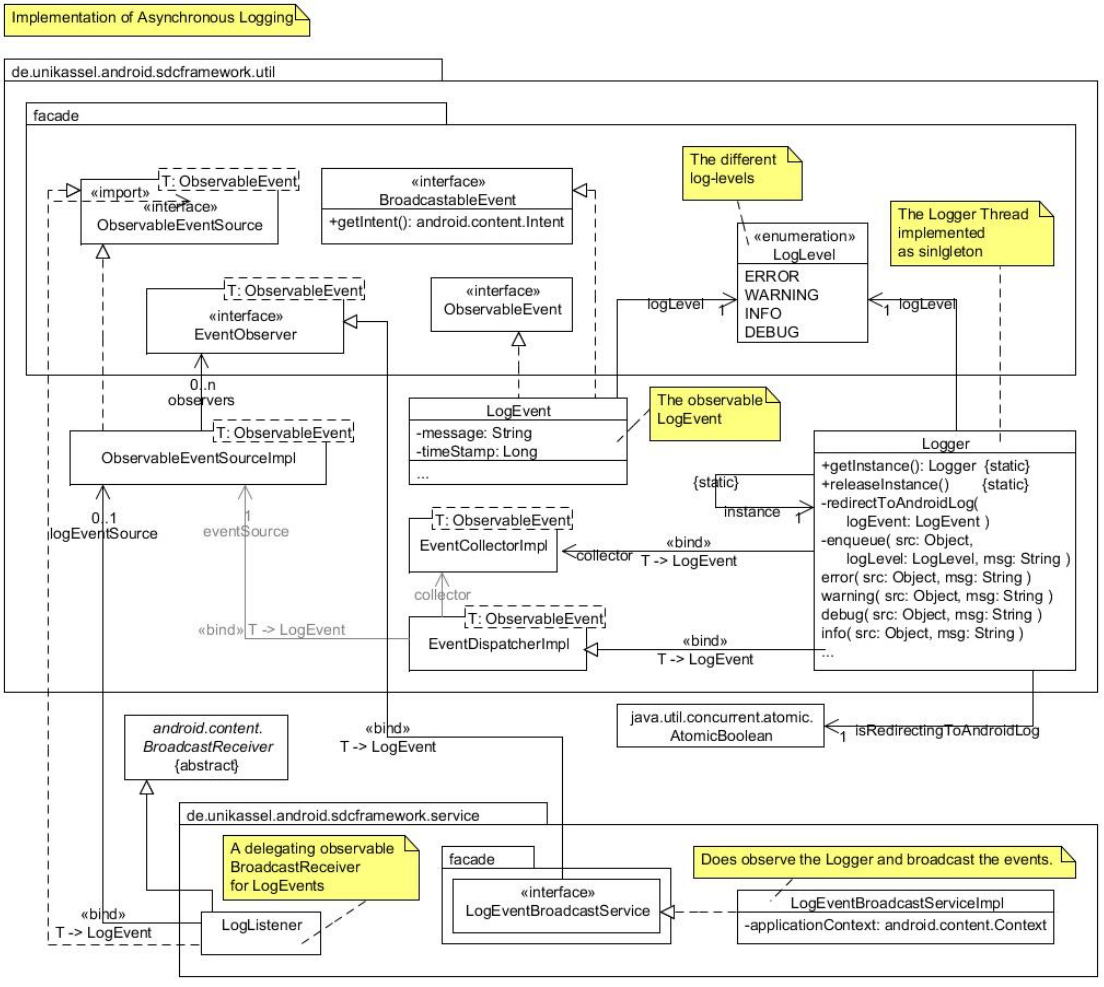

Figure 4.5: Components related to asynchronous logging

The *Logger* is an extension of the generic event dispatcher implementation, bounded to the *LogEvent* type and implemented as Singleton[6]. The implementation of the provided log methods, one for each available log level, does just enqueue a *LogEvent* representation of the message in the collector queue. The inherited *doWork*-method is overridden to

[6] the Singleton pattern is explained in [GHJJ96, p.157]

implemented a more special event dispatching. On the one hand the dequeued log event is redirected to the Android Log, on the other hand registered observers will be notified as usual. From inside of the framework, logging is simply done by invoking one of the log methods on the global *Logger* instance.
There are two related components, located in the package *service*, which does extend the basic logging concept by realizing the broadcast features. The *LogEventBroadcastServiceImpl* is an observer for *LogEvent*s. As the *LogEvent* type is broadcastable, it can transform each observed log event to an Android Intent[7] and broadcast it in the system. Any interested *BroadcastReceiver* can receive it. The receiver part is realized by the second component, the *LogListener*[8]. Due to the implementation of the *ObservableEventSource* interface, it is observable for the *LogEvent* type. Also it does extend the abstract Android *BroadcastReceiver*[9] class, to be able to receive broadcasted intents. The implementation of the inherited abstract *onReceive*-method does filter for intents of the type *LogEvent.ACTION*. Registered observers of the listener will be notified about observed log events.

4.2.1.5 Android Life Cycle Dependent Objects

Many components in the framework are influenced by the Android service life cycle[10]. The interface *LifeCycleObject* does define the necessary callbacks for such components. Amongst others, the asynchronous running management components of the core modules have to implement this interface.
Application state changes are propagated top down in the framework, from the *SDCService* over the *ServiceManager* down to the other life cycle dependent components. They do cause a call to the corresponding interface method: *onCreate*, *onResume*, *onPause* or *onDestroy*.

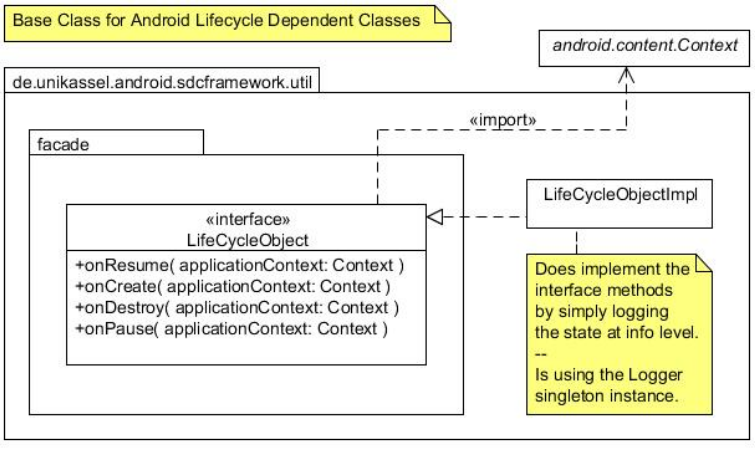

Figure 4.6: The basic type for Android life cycle dependent objects

The class *LifeCyclObjectImpl* is a basic implementation of the interface, simply logging the method calls. Thus life cycle dependent classes can either extend this base class or implement the interface.

[7] see http://developer.android.com/guide/topics/intents/intents-filters.html.
[8] e.g. the *SDCServiceController* activity does use this listener to receive the broadcasted log events.
[9] see http://developer.android.com/reference/android/content/BroadcastReceiver.html.
[10] see http://developer.android.com/reference/android/app/Service.html#ProcessLifecycle

4.2.1.6 Generic Chain of Responsibility Implementation

The classic *Chain of Responsibility pattern*[11] is modelled with two components, the so-called *client* and a processing element. In the generic implementation for the framework environment, the generic type does represent the *client* whereas the *ChainWorker* does define the interface for processing element types.
The class *AbstractChainWorker* is a generic abstract implementation of the *ChainWorker* interface. It does realize the *Chain of Responsibility pattern*, as a composition of linked *ChainWorker*s, each knowing its direct successor. An *AbstractChainWorker* does delegate the client request (represented by the *doWork*-method) to its successor, in the case the processing has failed. A class extending the *AbstractChainWorker* has to implement the inherited abstract method *process*, to realize the intended form of request processing for the bounded client type.

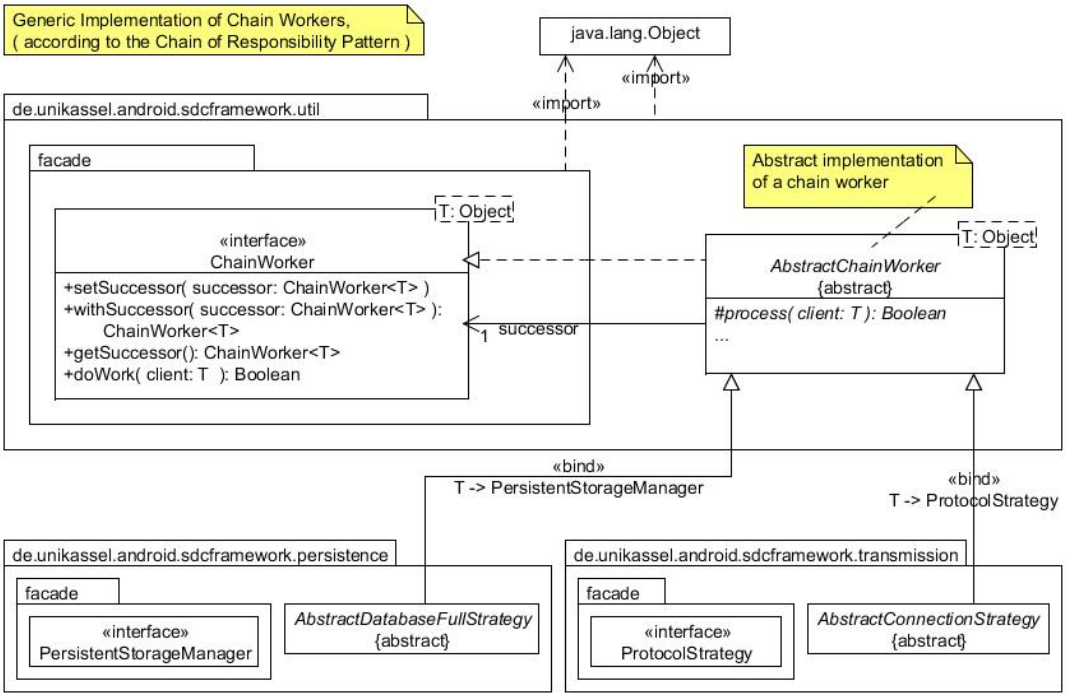

Figure 4.7: The generic implementation of the Chain of Responsibility pattern.

Regarding the framework, in all cases of application, the generic *Chain of Responsibility* implementation is used in combination with the *Strategy pattern*. That way, it is possible to apply different strategies in a specific order, while trying to solve a problem (like the database limit overrun).

4.2.1.7 Utility Classes for File Operations

The package *util* does also contain utility classes for file operations. While the class *FileUtils* is providing static methods for standard file operations (e.g. read, write and copy), the *FileCompressor* component can be used to create archive files. To support different archive types, the *FileCompressor* does use a *CompressionStrategy* instance.

[11] the Chain of Responsibility pattern is explained in [GHJJ96, p.410]

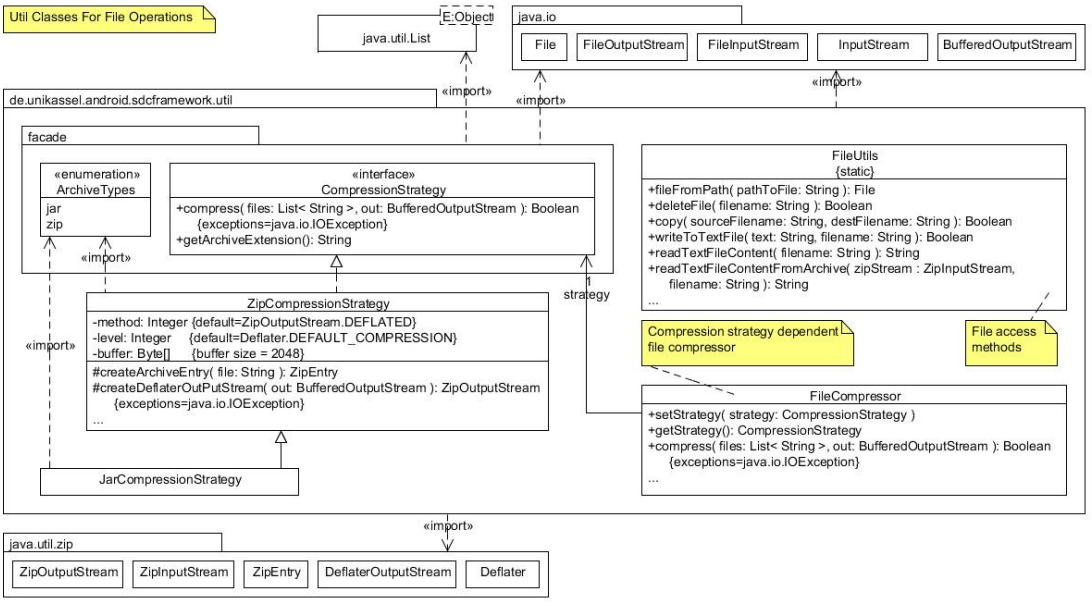

Figure 4.8: Utility components related to file operations

Actually jar and zip compression is supported, implemented in the *ZipCompressionStrategy* and *JarCompressionStrategy* classes. The jar strategy is regarded as specialisation of the zip strategy.

4.2.1.8 Asynchronous Sample Observation

Once it became apparent, that the usage of an event dispatcher for the sample observation task is inexpedient, a special base class for life cycle dependent and asynchronous working sample observers was composed of the existing types. It will be introduced now, even though the *Sample* data type is covered later. For the moment it is enough to know, that the *Sample* type is the framwork internal representation for sensor samples and does implement the *ObservableEvent* interface.

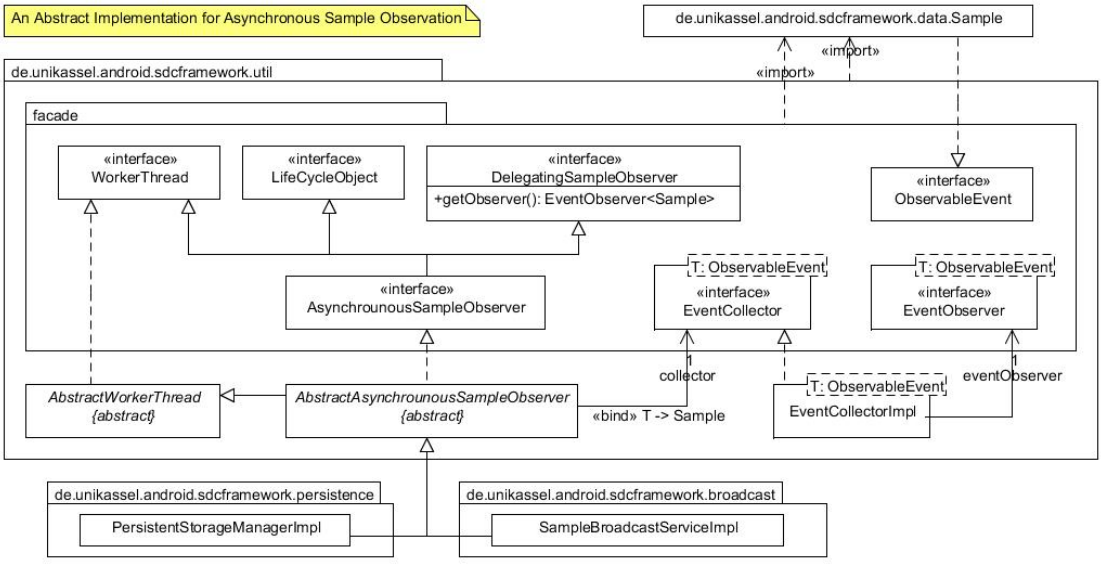

Figure 4.9: Asynchronous sample observation for life cycle dependent components

The interface *AsynchrounousSampleObserver* defines the visible behaviour. An asynchronous sample observer can handle life cycle changes, runs asynchronously as worker thread and can be queried for an observer instance bounded to the *Sample* type.
An abstract implementation is the *AbstractAsynchrounousSampleObserver* class. It does extend the abstract worker thread implementation and owns an event collector to observe and collect samples. At runtime the collector property is linked to an *EventCollectorImpl* object which is bounded to the *Sample* type. The default life cycle behaviour defines, that the thread starts working on resume and stops on pause, whereas on destruction the thread will terminate. The *doCleanUp()* method of the *WorkerThread* interface is implemented to empty the collector queue. Extending types, like the persistent storage manager and the sample broadcast service, have to define the cyclic executed work and may override the life cycle behaviour.

4.2.2 Serializable Data

Classes and types related to data serialization are located in the package *data* or below. This does include the sensor sample data representation, the configuration entries for the default configuration file and the device information[12]. All the pure Java classes, which are completely independent from the Android library[13], have been extracted to the subpackage *independent*.

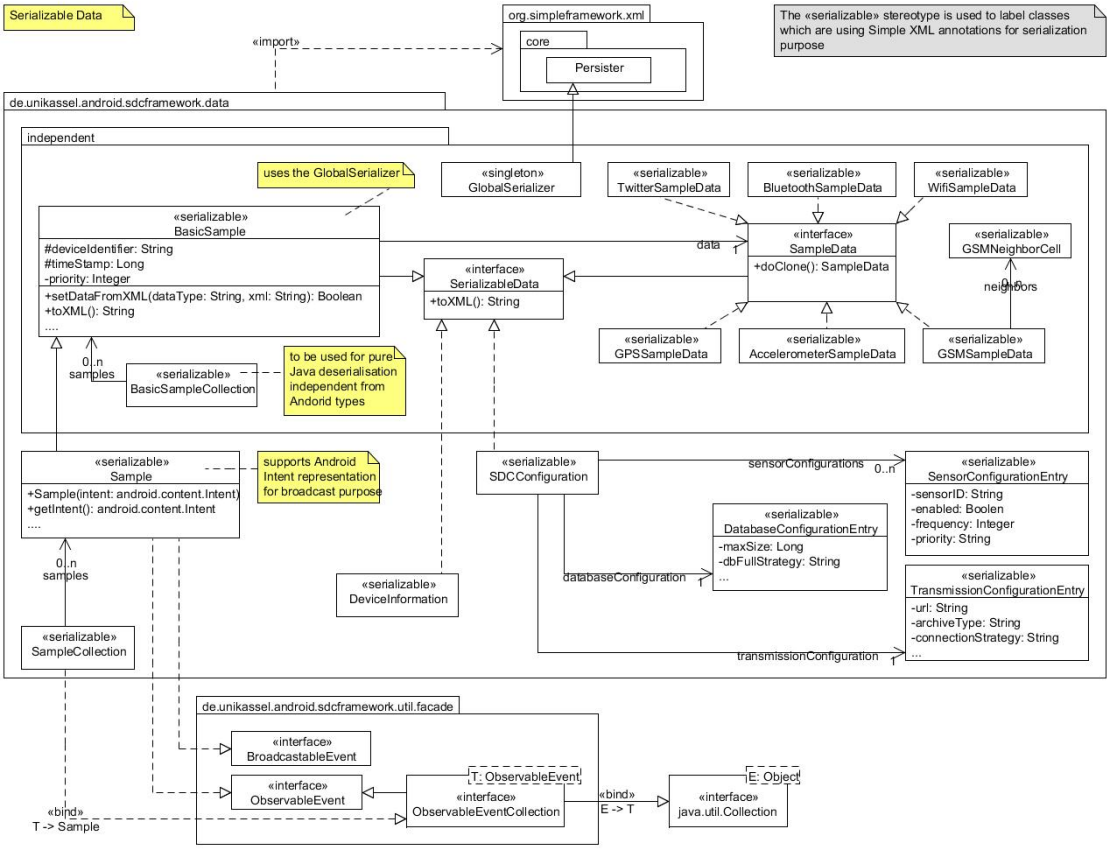

Figure 4.10: The package *data* and related classes

To cover the serialization part, the *Simple XML* framework in version 2.5.3 is linked[14].

[12] the device information is used to identify the origin of transferred samples.
[13] to be used to deserialize transferred sensor samples in a pure Java project.
[14] compare 3.5 *Serializable Data*

Serializability of objects is attained by the placement of annotations in the corresponding classes. A *Persister* object is used to parse or create the XML representations of objects. The static *GlobalSerializer* class does hold the global *Persister* instance. Any serializable class, implementing the *SerializableData* interface, is using the *GlobalSerializer* to implement the inherited *toXML*-method.
The pure sensor sample information is represented by the serializable *BasicSample* class, which is entirely independent from the Android API. Attributes of this class are the common sample properties, like a device identifier and a time stamp, and a reference to the device dependent sample data. Any concrete sample data type has to implement the *SampleData* interface, to be compatible with this reference. The *BasicSampleCollection* class does allow a pure Java deserialisation of transmitted XML files (containing a sequence of samples) into a collection of *BasicSample*s.
For the task of framework internal sensor data processing, an observable and broadcastable type was needed. This type is realized with the *Sample* class. It does extend the *BasicSample* class and implements the necessary interfaces from the package *util*. The corresponding collection type for *Sample*s is the *SampleCollection* class.
Beside sample data, the serializability feature is also required for the default configuration entries. The XML file, holding the default configuration for the framework service, is located in the projects asset folder. It will be loaded at start up, to determine the preconfigured service settings. The related serializable framework component is the *SDCConfiguration* class. Specific configuration parameters are organized in different configuration entries, analogously to the concrete *Configuration* types in the package *preferences*. For example, the default configuration for a single sensor device is covered by the *SensorConfigurationEntry* class.

4.2.3 Configuration Settings

Basically, the package *preferences* does contain two categories of types, one is related to preference and the other to configuration settings.[15] While the preference types do cover the access to the shared preferences, the configuration types are used to distribute configuration updates in the framework. In this section we will single out the configuration types, the related update events and the management components.

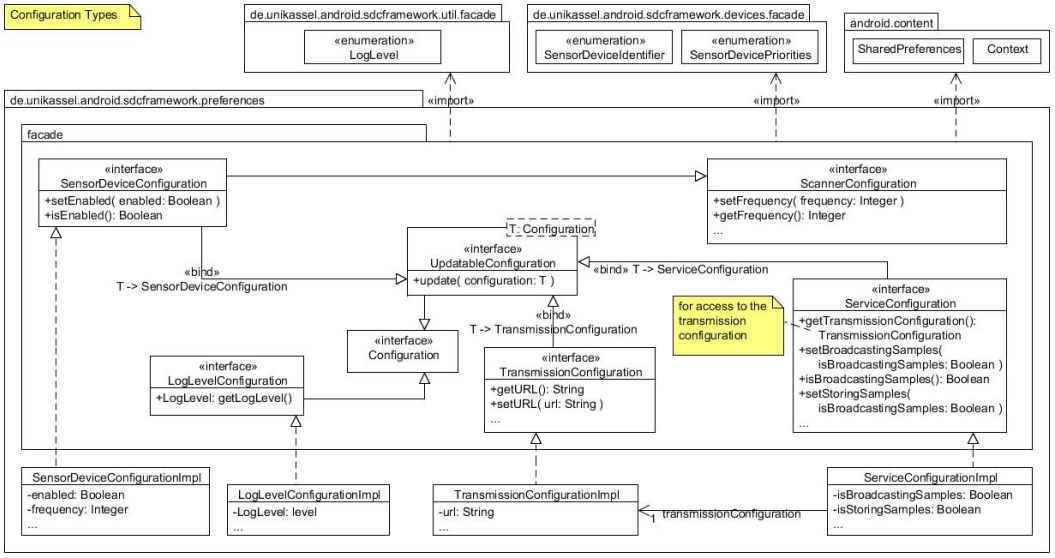

Figure 4.11: Overview of the classes related to configuration settings

[15] compare 3.7 *The Configuration Concept*

Any configuration type has to implement the *Configuration* interface, to be compatible with configuration change events. The *UpdatableConfiguration* interface is intended for types, which can be updated by another configuration instance of the same type. Each specific configuration type is covered by an interface for external access. The implementing classes are completely hidden behind that facade, with the *ApplicationPreferenceManager* as interface for the central access component.
In general, configuration types are organised according to the need to distribute them to the related components. While the *LogLevelConfiguration* type does refer to the *Logger* component, the *SensorDeviceConfiguration* type is related to the configuration settings of sensor devices and scanners, located in the *device* package. The *ServiceConfiguration* is a complex type, granting access to the *TransmissionConfiguration* (related to the *transmission* package) and to the settings for the *persistent* package.
As you see, the current organisation does lack consistency. Because the settings for the persistent storage module are located in the service configuration itself and not in an own type. This is due to the grown structure of this package. Initially, the *ServiceConfiguration* was intended, to cover all the settings related to the packages *persistence* and *transmission*. Finally it turned out, that a single configuration type for both modules is unhandy. Though transmission settings are already represented by an own type, but a general reorganisation of the configuration types is still pending.

Lets focus on the distribution of configuration change events now.

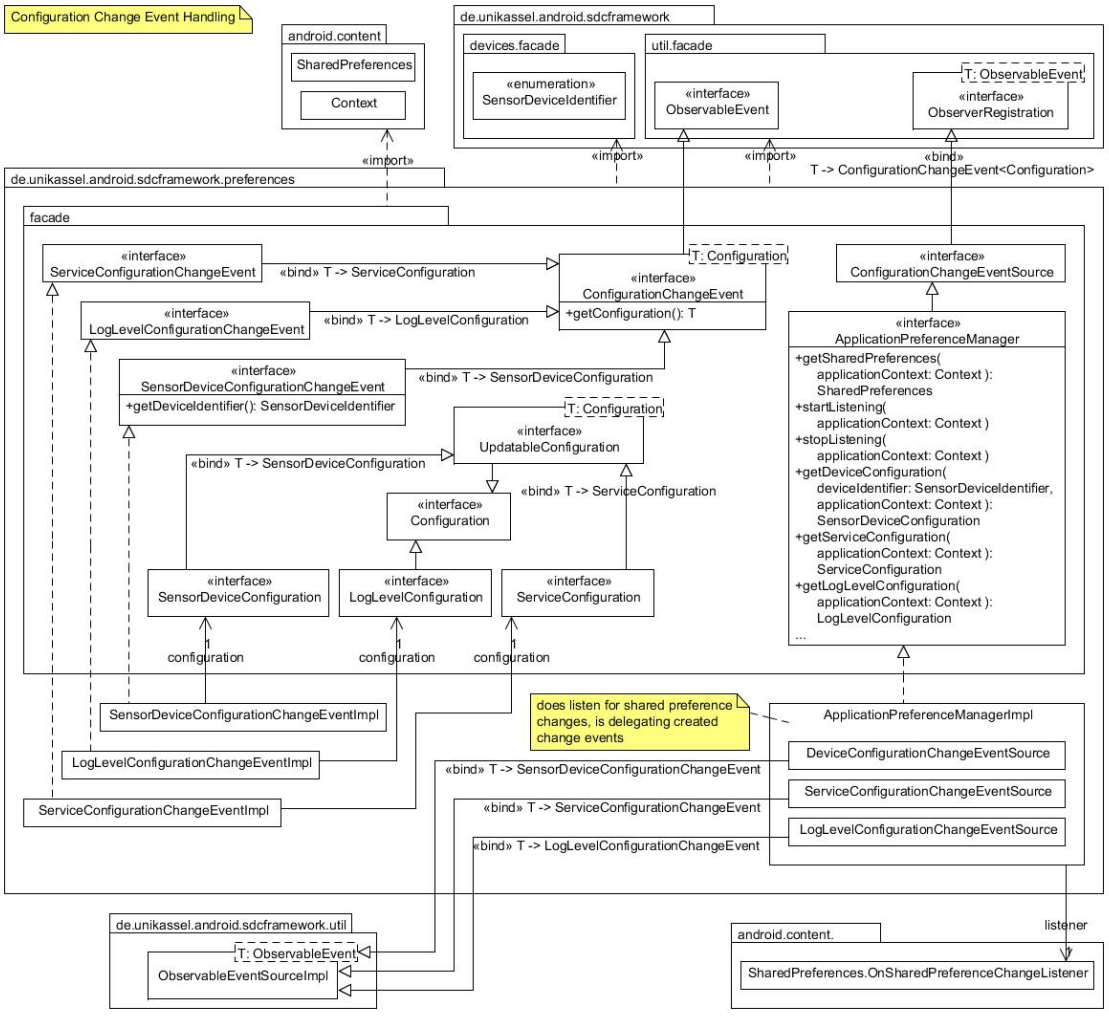

Figure 4.12: An overview of the classes related to the configuration change event handling.

The component for the central access to configuration values is covered by the *ApplicationPreferenceManager* interface implemented by the class *ApplicationPreferenceManagerImpl*. It does provide access to the current configuration settings and allows the registration of observers for *ConfigurationChangeEvent* types.
ConfigurationChangeEvent types are organized according to the initially planned configuration types. For notification purpose, the manger is delegating to internal type-bounded *ObservableEventSourceImpl* instances, one for each available event type. The internal observation of shared preference is controlled with the methods *startListening* and *stopListening*. As soon as a preference modification is detected, a configuration change event object of the related type is created and observers for this type will be notified.

4.2.4 Sensor Devices and Scanner

This section will cover the content of the package *devices*. First the basic types for sensor devices and scanners will be introduced, afterwards we will have a look into components related to sensor device management and finally an overview of the device and scanner hierarchy is given.

4.2.4.1 Basic Types

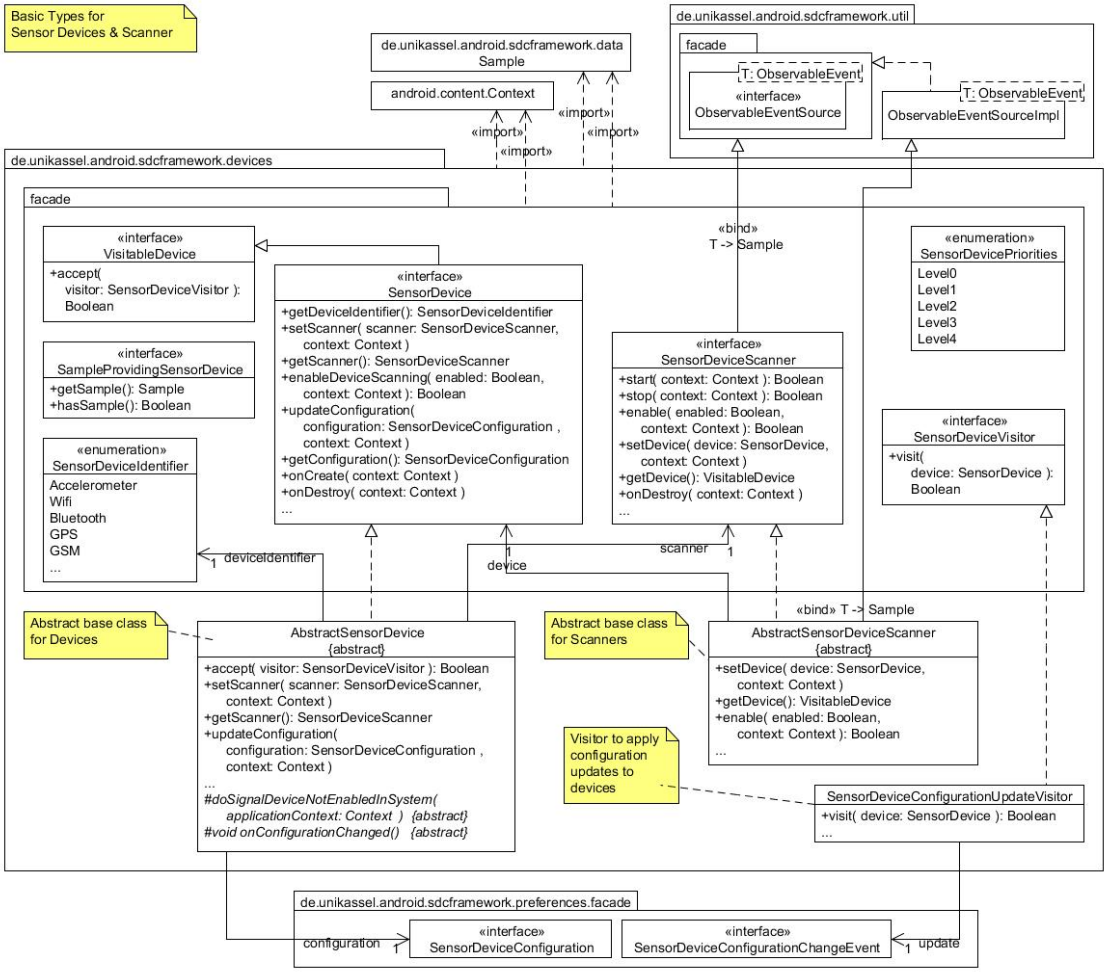

Figure 4.13: Interfaces and abstract base classes for sensor devices and scanners with related types.

The *VisitableDevice* interface is part of the *Visitor* pattern implementation for sensor devices. It does allow visitation by *SensorDeviceVisitor* types. For the moment, just the *SensorDeviceConfigurationUpdateVisitor* does implement this interface. The configuration update visitor is holding a device configuration update, to be applied to the device with the same device identifier.
As the type *SensorDevice* does extend the *VisitableDevice* interface, any sensor device is visitable by default. The class *AbstractSensorDevice* is the abstract basic implementation of the *SensorDevice* interface. A basic sensor device has a unique identifier for the device, a configuration and a reference to its associated scanner.
A sensor device instance is always associated with a scanner, that is responsible for asynchronous sampling. Scanner types are defined by the *SensorDeviceScanner* interface.
The class *AbstractSensorDeviceScanner* is the abstract basic implementation for scanner types. It does extend the generic type *ObservableEventSourceImpl* bound to the *Sample* type, hence it can be observed for scanned sensor samples. A basic scanner has a reference to its associated device.
The two abstract base types are intended to provide a basic behaviour for devices and scanners. Moreover, integrity for the scanner-device association is guaranteed.
For extending types, the *AbstractSensorDeviceScanner* does define the raw enable behaviour of a device scanner. This is done by recording the success of the calls of the abstract *start* and *stop* methods, which both remain unimplemented, as they do refer to the device specific implementation of asynchronism.
A class extending the *AbstractSensorDevice* type can by default accept visitors and handle device state changes. Its configuration can be updated and device scanning enabled or disabled. Given that all the configuration change handling is covered in the base type, extending types are requested to implement a more device specific reaction in the method *onConfigurationChanged*. For the case, that the device is framework-sided enabled for scanning but globally disabled in the system, the concrete device behaviour must be implemented in the method *doSignalDeviceNotEnabledInSystem*.

4.2.4.2 Management and Creation

A sensor device is in general identified by its unique device identifier. With the *SensorDeviceFactory* type and its implementing class *SensorDeviceFactoryImpl*, a central component for device production is provided. For a given device identifier, it does create the related device and scanner objects, both associated with each other.
The framework may support more sensor types than available in the system. Implemented as *Singleton*, the *SensorDeviceAvailabilityTester* component is responsible for the detection of available supported sensor devices. It must be configured for a set of supported device types[16], which is a subset of all known types. Configuration will always trigger a device detection run. The result is cached as a list of device identifiers and can be queried later.
Both components, the factory and the device availability tester, are used by the sensor device manager. By hiding all the implementation details, it is the central access component for the device module. Given that observer registration for the *Sample* type is supported, and *SensorDeviceVisitor* types will be accepted, it does act as observable provider for sensor samples and as entry point for configuration update visitors. While visitors are forwarded to the managed devices, observers will be delegated to the associated device scanners.
As implementing class, the *SensorDeviceManagerImpl* has a map holding the devices, a reference to the application preference manager (to access the current device settings independent from updates) and a factory instance for device creation. It does extend the

[16] this will be done at service start by the service manager component.

class *LifeCycleObjectImpl*, to adapt to the service life cycle. On creation, first the available device types are queried from the global *SensorDeviceAvailabilityTester* instance, then the corresponding device objects will be created by delegation to the factory. Device scanning is enabled on resume and disabled on pause. On destruction, the registered observers are removed and all device instances will be destroyed.

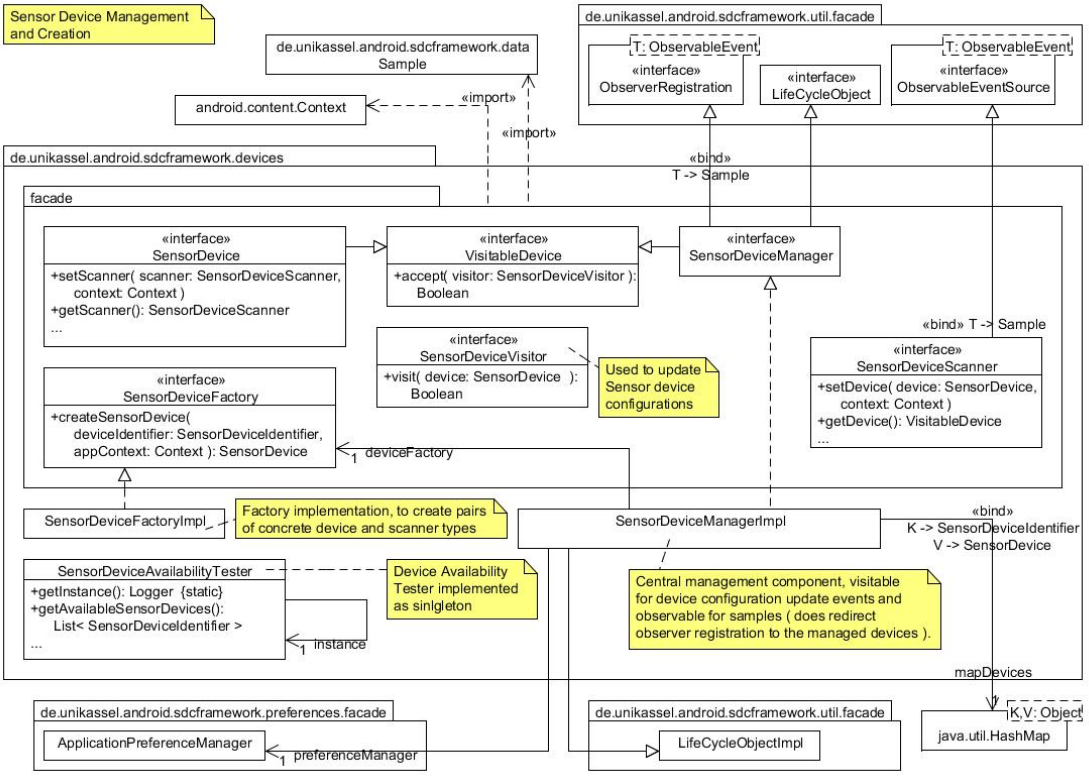

Figure 4.14: Components for device management and creation.

4.2.4.3 Device and Scanner Type Hierarchy

Some of the concrete device and scanner types do have a certain behaviour or property in common. The similarities have been extracted into abstract intermediate implementations, extending the related basic type. Whenever such a type has to request a more device specific behaviour, an abstract method is provided, to be implemented from an extending type.

Based on the analysis of the different kinds of sensor control (as provided by the Android API), two core categories have been identified:

- active devices, listening for sensor data and state changes while caching the latest sample and associated with a sample taking scanner type.
- passive devices, depending on broadcasts to receive information about system state changes and associated with a scanner type, which has to trigger periodical device scans and does receive samples by broadcasts as well.

Lets give attention to active devices first. Normally, sensor data is received in the callback method of a device specific listener and the latest sample must be stored in the device instance for scanner access. The behaviour for an active sample providing device is defined in the *SampleProvidingSensorDevice* interface. A compatible device scanner has to extend

the abstract *SampleTakingDeviceScanner* class. This type can fetch the latest sample from its associated device in the configured frequency. Internally, it is using a handler to execute the asynchronous sample task [17]. The sample task itself is realised by the class *SampleTakingTask*.
Using a listener to observe samples, does require a device to be aware of the scanner running state. For example, if device scanning becomes disabled, the listener must be detached to preserve system resources. The common behaviour for such a device is implemented in the *ScannerStateAwareSensorDevice* type. This is the base class for the concrete implementations of the GPS and GSM sensor devices.
For other sample providing and scanner state aware sensor types the Android API does already provide a common abstraction with the *Sensor*[18] class. The related framework component is the class *AbstractAndroidSensorDevice*, extending the *ScannerStateAwareSensorDevice* and implementing the default listener behaviour. A concrete implementation of this type is the *AccelerometerDevice*.

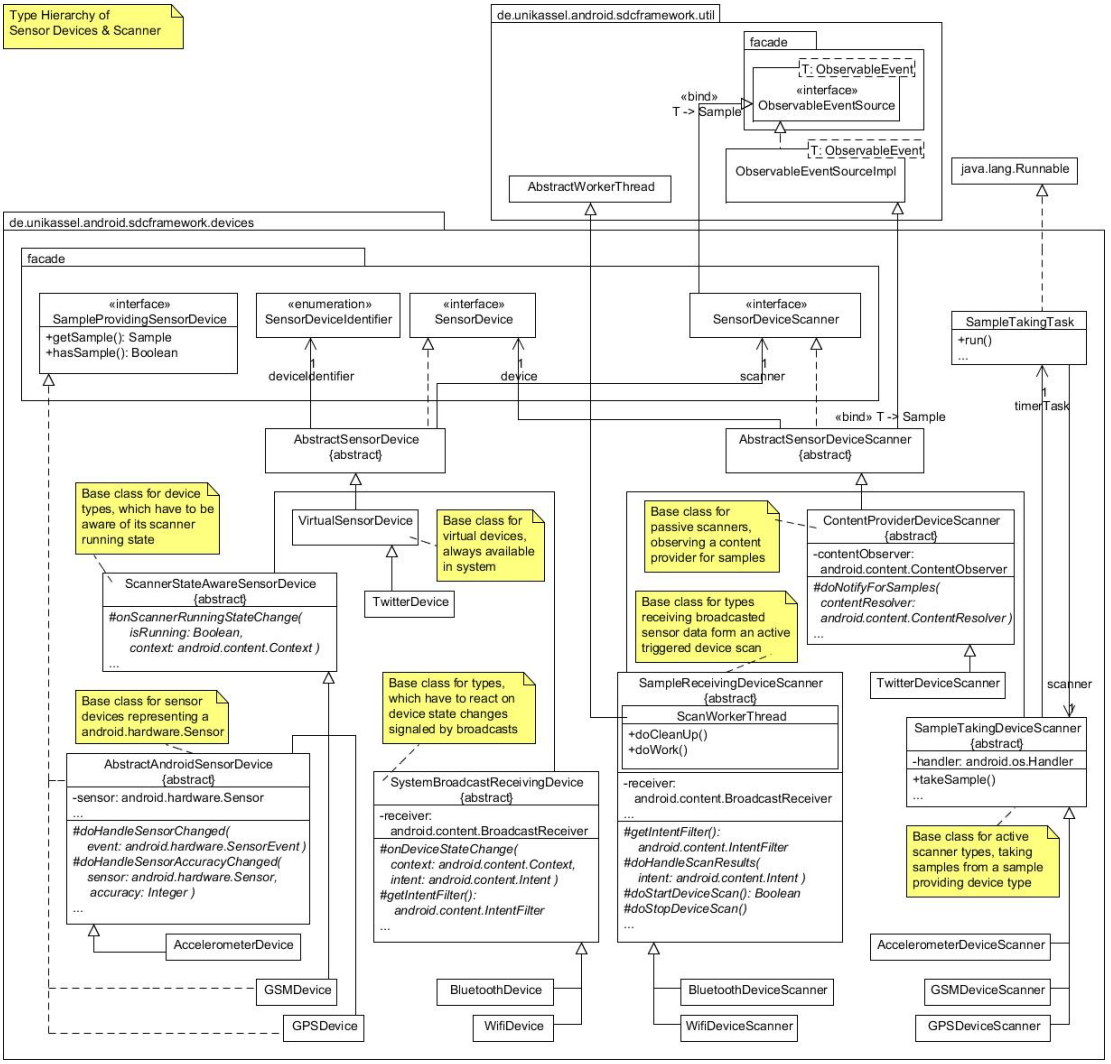

Figure 4.15: Hierarchy of sensor device and scanner types

In contrast to the sample providing and scanner state aware devices, there are broadcast dependent types, like Wifi and Bluetooth. Sensor data must be requested by the explicit

[17] the different concepts for asynchronism are explained in 3.8 Aspects of Parallel Processing
[18] see http://developer.android.com/reference/android/hardware/Sensor.html

initiation of a system scan. While the scan is running, the results are delivered by system broadcasts. The behaviour of such sensor devices is covered by the *SystemBroadcastReceivingDevice* type, to be associated with a *SampleReceivingDeviceScanner*.
A system broadcast receiving device, has to react on broadcasted system state changes. To do so, a concrete type must define the state-related *Intent* filter for the specific device type (in the method *getIntentFilter*), in addition it has to implement the state change handler (*onDeviceStateChanged*). A sample receiving device scanner is running a thread to control the device scan runs and does process the received scan results. Beside the related handler *doHandleScanResults*, extending types must implement the device specific methods to start and stop device scans and define the result-related *Intent* filter.
With the first framework extension a virtual device type was added. It does cover devices which do not refer to a physical sensor but an abstract data source to be scanned for samples. The *VirtualSensorDevice* type is always available in the system. By definition, the source for sensor samples is a content provider. The related scanner type is the abstract *ContentProviderDeviceScanner*. It does receive samples from a content provider and, in contrast to all other scanner types, it does ignore the configured sample frequency. This specific behaviour takes into account, that the virtual sensor type is used to record external data in the form of samples without loss. Based on these types, a virtual *Twitter* sensor was added to the framework, which is able to receive *Twitter* messages from another application using a content provider for the data exchange.

4.2.5 Sample Collection and Persistent Storage

The next section will introduce the components for database access, followed by the explanation of persistent storage management.

4.2.5.1 Database Access

Collected sensor samples are stored in a SQLite database in a uniform format[19]. The type *DatabaseSample* does reflect the database scheme and is used to transport samples through the internal database layer, leaving the responsibility for conversion to the controlling component.
In its function as *Adapter*[20], the class *DatabaseAdapterImpl* acts as interface to the SQLite database. For database creation and version management, it does use an internal helper class based on the *SQLiteOpenHelper* provided by the Android API.
The *DatabaseManager* implementation serves as central access point to the database layer. While it does provide some special methods for common tasks, it is able to execute any database command type, which implements the *DatabaseCommand* interface.
A database command representation is provided to reduce the extent of the database management interface, while allowing the usage of various different commands from inside of the persistence module. The behaviour of a database command can be parametrized. For example, the *DeleteSamplesCommand* can be configured for the count and to pay attention to the sample priority. All the concrete *DatabaseCommand* types do extend the *AbstractDatabaseCommand*, which does implement the interface for a generic command result type. The default behaviour includes the opening and closing of the database, as well as the handling of all *SQLException*s, leaving just the task to define the call of the related *DatabaseAdapter* method to a derived type. Let me point out, that any *SQLiteFullException*, occurring during command execution, is simply rethrown to be caught by the persistence storage manager which is responsible for the database full handling.

[19] a format description is given in 3.6 Independent Sample Data Representation
[20] the *Adapter* pattern is explained in [GHJJ96, p.171]

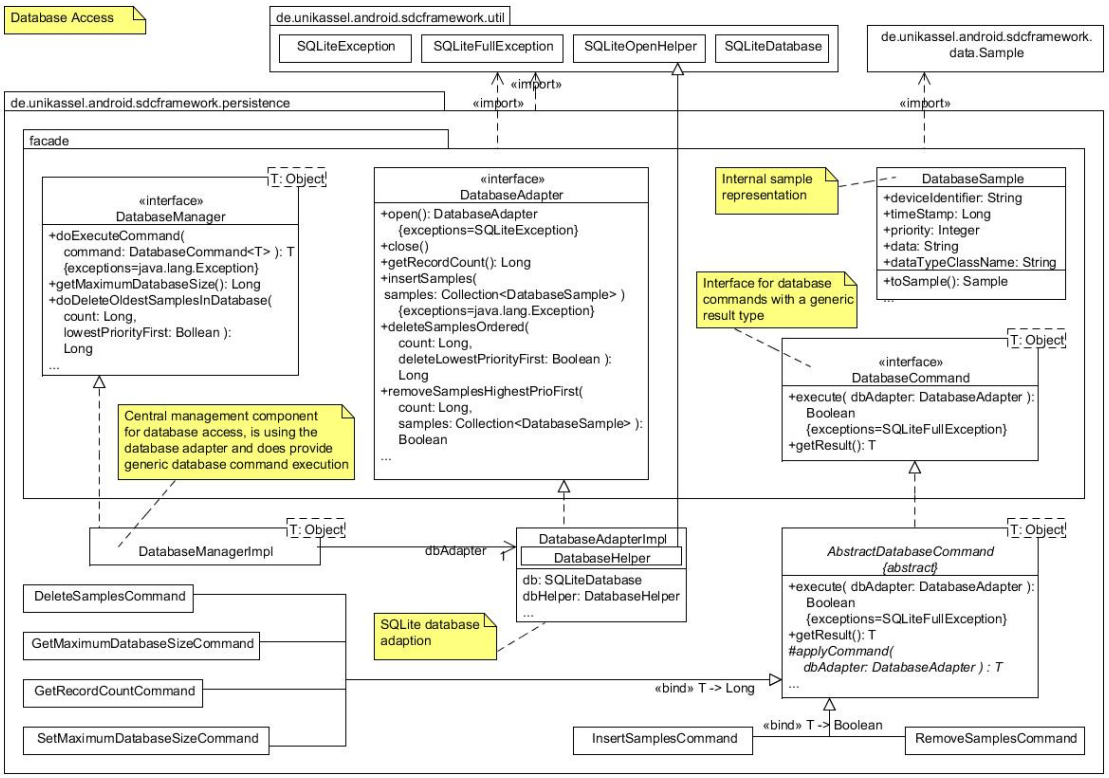

Figure 4.16: The classes related to database access

4.2.5.2 Persistent Storage Management

Turning to persistent storage management, the *PersistentStorageManager* can be regarded as interface to the persistence module. While delegating to a *DatabaseManagerImpl* instance for database access, it does serve as *DatabaseManager* itself for external components.

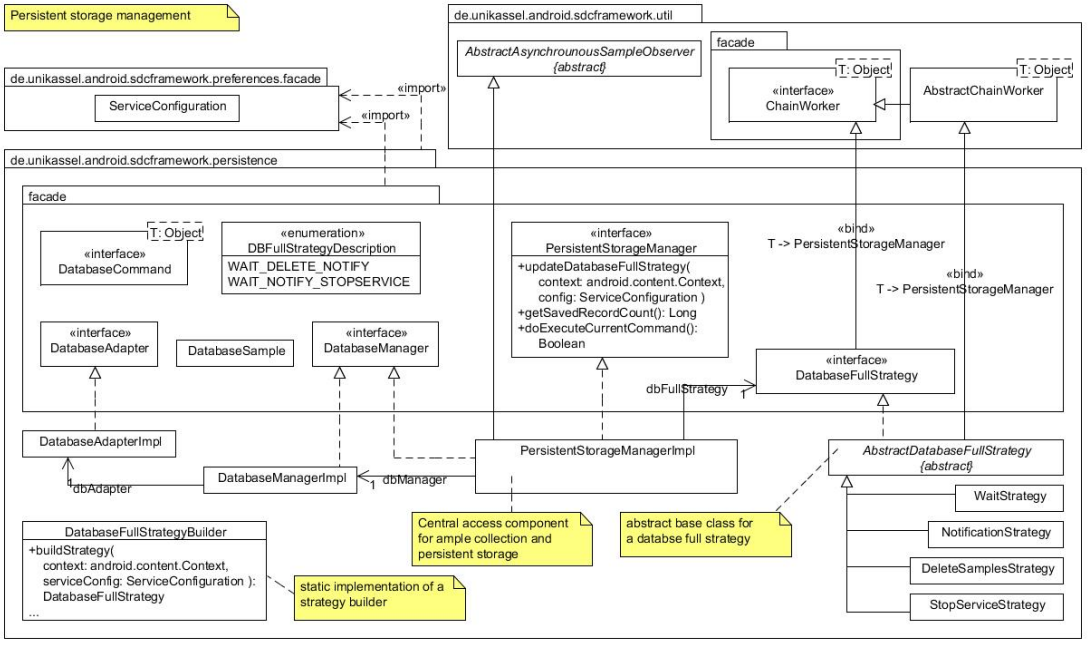

Figure 4.17: Persistent storage management and database full strategies

The *PersistentStorageManagerImpl* extends the *AbstractAsynchrounousSampleObserver*, hence it runs asynchronously adapted to the service life cycle and can be attached as sample observer to the device module. For the handling of a database size overrun, it is using a *DatabaseFullStrategy*.
A *DatabaseFullStrategy* type is a *ChainWorker*, acting as processor in a *Chain of Responsibility* with a *PersistentStorageManager* instance as client. As base type for any database full strategy, the *AbstractDatabaseFullStrategy* does extend the *AbstractChainWorker* implementation bounded to the *PersistentStorageManager*. It provides a basic implementation of the inherited *process* method, by invoking the *doExecuteCurrentCommand*-method of the persistent storage manager to execute the last command (which has failed) again. The extending concrete types do override this function to implement the specific strategy, calling the super class method in the end[21]. To sum it up, a strategy applies its solution and reports the result of the next command execution try. If the strategy fails, its successor is called.
The *DatabaseFullStrategyBuilder* is used to build an ordered chain of database full strategies according to the current configuration.

4.2.6 Sample Transmission

To introduce the package *transmission*, we will start with the upload management. Subsequently, the transfer manager is the key element we will focus on.

4.2.6.1 Upload Management

The *UploadManager* is responsible for the file upload to the configured remote host using the specified connection strategy. For this purpose, it does delegate to two different strategy types, a *ConnectionStrategy* and a *ProtocolStrategy*.
As base class for protocol strategy types, the *AbstractProtocol* is provided. Beside a common exception handling for malformed URLs and file access errors, the *uploadFile* method is implement to delegate to an abstract method (*doUploadFile*), to be implemented by a concrete protocol strategy.
The upload protocol can not be configured like the connection strategy, it is extracted from the provided URL by the upload manager. If the scheme is unknown, the *UnknownProtocol* type is linked as protocol strategy. For the moment just *http* as scheme is supported with the *BasicAuthHttpProtocol* as related type. The *FailSafeProtocol* does only exist for test purpose.
While the *ProtocolStrategy* is used stand-alone, the *ConnectionStrategy* is part of a *Chain of Responsibility*. If it fails to connect to the Internet, its successor is called. A concrete connection strategy is intended to extend the *AbstractConnectionStrategy*, which does provide the common behaviour as chain worker but leaves the task to implement the *ConnectionStrategy* interface. Ordered chains of connection strategies are defined in the *ConnectionStrategyDescription*. The *ConnectionStrategyBuilder* can build corresponding strategy chains for a configured strategy description value.
The purpose of the *ConnectivityWrapper* interface is the adaption of the Android *ConnectivityManager*[22] for the framework. It is implemented as *Singleton* with several test methods for connectivity and host reachability. The *AbstractConnectionStrategy* is using a reference to a *ConnectivityWrapper* instance, to implement a general test method for available network connections.

[21] with the exception of the *StopServiceStrategy*, which does trigger a service shut down
[22] compare http://developer.android.com/reference/android/net/ConnectivityManager.html

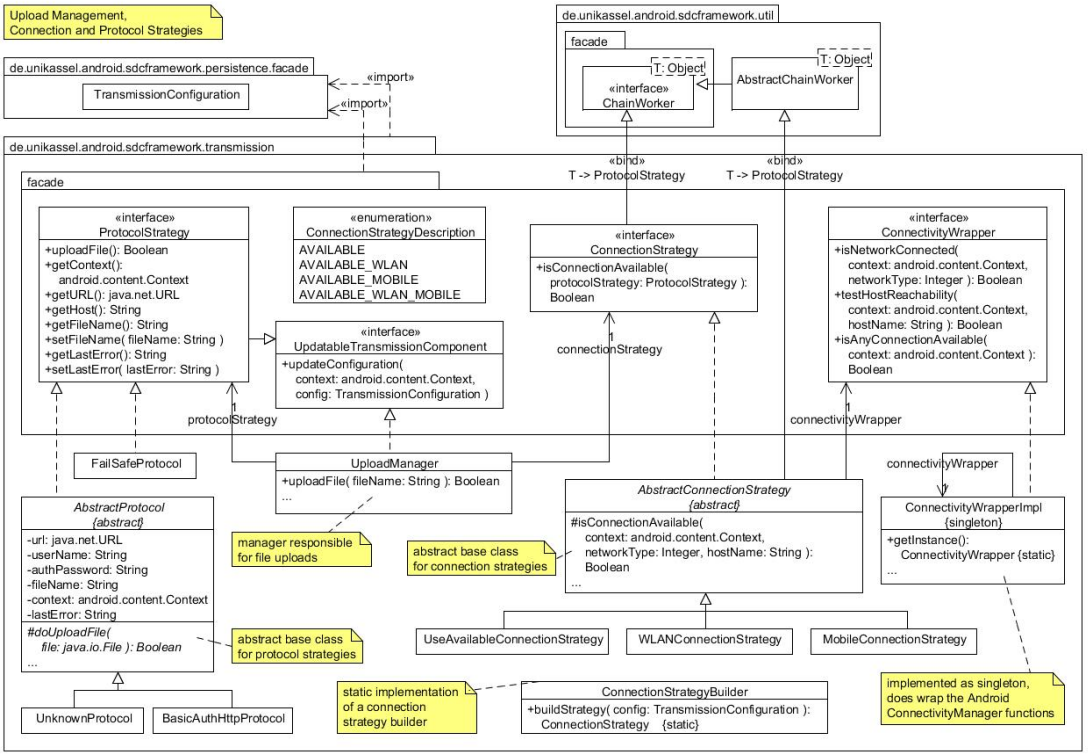

Figure 4.18: Types related to upload management.

The *UpdatableTransmissionComponent* interface does define the ability of a component to update its current settings from a forwarded *TransmissionConfiguration*. It is used to distribute configuration updates to the affected components in the transmission module.

4.2.6.2 The Transfer Manager

As the interface component for the transmission module, the *TransferManagerImpl* is responsible for the gathering of samples from the database, the creation of a temporary stored file with the serialized samples, the compression of the sample file together with the device information and the upload of the resulting archive. For this purpose, it does delegate to several other components, for example the upload manager.
The transfer manager does adapt to the android life cycle and runs asynchronously as worker thread. Internally, it is implemented as a state machine. The first state realizes an execution delay according to the configured frequency. It is followed by a wait state to gather enough samples for transmission and a subsequent state to prepare the archive. In the final state the archive is uploaded. There are several reasons to pause the thread execution at a specific point in order to wait for an event. For example, to await the next execution cycle or the availability of enough samples. In this case wait locks are used to send the thread sleeping.
For database access the transfer manager is delegating to a *DatabaseManager* instance. In this context I would like to point out, that the persistent storage manager and the transfer manager do share the same database manager instance[23]. This is due to the fact that SQLite databases are not intended to be accessed in a multi-process way. Using the same database adapter instance, respectively the same database connection, is a proper way to avoid concurrent access problems.

[23]this is guaranteed by the supervising service manager

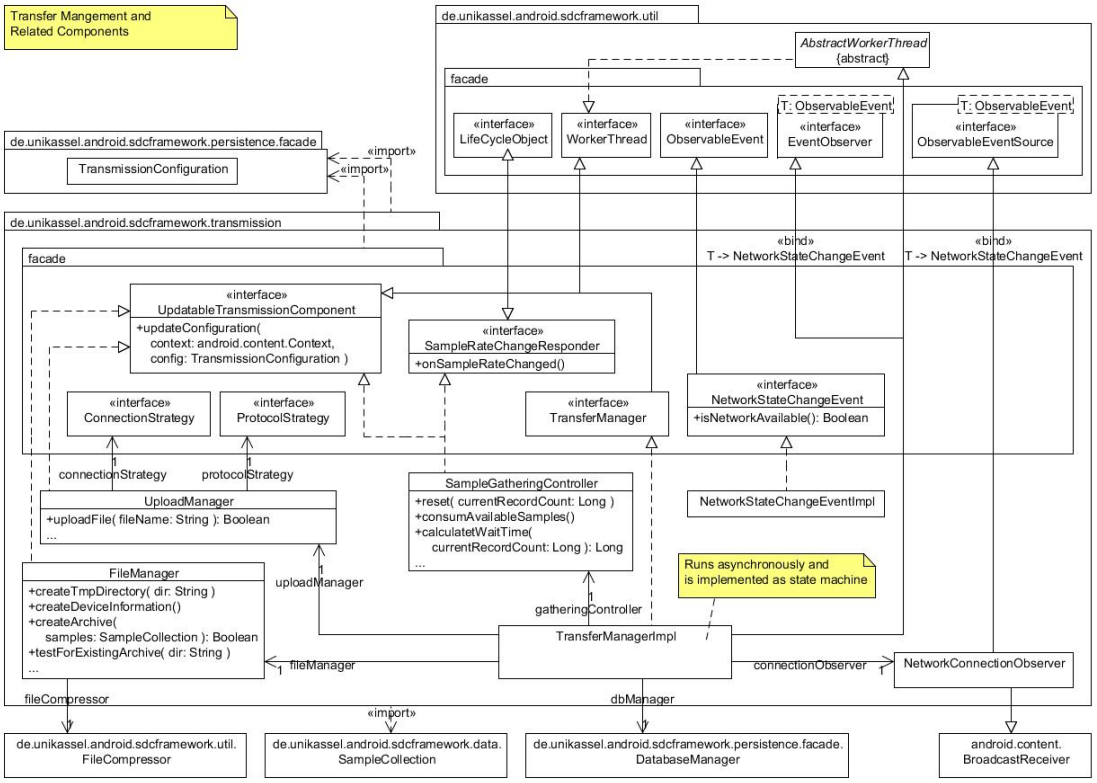

Figure 4.19: The transfer manager and related components.

Samples must be transmitted in a configurable extend. Due to the high degree of freedom in configuration, it may happen, that the configured transfer frequency is to high for the current sample rate, with the result, that there are not enough samples available in the database. Hence, the configured frequency is regarded as lower limit, while the real delay in execution is determined with the help of a special controller component. The *SampleGatheringController* estimates the minimal time span to wait, so that the required minimum of samples is available in the database.
For the upload preparation task, the transfer manager delegates to a *FileManager* instance, which does provide several functions to create and manage the required files. It does use a *FileCompressor* to store the files compressed in an archive. To upload the archive to a configured remote host, the transfer manager uses an *UploadManager* instance.
Android does broadcast general changes in the network connectivity. The *NetworkConnectionObserver* is the implementation of a broadcast receiver to monitor the global network state changes. It is observable for the *NetworkStateChangeEvent* type, which can be queried for the actual network availability. As registered observer, the transfer manager is notified about global connectivity changes. If an upload has failed due to a missing connection possibility, the execution is paused until the next network state change is notified.

4.2.7 Central Service Management

Beside two utility classes, related to the observation of the service running state, the package *service* does only contain the *ServiceManager*.
The *ServiceManagerImpl* is the core component of the framework service, responsible for the creation, proper association and destruction of all other service components. Another important task is the central observation of the preference manager for configuration changes along with the distribution of received change events to affected components.

By the implementation of the *LifeCycleObject* interface the behaviour is adapted to the service life cycle.

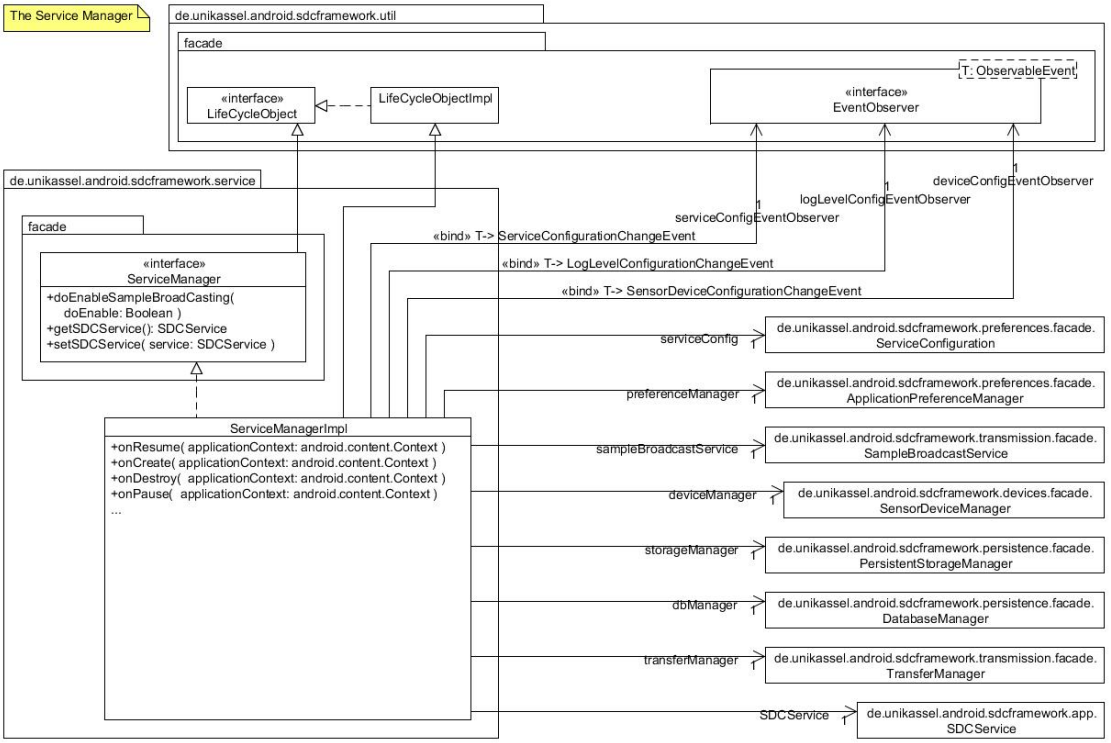

Figure 4.20: The service manager and its components.

4.2.8 The Framework Application

As Android application, the *SDCFramework* is composed of three application components[24], located in the package *app*:

- the SDCService (as main service, providing the core functionality),
- the SDCServiceController (as controlling activity, providing logging, service control and preference access),
- the SDCPreferenceActivity (as Android preference activity, to change preferences at runtime),

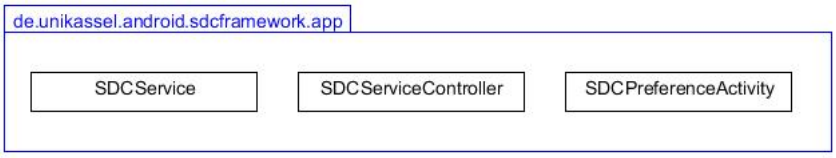

Figure 4.21: The components of the application package

Subsequently, you will find the related UML diagrams, displaying the components and how they are embedded in the framework, pointing out the direct related types and packages. With the end of this section the architecture overview is finally rounded off.

[24] the framework composition as Android application is explained in 3.2 *The Android Application Concept*

4.2.8.1 Main Service

The main framework component is the *SDCService*. It is implemented as Android service and does delegate to a *ServiceManagerImpl* instance.

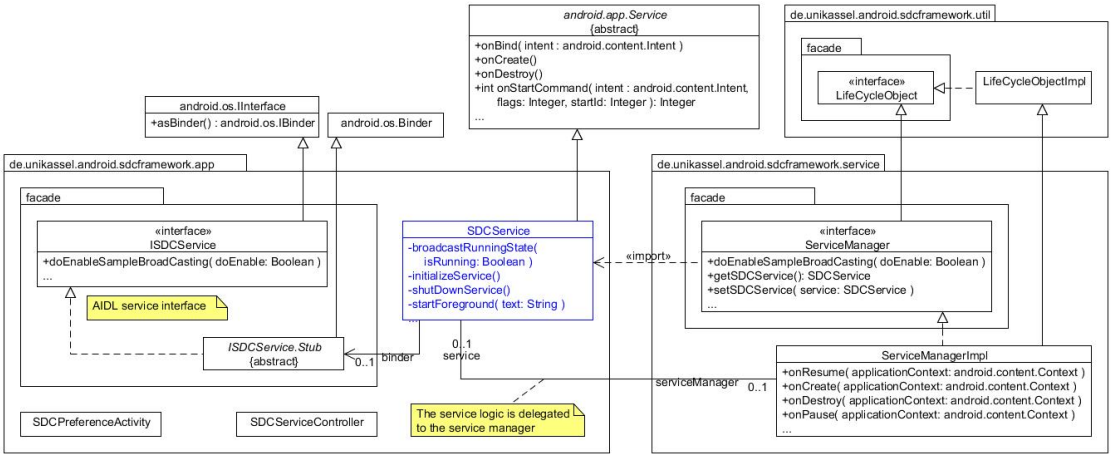

Figure 4.22: The SDCService in the context of the framework

The AIDL[25] interface *ISDCService* is provided for interprocess communication. Actually it does support an explicit state change of the sample broadcast service, allowing bounded activities to enable or disable it.

4.2.8.2 Control Activity

The *SDCServiceController* activity is a graphical user interface for the framework service. It does provide start and stop buttons, a log view displaying the received *LogEvents* and an option menu, with an entry to open the preference screen.

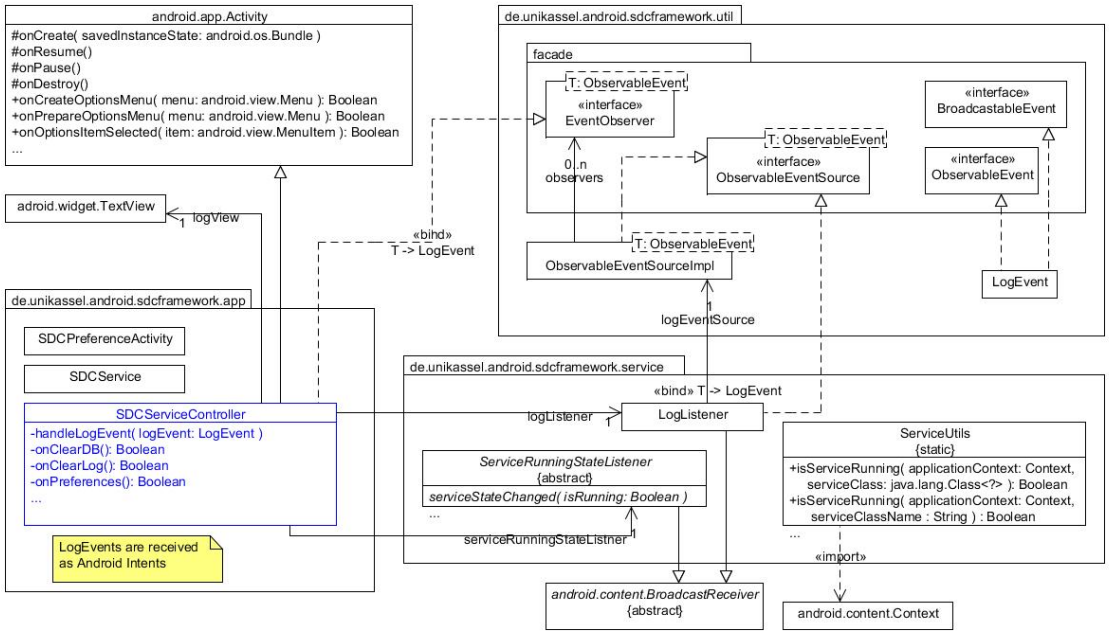

Figure 4.23: The SDCServiceController in the context of the framework

[25] Android Interface Definition Language, see http://developer.android.com/guide/developing/tools/aidl.html

4.2.8.3 Activity for Preferences

The *SDCPreferenceActivity* provides the preference screen, to change the preconfigured framework settings at runtime.

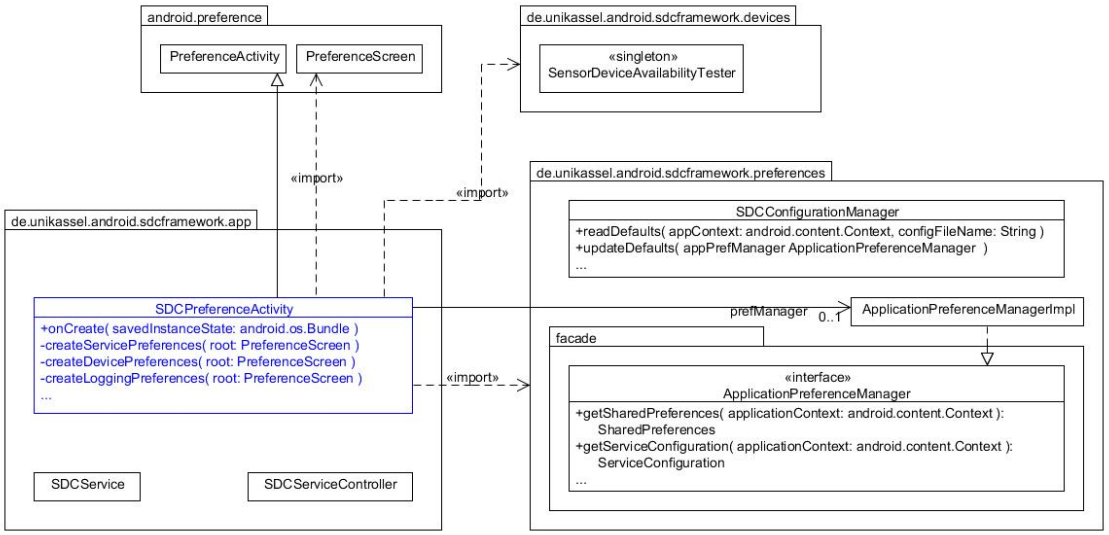

Figure 4.24: The SDCPreferenceActivity in the context of the framework

5. Conclusion

Considering the limited time[1] and the large extent of requirements, I would tend to classify the project result as success. But a detailed view does reveal a lot of weak points leaving areas of future work.
Lets start with the positive aspects. Regarding the intended area of application, the provision of sensor based data of mobile Android devices for a single configurable host, the requirements can be regarded as fulfilled. The core features do work as requested and the functionality is assured by unit test, furthermore a proper documentation is provided. With the implementation of the first framework extension for a virtual Twitter sensor, the module separation concept has been proven to work, because neither the persistence nor the transmission module have been affected. But integration tests with a large number of different mobile devices are still outstanding, first and foremost, a matter of availability.
Further weak spots are the grown structure of the package *preferences*[2], the accepted incompleteness regarding supported sensor types and the limited possibilities for remote control by other applications. If we leave the initial objective and the defined requirements behind and do think about further areas of application, considerable problems may arise. Just to point out one, the service can not be shared by different applications. For this purpose, some kind of registration for certain sample types to be transported to a specific remote host must be supported. At least the transmission concept does fail completely for such a task. For the moment just one configurable host is supported and transferred samples can not be limited by the type.

Let me just sum up the apparent areas for future work:

- Completion of the module devices by adding the remaining known sensor types.
- Concerning the organisation of configuration settings and the resulting upgrade granularity, a review and reorganisation of the package *preferences* is recommended. This is a large task, as interface changes will affect other packages as well.
- Regarding the framework idea, the AIDL service interface must be extended for all available configuration parameters. Thus, a bounded activity can remote control the service behaviour. At the moment, just the default configuration allows user independent customisation.
- For practical reasons, the ”stop service strategy” for database limit overruns must be changed to a less limiting solution, e.g. a temporary stop of sensor sampling.
- Currently, log information is just displayed in the open controller activity. To track errors and report crashes on a rolling time basis, a persistent log file is suggested.
- Some unit tests for the virtual sensor device extension are missing and extensive integration tests on different devices with several OS Versions are essential.
- An extension for pluggable sensor devices would be desirable.

Regarding extensions for a wider range of application areas, more detailed analyses are required. For the moment, I favour the idea to separate the core features in three different services or some kind of plug-in architecture, but this is content for a new project.

[1] the project is awarded with eight semester periods per week.
[2] as mentioned in 4.2.3 Configuration Settings

List of Figures

Bibliography

[And11] "Android developers site," Oct. 2011. [Online]. Available: http://developer.android.com/index.html

[BP09] A. Becker and M. Pant, *Android: Grundlagen und Programmierung*. Heidelberg: dpunkt, 2009. [Online]. Available: http://www.dpunkt.de/buecher/3436.html

[GHJJ96] E. Gamma, R. Helm, R. Johnson, and V. John, *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*, 1st ed. Bonn: Addison-Wesley, 1996, Design Patterns, 1995, Deutsche Übersetzung von Dirk Riehle.

[HKKR05] M. Hitz, G. Kappel, E. Kapsammer, and W. Retschitzegger, *UML@Work, Objektorientierte Modellierung mit UML 2*. Heidelberg: dpunkt.verlag, 2005.

[Sim11] "Simple XML framework project site," Oct. 2011. [Online]. Available: http://simple.sourceforge.net/

[Sta11] "Stackoverflow site," Oct. 2011. [Online]. Available: http://stackoverflow.com/questions/tagged/android

[Ven11] "The VENUS project page," Oct. 2011. [Online]. Available: http://www.uni-kassel.de/einrichtungen/en/iteg/venus/venus.html