AHL: A Toolkit for Model-Driven Engineering of Android Applications

by

Pedram Veisi

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

University of Alberta

Abstract

With the rise of smartphones and the increasing impact of mobile applications on everyday life, mobile software engineering has become a popular research topic. A desired outcome of these research efforts is efficient application development with lower cost, but with high-quality software products. A combination of domain-specific languages and code-generation techniques is a potential solution to this problem. In this thesis, we define a generic model for Android applications that work with peripheral devices, such as activity trackers, and propose a framework, namely AHL (Android Health Language), that implements our model and enables easy and rapid development of the core elements of a typical application reporting data collected from these peripheral devices. Our framework includes a domain specific language (DSL), AHL, that allows developers to describe their applications with an easy-to-use syntax. Then, the framework takes it from there and generates most of the code for the complicated components of a standard application falling into the domain of our problem. The generated code is functional and does not need any modifications. That will save developers from dealing with complicated Android concepts. Therefore, AHL can save time and reduce the cost of Android application development for developers. We explain the AHL framework, its models, the included DSL and the methodology we used to design and implement it. We also evaluate our work with two functional applications and compare them to the existing ones developed from scratch.

To my lovely parents and beautiful sister
For their endless love and support.

If you focus your mind on the freedom and community that you can build by staying firm, you will find the strength to do it.

– Richard M. Stallman

Acknowledgements

I am using this opportunity to express my gratitude to everyone who supported me throughout this work. My supervisor Dr. Eleni Stroulia and my committee members Dr. Ken Wong and Dr. Marek Reformat, my dear family and friends, and everyone who has taught me something in my life.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Portability, from a customer's point of view, was the only difference between landline telephones and mobile devices for years. However, during the recent years with the help of powerful hardware and various sensors, smartphones have become a replacement for desktop and laptop computers, digital cameras, navigation devices, health and fitness gadgets, etc. All these functions in a single portable device have made smartphones irresistible to users. Obviously, these features are only possible with the support of mobile platforms for applications and the business models that allow third parties, including software companies and independent developers, to develop applications, improve the platform and generate revenue. Hence, many developers are drawn to mobile applications development.

The size of the target market, especially for independent developers and small businesses, is a significant factor considered in choosing a platform to develop applications for. Android, which currently dominates the world market of smartphones with nearly 83% share of the market [1], is a primary choice for businesses and independent developers. Figure 1.1 shows changes in smartphone OS market share over a three years period, from the second quarter of 2012 to the same period in 2015.

Although cross-platform technologies, such as PhoneGap [1] provide the possibility of developing applications for major platforms (including Android and iOS) at the same time, native applications offer a far better user experience

[1] http://phonegap.com

and performance since the overhead is minimum in native code compared to cross-platform code. Therefore, most developers focus on a single platform, and native application development is far more popular. According to a cross-platform tools benchmarking study by research2guidance, "in the USA 11.8% (iOS) and 14.9% (Android) of the top 2,000 apps are developed with a CP[cross-platform] Tool" [2].

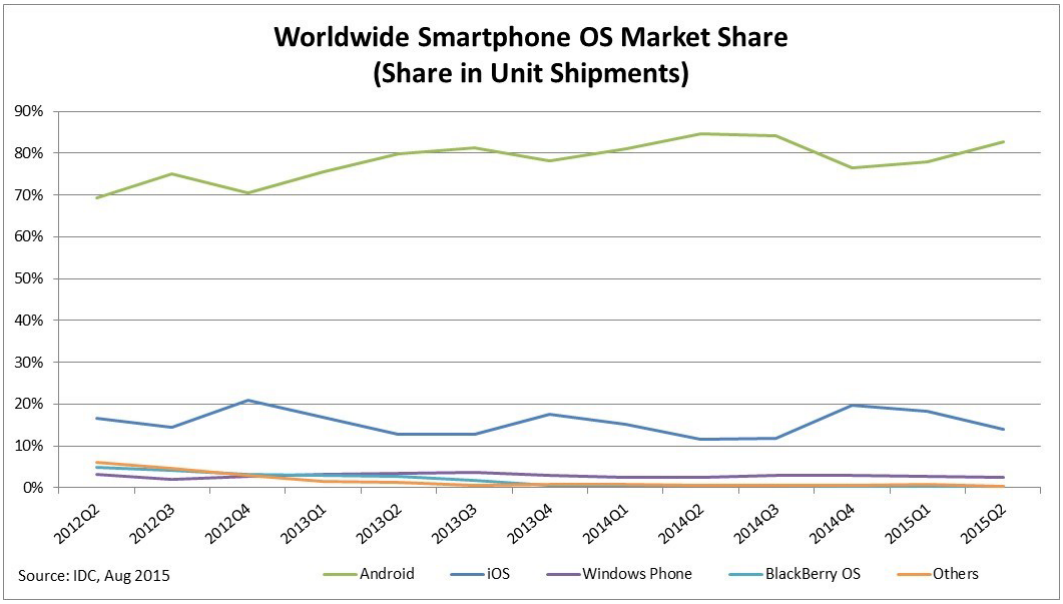


Figure 1.1: Smartphone OS Market Share [1]

1.1 Motivation

Despite all the efforts of Google engineers to make Android development easier, many developers, especially inexperienced ones, face major challenges during development. The complexity of the platform in the competitive market calls for tools to make the process easier and faster. Such tools can save time and money and increase the efficiency of development teams.

Developers work to create applications for different purposes. Among all, one stands out and has many users: health and fitness. Applications in this field are divided into two main categories: those that rely on pre-installed sensors and components on smartphones such as accelerometer and GPS, and the others that come with an accompanying physical device such as Fitbit,

Garmin, and smart watches. The former are limited to tracking a few kinds of activities like walking, running or biking. Google Fit on Android and Apple Health on iOS fall under this category. They usually have a significant impact on battery consumption and in some cases their accuracy depends on the way that users carry their phones. In contrast, accompanying physical devices do not have these limitations for tracking the same activities and enable providers to focus on tracking ones that are not trackable with smartphones like sleeping, using wheelchairs, etc. Most of these devices are produced as wearables that are very convenient for users to use. They also take advantage of Bluetooth for communication with smartphones. Most of these devices can store activity data for a few days, and can usually work for more than a week on a single charge.

Wearables market is a growing one. Tractica, a market intelligence firm that focuses on human interaction with technology, published a white paper in the third quarter of 2015 with the title of "Wearables: 10 Trends to Watch" and predicted that the wearable market will experience a big growth over the next few years (Figure 1.2) [3]. In another report, CCS Insight, an industry analyst firm focusing on mobile communications and the Internet, stated that wearables market will be worth $25 Billion by 2019 [4].

However, in this case, the convenience of the user means more work on the development side. Developers working on the related applications have to spend more time and resources to manage Bluetooth communications and data storage in the background that includes implementing low-level Bluetooth communication functions and services for handling these communications, reading from the physical device and storing it in a database. The common set of functionalities in this family of applications can be reused for development efficiency and lower costs and for reusing good-quality code.

Applications that work with wearables or other peripheral physical devices have a few modules in common: Bluetooth communication, data storage on the phone, a background service and some UI components such as pairing devices and visualizations.

This work provides a tool which generates most of the code for all of the

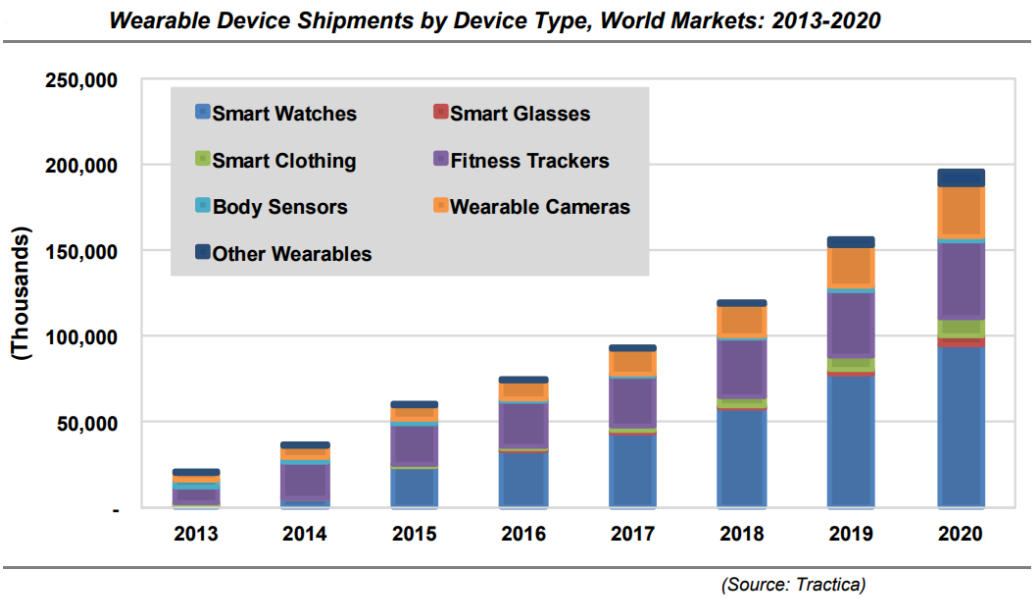


Figure 1.2: Wearbale Market Growth Over the Next Few Years

four modules common in this family of applications. A developer fills in a configuration file, and the provided tool generates the code required for database management using Android Content Providers, an Android service and all the other classes related to periodic task scheduling, and all the necessary classes for Bluetooth communication that are not device specific.

1.2 Contributions

Our framework offers support for a systematic development of applications working with wearable, activity trackers and other kinds of BLE related devices. We propose an abstract model that describes such applications and provides a toolset that enables Android developers to generate all the required source code for the challenging parts of them.

To generate the source code, the developer must provide some information including hardware information required for Bluetooth communications, data packet structure transmitted by the device and other information about the application itself such as the package name, desired name for the database file, etc. The user fills out a configuration file using a domain specific language

we have designed to provide this information. A plugin for JetBrains IntelliJ IDEA, the IDE that Android Studio is based on, takes it from there and generates the source code based on the provided configuration and the models designed for this purpose as part of our framework. At last, an accompanying Java class, that is run by the developer, finishes the work by creating Java packages and moving all the generated and pre-written Java classes to the right place.

The generated components are as follows:

- **Bluetooth Communication**: using the information provided by the user, the framework will generate the code to handle searching for BLE devices and pairing with them. The code for reading from these devices should be implemented by the developer since it is hardware dependent.

- **UI**: this component provides an interface for the Bluetooth Communication component. UI requirements for listing Bluetooth devices, pairing with them and removing them from paired devices are included in the produced code by our framework.

- **Data Storage**: the framework generates an Android content provider for data and database management. Content providers manage access to data by the application that implements them and other application that might be given access to by the developer in a standard way.

- **Service**: for such application, an Android service should be generated to link Bluetooth communication and data storage. The service will be activated at specific periods of time and is in charge of calling specific methods from the Bluetooth communication component for initiating the connection and reading the data from the device and then sending the data to the data storage component for storing it in the database. This component also includes other necessary classes, namely broadcast receivers, required for periodic task scheduling in Android.

In summary, our work makes the following contributions:

- **First,** it proposes a modeling language to describe applications that work with a peripheral BLE device.
- **Second,** it offers an easy and step by step environment for developers using an IntelliJ IDEA plugin to create their applications.
- **Third,** it evaluates this methodology by creating an example application for a wheelchair usage tracking device (i.e. Redliner) and a BLE Sticker Reader application (i.e. Estimote Stickers) and comparing them with their respective applications developed from scratch.

1.3 Thesis Outline

The remaining of this thesis is organized as follows. Chapter 2 is dedicated to the literature review and related work. Chapters 3 and 4 discuss the background of our work, related concepts and definitions, and technologies that are used to design and implement the framework. In chapter 3, we describe our frameworks architecture and each of its components in details and discuss related technologies such as Android Content Providers. Then in Chapter 4, we discuss MPS, introduce our models and describe how they work and discuss the process of code generation. Following that discussion, we demonstrate the process of generating all of the components for the Redliner application in details using a step by step approach in chapter 5. After that, we evaluate our work in chapter 6 by comparing the Redliner and Estimote Reader applications created from scratch with the ones with generated components. At last, we conclude with a discussion about the possible future work in chapter 7.

Chapter 2

Literature Review

In this chapter, we start with discussing Domain Specific Languages (DSLs) (Section 2.1.1) and the tools that are available to design and implement them (Section 2.1.2). Then we introduce code generation tools created to generate code based on models (Section 2.1.3). Finally we finish this chapter but reviewing research works dedicated to automatic code generation for Android applications.

2.1 Model-Based Code-Generation Environments

In this section, first, we explain Domain Specific Languages (DSLs), DSL designing tools and Model to Text Transformation. Then, we discuss a few popular environments such as Acceleo, Xpand, Xtext and JetBrains's MPS and explain how they transform models to source code.

2.1.1 Domain Specific Languages

General-purpose modeling languages such as Unified Modeling Language (UML), as the name suggests, are generic and general. Even though they can be utilized for many applications, in some cases they are not enough, and a tailored engineering tool is required. Domain Specific Languages (DSLs) try to address this issue. They focus on a particular domain and provide a high-level abstraction and tools for developing solutions for problems in that domain. Mernik et.al. [5] carry out an in-depth survey of when and how to develop

DSLs. SQL, HTML, Verilog, Unix shell scripts, Make and MATLAB, are a few examples of domain specific languages.

DSLs have many applications, but despite all their advantages, they also have some disadvantages. For instance, it takes time to learn them as a new language given their limited applicability. Additionally, designing, implementing and modify them require expertise, time and capital.

2.1.2 DSL Designing Tools

Designing a DSL requires a deep analysis of the domain and its features. After defining the system, one of the several available tools is used to develop the language. All of these tools are provided in two forms of graphical or textual editors. Graphical editors provide a GUI for modeling a system and defining components. On the other hand, models are defined and written in text in textual editors. Either way, both kind of editors serve the same purpose of defining models for a language.

Xtext [6], EMFText [7] and JetBrain's MPS [8] are a few examples of popular textual tools. Developed as plugins for major IDEs (Xtext and EMFText for Eclipse and MPS for IntelliJ IDEA), they allow developers to define text syntax for a DSL described by metamodels such as Ecore [9]. Each of these tools has unique features, but all of them offer some standard ones, such as code completion, syntax coloring, quick fixes, etc., that make designing a language easier.

Many graphical tools are also available for designing DSLs. Some of the major ones include GMP [10], AToM3 [11], MetaEdit+ [12] and Visual Studio DSL Tools [13]. Each of these tools has their benefits and shortcomings. AToM3 and MetaEdit+ are standalone applications whereas GMF, and Visual Studio DSL Tools are designed as plugins for Eclipse and Visual Studio respectively. MetaEdit+ is a proprietary tool and available with a paid license that makes it more suitable for industrial purposes. However, all other three are free (Visual Studio DSL Tools for Visual Studio users) that makes them a better choice for many researchers and independent developers. There are a few research works comparing these tools and investigating their similarities

and differences.

In [14], Pelechano et.al. compare Microsoft DSL tools and Eclipse modeling plugins. To evaluate these two, they divided a group of 48 last year undergraduate students of computer science into two groups and each group was asked to develop a DSL (including code generation) with a different tool. After that, they conducted a survey about the experience of working with these tools. The survey includes questions about ease of use, usability and quality of graphical designers, complexity in defining the code generator, etc. Their results show that Eclipse DSL tools have been better accepted than DSL tools and respondents find it more simple, robust and stable.

In [15], De Smedt compares three graphical DSL editors: AToM3, MetaEdit+ and Poseidon for DSLs in terms of speed of development, documentation, platform (Windows, Linux, Mac OS), price, availability of APIs, etc. He concludes that MetaEdit+ does better in their comparison than the other two. It is available for all three major operating systems and provides an extensive transformation and generation tool. However, MetaEdit+ is not free and the price of a standard license is high. He also argues that AToM3 offers almost the same functionality as MetaEdit+ free of cost. When it comes to Poseidon for DSLs, he finds its functionality very limited.

2.1.3 Code Generators

DSLs define a high-level model of a system, and code generators such as Xtend [16], Acceleo [17] and MPS Generator [18] use that model to generate source code. Users use the editor provided with these tools to write down DSL code, and they translate this code to actual source code in the target language. Xtend is specific to Java. The Xtend language is an extension on top of Java and generates Java code behind the scenes. Therefore, its basic syntax is Java and it provides additional features such as extension methods and operator overloading with a new syntax. Xtend is used in Xtext DSL projects to define language aspects including typing rules and generators. All generator rules are defined in Xtend language and then, the DSL code is transformed into Java code. Acceleo and MPS on the other hand, are more general and can

generate code in any format (Java, XML, HTML, JSON, plain text, etc.). This flexibility comes with the cost of writing more code. Using these tools, the user defines generator rules for every line of the generated code. Other than the mentioned features, MPS also enables developers to extend an existing language. This feature offers a great deal of flexibility to DSL designers, and it can be used to add new features to programming languages like Java or another DSL.

2.2 Code Generation for Android

Most of the research works targeting code generation for Android focus on Java code that is responsible for functionality rather than UI. One reason for that is the availability of code generators for Java that can be adopted and used to generate Android code.

In [19], a model-driven approach for developing Android applications is proposed. To produce Android code based on defined models, Abilio G Parada et al. extend GenCode [20] tool that generates Java code based on UML and sequence diagrams. GenCode uses UML diagrams to generate the classes and sequence diagrams to generate function calls and the program flow. Their approach is general and does not target any specific family of applications. To demonstrate their work, the authors use one of the available Android SDL samples (Snake), define its behavior using UML and sequence diagrams and generate the Java code for it.

In [21], the authors target high-performance image processing as their domain. They propose a code generator for RenderScript [1], a framework for running computationally intensive tasks on Android, and FileScript, a stricter version of RenderScript that provides wider compatibility with various CPUs and GPUs. They extend Heterogeneous Image Processing Acceleration (HIPA^{cc}) Framework [2] that includes a DSL based on C++ for defining images and filter masks, and integrate a generator for RenderScript and FileScript into the

[1] https://developer.android.com/guide/topics/renderscript/compute.html
[2] http://hipacc-lang.org

framework. For evaluation, they apply image processing filters and operators (e.g. Sobel and Laplace operators for edge detection and Gaussian blur filter) on a 2048×2048 image using their generated code and compare the results on CPU and GPU with the results from applying the same functions using OpenCL on GPU and OpenCV on CPU. They use an Arndale Board[3] with a Samsung Exynos 5250 running Android 4.2 as their testbed. In [22], the authors discuss an approach that uses ATL and Acceleo to generate code for heterogeneous Android applications. They use ATL to define model-to-model transformation specifications and transform models created by the user into pre-defined intermediate models and then use Acceleo to generate source code from the intermediate models. To demonstrate their work, they generate the code for the Snake sample application included in the Android SDK with some improvement (more directions for movements and new obstacles and entertainment elements) and call it Snake Plus. In [23], the same authors make their work platform- independent and propose sample models for Android and Windows Phone application development. The process of code generation from models is the same as their previous work.

There are also some works dedicated to generating GUI code for the Android platform. In [24], the authors take a model-driven approach for generating GUI code for multiple platforms. Their code generation happens in three steps. First, the system's GUI is modeled in class and object diagrams. Then, these models are transformed into platform-independent XMI files using JDOM API[4]. Finally, they adopt a model-driven architecture (MDA) approach to transform the models into GUI source code specific to a platform. The transformation rules in the last step are defined using ATL (Atlas Transformation Language). Also, their MDA approach takes scripts that describe the target platform as input. They evaluate their work by demonstrating the process of generating Android GUI code using their framework. Related platform scripts for Android are also provided. Mohamed Lachgar, et al. [25], take the same approach of model driven code generation, but they provide a DSL, designed

[3]http://www.arndaleboard.org/wiki/index.php/Main_Page
[4]http://www.jdom.org/

using Xtext, for defining the application and generating the UI code. The system is defined in their DSL and this definition is transformed into intermediate models specific to a platform using Xtend. Then, the source code is generated from intermediate models using Xtend generator. Their framework covers a wide range of platforms including web and mobile operating systems. They apply their approach to the Android platform and Server Faces Framework and generate GUI for two sample application to demonstrate their work.

Chapter 3

Background

We start this chapter with a discussion of Android related technologies that are used in this work including Bluetooth Low Energy (BLE), Android Content Providers, Android services and Android alarms and broadcast receivers. After the first section that lays out the Android technologies used in this work, we introduce a general architecture for applications that work with BLE devices and describe all of the important components that are common among them (Section 3.2).

3.1 Android Related Technologies

In this section, we discuss Android related technologies and components that are, or should be, implemented in every application that works with an external physical device. We will briefly talk about Bluetooth Low Energy (BLE) and its key concepts in Android (Section 3.1.1), Android content providers; what they are and how they work (Section 3.1.2), Android services, which are the components provided by the Android platform to support long-running operations (Section 3.1.3) and Android alarms and broadcast receivers (Section 3.1.4).

3.1.1 Bluetooth Low Energy

Bluetooth Low Energy (BLE, Bluetooth LE or Bluetooth Smart) [1] is a new Bluetooth technology specifically designed for considerably lower energy consumption. BLE is natively supported on all major desktop and mobile operating systems including Android (version 4.3 and above). Even though there are studies that show BLE is not as energy efficient as expected [26], but it is still more efficient than older Bluetooth protocols. For that reason, it is widely adopted by the industry, especially for health and fitness devices that are expected to have a lower power consumption.

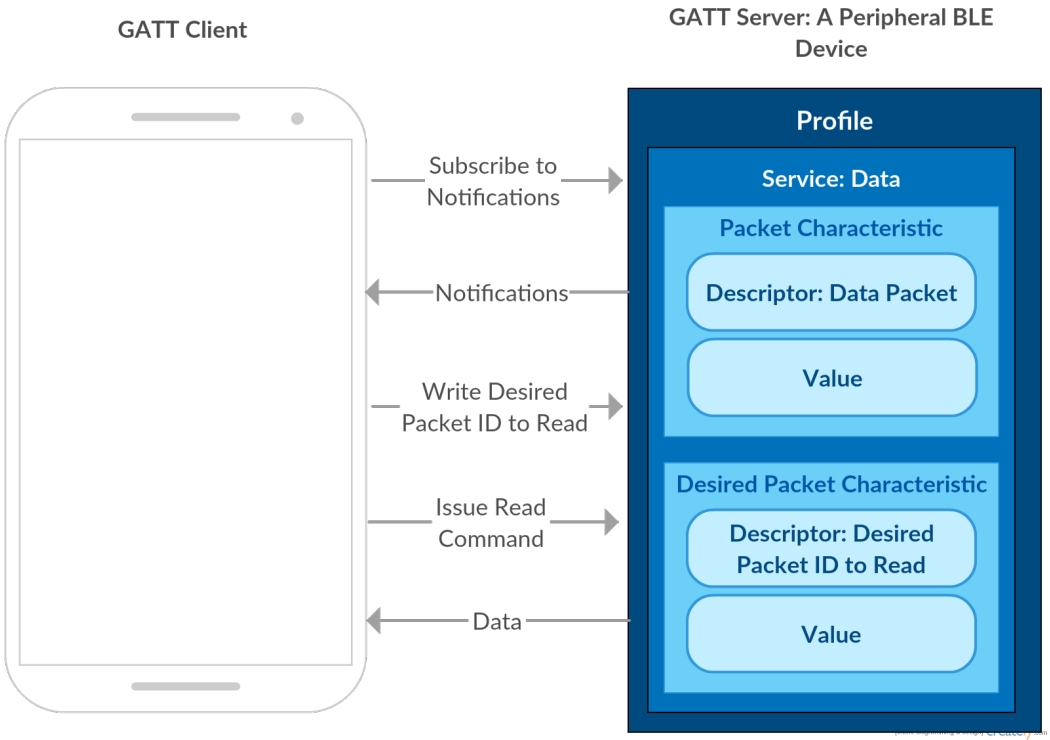


Figure 3.1: Process of a BLE Communication

Figure 3.1 shows an example of a communication between a phone and a BLE device. The BLE devices, which can be an activity tracker, heart rate monitor, etc., gathers data and stores them in packets and provides them when there is a request from the phone. Each packet has a unique ID that identifies that packet. To read the packets, the phone needs to specify the packet IDs.

[1] https://www.bluetooth.com/what-is-bluetooth-technology/bluetooth-technology-basics/low-energy

The important BLE concepts that are related to our work are explained by referring to this example.

- **Characteristic**: A characteristic is a data container. Each characteristic has at least two attributes: *descriptor* which is metadata about the data and *value* which is the actual data. Value is not typed and is considered a non-typed container that can be cast to any type but it has a limit on its length which is 512 bytes. Our example BLE device has two characteristics: *packet* that stores the packet data and *desired packet* that stores the ID of the requested packet by the phone.

  Each characteristic supports four operations: read, write, notify and indicate. Read and write are requested by the Bluetooth client (the phone in our example) and, as the names suggest, are used for reading the current values and writing a new one. Notify, on the other hand, is initiated by the server (the external BLE device) to notify clients that new data is written to a characteristic. So, if a client subscribes to notifications on a characteristic it receives a notification from the server when new data is available. Indicate works in the same way as notify except for one crucial difference: indications are acknowledged by the client, while notifications are not. Obviously, the decision of using notifications or indications for tracking a characteristic depends on the required level of reliability and power consumption restrictions. In Figure 3.1, the phone subscribes to notifications on the *packet* characteristic and if there is new data available, the BLE device sends a notification to the phone. After receiving a notification, the phone writes the desired packet ID on the *desired packet* characteristic using the write operation and then issues a read command for the *packet* characteristic. The target packet will be returned back to the phone. If there are multiple packets available the process of writing a packet ID and reading a packet must be done one by one.

- **Service**: A BLE service is a collection of related characteristics. In our example, the data service includes *packet* and *desired packet* characteristics. On an activity tracker device, a data service can contain characteristics such

as the number of available packets, the actual packet data, etc. A device information service on a device can include information such as battery level, temperature, etc.

- **Profile**: Profile is a pre-defined collection of services. It does not exist on the BLE device; rather it is a concept for grouping services. A device supports a profile based on its application (heart rate sensors, movement sensors, etc.). Two BLE devices that are compatible with each other support the same profile.

- **GATT**: GATT stands for Generic Attribute Profile, and defines the protocol to exchange profile and user data over a BLE connection using **characteristics** and **services**.

- **GATT Server**: In a BLE setup, the server is the peripheral device that has data, characteristic and service definitions on it.

- **GATT Client**: Client is the device (a phone, tablet, etc.) that sends requests for data to the server.

- **UUID**: A Universally Unique Identifier is a 128-bit identifier that is globally unique (with a high probability). One of its applications is identifying BLE elements such as characteristics and services. Each element is assigned a UUID that is used to access that element. This approach makes it possible to access each characteristic independently from the others.

Android 4.3 (API Level 18) introduced built-in support for BLE [27]. Even though APIs for scanning for devices, establishing connections, reading characteristic and writing them are provided by Android platform, synchronization for reading or writing multiple characteristics at the same time must be handled by the developer. Communicating data of various characteristics must be done in a serial fashion. More specifically, *characteristicRead*, which is the function that handles read operation on Android, executes synchronously and the client must wait for the results before issuing another call. Common

synchronization techniques such as synchronous queues, locks, condition variables, etc. can be used to implement the serial communication and overcome this issue.

3.1.2 Android Content Providers

Content providers are the standard way of handling data access and storage on Android. They provide a layer of abstraction between the underlying data storage and rest of the code. In this way, the application code is independent of the underlying data persistence approach, and this provides flexibility to developers. Data persistence can be implemented using a common Database Management System (DBMS) such as SQLite, or REST APIs.

Content providers offer data encapsulation based on URI's. These URIs can be used to perform create, read, update and delete operations in applications. Additionally, implementing content providers makes it possible to share data with other applications in a standard and secure fashion. Access to content providers and possible operations can be managed using Android permission system that works in the same way as requesting access to resources or hardware. So, an application that has the right permissions for accessing or updating the content provider can perform the operation using the designated URI.

The concepts related to content providers that are used or mentioned in this work are as follows.

- **Authority**: Authority is a string that identifies a provider and is used to access it. For that reason, it must be unique, and it is a good practice to use the reverse domain name notation, the same naming convention used for Java packages, to define it.

- **Content URIs**: A content URI identifies specific data in a provider. Authority plus a name that points to a table, identifies a table in a provider. In general, a table is accessed using a URI in the following format:

```
content://authority/table-name
```

- **Cursors**: Cursors are sets of data returned by a database query. They provide various methods for accessing records and iterating through them. As a storage management system, Android content providers use cursors for data access and modification.

- **Content Resolvers**: *ContentResolver* class provides client objects that can be used to access data in content providers. A Content Resolver client object has methods that call corresponding methods in a provider object (an instance of the Content Provider class or its subclasses) to perform the desired operations. These methods provide create, query, update, and delete operations. When a content resolver is used, all of the inter-process communications between a provider and its client are automatically handled by Android.

- **Access Permissions**: As discussed before, content providers can be used to share data between apps. This raises many security concerns about illegitimate access and modifications. To address this issue, Android extends its permission system to content providers. A provider's application specifies a permission that other applications should request to access or modify data in a content provider. A usual permission is defined like this:

```
ca.redliner.READ_REDLINER_PACKETS
```

  The client requests the permission in *AndroidManifest.xml* like any other permissions (such as access to the Internet or location, reading contacts, etc.):

```
<uses-permission android:name="ca.redliner.
   READ_REDLINER_PACKETS">
```

  As a result, the end user will see the permission while installing the application and will be informed what data is being accessed.

In summary, implementing and accessing a content provider happens in

this order:

1. The provider implements a subclass of *ContentProvider* class.
2. The provider implements query, insert, update and delete methods.
3. The provider declares the content provider in the AndroidManifest.
4. The provider specifies a unique authority and paths to data tables.
5. The client requests the permissions required for accessing the provider's data.
6. The client creates a *ContentResolver* object and uses content URIs to access the data or modify it.

Even though we discussed the general application of content providers, i.e. provider being one application and client being another one, both provider and client can be one Android application.

Implementing a content provider is complicated and requires expertise and experience. A developer needs extensive knowledge of how they work and how they operate within the Android platform. Also, the amount of written code is high and extensive testing is required to ensure that implemented components are working properly. For those reasons, many developers neglect to implement one and write plain SQL in their application code for handling data storage and end up with non-scalable software. Content providers are essential to the scalability of an application and quality data management in Android.

To generate a content provider, we integrated Android Content Provider Generator[2] tool into our framework. Other than a content provider, this tool creates a Content Resolver that is a single and global instance in Android applications providing access to content providers.

Android Content Provider Generator also generates helper and wrapper methods for creating structured data and inserting them into the database, selections, projections, etc. This tool requires at least two JSON files to function. One is for defining application-specific configurations that include entries

[2]https://github.com/BoD/android-contentprovider-generator

such as package names, database file name, database related class names, etc. It also needs at least one table definition. If the database designed for an application has more than one table, each table must be defined in a separate file. All constraints for a table are also defined in the same file as the table definition.

3.1.3 Android Services

Some long-running operations such as network transmissions or data communications with external hardware devices should be done in the background and do not need a user interface. Android services are implemented to perform this kind of operations, and by using them, the life cycle of the operation does not depend on the application's life cycle. So, when the user is not using the application or when it is closed, the background operation can continue performing its task. Also, when Android is low on memory, some processes are killed by the system so that their resources can be reclaimed. To determine which processes should be killed, Android puts each running process in an importance hierarchy based on the application's running components and their status. The process types based in order of importance are: foreground processes, visible processes, service processes and background processes. Foreground processes have the highest priority. An application running an activity at the top of the screen that the user is interacting with or a service that is currently executing code in the background are considered foreground processes. When a service is not executing code, it is a service process. In both cases, the chances of a service being killed are very low [28]. That is one of the main reasons for considering Android service for long-running operations. However, in cases that the work needs to be done only while the user is interacting with the app a simple thread is sufficient.

Since periodic data transfers are required between a BLE device and a smartphone, an Android service is the perfect tool for handling communications. Since, BLE supports notifications from the device when data is available, notifications set on different characteristics can wake the application which in turn runs the service to read the data, store it in the database and update the

application.

3.1.4 Android Alarms and Broadcast Receivers

For scheduling and managing periodic tasks in Android, alarms and broadcast receivers are used on top of services. Android alarms are designed for running time-based operations outside the lifetime of an application. Developers set an alarm and when it goes off a message is sent to a corresponding broadcast receiver that is defined for that alarm. Broadcast receivers are Android components that let applications register for application or system-wide events. For periodic tasks such as reading data from a BLE device, an alarm is set that sends a message to a broadcast receiver which in turn runs a service. The service will take it from there and starts the reading operation.

3.2 Generic Architecture of BLE Applications

To define our models for automatic code generation, we designed a general architecture for BLE applications that communicate with peripheral devices. Fig. 3.2 demonstrates how this architecture is laid out and shows its essential components that are as follows:

- **Device Scanner**: This component is in charge of searching for devices using the BLE component, pairing with them and storing their information.
- **Service**: Service component handles all the background work and connects all the other components to each other.
- **BLE**: All Bluetooth communications, including searching for devices, connecting to and disconnecting from them and writing and reading data, are handled in this component.
- **Data Storage**: The content provider and all its parts are a part of this component.

We go through each of these components and discuss how they work and interact with other components.

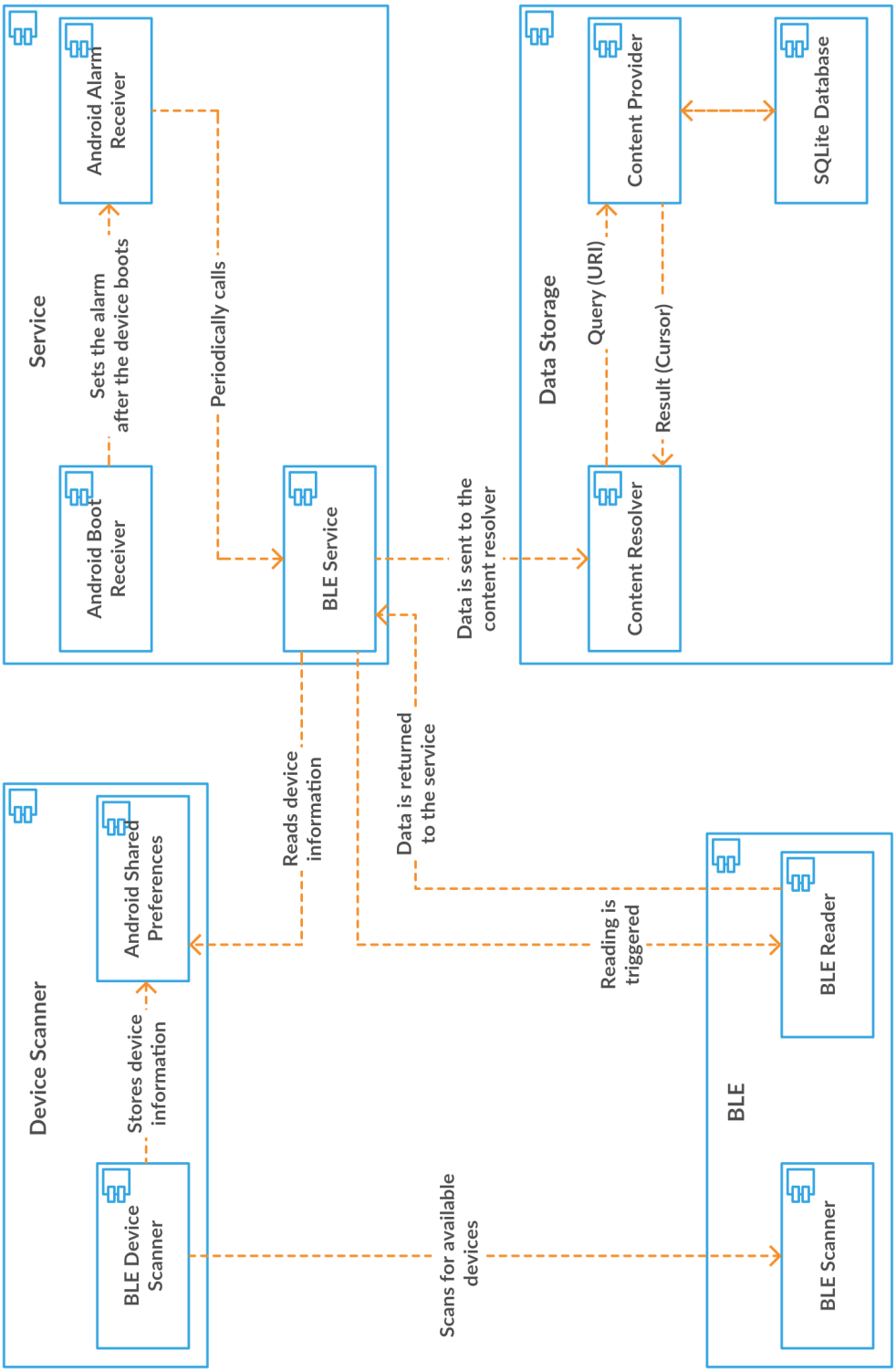


Figure 3.2: Common Components of Android Applications Working with BLE Devices

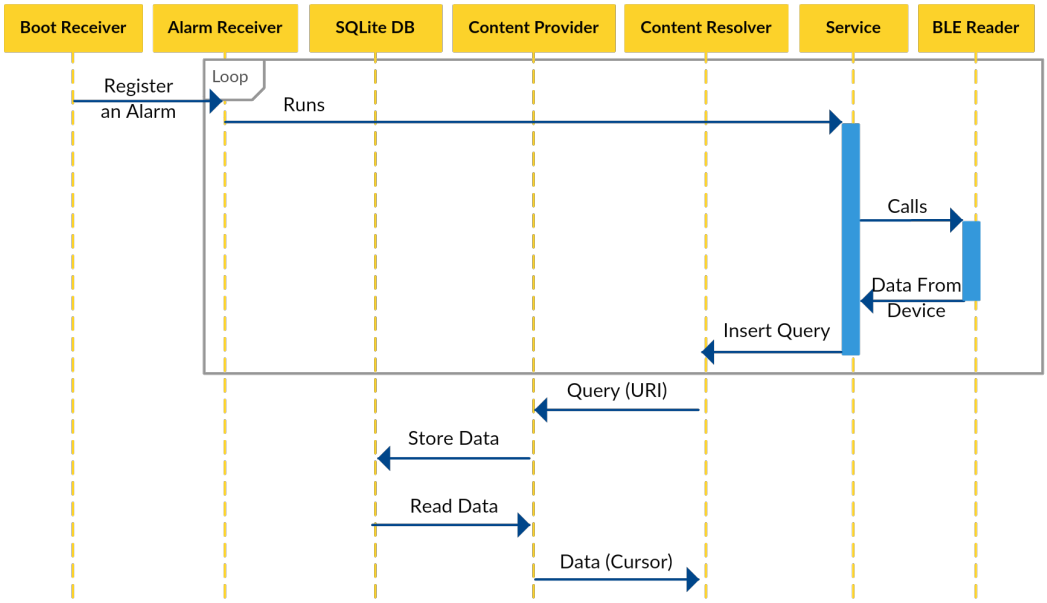


Figure 3.3: Sequence of Events Happening During Reading and Storing BLE Data in Our Architecture

3.2.1 Device Scanner

The device scanner calls the BLE scanner (from the BLE component) and lists all the nearby devices. The end user of the application picks the device they want to pair with the application, and the hardware information about that device is stored using Android Shared Preferences [3]. Later when the service is started to read the data from the device, this information is retrieved from Shared Preferences and passed to the BLE reader. Figure 3.4 depicts this process.

3.2.2 Service

When the boot process is finished in Android, a BOOT_COMPLETED broadcast message is broadcast. This message helps developers find out when the boot process is finished so they can schedule tasks including periodic ones. A boot receiver, which is a subclass of Android Broadcast Receiver, listens for this message and initiates the tasks it is asked to run.

[3]Shared Preferences, as a kind of data storage, allow developers to save and retrieve data in the form of key, value pair in a simple and easy way.

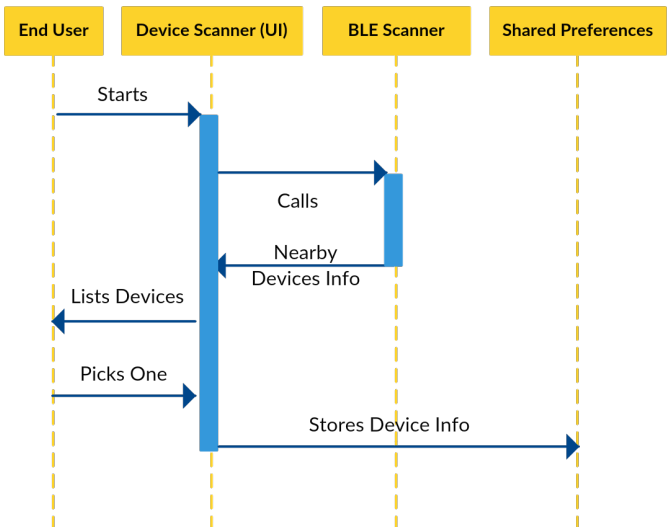


Figure 3.4: Sequence of Events When Pairing a BLE Device With a Phone

Also, for applications that perform time-based operations outside of their lifetime, such as communicating with a peripheral device or periodic network communications, repeating alarms [4] should be set for specific periods of time to initiate tasks like running a service. This is done using Alarm Receiver class which is another broadcast receiver. Boot receiver and alarm receiver classes are usually used together for implementing periodic tasks in Android.

In our design, when the alarm set by the boot receiver class goes off, the alarm receiver gets the message and calls the Android service. This service is responsible for collecting and storing data. At first, it reads the device information stored in a shared preferences file by the device scanner component. Then, it triggers the BLE read action and waits for the results to come back from the BLE component. At last, when the data is ready, it will be passed to the content resolver with an insert query by this service.

As shown in figure 3.2, the service connects all the components in our architecture to each other.

3.2.3 Data Storage

This component is fully implemented using Android content providers. The data collected by the service running in the background is handed to a content

[4]https://developer.android.com/training/scheduling/alarms.html

resolver which is an abstraction layer for content provider operations. The data is automatically handled by the content provider based on the query from the content resolver. Implementing a content provider makes this architecture more flexible. The data can be shared with other applications if required, and the underlying data persistence mechanism can be any DBMS or REST APIs. We used SQLite which is very lightweight and is used as the default DBMS in Android application.

3.2.4 BLE

BLE scanner component is used by two other components: BLE device scanner which is an activity listing surrounding devices and the BLE service that is in charge of reading and storing data periodically. The scanner is generic and can be used in any application that works with BLE devices. On the contrary, BLE reader is device specific, and the implementation heavily depends on the protocol defined on the physical device. The possibility of accessing each characteristic individually or getting a notification when new data is written to a characteristic are among the main benefits of the BLE standard since they reduce energy consumption. At the same time, reading from a device should be done in a synchronous matter, and characteristics should be read one by one. So, reading the data altogether is not possible, and even if there is a synchronization mechanism in place to do so, it won't be energy efficient. For these reasons, BLE reader is the only component in our architecture that should be implemented by the user and is neither automatically generated nor pre-written.

Chapter 4

The AHL Application Construction Framework

We start this chapter with a discussion about Android Content Provider Generator library and how it works (Section 4.1). Then we introduce MPS (Section 4.2), how we designed our DSL with it (Section 4.3) and the concepts of our language (Section 4.3.1). After that, we put forward the plugin we have created for IntelliJ IDEA that adds our language to this IDE (Section 4.4). In the end, we discuss the code generation process (Section 4.5) and AHL Wrapper, a Java class that finishes the work by running Android Content Provider Generator and moving generated classes to their right place in an Android project after code generation (Section 4.6).

4.1 Android Content Provider Generator

As mentioned before, Android Content Provider Generator (ACPG) makes the tedious task of implementing a content provider easy and quick. It takes JSON files as input and generates all the Java classes for data types, a content provider, content resolvers and helper classes for creating structured data.

To generate a content provider, ACPG needs one general configurations file and at least one table definition. The configuration file provides the library with some information about the system including authority, applications package name, Java class names for different classes (provider, helper classes, etc.), provider package name and so on. Table definition files include

column names, column types, optional documentation for the table and its columns, whether a column is nullable or not, and constraints such as field being unique, foreign keys, conflict resolution, etc.

Listing 4.1 depicts definition of a simple table with two columns. This table has two columns: *packet id* and *start time*, both of type Integer. *Start time* can be null but *packet id* cannot. Additionally, this table has a constraint; *packet id* should be unique, and an insert operation with an already existing id will be ignored (line 22). As the code shows, developers can also write documentation lines in this file.

```
1
2  {
3    "documentation": "Packets read from the Redliner device",
4    "fields": [
5      {
6        "documentation": "Packet ID",
7        "name": "packet_id",
8        "type": "Integer",
9        "nullable": false
10     },
11     {
12       "documentation": "Start Time",
13       "name": "start_time",
14       "type": "Integer",
15       "nullable": true
16     },
17   ],
18
19   "constraints": [
20     {
21       "name": "unique_packet_id",
22       "definition": "UNIQUE (packet_id) ON CONFLICT IGNORE"
23     }
24   ]
25 }
```

Listing 4.1: Database Table Definition

A sample query to the database using classes generated by ACPG can be done in the way shown in Listing 4.2. *PacketSelection* and *PacketColumns* are examples of generated helper classes for querying, inserting and updating data. A selection class is created for every table that includes helper methods for making selections. In our example of a Packets table, rows that have

packet id of 0000 or start time of 0000 will be selected (line 3). Later in line 5, this selection is passed to the *query* method to return the rows with the desired values. There is also a *columns* class generated for every table. These classes include column names and the URIs of tables for easy access. *PacketColumns.CONTENT_URI*, in this example, provides the URI for the Packets table.

```
1 PacketSelection selection = new PacketSelection();
2
3 selection.packetId("0000").or().start_time(0000);
4
5 Cursor c = context.getContentResolver().query(PacketColumns.
    CONTENT_URI, projection, selection.sel(), selection.args(),
     null);
```

Listing 4.2: Sample Database Query

4.2 JetBrains Metaprogramming System (MPS)

We used MPS for implementing the idea behind this work, mainly because it integrates with IntelliJ IDEA and Android Studio, the official Android IDE, is an IntelliJ IDEA variant. MPS implements a non-textual presentation of code. In this approach that takes advantage of Projectional Editors, developers do not write code in a plain text format. Every expression of a language is broken into cells, and each cell only accepts special keywords or properties that are correct in the context. Since DSLs are mostly being used by domain experts, not professional programmers, projectional editors make things easier by asking for a specific value for each cell. Also, this approach eliminates the need for a parser and provides flexibility for designing languages.

MPS always maintains code in an Abstract Syntax Tree (AST). The AST consists of nodes that have properties and children. A DSL can be created using three approaches: first, writing a generator that converts developer's AST to an intermediate AST that is understood by the MPS generator and will be transformed into code. This method uses model to model and model to text transformations. Second, writing text generators to directly transform

the AST to code, which is a model to text transformation. The third method is a combination of these two.

For model to model or model to text transformation, the generated code depends on the templates written by the developer. As an example of model to text transformation, if we have a data model called packet defined using the syntax shown in Listing 4.3 and the developer decides to generate a Java class from this model, two templates should be created to associate the model properties with the desired elements of a Java class and generate the code. First, a text generator is defined to create the Packet class that can have any number of properties. Listing 4.4 shows this generator.

```
1 Packet[
2   packetId:long,
3   packetData:String
4 ]
```

Listing 4.3: Sample Data Model

```
1  text gen component for concept Packet {
2  file name : <Node.name>
3  extension : (node)->string {
4    ".java";
5  }
6  encoding : utf-8
7  text layout : <no layout>
8    (context, buffer, node)->void {
9      append {public class } ${node.name} { {} \n ;
10     increase depth ;
11     append $list{node.listOfPacketProperties} \n ;
12     decrease depth ;
13     append {}} ;
14   }
15 }
```

Listing 4.4: Text Generator for Packet Class

The generated file name and its extension are defined at the top of every text generator. Then, starting from line 9, the text that we want to generate is appended to the output buffer. *node* in this piece of code represents a Packet model. So, if we create a model of type Packet and name it Data the following

line:

```
append {public class } ${node.name} { {} \n ;
```

results in the following output:

```
public class Data {
```

The *$listnode.listOfPacketProperties* line adds all the generated code from text generators of every property to the Packet model (*listOfPacketProperties* is a property defined for this model). It is also possible to manage indentation in the generated class with *increase depth* and *decrease depth* commands.

Listing 4.5 shows the text generator that generates the code for the declaration of each property in the Packet model and the associated getters and setters. Each property is defined using the PacketProperty model that includes a name and a type for that property. *node* in this text generator represents a PacketProperty model. It is worth mentioning that this generator does not create a file (hence no sign of defining file name or extension) and just generates the code that is appended to the output buffer in the Packet model generator.

```
1  text gen component for concept PacketProperty {
2    (context, buffer, node)->void {
3      append \n ;
4      append {private } ${node.type} { } ${node.name} {;} \n \n ;
5
6      append {public } ${node.type} { } {get} ${node.name} {(){}
          \n ;
7      increase depth ;
8      append {return } ${node.name} {;} \n ;
9      decrease depth ;
10     append {}} ;
11
12     append \n \n ;
13
14     append {public void set} ${node.name} {(} ${node.type} { }
          ${node.name} {) {} \n ;
15     increase depth ;
16     append {this.} ${node.name} { = } ${node.name} {;} \n ;
17     decrease depth ;
18     append {}} ;
```

```
19
20      append \n;
21
22    }
23  }
```

Listing 4.5: Text Generator for Properties

Using these two generators the Packet model is transformed into a complete .java file with all the properties, access modifiers and getter and setter methods. This resulting Java class is demonstrated in Listing 4.6.

```
1  public class Packet {
2
3    private long PacketId;
4
5    public long getPacketId(){
6      return PacketId;
7    }
8
9    public void setPacketId(long PacketId) {
10     this.PacketId = PacketId;
11   }
12
13   private String PacketData;
14
15   public String getPacketData(){
16     return PacketData;
17   }
18
19   public void setPacketData(String PacketData) {
20     this.PacketData = PacketData;
21   }
22
23 }
```

Listing 4.6: Generated Java Code

4.3 Designing AHL in MPS

In order to design AHL, we used the following aspects of a language provided in MPS for defining a DSL:

- **Structure**: Structure part of a language defines all the concepts included in that language. We defined our basic concepts such as a BLE characteristic

or a database column. Then, more complex concepts were defined based on these basic ones: a database table consisting of multiple database columns, a BLE service consisting of multiple characteristics and characteristic packs, etc.

- **Editor**: Each concept has an editor. These editors define how a concept is displayed to the user when a file of that concept is created. Fig 4.1 shows the editor for each database column in AHL.

```
<default> editor for concept DbColumn
  node cell layout:
    [- { name } : { type } : { nullable } : { documentation } -]

  inspected cell layout:
    <choose cell model>
```

Figure 4.1: A Simple Concept Editor

After defining this editor, a database column will be portrayed in the same way in any place that the database column concept is used. Figure 4.2 shows the effect of this editor when the database column is used in the table concept.

```
Table: <no name> {
  <no name> : <no type> : false : <no documentation>
  <no name> : <no type> : false : <no documentation>
  <no name> : <no type> : false : <no documentation>
}
```

Figure 4.2: Effect of the Concept Editor

- **TextGen**: TextGen aspect defines model to text transformation in MPS. TextGens can generate source code files or snippets based on the type of the model. There is no limit on the target language or file format since these attributes are defined by the DSL developer.

- **Generator**: Generators in MPS perform model to model transformations. A project can have multiple generators, and each generator can have multiple rules including pre and post processing scripts, reduction and mapping

rules, etc. In Section 4.5, we will discuss generators and the way we used them in details.

4.3.1 AHL Concepts

In MPS, concepts are definitions that describe the abstract structure of a syntax element in a DSL. Therefore, creating a DSL starts with defining its concepts. Our language includes the following concepts (Figure 4.3):

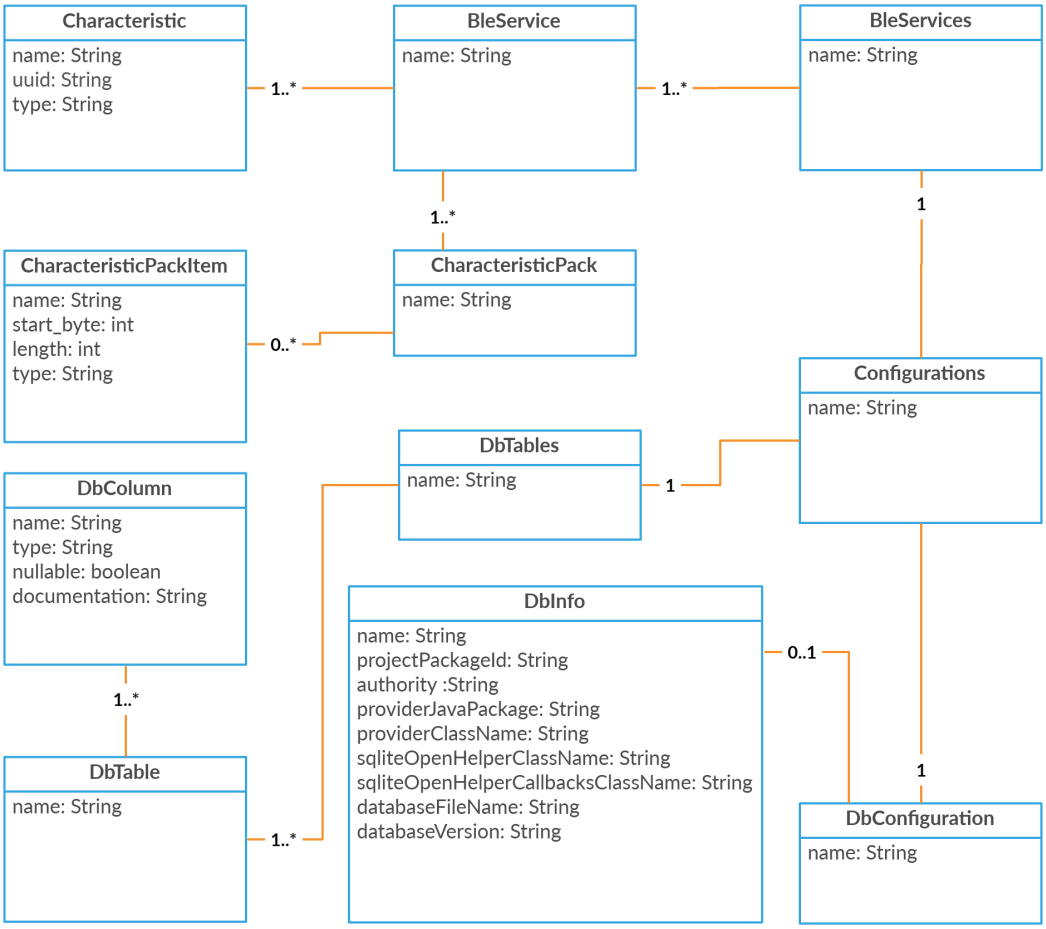


Figure 4.3: AHL Metamodel

- **Characteristic**: This concept defines a single BLE characteristic. Every characteristic concept has three properties: name, uuid, and type.
- **CharacteristicPack**: In some cases, developers bundle related data items in a pack and access each item using its length and start byte. Character-

isticPack defines that kind of data representation and has name and uuid properties.

- **CharacteristicPackItem**: This concept defines single items in a characteristic pack and has four properties: name, start_byte, length, and type.
- **BleService**: BleService concept defines a single BLE service. Each concept of this type includes a list of characteristics and a list of optional characteristic packs. Depending on the BLE profile, there can be one or multiple BLE services on the physical peripheral device.
- **BleServices**: This concept packs single BleService concepts together. The main purpose of this concept is an easier model to model transformation that will be explained later.
- **Configurations**: Configurations is the biggest concept in our language and includes database configurations, database tables, and BLE services. This concept is mainly designed so that developers can easily provide all of the information required for code generation in one single file.
- **DbColumn**: This concept defines a database column that is used in DbTable concept. DbColumn properties are: type, nullable and documentation (comment).
- **DbInfo**: DbInfo defines information about the content provider and database classes to be generated, including database file name and version, class and package names for the content provider, etc.
- **DbTable**: DbTable defines a database table and includes a list of database column concepts.
- **DbTables**: This concept defines a list of database tables. Mainly used for an easier model to model transformation.

4.4 The Application Specification Plugin

MPS comes with tools that give DSL developers an option to create IntelliiJ IDEA plugins for their languages and distribute it to users. DSL developers can create a build script in their MPS project and define features of their plugin. They also need a working installation of IntellJ IDEA with MPS plugin installed on it that will be referenced in this script and will be used for generating the DSL plugin. This script will generate an Apache Ant build script that includes all the information about the designed language, the required features and the path to IntelliJ IDEA installation. Running this Ant script, either in MPS or independently using Ant build tools, will create a zip file as the plugin which can be installed on IntelliJ IDEA.

We used this feature to create a plugin that packs our language and its concepts, editors, model transformations and code generators. This plugin, in combination with MPS plugin for IntelliJ IDEA, provides the tools for creating AHL models and write the code in this language.

To use AHL scripts and code generators, the developer creates an MPS module in their Android project and specify AHL as the language they want to use. After that, they will be able to create AHL root models (Fig. 4.4). These models include database configurations, database tables, BLE Service (a single service), BLE Services (a collection of services), and BLE Characteristic Packs (a collection of BLE characteristics) that can be created separately. Another option is creating an AHL configuration model that includes all of the other models (Fig. 4.5). Both methods generate the same code. We provided both options to make organizing and managing models easier for small and big projects.

4.5 The Process of Code Generation

In this section, we describe the steps involved in the process of generating code using our framework.

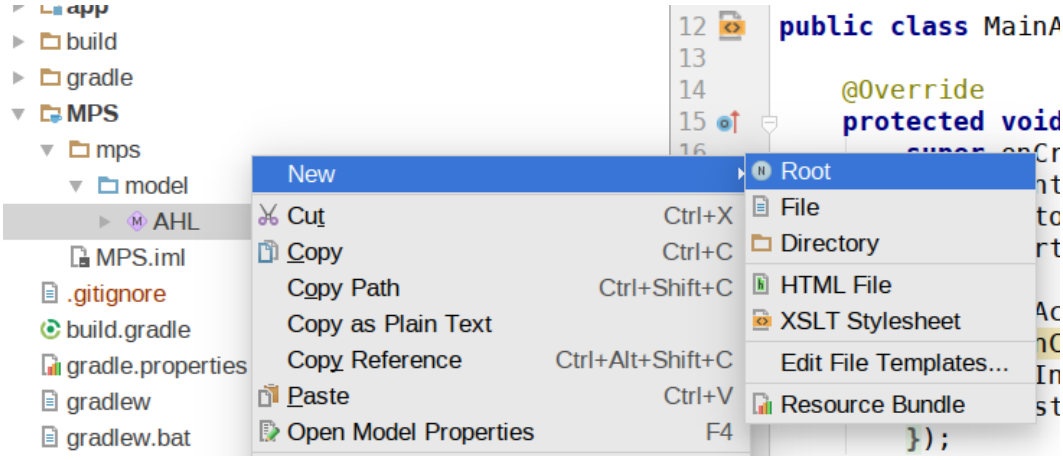


Figure 4.4: Creating Models in IntelliJ IDEA

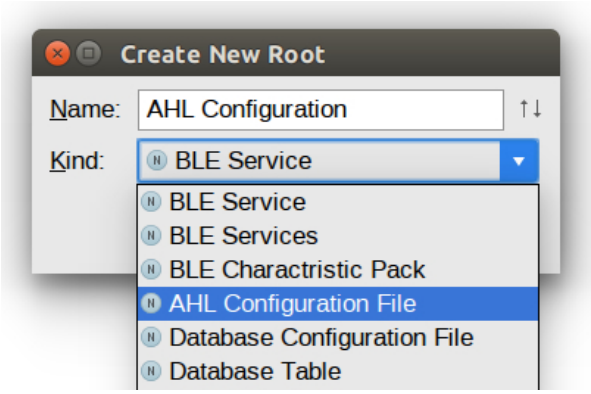


Figure 4.5: Available AHL Models

4.5.1 Creating Concepts and Writing AHL Code

As previously mentioned, all AHL concepts are included in the provided plugin and are available to the user. These concepts can be created like a new file in the project. After creating concepts, the user can start writing AHL code. The code written in each concept follows the rules defined in the editor for it.

4.5.2 Running MPS and AHL Wrapper

We mentioned that model to text transformation can be defined using the TextGen aspect of a language in MPS. We created TextGen for five of our major models in ordered to generated code for solutions defined based on the following concepts:

- **CharacteristicPackItem**: Every characteristic pack item is transformed

into a Java property with corresponding getters and setters. The generated code from this TextGen for each item will be used in the class generated for each characteristic pack.

- **CharacteristicPack**: Each characteristic pack is turned into a java class. The TextGen for this concept defines the class and appends the code generated from each characteristic pack item.

- **BLEService**: The TextGen for each BLE service generates a java class with single characteristic and characteristic packs as properties and getters and setters for each property. It is taken into account that each characteristic pack is a class and a type itself.

- **DbInfo**: This model is transformed into a JSON file named _config.json which is one of the files required by the Android Content Provider Generator library for generating the content provider.

- **DbTable**: There can be one or more database tables defined based on DbTable. This concept has a TextGen that transforms it into a JSON file describing the table based on the Android Content Provider Generator library standards. Each table defined based on DbTable will be converted to a separate JSON file.

Other than having separate concepts for different components of our framework, we have also defined an all-in-one Configuration concept that includes all the other concepts. Using this concept, users can provide all the information required by the framework in a single solution file. However, it is not possible to have multiple TextGens for a single concept and generate multiple source code files from it. So, we defined model to model transformations using the generator aspect of languages in MPS to map each part of Configuration to other concepts and then execute the model to text transformations using already defined TextGens.

As a part of the code generation process, we created a Java class called AHL Wrapper that runs the Android Content Provider Generator library as

one of its tasks. This library generates the code for the data storage component of our architecture.

4.6 AHL Wrapper

MPS does not support creating project structure and generating files in multiple folders or packages. All the generated files and classes are placed in a single folder. Additionally, even though configuration files for Android Content Provider Generator are generated using MPS, running it as a Java application cannot be done with MPS. To overcome these issues, we created a wrapper class written in Java that performs the following tasks after the MPS code generation is done:

- **Creating packaging structure**: At first, the wrapper class creates all the required Java packages in the project.
- **Moving files and classes**: Then, the wrapper class will move generated and the pre-written classes and resource files to their right place in the Android project.
- **Running Android Content Provider Generator**: Using the JSON configuration files generated by MPS and moved by the wrapper as the input, ACPG library is called to generates the content provider in its right package in the Android project.
- **Giving the update instructions for Gradle build files and AndroidManifest.xml**: At the end the wrapper class prints out some instructions for updating two sets of files: a) Gradle build files for adding the required libraries to the Android project and b) the AndroidManifest.xml for requesting permissions and adding the service and the content provider definitions to the Manifest.

After defining the configurations and building the Android project for the MPS generator to kick in and generate the files, the user runs the wrapper to perform the mentioned tasks and finish up the work.

Chapter 5

Developing an Application with AHL Framework

In this chapter, we introduce Redliner as an activity tracker system for wheelchair users. Then we go through the steps to generate the code for multiple components of the accompanying Android application using AHL IntelliJ IDEA plugin and AHL Wrapper. In order to do this, we create an Android project and start the code generating process. When it's done, the generated code will be added to the project and is ready to use.

5.1 What is Redliner?

Redliner is a wheelchair activity monitor that can measure upper body over-exertion (Redline events) for wheelchair users. By measuring different metrics, the Redliner system allows users to be aware of their activities and be notified in case of over-exertion. A peripheral device, which includes multiple sensors and has a BLE module installed on it, is attached to the wheelchair and connects to Android phones. The Redliner activity monitor, in the form of an Android application, periodically connects to this device via BLE, reads the data and stores it in a database on the device. Then the app uses this data to visualize multiple metrics such as Redline events, velocity, the number of pushes, etc.

5.2 Generating the Redliner Application with AHL

In this section, we will describe the process of generating the code for the Redliner application. We assume that IntelliJ IDEA with MPS plugin is already installed and ready to use.

- **Step 1 - Installing the AHL Plugin:** We install the AHL plugin for IntelliJ IDEA which is provided as a zip file. This can be done in the *Plugins* section of IDEA settings by choosing the *Install plugin from disk...* option.

- **Step 2 - Creating a New Android Project:** IntelliJ IDEA community edition comes with Android Studio plugin pre-installed. We create a standard Android project in IntelliJ IDEA (Figure 5.1).

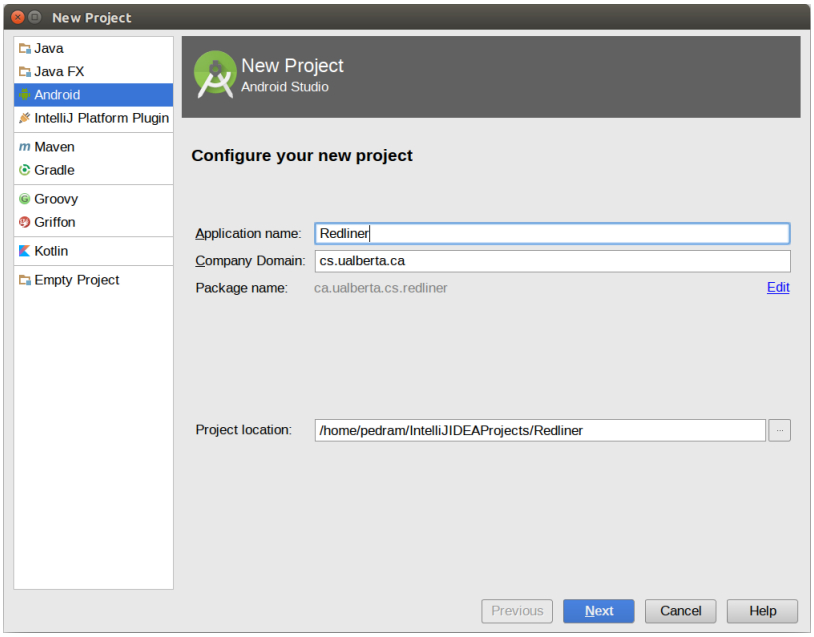
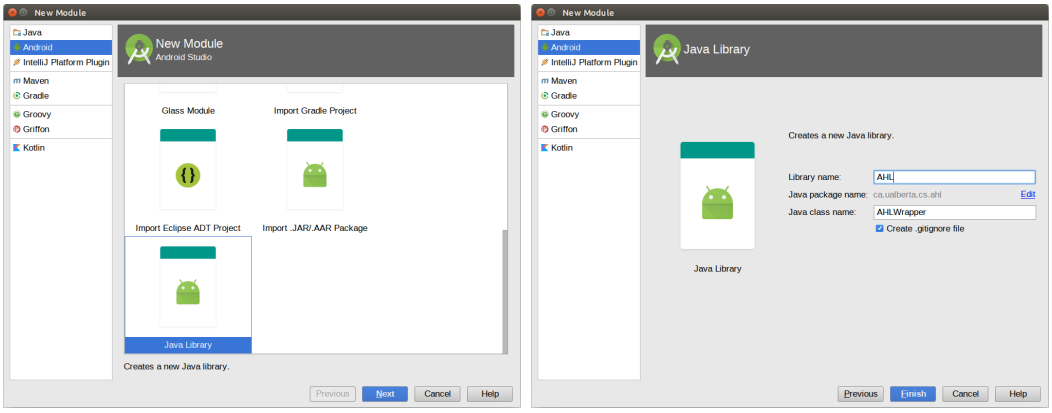


Figure 5.1: Creating a New Android Project

- **Step 3 - Creating a Java Library Module:** We create an Android Java Library module for the application. This can be done by a right-click on the project's name and selecting $New > Module > Android > JavaLibrary$



(a) Creating an Android Java Library (b) Java Library Configuration

Figure 5.2: Android Java Library Module Configuration

(Figure 5.2a). We will name this module AHL and configure it in the way demonstrated in (Figure 5.2b.

- **Step 4 - Copying the AHL Development Bundle into the Project:** Then we copy the AHL Development Bundle folder into the root of the newly created Java library module. This folder includes the following items: a set of generic Java classes that are already written and can be used without any changes, Android Content Provider Generator library, a src_gen folder as the destination for the MPS code generator and an src folder that includes the AHL Wrapper Java class.

- **Step 5 - Creating an MPS Module:** Now it's time to setup MPS. We add an MPS module to the project. MPS is listed under the Java section in the *New Module* window (Figure. 5.3). The relative path to models folder can be customized while creating the module. This path only matters to the MPS module and can be changed or left as default. Either way, it won't affect the code generation process.

- **Step 6 - Configuring the MPS Module:** Now if we right click on the project (or on any other module) and choose *Open Module Settings* item, we can access the module settings for the project and configure our MPS module. Under the Paths tab, we set the output folder for the generated code to *src_gen* in our AHL module. Also, we add AHL as a language

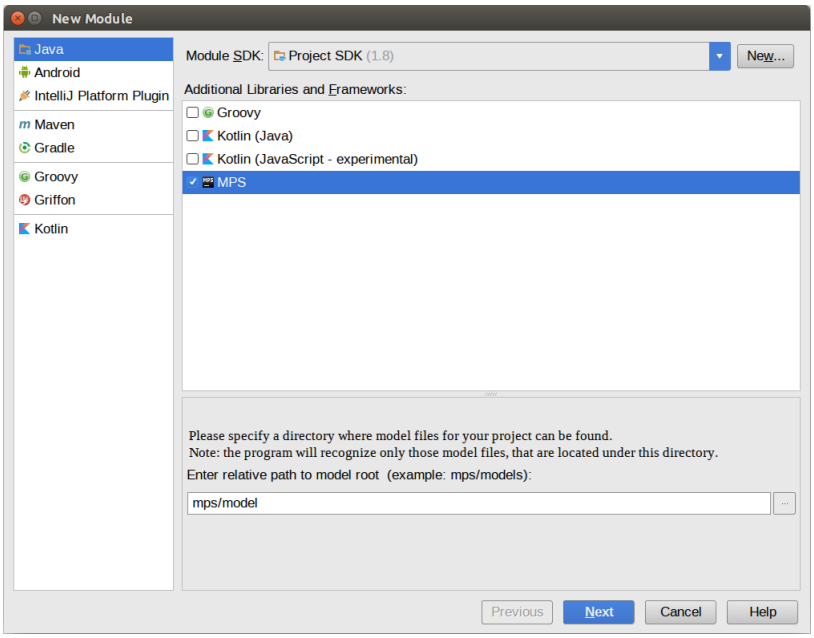


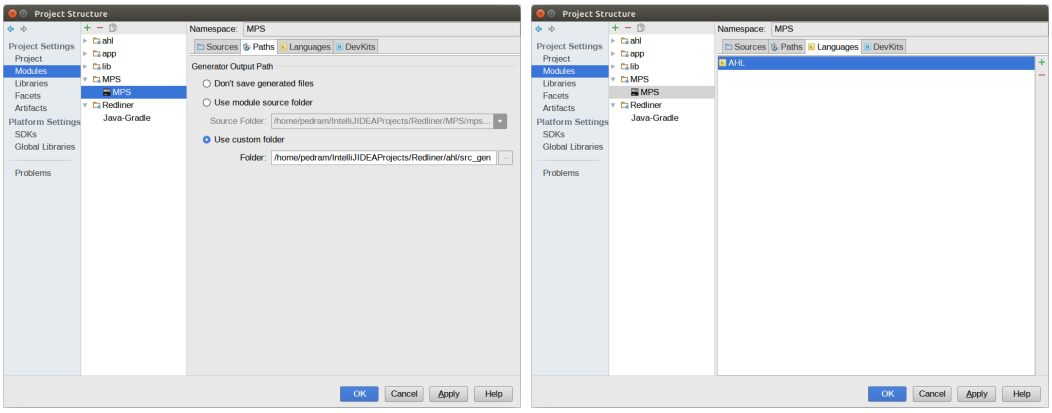
Figure 5.3: Adding a MPS Module to the Project

to the MPS module under Languages tab. Figure 5.4 demonstrates these configurations.

After these steps, the project structure will look like Figure 5.5. The *app* module, the default module for Android applications, can be seen in the structure.

- **Step 7 - Creating AHL Concepts:** Now we can right click on the model folder in the MPS module (specified in Step 5, while creating the module) and create a new root model and name it AHL. AHL defined concepts can be created under this model. We create an *AHL Configuration File* to define all the configurations for Redliner. As explained before, there is an alternate way of defining configurations. Each section of the *Configuration* concept can be a separate file which is beneficial for big projects. However, for the sake of simplicity, we will stick to defining everything in a single file using the *Configuration* concept.

  The newly created file looks like Figure 5.6. By pressing Enter key on << ... >> signs, the definition for that sections will be generated. Since MPS uses projectional editors, everything that appears in this file is already



(a) Output Path Configuration (b) Adding AHL to MPS

Figure 5.4: MPS Module Configuration

defined in the editor for the Configuration concept and the user cannot add anything extra to the file.

- **Step 8 - Writing AHL Code:** We fill out the configuration file based on the Redliner project specifications. The database includes one table with all the metrics from the device as columns (Figure 5.7). The BLE Service part is also completed with information from the Redliner BLE protocol (Figure 5.8). This protocol defines a Data service with three characteristics representing packet information required for reading packets, and two characteristic packs that package the packet data.

- **Step 9 - Generating MPS Code:** Now we will make the project. This can be done through the *Build* menu in IntelliJ IDEA, choosing *Make Module* from the right click menu on the root module of the project, or using Ctrl + F9 shortcut. The make command will trigger the MPS generator that defines the model to model transformation and all the TextGen aspects that perform source code generation. In the end, the code will be generated in the *src_gen* folder of the AHL module.

- **Step 10 - Running the AHL Wrapper:** At last, we run AHL Wrapper to finish the process. At first, AHL Wrapper creates the package structure in the Android module. Then it runs the Android Content Provider Generator library, using the JSON files generated by MPS, to create the con-

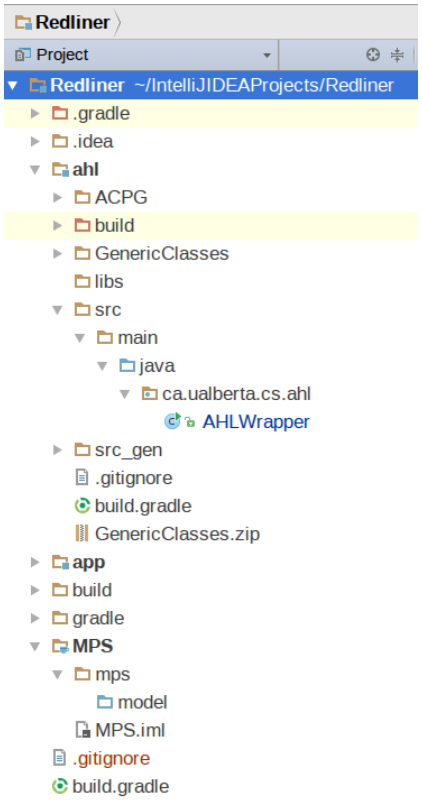


Figure 5.5: Project Structure After the Setup

tent provider. At last, all the generated classes and the pre-written generic classes will be moved to their right packages in the project. After this point, AHL and MPS modules are no longer needed and can be removed from the project.

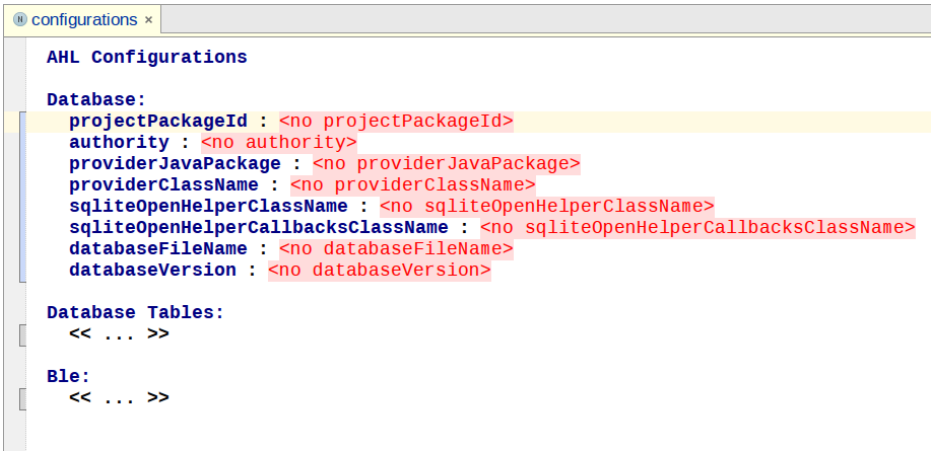


Figure 5.6: AHL Configuration File

```
AHL Configurations

Database:
  projectPackageId : ca.ualberta.cs.redliner
  authority : ca.ualberta.cs.redliner.packets.provider
  providerJavaPackage : ca.ualberta.cs.redliner.provider
  providerClassName : PacketsProvider
  sqliteOpenHelperClassName : MySQLiteOpenHelper
  sqliteOpenHelperCallbacksClassName : MySQLiteOpenHelperCallbacks
  databaseFileName : packets.db
  databaseVersion : 1

Database Tables:
  Table: packets {
    packet_id : Integer : true : Packet ID
    start_time : Integer : true : Start Time
    day : Integer : true : Day of Month
    month : Integer : true : Month of Year
    distance : Integer : true : Distance
    velocity : Float : true : Velocity
    pushes : Integer : true : Pushes
    seconds_active : Integer : true : Seconds Active
    redline_events : Integer : true : Redline Events
    gps_lat : Float : true : GPS Latitiude
    gps_lng : Float : true : GPS Longitude
    slope : Float : true : Slope
    lateral_slope : Float : true : Lateral Slope
    pos_alt_delta : Float : true : Positive Altitude Delta
  }
```

Figure 5.7: Redliner Database Configuration

```
    lateral_slope : Float : true : Lateral Slope
    pos_alt_delta : Float : true : Positive Altitude Delta
  }

Ble:
  bleService: DeviceData : 401dc6a0-3f8d-11e5-afbb-0002a5d5c51b {
    << ... >>

    char: oldestPacketId : 0000c6a1-0000-1000-8000-00805f9b34fb : int
    char: mostRecentPacketId : 0000c6a2-0000-1000-8000-00805f9b34fb : int
    char: desiredPackatToRead : 0000c6a3-0000-1000-8000-00805f9b34fb : int

    charPack: PacketHi : 0000c6a4-0000-1000-8000-00805f9b34fb {
      id : 0 : 4 : int
      startTime : 4 : 4 : int
      distance : 8 : 2 : int
      velocity : 10 : 4 : float
      pushes : 14 : 2 : int
      secondsActive : 16 : 2 : int
      redlineEvents : 18 : 2 : int
    }
    charPack: PacketLo : 0000c6a5-0000-1000-8000-00805f9b34fb {
      gpsLatitiude : 0 : 4 : float
      gpsLongitude : 4 : 4 : float
      slope : 8 : 4 : float
      lateralSlope : 12 : 4 : float
      positiveAltitudeDelta : 16 : 4 : float
    }
  }
```

Figure 5.8: Redliner BLE Configuration

Chapter 6

Evaluating the AHL Framework

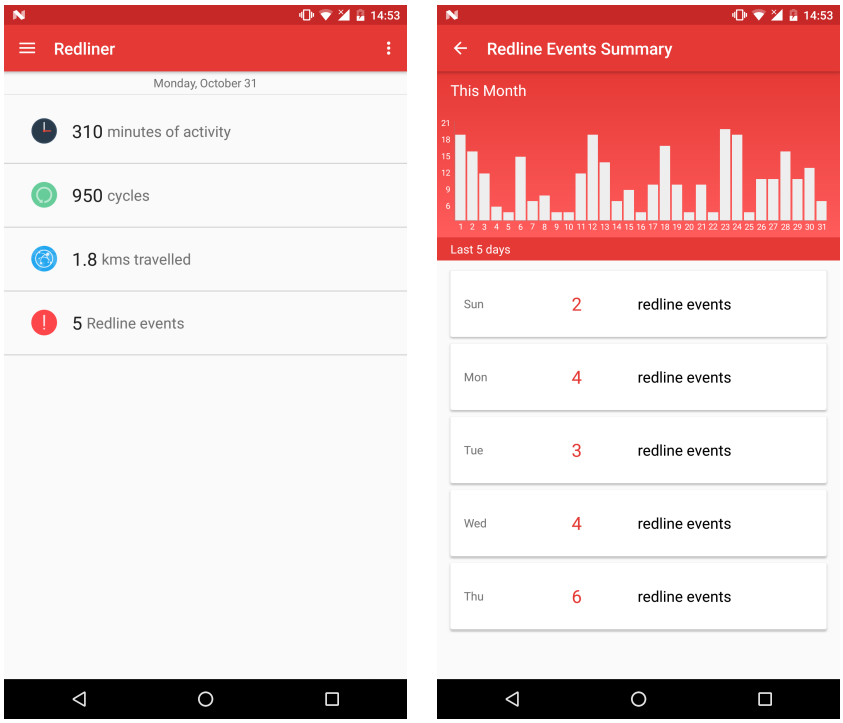
In chapter 5, we briefly described the Redliner application and discussed the process of developing an application and generating the code for its essential components as proof of concept. In this chapter, we compare the application that was generated through that process with the original Redliner application developed from scratch in terms of functionality and more quantifiable metrics such as line of code and number of classes. To investigate further, we also created an application for a BLE nearable technology called Estimote Stickers and compared to the application developed by another developer it in the same way.

6.1 Redliner vs. AHL Redliner

Other than the common functionality components that were described in the previous chapter, Redliner has a UI component that is consist of the following activities [1]:

- **Dashboard**: This activity lists different metrics and provides a link to visualizations for each one (Figure 6.1a).
- **Visualizations**: Each metric is visualized based on time in a separate activity (Figure 6.1b).
- **Device Manager**: This activity lists nearby Bluetooth devices and allows

[1] Activties are interactive screens in Android application.



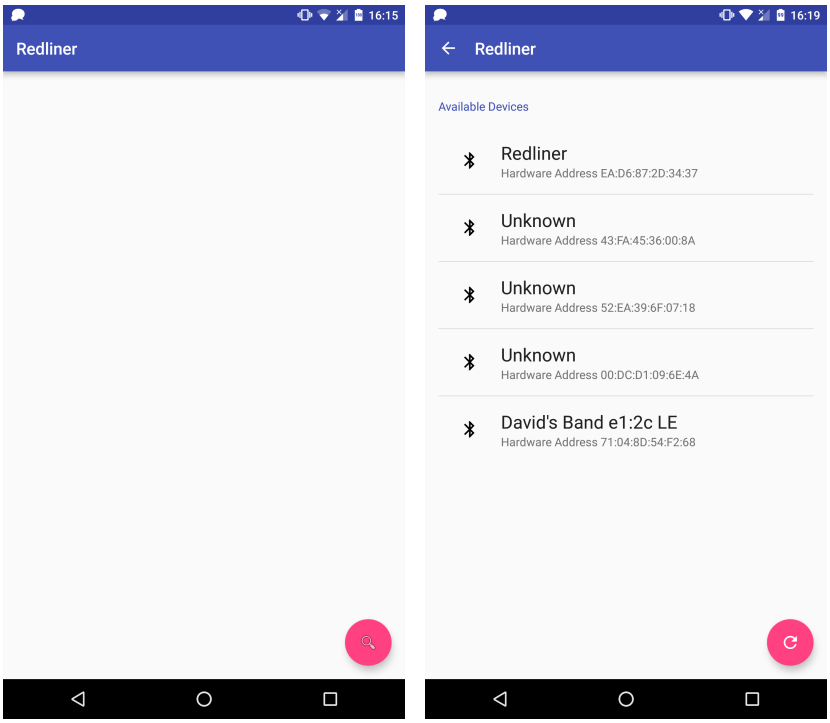
(a) Dashboard (b) Visualizations

Figure 6.1: Redliner Activities

the user to pick the Redliner device and pair it with the phone. Removing a paired device is also possible in this activity.

Out of these three kinds of activities, only the last one is available in the application generated by our framework. Figure 6.2 shows the AHL produced activities including the device manager that has the complete functionality of the same activity in the Redliner application (6.2b). The other generated activity is an empty main activity with a floating action button that launches the device manager (Figure 6.2a). Clearly, developers can remove this button and launch the device manager from anywhere that is appropriate based on their design.

On the other hand, most of the code that provides functionality in the background is produced by AHL. To verify that the generated application is working properly, we added the BLE Reader code from the Redliner Android application to AHL Redliner, generated AHL Redliner APK file and installed it on a Nexus 6 phone running Android 7, Nougat. Then we turned on a



(a) Main Activity (b) Device Manager Activity

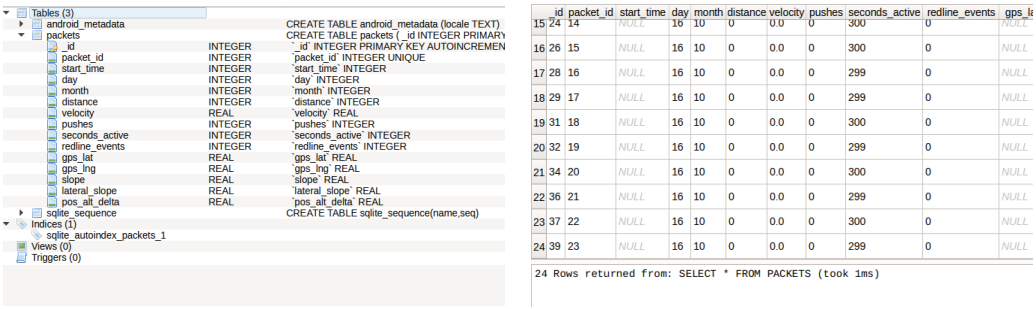
Figure 6.2: AHL Redliner Activities

Redliner prototype device that generates random dummy data. A blue LED light on the device was indicating that the Android service in the application is successfully connecting to the device and reading the packets every five minutes. In order to investigate this further and at the same time ensure that the database system is working as well, we pulled the application's database, named *packets.db*, from the Android phone (this name was defined in the AHL configuration file in the process of generating the code). We used DB Browser for SQLite tool [2] to browse the database. Figure 6.3a shows that the structure defined in AHL configuration file is successfully created. Figure 6.3b proves that the dummy data from the device is inserted into the database.

6.1.1 Functionality Comparison

If we exclude the UI component, that is partly generated, most of the other functionalities of the Redliner application are available in AHL Redliner. List-

[2] http://sqlitebrowser.org/



(a) Database Structure (b) Database Rows

Figure 6.3: AHL Redliner Database

ing, pairing and unpairing devices (the background BLE code), data persistence using Android content providers and all the service components are produced by AHL. The only missing part is reading from a BLE device which, as discussed before, is heavily device dependent. We provided a comparison of the two applications in terms of functionality in Table 6.1. Dashboard and graphical representations are a part of the UI component. Listing devices, pairing and removing a device functionality have both UI code and background code and all of it is produced by the framework.

Functionality	**Redliner**	**AHL Redliner**
Dashboard	✓	x
Listing Nearby Devices	✓	✓
Pairing a Device	✓	✓
Remove a Paired Device	✓	✓
Reading from a Device	✓	x
Data Persistence	✓	✓
Boot Receiver	✓	✓
Alarm Receiver	✓	✓
Android Service	✓	✓
Graphical Reperesentations / Charts	✓	x

Table 6.1: Redliner vs. AHL Redliner in Terms of Functionality

6.1.2 LOC Comparison

In order to evaluate our work in a more quantifiable manner and show the amount of the generated code, we performed a comparison in terms of lines of code (LOC). The results are provided in Table 6.2. First, this comparison shows all of the classes and nearly all of the code that provide some functionality in the background are generated by AHL. The only background code that must be implemented by the user is a part of the BLE communication class that is responsible for reading from a device. That is the only difference between these two application when it comes to the background code.

Moreover, even though generating UI code is not the focus of our framework, 25% of XML and 24% of Java code for the UI is generated through AHL. Java classes related to UI include activities, fragments and Android list adapters.

It is worth mentioning that we excluded all comments and empty lines when counting lines of code to get a better understanding of the actual percentage of the code generated by AHL.

	# of Classes	# of UI Related Classes	LOC (Java - Functionality) *	LOC (Java - UI) *	LOC (Java - Total) *	LOC (XML)
Original Redliner	36	14	2235	910	3145	1263
AHL Redliner	25	3	1970	216	2186	315
% of Generated Code	69%	21%	88%	24%	70%	25%

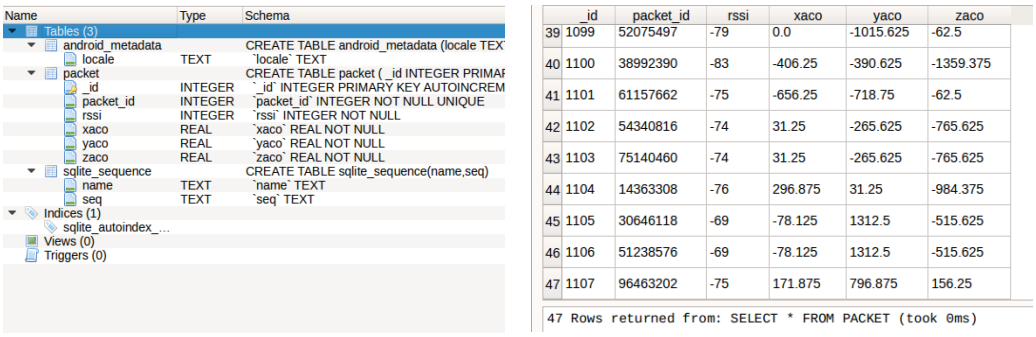
* Comments and empty lines are not counted.

Table 6.2: Redliner vs. AHL Redliner in Terms of Line of Code (LOC)

6.2 Estimote Reader vs. AHL Estimote Reader

Estimote Stickers [3] are customizable BLE sensors that can be attached to objects and provide context awareness by broadcasting data packets. These data packets include information like x, y and z accelerometer values, temperature and a boolean value that indicates whether the sticker is in motion. These stickers are used to track movements in a research project called Smart Condo

[3] http://estimote.com/



(a) Database Structure (b) Database Rows

Figure 6.4: AHL Estimote Reader Database

[4]. As a part of that project, an Android application connects to the Estimote stickers, collects their data and stores them in a file. This application is in early stages of development, and some of its requirements are not implemented yet. As a method to evaluate our work, we followed the steps described in chapter 5 and created an Estimote Reader application with AHL.

For the database, we defined one table with a column for the following values existing in the packets: RSSI (Received Signal Strength Indication) value and x, y, z values from the sticker's accelerometer. Figure 6.4a shows that the database is created and Figure 6.4b shows that the data from the stickers are successfully inserted into the database.

6.2.1 Functionality Comparison

We sat down with the researcher who is working on the Estimote Reader application and managed to develop a fully functional application with AHL in just an hour. In order to adapt the generated application to the project requirements, we had to make some changes. Estimote provides an SDK for handling communications with their devices. This SDK provides function calls for searching for devices and reading from them. Therefore, we removed the BLE Scanner component of our framework. Also, since there is no need to pair stickers with the phone, we also removed the Device Scanner component which was providing an interface for pairing and managing devices.

[4]http://www.hserc.ualberta.ca/Resources/Spaces/SmartCondo.aspx

Other than the BLE Reader component, that is provided by Estimote SDK in this project, AHL covers all the other components of the Estimote Reader application. We compared the original Estimote reader to the AHL produced application in terms of functionality in Table 6.3.

Functionality	**Estimote Reader**	**AHL Estimote Reader**
Reading from a Device	✓	x
Data Persistence	✓	✓
Boot Receiver	✓	✓
Alarm Receiver	✓	✓
Android Service	✓	✓

Table 6.3: Estimote Reader vs. AHL Estimote Reader in Terms of Functionality

6.2.2 LOC Comparison

Since Estimote Reader is in an early stage of development, our framework generated more code than the original application. More specifically, there is no service implemented in Estimote Reader and the data persistence mechanism is not standard. Reading from stickers and storing the data is triggered when the application is launched and it stops when the it is closed by the user. This means that there is no background data collection. Also, this application stores the data in a file and this file is supposed to be transferred to a server over the network for further processing. This is not a standard way of handling data storage in Android; rather a temporarily solution. A proper database makes this data exchange much easier, especially when it is done regularly which makes transferring the whole file impractical. The content provider generated by AHL solves this problem and makes data storage standard and scalable in this application.

In terms of user interface, the only UI part in AHL Estimote Reader is the MainActivity class which is the default activity in Android. That explains the generated UI code even though AHL Estimote Reader does not cover any

UI requirements. The original application lists the surrounding stickers in the main activity which is not necessary to its functionality and is implemented for debugging purposes. That is the reason for the higher amount of XML and Java UI code in the original Estimote Reader.

	# of Classes	# of UI Related Classes	LOC (Java - Functionality) *	LOC (Java - UI) *	LOC (Java - Total) *	LOC (XML)
Estimote Reader	9	4	285	368	653	233
AHL Estimote Reader	21	1	1191	24	1215	112
% of Generated Code	233%	25%	418%	7%	186%	48%

* Comments and empty lines are not counted.

Table 6.4: Estimote Reader vs. AHL Estimote Reader in Terms of Line of Code (LOC)

Chapter 7

Conclusion and Future Work

In this thesis, we introduced the AHL framework for a systematic development of applications working with peripheral BLE devices. We defined a general architecture for applications communicating with such devices and described the common components among them. Then, using JetBrains MPS, we created a domain specific language that allows developers to define their application requirements in configuration files and generate the code for the components defined in the general architecture. These components include an Android service and related alarm manager and broadcast receivers, a content provider incorporating a content resolver and an SQLite database, and UI elements for detecting and managing Bluetooth devices. Parts of the BLE communications code, which are not device specific, are also generated by our framework.

We provided an IntelliJ IDEA plugin that makes the code generation possible from within an Android application project in IDEA. This plugin includes our domain specific language and all of its tools and enables users to add a model module to their application and define their requirements. After that, the AHL framework will use these defined requirements and add the generated code to the project.

Using this framework does not require prior knowledge of the generated components and their concepts. These components are complicated and time-consuming to implement. The code generated by AHL is fully functional and does not need any modifications. This means developers do not have to learn how to implement or modify these components because using them does not

require knowledge of how they work.

The contributions of this work are as follows:

- **We defined a general architecture for Android application working with physical BLE devices.** This architecture laid out all the common components among applications working with BLE devices. Using this, we build other parts of our framework to automate the process of implementing as many of those components as possible.

- **We developed a modeling language that describes components of an application working with BLE devices.** Using this language developers can define their components' configurations such as database tables and their BLE specifications and the AHL framework takes it from there and produces the code.

- **We developed a framework that allows Android developers generate code for their application in an easy and efficient way.** AHL turns the tedious task of implementing content providers, services, etc. that usually takes weeks into a fast process that can be done in a few hours. The Redliner Android application is a part of the Redliner platform which will hit the market in the near future. We generated most parts of this application in less than 30 minutes and illustrated the process in 10 steps.

- **We evaluated our framework by developing two applications and comparing them to the applications developed from scratch for the same purpose.** We added the code already written for reading from the Redliner BLE device to the AHL generated Redliner and compared this functional application to the one built from scratch. On top of that, we developed an application for working with Estimote Stickers in an hour and compared it to the original application developed by another developer.

In the future, we are planning to update our DSL with new syntax, so it is possible to define the order in which BLE characteristics will be read by an

application. Using this methodology, our framework will be able to generate some parts of the BLE Reader component. However, the parts related to specific BLE protocol on each physical device will still be left for developers to implement. Also, it is possible to make AHL framework more general in the future. By augmenting it with more auto-generated components, it can be used for other types of applications. Naturally, developers will define the components they need and the code only for those will be generated by the framework.

Bibliography

[1] Idc: Smartphone os market share 2015, 2014, 2013, and 2012. http://www.idc.com/prodserv/smartphone-os-market-share.jsp, [Online; accessed 22-July-2016].

[2] Cross-platform tool benchmarking 2014. http://research2guidance.com/cross-platform-tool-benchmarking-2014/, [Online; accessed 09-December-2016].

[3] Clint Wheelock Aditya Kaul. Wearables: 10 trends to watch. Technical report, Tractica, Boulder, Colorado USA, 2015.

[4] Wearables market to be worth $25 billion by 2019. http://www.ccsinsight.com/press/company-news/2332-wearables-market-to-be-worth-25-billion-by-2019-reveals-ccs-insight, [Online; accessed 10-November-2016].

[5] Marjan Mernik, Jan Heering, and Anthony M Sloane. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*, 37(4):316–344, 2005.

[6] Xtext - language engineering made easy! https://eclipse.org/Xtext/, [Online; accessed 10-November-2016].

[7] Emftext. http://www.emftext.org/index.php/EMFText, [Online; accessed 10-November-2016].

[8] Jetbrains meta programming system. https://www.jetbrains.com/mps/, [Online; accessed 10-November-2016].

[9] Eclipse modeling framework. http://www.eclipse.org/modeling/emf/, [Online; accessed 10-November-2016].

[10] Graphical modeling project. http://www.eclipse.org/modeling/gmp/, [Online; accessed 10-November-2016].

[11] Atom3 a tool for multi-formalism meta-modelling. http://atom3.cs.mcgill.ca/, [Online; accessed 10-November-2016].

[12] Metaedit+ domain-specific modeling tools. http://www.metacase.com/products.html, [Online; accessed 10-November-2016].

[13] Overview of domain-specific language tools. https://msdn.microsoft.com/en-us/library/bb126327.aspx, [Online; accessed 10-November-2016].

[14] Vicente Pelechano, Manoli Albert, Javier Muñoz, and Carlos Cetina. Building tools for model driven development. comparing microsoft dsl tools and eclipse modeling plug-ins. In *DSDM*, 2006.

[15] Philip De Smedt. Comparing three graphical dsl editors: Atom3, metaedit+ and poseidon for dsls. *Preprint, Submitted to Elsevier, University of Antwerp*, 2011.

[16] Xtend - modernized java. https://eclipse.org/xtend/index.html, [Online; accessed 10-November-2016].

[17] Acceleo. https://eclipse.org/acceleo/, [Online; accessed 10-November-2016].

[18] Generator overview mps 3.3 documentation confluence. https://confluence.jetbrains.com/display/MPSD33/Generator+Overview, [Online; accessed 10-November-2016].

[19] Abilio G Parada and Lisane B de Brisolara. A model driven approach for android applications development. In *Computing System Engineering (SBESC), 2012 Brazilian Symposium on*, pages 192–197. IEEE, 2012.

[20] Abilio G Parada, Eliane Siegert, and Lisane B de Brisolara. Generating java code from uml class and sequence diagrams. In *2011 Brazilian Symposium on Computing System Engineering*, 2011.

[21] Richard Membarth, Oliver Reiche, Frank Hannig, and Jürgen Teich. Code generation for embedded heterogeneous architectures on android. In *Proceedings of the conference on Design, Automation & Test in Europe*, page 86. European Design and Automation Association, 2014.

[22] Woo Yeol Kim, Hyun Seung Son, Jae Seung Kim, and Robert Young Chul Kim. Adapting model transformation approach for android smartphone application. In *Advanced Communication and Networking*, pages 421–429. Springer, 2011.

[23] Woo Yeol Kim, Hyun Seung Son, and Robert Young Chul Kim. Design of code template for automatic code generation of heterogeneous smartphone applications. In *Advanced Communication and Networking*, pages 292–297. Springer, 2011.

[24] Ayoub Sabraoui, Mohammed El Koutbi, and Ismail Khriss. Gui code generation for android applications using a mda approach. In *Complex Systems (ICCS), 2012 International Conference on*, pages 1–6. IEEE, 2012.

[25] MOHAMED LACHGAR and ABDELMOUNAÏM ABDALI. Modeling and generating the user interface of mobile devices and web development with dsl. *Journal of Theoretical & Applied Information Technology*, 72(1), 2015.

[26] Meera Radhakrishnan, Archan Misra, Rajesh Krishna Balan, and Youngki Lee. Smartphones and ble services: Empirical insights. In *Mobile Ad Hoc and Sensor Systems (MASS), 2015 IEEE 12th International Conference on*, pages 226–234. IEEE, 2015.

[27] Android api guides - bluetooth low energy. https://developer.android.com/guide/topics/connectivity/bluetooth-le.html, [Online; accessed 20-July-2016].

[28] Processes and application life cycle. https://developer.android.com/guide/topics/processes/process-lifecycle.html, [Online; accessed 27-July-2016].