CARL VON OSSIETZKY universität OLDENBURG

Faculty II – Computer Science, Law, and Economics
Software Engineering Group

Energy Refactorings

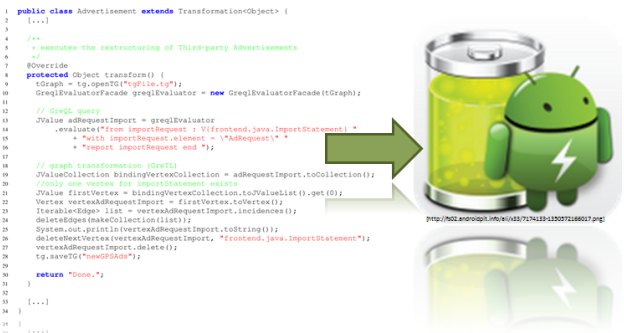
Master's thesis

submitted by

Marion Gottschalk

matriculation number:

9854680



Supervisor:

Prof. Dr. Andreas Winter
M. Sc. Jan Jelschen

Oldenburg, October 13, 2013

Abstract

Nowadays, energy-efficiency is an important topic for the information and communication technology on various levels, such as in mobile devices, computers, servers, and data centers. Currently, 1 % of the German energy consumption results from mobile devices, such as smartphones and tablets. Due to the raising sales volume of these devices, it can be anticipated that the energy consumption, and hence, their percentage share of the complete energy consumption in Germany also rises. A reason for the increasing sales volume is the improvement of the mobile device's functionality on hardware and software level, such as faster processor and more complex applications. Both levels have a great influence on the energy consumption. Meanwhile, researches on both levels exist to optimize their energy use.

In this master's thesis, the software level, or to be more exact, the applications level on mobile devices is considered. The applications level on the operating system Android is open for each vendor and user. This means, that vendors and users can decide which applications are installed. However, these applications are programmed by many different vendors and programmers, and only a few guidelines by different organizations, such as Android [Anda] and Samsung [Samb], exist which can be used by programmers. Hence, some applications are energy-inefficient and reduce the battery duration of mobile devices. The source code of these applications can be optimized, to get more energy-efficient applications. Therefor, an Energy Refactoring Catalog is created to define energy-inefficient code parts and their transformation to efficient code. This enables a semi-automatic process to reduce the energy consumption for individual Android applications and is presented in this work.

Content

1 Introduction

The first part of this master's thesis introduces the topic *energy-efficient programming and applications* to explain why the topic *Energy Refactorings* is sensible. Firstly, a motivation is given to demonstrate the field of application and its room of improvement. Secondly, an approach is described which represents a possibility to modify applications concerning their energy consumption. Thirdly, an overview about related work in the area of energy-aware programming is given. Fourthly, the work packages for this master's thesis are illustrated, and in addition, it includes the objective of this thesis. Finally, the structure of the master's thesis is described.

1.1 Motivation

The increasing number of mobile devices, such as smartphones and tablets, in households and industry is partly responsible for the increasing energy consumption. Meanwhile, the energy consumption for information and communication technologies (ICT) exceeds 10 % of the total energy consumption in Germany [Sto08]. This 10 % can be categorized as households, industry, servers and data centers, and mobile devices. The category *mobile devices* represents 11,6 % of the 10 %, so that it is the fourth biggest energy consumer of ICT [Sto08]. Thereby, the energy consumption of mobile devices accounts approximately 1 % of the German energy consumption. This amounts approximately 22.9 PJ (1 PJ is equal to $1 * 10^{15}$ J) which are only consumed by mobile devices within one year. This corresponds to the complete energy production of the atomic power plant Emsland [RWE] within seven months. Furthermore, the prognosis for sale of mobile devices is positive and increases every year [Sta]. On account of high energy consumption and demand of mobile devices, it is also important to improve the energy-efficiency in this area to reduce energy costs for users and to save the environment [Hom09].

Next to the increasing energy consumption and sales volume for mobile devices, users requirements are important for application vendors and devices' manufactures. Users carry their smartphone with them the whole day and want to use it whenever they want. Hence, mobiles devices' batteries need to last the whole day. But this is not often the case, as the Blackberry Z10 shows [Kre13]. First tests have shown that batteries of the Z10 are empty after about five hours. This are only 21 % of a day, and thus, it is not long enough for users requirements. Blackberry's solution is an external battery which charges smartphone batteries to extend their operating time. Due to the external battery, users have to carry two things with them when they need their smartphone for more than five hours. This aspect reduces user satisfaction enormously and lowers the sales volume, and thereby, the profit. Therefore, manufacturers should be interested in energy-efficient software to extend the battery duration of their mobile devices.

The energy-efficiency of mobile devices can be improved by hardware or software optimization. Hardware optimization means to make changes on hardware components of mobile

devices, like processors, GPS, and WLAN modules, to save energy. A widespread hardware optimization are processor optimizations using new techniques, e.g. Tri-Gate-Transistors [Kam11] and specialized processors for mobile devices, such as Snapdragon [Sna11]. Also, other hardware components are optimized, like the GPS modul, to reduce the startup time of it [OnV12]. Another possibility is software optimization which describes changes on source code of different areas of software to reduce the energy consumption of mobile devices. Therefore, software optimization can be divided into operation system level and application level [Sin01]. On the operating system level, it is possible to control hardware components intelligently. This refers to control e.g. the processor by reducing the frequency or voltage without any loss in performance, but a lower energy consumption [Sin01]. Optimization on application level is less common as on the other two levels (hardware and operating system). Although, some free applications, which are used on mobile devices, consume more energy than necessary, for example through third-party advertisement or tracking user data [PCHZ12]. Thus, it reduces also the battery duration, similar to energy-inefficient hardware. Hence, applications should be also designed more energy-efficient to extend the battery duration of mobile devices. More energy-efficient applications are realized through transforming applications code to remove energy-inefficient code parts. Thereby, several energy-inefficient code parts could exist within one application. As a result, the objective of this master's thesis is determined:

Creating an Energy Refactoring Catalog which defines energy-inefficient source code parts and removes them by a semi-automatic transformation.

1.2 Approach

To reach the objective of this master's thesis, a process to generate energy-efficient source code must be created. This process is needed to execute the different Energy Refactorings (cf. Section 4). It is depicted in Figure 1.1 and illustrates several reengineering techniques which are explained in Section 2.1. Moreover, the process bases on the reengineering reference framework by Ebert et. al [EKRW02]. In order to gain a better understanding, the steps of the process, *parse*, *analyzing*, *restructuring*, and *unparse* are described.

Parse: The source code of an application is parsed to get an abstract representation of it, which is stored into a central repository to execute efficient analyzes.

Analyze: Secondly, the abstracted code can be analyzed, e.g. by static code analysis (cf. Section 2.1.2) which has proved itself in software evolution, to identify Energy Code Smells (cf. Section 3).

Restructure: Thirdly, energy-inefficient parts are changed or deleted by restructurings on the abstracted view. This means a conversion of the existing code to more energy-efficient code on a higher abstraction without changing the intended functionality.

Unparse: After restructuring, the abstracted view is unparsed to source code, hence, it can be compiled and executed.

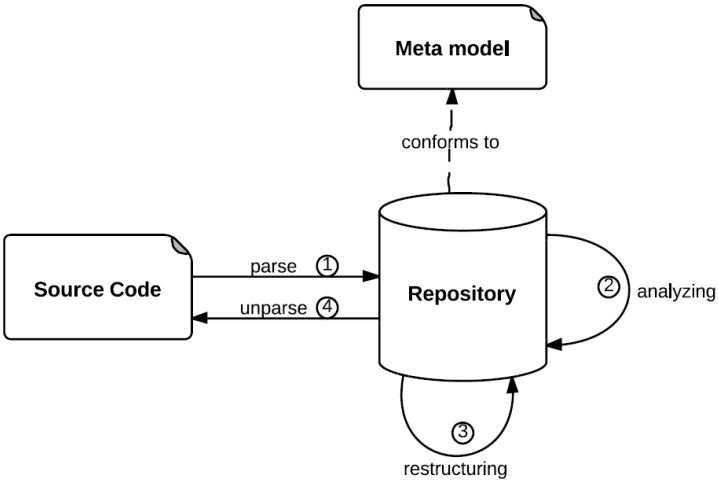


Figure 1.1: Process to generate energy-efficient source code [GJJW12]

In this master's thesis, the steps *analyze* and *restructure* are realized to generate energy-efficient source code in a semi-automatic process. The first step (*parse* is realized by the pro et con parser, used by the SOAMIG project [FWE+12], which can be used for this master's thesis. The source code must be parsed to get an abstract representation of the code and to enable an efficient code analysis. For this purpose, a graph-structured repository which conforms to a meta model is used (cf. e.g. Ebert et al. [ERW08]). The meta model is needed because it defines a clear, precise, and target structure and documentation of the underlying data structures which enable an efficient code analysis, e.g. by querying applications [JCD02]. For the second and third step techniques of the TGraph approach (cf. Section 2.2) are used: GReQL (Graph Repository Query Language) [HE11] and the JGraLab API, which includes the functionality of GReTL (Graph Repository Transformation Language) [HE11]. These are necessary to detect and remove energy-inefficient code. A short sequence of the execution of this process is given to improve the understanding of it: At first, source code of a freely available application is parsed and stored as a TGraph (first step) which conforms to a Java meta model [FWE+12], e.g. Android applications are implemented with Java. Hereafter, queries via GReQL are made to identify Energy Code Smells (second step) on TGraphs. The JGraLab API is used to transform the TGraph to get more energy-efficient code (third step). Finally, the transformed TGraph is parsed back into source code (fourth step) with a simple Java generator. The detailed process of the second and third step are described in Section 3.

1.3 Related Work

This section gives an overview about further researches in the area of energy-aware programming for mobile devices. Hence, four author groups and their approach are briefly described.

Firstly, the approach of Pathak et. al [PCHZ11] is presented. This approach deals with the energy use of mobile devices, and it defines *Energy Bugs* which bases on hardware, software or external mistakes which raise the energy consumption. For this master's thesis, the part with the *Software Energy Bugs* is considered. This part is also divided in operation system and application level, like this work, and hence, the application level is only considered in the further step. They describe three types of *Applications Energy Bugs*: *No-Sleep Bug*, *Loop Bug*, and *Immortality Bug* [PCHZ11, p. 3]. *No-Sleep Bug* describes the usage of wake locks within applications which are responsible for preventing the shutdown of hardware resources. *Loop Bug* describes applications which repeat the same activity over and over again without reaching an intended result. These both *Applications Energy Bugs* are also described in Section 5, but a reference to this work, i.e. they are defined and include an approach to detect and remove them by a semi-automatic process. *Immortality Bug* describes one application which is stopped but restarted by another application. In [PCHZ12] more detailed information and measurement results are described concerning on *Applications Energy Bugs*. The approach by Pathak et. al is similar to the approach in this work, energy-inefficient code parts are defined, however, in this master's thesis also an approach is created to remove the energy-inefficient code.

Secondly, Höpfner and Bunse [HB11] research the energy consumption of different sorting algorithms and substitute them. Therefor, applications with the same functionality but with different sorting algorithms are measured and compared. It shows that the *Insertionsort* consumes less energy than *Quicksort* and *Mergesort*. However, this is only shown for a special hardware and must be validated for other. In their work, the sorting algorithms are changed manual and no process is described how the modification can be made automatic or rather semi-automatic. This represents the difference between their work and this master's thesis. However, the modification of sorting algorithms could be realized with the approach of this thesis how it is described in Section 5.

Thirdly, Bunse and Stiemer [BS13] attend to a special type of code parts which are maybe energy-inefficient. They validate some *Design Pattern* (*facade*, *abstract factory*, *observer*, *decorator*, *prototype*, and *template method*) by Gamma et. al [GHJV93]. Therefor, they implemented two comparable applications, one with a *Design Pattern* and one without. Both applications are validated with the application PowerTutor [GZT], and the measurement results are compared. It shows that some of the *Design Patterns* (*decorator*, *prototype*, and *abstract factory*) are energy-inefficient. This type of energy-inefficient code can be also defined as Energy Code Smell (cf. Section 3) and the analysis and restructure can be applied on these code parts. Hence, the work by Bunse and Stiemer [BS13] is similar to this master's thesis, it shows code parts which raise the energy consumption of applications, but they do not describe an approach to remove energy-inefficient code.

Finally, the work by Wilken et. al [WRP+12] is described. They develop an approach to profile the energy consumption of mobile devices and comparing them for similar applications, i.e. the energy consumption of applications which have the same functionality are compared. This information should help users to decide which application they install on their mobile device. The visualization of the energy consumption should be represented through energy

labels similar to the labels for fridges. This work does not seek for energy-inefficient code parts but for energy-inefficient applications. In addition, energy-inefficient applications are not modified to save energy, they are only compared whit each other to help users in their decision which applications they should install to save energy.

1.4 Work Packages for the Thesis

The main objective of this master's thesis is to create an overview of energy-inefficient source code parts and to verify them by an energy measurement. Moreover, the code parts are optimized to generate energy-efficient source code for existing applications by a semi-automatic process. To reach these objectives, this work packages need to be done:

- Literature study about required techniques and other possibilities to reduce energy consumption on applications level.
- Defining at least five platform-independent Energy Refactorings (cf. Section 4).
- Implementing at least three restructurings for the Android platform.
- Apply the implementation on different freely available application, like GPSPrint [Rob12], Standup Timer [Woo11], MyTracks [MyT], etc.
- Evaluating the Energy Refactorings using the energy measurement tool by Schröder [Sch13] to check their energy consumption.
- Intended result: The complete process for Energy Refactorings (name, definition, motivation, constraints, example, analysis, restructuring, and evaluation) (cf. Section 3.2) must be demonstrated. The number of implemented restructurings can be changed when the implementation needs more time than expected.

The work packages are still being taken up in the following sections to reach the main objective at the end.

1.5 Structure of the Thesis

This master's thesis is structured into seven parts and an appendix. Section 1 represents the first part which contains an introduction into the topic of the master's thesis *Energy Refactorings*, its objectives, and an approach to reach energy-efficient code. The second part in Section 2 gives an overview about the used techniques, hardware, and applications. These techniques are needed for the approach described above, and the hardware and applications are needed for the validation of the approach. A definition of Energy Code Smells and a template for the Energy Refactorings are given in Section 3. Afterward, the Energy Refactoring Catalog is presented in Section 4 which includes five Energy Code Smells which are detected, restructured, and validated. Further Energy Refactorings are described in Section 5 but not implemented and validated. In Section 6 the implementation of the validated Energy Refactorings are presented and explained. Finally, the master's thesis ends with a conclusion, which summarizes the validation results of the Energy Refactorings, and gives an outlook in Section 7.

2 Basic Techniques

This section gives an overview about used techniques, mobile platforms, and applications which are important for this master's thesis. Firstly, reengineering techniques are introduced to get the basic techniques to analyze and restructure source code. Secondly, the TGraph approach is described, which represents the technique and tooling for Energy Refactorings (cf. Section 4) to detect badly designed code parts and to restructure them. Thirdly, the OS Android as mobile platform is described because applications on this OS are used to validate Energy Refactorings. Fourthly, the mobile devices HTC One X and Samsung Galaxy S4 are presented, which are also needed for validation. The energy consumption of several applications and possible Energy Code Smells (cf. Section 3) are checked for these devices. Fifthly, several energy measurement techniques and the measurement process are described which are used to evaluate the Energy Refactorings. Lastly, Android applications which contains Energy Code Smells are introduced. These applications are needed for the evaluation, and hence, its functionality should be known.

2.1 Reengineering Techniques

Software reengineering is a process to improve the software quality by reconstituting it in a new form without changing its functionality [CC90, p. 15–16]. The Reengineering process by Kazman et. al [KWC98] is illustrated in Figure 2.1. It shows a process which can be subdivided into three steps. First, source code is represented in a higher abstraction, the architecture level. Second, on this level transformations are executed to modify the architecture. And finally, the new architecture is developed to new source code. In addition, several techniques are applied which are needed to improve the software quality, e.g. : reverse engineering, forward engineering, source code analysis, restructuring, and refactoring.

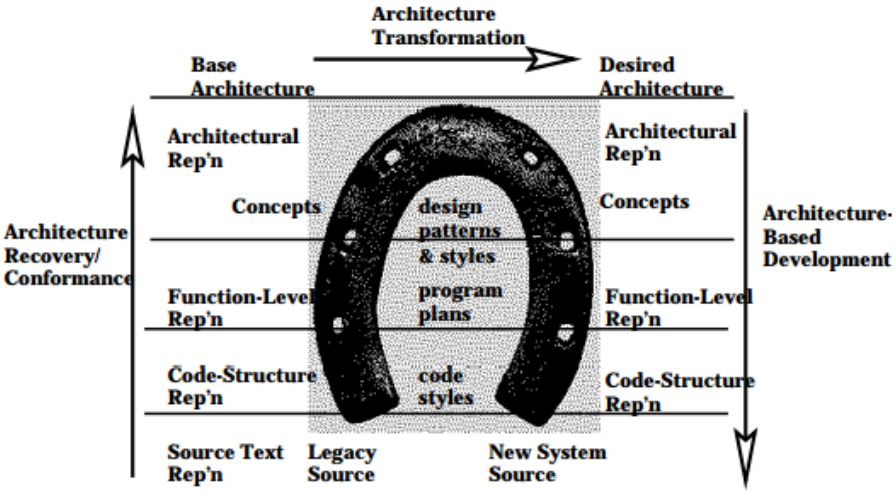


Figure 2.1: Horseshoe model [KWC98]

In this master's thesis, only the code level is considered in which the code is represented in a higher abstraction. However, this abstraction illustrates the code structure, so that the architecture level is not achieved. But, the same techniques are applied to get the code structure and to transform it. Hence, the Horseshoe model in Figure 2.1 represents the transformation on a higher level than it is needed here, but it helps to arrange the topic of this thesis.

2.1.1 Reverse- and Forward-Engineering

For improving the software quality, a reverse engineering is needed to get an abstracted view of the source code. Thereby, all components and their interrelationship are identified and another representation form is created. In this master's thesis, source code is represented in TGraphs which is described in Section 2.2. The reverse engineering does not make changes on existing source code, it is only a process of examination, not change or replication [CC90, p. 15]. The abstracted view is needed for efficient source code analyses to detect Energy Code Smells. The opposite of reverse engineering is forward engineering. Forward engineering describes the traditional process of creating source code from an abstract view [CC90, p. 14]. The aim of this thesis is to improve existing applications, and hence, forward engineering described the process in which source code is generated through the modified abstract view. In Figure 1.1 the process of creating energy-efficient source code is illustrated, and the steps 1 and 4 depict the reverse- and forward-engineering process.

2.1.2 Source Code Analysis

After reverse engineering, source code analyses can be performed. During source code analyses, information about source code and its behavior are collected and considered [Bin07, p. 1]. The source code the application are only analyzed and not changed. In this section, two variants of source code analysis are explained which can be used to detect Energy Code Smells (cf. Section 3). These variants are: *static analysis* and *dynamic analysis*.

Static Analysis

Static source code analyses examine the complete source code and its behavior which could arise at run time [Ern03, p. 25]. The run time environment and the input values during run time are not relevant for this analysis, because only correlations between several source code parts are considered. For a complete static analysis, all possible states of the applications must be considered. Therefore, an abstracted model is used which represents the behavior of source code. This model is more compact and easier to use than no abstracted source code [Ern03, p. 25]. The static analysis is mostly used for proofs of correctness and type safety, e.g. compiler optimizations are standard static analyses [Ern03, p. 25–26]. For the static analysis in this thesis a graph query language is used, which works on a graph representation of source code (cf. Section 2.2). The graph representation is needed to detect Energy Code Smells which are bad for the energy consumption instead of software quality by Code Smells [FBB+02].

Dynamic Analysis

Dynamic source code analyses operate by running an application and observing its executions [Ern03, p. 25]. The dynamic analysis does not need a model or abstraction. The static analyses are preciser than dynamic analyses, because all source code parts are considered in static analysis in which the dynamic analysis considers only predefined use cases. Typical dynamic analyses are testing and profiling. Due to dynamic analysis, values, such as run time and memory cost, can be determined, hence, dynamic analyses do not need more time than applications execution. In contrast, complex static analyses need sometimes more time than applications execution [Ern03, p. 25]. A disadvantage of dynamic analyses is the assignment of results respectively errors to explicit source code, because it is possible that results are not the same for other applications executions, e.g. when input values change [Ern03, p. 26]. These aspect should give a small overview about dynamic analyses. In this master's thesis, dynamic source code analyses are not considered to detect Energy Code Smells. But it is possible to detect the energy-inefficient code through dynamic analyses, e.g. creating user profiles to switch-off component, such as screen and Wi-Fi [JGJ+12, p. 4]. This is a further area of research to get more energy-efficient applications.

2.1.3 Restructuring

After source code analyses, restructurings can be applied when code parts (e.g. Energy Code Smells in Section 3) were detected through the analyses. Restructurings are transformations of source code into *new* source code wherein the functionality is the same as before, but it is realized by different source code. This technique is often used as a form of preventive maintenance, e.g. unstructured code is transformed into structured code [CC90]. In this thesis, the transformed code should be more energy-efficient as before (cf. Energy Refactorings in Section 4), but it does not involve modifications concerning its functionality. An example is the usage of advertisements (cf. Section 4.1), if advertisements are detected through source code analysis, the transformation would modify or rather remove these code parts. But the other parts are not changed, so that the functionality is the same.

2.1.4 Refactoring

The combination of source code analyses, here detection of Energy Code Smells, and restructuring is called refactoring in this master's thesis. Refactoring is used to optimize source code after it has been written in which no functionality is changed but the software quality is improved. Therefore, badly designed source code can be restructured to well-designed code. Generally, bad code, such as code clones, is sought by static analysis. Afterward, detected code parts are restructured which means that the bad code is optimized. Fowler et. al [FBB+02] differentiate between Code Smells and refactorings. Refactoring only denotes code restructuring, and Code Smells describes badly programmed code [FBB+02, p. 53]. In this thesis, refactorings are named Energy Refactorings (cf. Section 4) which represent a static source code analysis and restructuring with the aim to save energy within an application.

2.2 Java TGraphs

The source code structure of an application can be depicted in several forms which are named by Ebert et. al [EKW97, p. 4]: abstract syntax trees [WBM95], relational data bases [CNR90], TGraphs [ERW08], etc. The former two techniques are not considered any further in Ebert et. al [EKW97] and in this master's thesis. TGraphs are used for the abstract representation of source code, although each programming languages can be represented through its nodes and edges. Due to the SOAMIG project [FWE+12], the tooling for the Java programming language is available and it is used for this master's thesis. Due to the abstract representation, it is possible to perform efficient analyses and transformations. Generating the abstract view of source code is the first step of the refactoring approach in Figure 1.1 on page 3 to analyze (step 2) and restructure (step 3) Energy Code Smells (cf. Section 3). In the next step, TGraphs are defined, and an example for them and their usage (analyzing and restructuring) is given.

2.2.1 TGraphs

In this master's thesis, TGraphs are used to represent Android applications on a higher abstraction level to analyze and restructure the source code. This enables the detection of Energy Code Smells (cf. Section 3) by efficient graph queries and the restructuring on this abstraction. In this section, a very small example of an Android application is given to illustrate and explain the TGraph approach.

Figure 2.2 illustrates a simple extract of an Android application. The extract is a part of the Android application GpsPrint (cf. Section 2.6) and represents the power-on of the GPS component of a mobile device during the application's runtime. In this application, the GPS is started twice in line 3 and 6, but to start it in line 6 would usually be enough to achieve the application's functionality. The Android Life Cycle (cf. Section 2.3.2) specifies that applications are visible when the method `onResume()` (line 5) is called, hence, resources which are started in `onCreate()` (line 2) are started too early, because their functionality is not needed yet. Furthermore, it represents the Energy Code Smell *Binding Resources Too Early*, which is described in Section 4.2. *Binding Resource Too Early* means that hardware resources are started at an early stage when they are not needed by the application. Hence, the method call for GPS in line 6 is enough and the call in line 3 can be deleted. In the next step, the code is represented by a TGraph to detect and remove this Energy Code Smell.

```
1  public class GpsPrint extends Activity {
2     public void onCreate(){
3        requestLocationUpdates();
4     }
5     public void onResume(){
6        requestLocationUpdates();
7     }
8  }
```

Figure 2.2: Java Code of GpsPrint (extract)

The TGraph in Figure 2.3 is colored into four different colors to show the relation to the source code in Figure 2.2. The red color presents the definition of the class `GpsPrint` and the inheritance relationship to the class `Activity` (line 1). Orange and yellow stand for the methods `onCreate()` and `onResume()` (lines 2 and 5). The method call `requestLocationUpdates()` (lines 3 and 6) for the GPS power-on is colored beige.

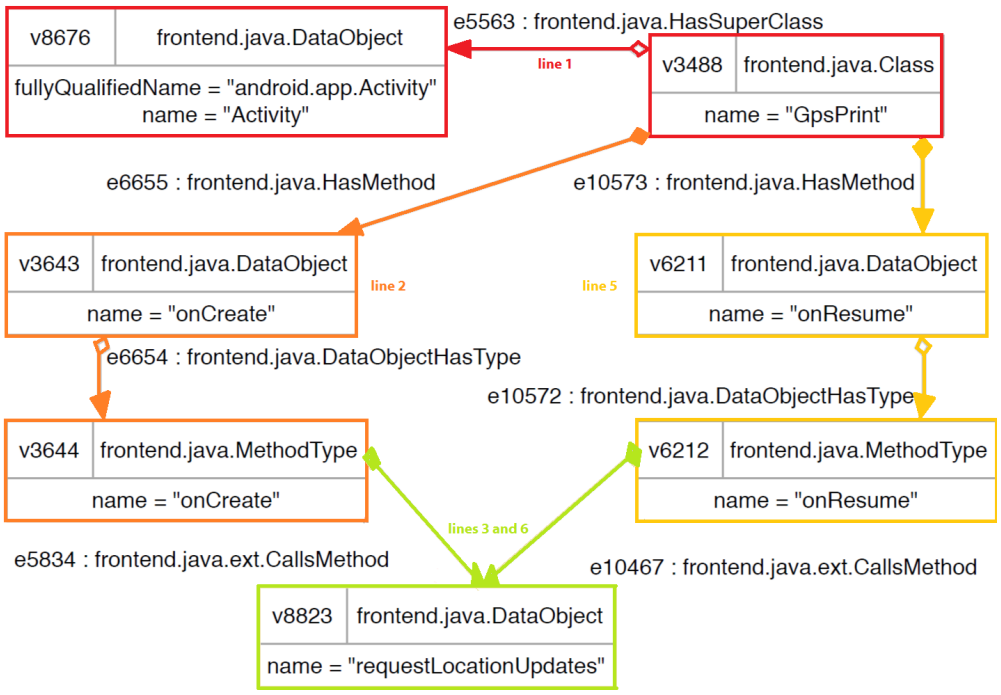


Figure 2.3: TGraph of GpsPrint (extract)

TGraphs are directed graphs, whose nodes and edges are typed, attributed, and ordered [ERW08, p. 2]. These attributes are also depicted in Figure 2.3. Edges and nodes have an identifier and a type, nodes also have a name. The direction of edges shows which node connects further nodes. The identifier of nodes and edges gives information about the TGraph element. It consists of a letter and a number. The letter "v" stands for vertex and "e" stands for edge, and its number is a consecutive number. The edge type, such as *frontend.java.HasSuperClass*, is defined by a Java meta model and shows how nodes are connected. The node type, such as *frontend.java.Class* (also defined in the Java meta model), gives information about the nodes function. Nodes also have attributes, such as *name* and *fullyQualifiedName*, to represent the source code, e.g. `requestLocationUpdates` is the *name* of the node v8823 and depicts the method in line 3 and 6.

TGraphs can beused to represent the abstract syntax of programming languages, here Java, and are accompanied by a meta model to define, analyze, query, visualize, and transform TGraphs [ERW08, p. 2], in which TGraphs have always a similar structure when the same programming language is used. A meta model for Java up to level 6 was developed during the SOAMIG project [FWE+12], so that TGraphs conforming to a Java meta model can be created. In this master's thesis, Android applications are used which are implemented in Java

6 (cf. Section 2.3). Hence, TGraphs for Android applications can be generated. The Java meta model illustrates the relation between several types of relevant source code artifacts [FWE+12, p. 167]. A small part of it is shown in Figure 2.4, and hence, it serves as meta model for the given example in Figure 2.3.

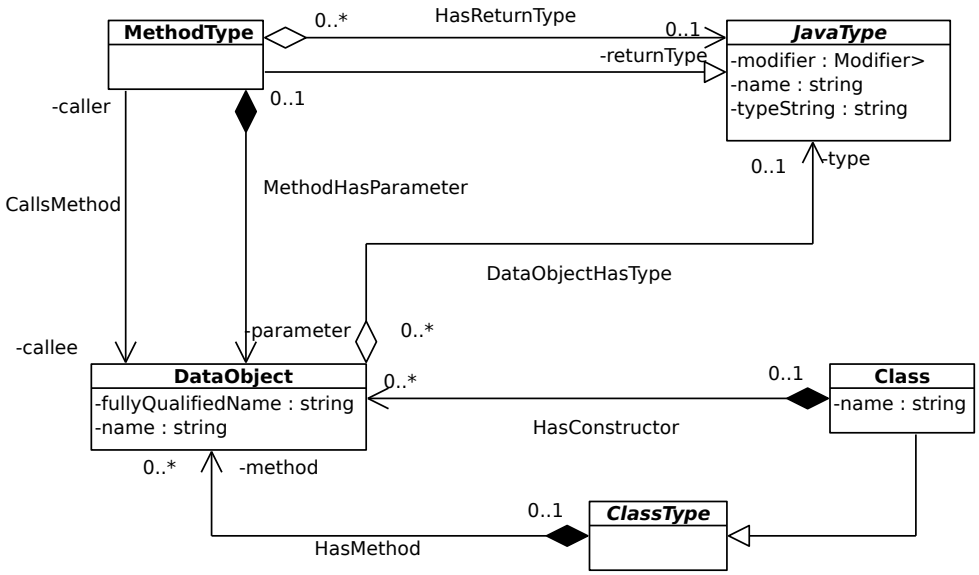


Figure 2.4: Java meta model (extract) [FWE+12]

The Java meta model shows that each TGraph of an application consists of nodes of different types. Firstly, it has at least one *Class* node which has any number of constructors. Also, it has any number of methods of the type *MethodType* which are connected to the *Class* node by a *DataObject* node via edges. Therefore, *DataObject* nodes are linked to the *class* node by a *HasMethod* edge and calls methods by a *CallsMethod* edge. *DataObject* nodes are used to store a name and a fully qualified name, and *MethodType* nodes save values for a modifier, a name, and a type. Further nodes and edges exist to represent different source code components which are not displayed here.

The functionality for analyses and restructurings of TGraphs are realized with JGraLab [Kah06]. The JGraLab API enables graph queries with GReQL (Graph Repository Query Language) to analyze TGraphs, and a graph transformation with Java methods which realizes the functionality of GReTL (Graph Repository Transformation Language) to restructure TGraphs. These techniques are needed to detect and remove Energy Code Smells (cf. Section 3), and hence, they are described in the next two sections.

2.2.2 Graph Query

The TGraph analysis is a static analysis and can be done with GReQL, a query language developed by the Institute for Software Technology of the University of Koblenz [FWE+12,

p. 168]. GReQL can be used to extract static information and dependencies in source code which are represented within the TGraph. An example for a GReQL query is given in Figure 2.5. It analyzes the TGraph in Figure 2.3 and seeks for the node `requestLocationUpdates` which is called by the node `onCreate()` and `onResume()`, but in this query, it is sufficient to know that this method is called by `onCreate()` because this is the Energy Code Smell.

```
1  from callee : V{frontend.java.DataObject}, caller : V{frontend.java.MethodType}
2  with callee.name="requestLocationUpdates" and caller.name="onCreate" and
       caller -->{frontend.java.ext.CallsMethod} (-->{frontend.java.DataObjectHasType}
       -->{frontend.java.ext.CallsMethod})* callee
3  report callee
4  end
```

Figure 2.5: GReQL example

A GReQL query can be constructed in a form with the key words: *from*, *with*, *report*, and *end*. The query starts with *from*, where the definition of nodes and edges takes place. In this example, two nodes of several types are defined, `callee` is a `DataObject` node and `caller` is a `MethodType` node. These are the node types which are demonstrated in the Java meta model in Figure 2.4. After the key word *with* the query conditions are denoted, e.g. `callee.name="onCreate"` is one condition which says, that the node `callee` must have the name *onCreate*. The condition `caller <-{frontend.java.ext.CallsMethod} [...] callee` means that the node `callee` must be called by the node `caller` through an edge with the type *frontend.java.ext.CallsMethod*. But, it is possible that `callee` is not called directly from `caller`, i.e. further nodes of the types `DataObject` and `MethodType` exist between the nodes `v3644` and `v8823` in Figure 2.3. This is queried by `(->{frontend.java.DataObjectHasType} ->{frontend.java.ext.CallsMethod})*`. These edges must be appeared together in this order, but the symbol $*$ says that this order can be repeated as often as required. In addition, two further symbols exist to influence a query: `[]` and `+`. `[]` means that these edges and nodes within the bracket are optional, and hence, it can be zero times or once. `+` represents the transitive closure, i.e. all edges and nodes, which are marked by it, occur at least once. After the key word *report* nodes and edges can be denoted which should be stored to work with them during the transformation. The key word *end* ends the query.

2.2.3 Graph Transformation

The TGraph restructuring can be done with JGraLab which realizes the GReTL functionality, a transformation language also developed by the Institute for Software Technology of the University of Koblenz [FWE+12, p. 168]. GReTL is a dedicated graph transformation language and its wide functionality is not needed for the graph transformations in this thesis, hence the functionality of the implementation of the JGraLab API suffices. The JGraLab API extends GReQL. It uses the return value of a query as a basis of the transformation. In Figure 2.6, an example for a TGraph transformation is depicted, which uses the Java API of JGraLab. This transformation is made for the TGraph in Figure 2.3 and based on the return

value `callee` of the query in Figure 2.5 which is depicted as the object `callee` in line 3 in Figure 2.6. The objective of this transformation is to remove the edge between `onCreate` and `requestLocationUpdates`.

```
1  protected Object transform() {
2     [...]
3     for (Edge edge : callee.incidences(EdgeDirection.IN)) {
4        if (edge.getAlpha().getAttribute("name").equals("onCreate")){
5           edge.delete();
6        }
7     }
8  }
```

Figure 2.6: JGraLab example

The TGraph transformation can be done through simple Java code. Each transformation is made in the method `transform()` which is a part of the provided class `Transformation` by JGraLab which is also a part of GReTL [Hor10, p. 3]. The method `transform()` is override and includes all transformation methods which are needed to realize the transformations behavior for the Energy Refactorings (cf. Section 4). In this case, all ingoing edges of the detected node `callee` are checked in line 3 by `callee.incidences(EdgeDirection.IN)`. The GReQL query for this transformation is in Figure 2.6 in line 2 within the brackets, and hence, it is a part of the transformation and the query result can be used directly. The condition `edge.getAlpha().getAttribute("name").equals("onCreate")` is needed to check whether the edge comes from a node with the name `onCreate` (line 4), due to the GReQL query before, it is clear that this edge goes to `requestLocationUpdates`. If an edge is found, it will be deleted (line 5). Here the edge `e10467` are seek, one of the incoming edges of the node `requestLocationUpdates` in Figure 2.3. The method `getAlpha()` returns the node from which the edge comes from (`getOmega()` returns the node to which the edge leads), in this case, two values must be checked because two incoming edges (`e5834` and `e10467`) exist.

2.3 Android

The implementation and evaluation of Energy Refactorings (cf. Section 4) is based on Android applications, and hence, some Android developer foundations and its application life cycle are presented here. The Android life cycle includes several states of one application and reactions of applications when another application starts. This presentation should help to define the Energy Refactorings and to realize their restructuring in Section 4.

2.3.1 Foundations

Android is a mobile platform which is based on a Linux system. Applications for Android are written in Java 6. Due to its openness and its multiplicity of partnerships, Android is very popular among users and programmers and easy to extend [Andd].

An Android application is deployed as a single `.apk` file which contains all needed source code files and resources for installing an application. After installing, each application gets its own Linux user ID and runs in its own process which is started by executing the application [Ande]. The several states of an application or rather a process are described in Section 2.3.2. Android keeps the *principle of least privilege* which gives a high security because each application only receives access to resources and components which are required for its work. The access to resources and components must be defined into the manifest file and it must be allowed by the user before installing. The manifest file `AndroidManifest.xml` is read first by the system and declares required user permissions, minimum API level, hardware components, and API libraries. Applications' resources, e.g. images and audio files, are used to create the user interface which should not declare in source code but rather in XML files where the layout is defined and resources are loaded [Ande].

The communication between applications is realized by the system because each application runs in its own process and has no permission to communicate with other applications. Therefore, an application requests another application through the system which starts the other application and sends results back to the requesting application. The system also decides between several application types: *activities*, *services*, *content providers*, and *broadcast receivers* [Ande]. Theses types have different tasks and visualizations. *Activities* are single screen applications with an user interface, e.g. games or email applications. *Services* are applications which run in background without an user interface and perform long-running operations, e.g. music player. *Content providers* manage several applications data and store them in the file system to share these with other applications, e.g. SQLite databases. So, applications have access on these data and can modify them. *Broadcast receivers* are applications which notify the system or rather other applications about new events, such as the screen is off or a download is finished. The following Android life cycle and later the restructurings are only based on the type *activities* [Ande].

2.3.2 Android Life Cycle

To understand the structure and behavior of Android applications, the Android life cycle is described and visualized. Some of the described Energy Refactorings in Section 3 are based on the behavior of several states in the life cycle.

The Android life cycle [Andb], illustrated in Figure 2.7, describes several states in which an application can be in and manages all active applications in a system. It is possible that several applications run at the same time but in different states. Thereby, only one application can be in `Foreground` wherein several applications can be in `Background`, `Hidden`, or `Inactive`. The order of the active applications is organized as a stack. A newly started application is placed on the top of the stack and the previously application is in `Background`, and thereafter, in `Hidden` when the new application is resumed. The previously application does not come back to `Foreground` again before the new application is exited. If there

are too many applications which run in `Hidden`, the system will finalize applications to get more memory for the running application.

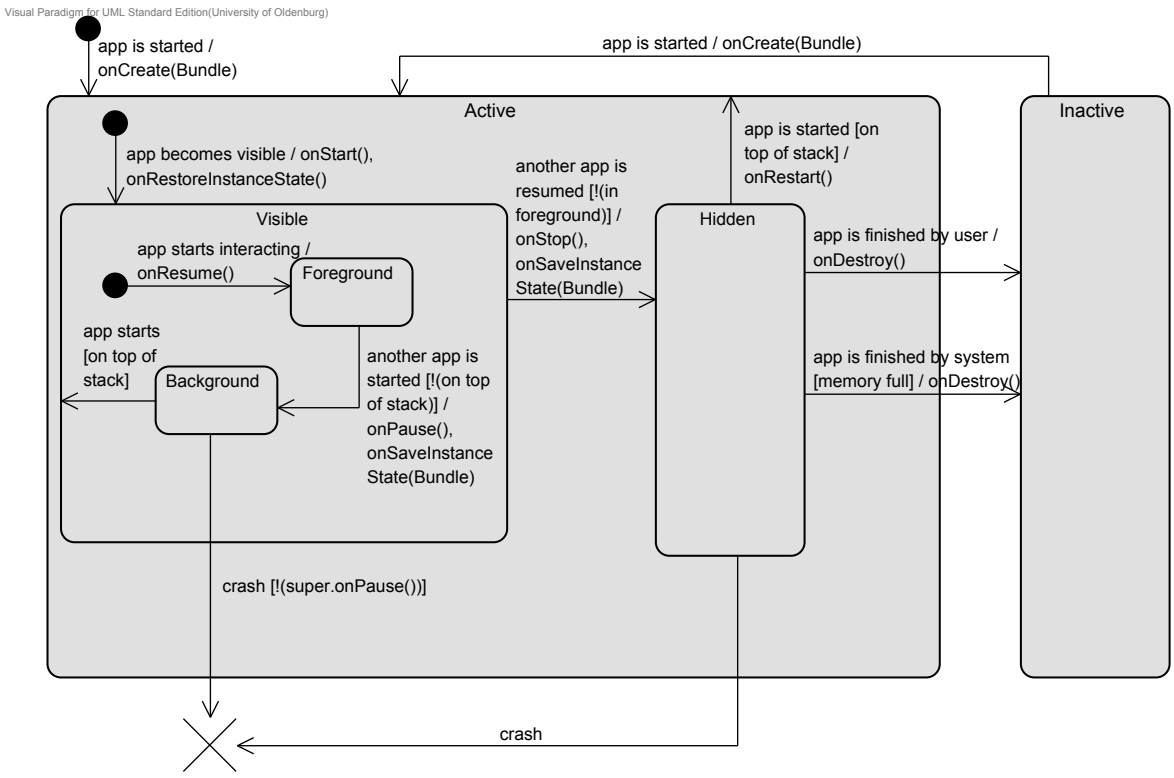


Figure 2.7: Android Life Cycle, derived from [GJJW12]

The life cycle for one Android application begins with the start of this application by the user and its state is `Active`. After starting, the application moves direct into `Visible` and becomes visible for the user, and also, the last saved instance of it is loaded. The application is now in `Foreground` and interacts with the user. If the user starts another application, this application will switch into `Background` in which the current instance of it is saved to load it again when it is again in `Foreground`. Therewith, changing between `Foreground` and `Background` takes place quickly enough the code of executed methods should be fairly lightweight. In this state or in `Hidden` the application can be terminated by the system. All applications which are not in `Foreground` move into `Hidden` after resuming a new application. Thereby, the current instance is saved in a bundle. In this state the user or the system can finish the application and it is `Inactive` when more memory is needed for the application in `Foreground`. An inactive application can be started again in which the last saved bundle is passed to load the previous instance of the application.

2.3.3 Other Operation Systems

Other mobile devices' OSs are Windows Phone and iOS. These both OSs use other programming languages than Android. Windows Phone uses C, C++, C#, and JavaScript, and hene, it allows more diversity for the programmer [Mic]. iOS uses the Cocoa Touch Framework which is based on Objective C [Khu]. Due to the several programming languages, several programming platforms are needed to create applications for the three OSs, Android, Windows Phone, and iOS. In this master's thesis Energy Refactorings (cf. Section 4) are described which can be applied on all this platforms, but in this case, they are only realized for Android applications.

2.4 Tested Hardware

In this section, mobile devices are presented which are used for the evaluation of this master's thesis. Both mobile devices are smartphones which use the OS Android: the HTC One X and the Samsung Galaxy S4. The presentation contains some software and hardware details.

HTC One X

The first mobile device for the evaluation of Energy Refactorings is the HTC One X [HTCa], and it is only named HTC in this work. It is depicted in Figure 2.8a. Hardware and software of the HTC are briefly described. Firstly, some technical details are given. The processor is a 1.5 GHz Quad Core processor, Tegra 3 from Nvidia, with this processor applications or tasks can run at the same time [She07]. The memory has a total storage of 32 GB and the RAM has a 2 GB storage. The network modules contain 2G (GSM/GPRS/EDGE) and 3G (UMTS/HSPA/CDMA), hence, the data transfer is based on known technologies. Additionally, modules for the communication and data transfer are build in, such as Wi-Fi and NFC (Near Field Communication). Furthermore, some sensors are build in, such as a gyro sensor, accelerometer, and proximity sensor, which are used by applications and the OS. The HTC uses Android 4.1.1 as OS with HTC Sense 4+ and HTC BlinkFeed. It has a 4.7 inch HD, super LCD 2 screen with a resolution of 1280x720 pixel. The Li-ion battery has a performance of 1800 mAh which allows a maximum standby time of 440 hours. These information are listed to give an overview about considered components during the energy measurements in Section 4.

Samsung Galaxy S4

The second mobile device which is used for evaluation is the Samsung Galaxy S4 [Sama]. This device is named S4 in further sections and is depicted in Figure 2.8b. Here, hardware and software of the S4 are also briefly described. Firstly, technical details are given similar to the HTC. The processor is a 1.9 GHz Quad Core processor, Snapdragon 600 from Qualcomm, it is a processor with four cores like the HTC, so that applications and tasks can run simultaneously. The memory storage amounts 16 GB and the RAM has a 2 GB storage. The

network modules contain 2G (GSM/GPRS/EDGE), 3G (UMTS/HSDPA/CDMA), and 4G (LTE), hence, this device based on the latest technologies. It uses the OS Android 4.2.2 with the user interface TouchWiz. The screen is a 5 inch Full HD Super AMOLED screen with a resolution of 1920x1080 pixel. The Li-ion battery has a performance of 2600 mAh which allows a maximum standby time of 370 hours with running 3G modules. These information are given for a better understanding in the evaluation of the Energy Refactoring *Backlight* in Section 4.4. The S4 is not used for the other evaluations in Section 4.

(a) HTC One X [HTCa] *(b) Samsung Galaxy S4 [Sama]*

Figure 2.8: Mobile Devices

2.5 Energy Measurement

In this master's thesis, the evaluation is done with energy measurements to reenact these the measurement techniques and its process are explained. Whereat, the mobile device settings are described which are adjusted before energy measurements start. Additionally, the number of measurements and its efficiency briefly described. Finally, due to a HTC firmware update, the HTC and S4 power profile and their influence are considered.

2.5.1 Measurement Techniques

Energy measurements are performed by an Android application called Andromeda. This application was programmed by Marcel Schröder during his diploma thesis [Sch13]. A screenshot shows the main functionality of Andromedar in Figure 2.9. The figure depicts five functions which can be adjusted by the user. Firstly, the energy measurement interval of the *"Input heartbeat"* is defined. In this case, 60 seconds were selected to store the current

energy consumption in these intervals. Secondly, the measurement duration *"Stop measurement after X minutes"* is determined, which amounts to 140 minutes in this case. Thirdly, measurement methods *"Select measurement method"* can be selected which is useful for the user when it is known which measurement method is the best in the following use case. Three measurement methods exist: *Delta-B, System File*, and *Energy Profiling*. The default selection chooses all implemented measurement methods. Fourthly, the output format *"Select output format"* of the measurement results can be selected. Currently, the application can only use excel. Finally, the user can start and stop the measurement. After measurement, excel files for each measurement method are stored on the flash drive from where the user can download them to a computer to evaluate the results. In the next step, the several measurement techniques are briefly described based on the description of Josefiok et. al [JSW13].

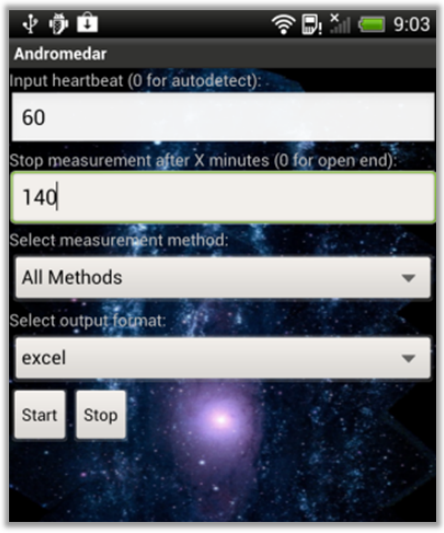


Figure 2.9: Screen shot of Andromedar

Delta-B

The *Delta-B* technique calculates the energy consumption through the battery level which can be read out with the Android BatteryManager API [Andg]. When the battery level changes, Andromedar calculates a new value of the current energy consumption by the usage of the BatteryManager API. The BatteryManager API gets the information about the battery level through Intents. An Intent is a class which contains some standard broadcast actions, e.g. to provide information about the battery change to other application which uses the BatteryManager API [Andf]. Otherwise, Andromedar uses the same value for energy consumption as before. How many values are the same is linked to the Input heartbeat and the scenario which is measured. Hence, the measurement results can look like a step function in the graph. After measurement, the average energy consumption is determined. Due to the dependence on the Android battery level which provides the most common variant of a maximum of 100 measurement results, this technique is not very accurate to determine the concrete energy consumption, but shows a trend. Furthermore, device manufacturers can set the battery level to another higher number than 100 to get more detailed measurement results. This technique works for the HTC and S4.

System File

Some mobile device vendors create an internal system file, which stores information about the current battery discharge and battery voltage. With these information, Andromedar calculates the energy consumption. HTC uses this system file, so that this technique can be used

on the HTC to measure the energy savings by executing Energy Refactorings (cf. Section 4). The HTC system file is updated every 60 seconds by the OS and gives exact information about the current energy consumption. Therefore, more than 100 values can be measured, and hence, the *System File* technique is more exact than *Delta-B*. This measurement technique does not work for the S4, maybe because no internal system file exists or it is saved on another position than the HTC system file.

Energy Profiling

Energy Profiling uses an internal XML file, which can be created by the vendor or someone who has detailed information about the hardware components. This XML file gives information about the average power of each hardware component built in the mobile device. In this case not the energy consumption is measured, but the active time of all hardware components, i.e. Andromedar measures the time between switching on and off of each component. After measurement, the active time of each component in each certain state and its average power are multiplied from which the energy consumption of the HTC and S4 results. This measurement technique is highly dependent on the hardware information and when this information provides wrong values, the *Energy Profiling* results are of no use. But when the information about the average power are very good, this technique provides the best results. When the active time of hardware components changes after restructuring, the energy consumption will be also change. Otherwise no difference between measurements are visible. This technique also works for both mobile devices.

2.5.2 Process of the Evaluation

The energy measurement of applications is the evaluation in Section 4. For each evaluation, the same process should be applied which is briefly described hereafter. Thereby, three aspects are considered: mobile device settings, number of measurements, and efficiency of measurement techniques.

Mobile Device Settings

The measurements for each Energy Refactoring (see Section 3) should always be executed under the same conditions. Therefore, a list with mobile device settings for the described mobile devices is created to guarantee correct, comparable, and reproducible measurements. This list contains screen, notification, application, wireless & network, and gesture settings. Also, information about miscellaneous settings or specifics are given. For different Energy Refactorings and applications some settings are different, hence, these settings are described when they are needed in Section 4.

- Screen settings:
  The screen is permanently on but on lowest brightness because applications are run only when the screen is active, and the lowest brightness is chosen to reduce the energy con-

sumption of the screen. The auto rotation screen is also off because the rotation sometimes interrupts e.g. the GPS connection. This was observed by first measurements with the application GpsPrint 2.6.

- Notification settings:
  Automatic notifications are off, such as email client, play store, calendar, etc. This is necessary to make comparable energy measurements which are not interrupted by individual notifications, such as emails or updates.

- Application settings:
  All background applications, such as HTC Services, Google Services, 7digital, and so on, are off so that only the tested application runs. Also, the HTC and S4 power saver are off.

- Wireless & networks settings:
  The modules mobile data, blue-tooth, GPS, NFC, and Wi-Fi are off.

- Gesture settings:
  Additional gestures, such as the three finger gestures, are off to reduce the error rate when energy measurements are started, e.g. unwanted touches on the screen which makes repeatable measurements difficult.

- Miscellaneous:
  The HTC does not have a sim card installed, with the result that the HTC seeks for it, permanently. The influence of this seek is tested and the result is recorded in Appendix A. Also, no other applications are installed apart from the standard applications, the tested application, and Andromedar.

Number of Measurements

The number of energy measurements is important to consolidate results and to exclude anomalies within the measurements. Hence, an average of ten measurements for each Energy Refactoring and application is created to get significant results. Therefor, ten measurements before and after the restructuring of the Energy Refactorings (cf. Section 4) are made to compare the energy consumption.

Efficiency of Measurement Techniques

For each energy measurement, all measurement techniques are used to show differences between the techniques. *Energy Profiling* is dependent on vendor information. When these information are wrong, this technique is not useful. Also, this technique cannot consider differences between the traffic for the Wi-Fi module. This means, that energy savings by reducing the traffic are not recognized because this technique only measures the runtime of components, such as Wi-Fi, and calculates an average energy consumption of these components, i.e. if the runtime is identical but the traffic is a less different, the energy consumption will be the same.

The other two measurement techniques, *Delta-B* and *System File*, calculate the energy consumption through information about the battery and show mostly the same trend for the energy consumption. Though, *Energy Profiling* is more precise than the others, when the vendor information are detailed enough.

2.5.3 HTC Power Profile

The power profile of the HTC contains values of the electric current of several components to calculate the energy consumption for the measurement technique *Energy Profiling*. It is read-out by the class `com.android.internal.os.PowerProfile` which is a part of the Hidden-API. The power profile is briefly described because the measurement results of the *Energy Profiling* are different to the results of the *delta-B* and *file-based measurement* in this thesis. The reason is the firmware update deployed on 22nd April 2013 [HTCb]. The firmware update to Android 4.1.1 changes the HTC power profile and is shown in Figure 2.1. In this figure the new and old power profile are illustrated to compare them. On the left side, the new power profile for Android 4.1.1, and on the right side, the old power profile for Android 4.0.4 are depicted.

Firstly, the number of considered components are different. The new power profile has four more states for the CPU. All other components are the same, but with a new value for the current. Secondly, the high difference between the values of *Screen_BSOn* should be noticed. All measurements for the evaluation are done with a switched on screen, and the difference amounts to over 97 mA which is the main reason for the differing results for *Energy Profiling* in comparison to the other two measurements in Section 4. Also, the *GPS_BS* and *Wifi_BS*, which are often used for the measurements, differs from old values. The difference for *GPS_BS* amounts 169 mA and it amounts 3 mA for *Wifi_BS*. Hence, the reported energy consumption is substantially lower than before the firmware update.

In this thesis, the new power profile for Android 4.1.1 is used, and hence, the results differ to the other results and to the results in the diploma thesis by Schröder [Sch13]. The power profile is not changed to the old profile because Andromedar reads-out the electric current values at runtime and calculates the energy consumption of each period, and hence, the measurement results cannot be changed in the excel file belatedly. But, the active time of components are also saved in the excel file which contains the measurement results, hence, the application *Andromedar Analytics* can calculate new measurement results and demonstrates them as web-application. However, the measurement results with the new Power Profile contain more CPU states which are not existing in the old one, hence, the profile cannot changed directly. In Appendix C an example is given which contains a modified version of the new Power Profile which is similar to the old one. In addition, the modified measurement results are compared with the results within the evaluation in Section 4. But it must be considered that the *Energy Profiling* and the other measurements show only a trend of the consumed energy.

Component (OS 4.1.1)	Current in mA
Screen_BSOn	2,19
Screen_BS0	4,861
Screen_BS1	14,583
Screen_BS2	24,305
Screen_BS3	34,027
Screen_BS4	43,749
CPU_BS0	10
CPU_BS1	20
CPU_BS2	30
CPU_BS3	46,9
CPU_BS4	79,4
CPU_BS5	84,4
CPU_BS6	87,8
CPU_BS7	90,6
CPU_BS8	106,1
CPU_BS9	110,3
CPU_BS10	118,3
CPU_BS11	133,3
CPU_BS12	160,6
CPU_BS13	179,2
CPU_BS14	233,6
CPU_BS15	253,3
CPU_FILE_IDLE	0,1
GPS_BS	1
Wifi_BS	0,1
Wifi_DATASend	0,1
Wifi_DATASend BYTES	0
Wifi_DATAReceived	0,1
Wifi_DATAReceived BYTES	0
Mobile_BSOn	1
Mobile_BSScan	0
Mobile_BSRadio0	0,2
Mobile_BSRadio1	0,1
Mobile_BSRadio2	0,1
Mobile_BSRadio3	0,1
Mobile_BSRadio4	0,1
Mobile_DATASend	1
Mobile_DATASend BYTES	0
Mobile_DATAReceived	1
Mobile_DATAReceived BYTES	0
Bluetooth_THREAD	0,1
Audio_THREAD	0,1

Component (OS 4.0.4)	Current in mA
Screen_BSOn	100
Screen_BS0	16
Screen_BS1	48
Screen_BS2	80
Screen_BS3	112
Screen_BS4	144
CPU_BS0	66,6
CPU_BS1	84
CPU_BS2	90,8
CPU_BS3	96
CPU_BS4	105
CPU_BS5	111,5
CPU_BS6	117,3
CPU_BS7	123,6
CPU_BS8	134,5
CPU_BS9	141,8
CPU_BS10	148,5
CPU_BS11	168,4
CPU_FILE_IDLE	2,8
GPS_BS	170
Wifi_BS	4
Wifi_DATASend	120
Wifi_DATASend BYTES	0
Wifi_DATAReceived	120
Wifi_DATAReceived BYTES	0
Mobile_BSOn	300
Mobile_BSScan	0
Mobile_BSRadio0	3
Mobile_BSRadio1	3
Mobile_BSRadio2	3
Mobile_BSRadio3	3
Mobile_BSRadio4	3
Mobile_DATASend	300
Mobile_DATASend BYTES	0
Mobile_DATAReceived	300
Mobile_DATAReceived BYTES	0
Bluetooth_THREAD	0,3
Audio_THREAD	88

Table 2.1: HTC Power Profile

2.5.4 S4 Power Profile

As regards to the completeness, the S4 power profile is also presented in Figure 2.2. At first sight this power profile may look the same as the HTC power profile in Figure 2.1. It also has five several states for the screen, but one state less for the CPU than the HTC. The remaining states for GPS, Wi-Fi, blue tooth, etc. are identical. The difference to the HTC power profile consist in the average current consumption, e.g. the switched on S4 screen consumes 120 mA whereas the new power profile of the HTC only needs 2.19 mA according to the power profile. A further example shows, that the basic power of the S4 Wi-Fi amounts 0.64 mA whereas the HTC Wi-Fi amounts 0.1 mA. Also, the other current values are different to the HTC. Hence, the values of the S4 power profile look more realistic, but in this thesis, the vendors' power profile cannot be checked, so that these values are taken for the evaluation in Section 4.

Component	Current in mA
Screen_BSOn	120
Screen_BS0	13
Screen_BS1	39
Screen_BS2	65
Screen_BS3	91
Screen_BS4	117
CPU_BS0	118
CPU_BS1	113
CPU_BS2	108
CPU_BS3	104
CPU_BS4	102
CPU_BS5	100
CPU_BS6	98
CPU_BS7	96
CPU_BS8	77
CPU_BS9	75
CPU_BS10	75
CPU_BS11	74
CPU_BS12	74
CPU_BS13	74
CPU_BS14	61
CPU_FILE_IDLE	4
GPS_BS	0,13
Wifi_BS	0,64
Wifi_DATASend	145
Wifi_DATASend BYTES	0
Wifi_DATAReceived	145
Wifi_DATAReceived BYTES	0
Mobile_BSOn	117
Mobile_BSScan	0
Mobile_BSRadio0	1,23
Mobile_BSRadio1	1,23
Mobile_BSRadio2	1,23
Mobile_BSRadio3	1,23
Mobile_BSRadio4	1,23
Mobile_DATASend	117
Mobile_DATASend BYTES	0
Mobile_DATAReceived	117
Mobile_DATAReceived BYTES	0
Bluetooth_THREAD	0,9
Audio_THREAD	35

Table 2.2: S4 Power Profile

2.6 Used Applications

Applications which are described in this section are used to validate the energy saving through Energy Refactorings (cf. Section 4). All applications are free or self-programmed Android applications. The applications *GpsPrint*, *GpsStarter*, and *TreeGenerator* are briefly presented, hereafter.

GpsPrint

GpsPrint is a free Android application by Robotmafia [Rob12] with about 1642 LOC. A screenshot for the application is illustrated in Figure 2.10. GpsPrint localizes the position of a mobile device and seeks via Internet for an address. If an Internet connection is unavailable, it will be displayed by a hint on the screen (*"No internet connection for address search"*). Also, the accuracy of the GPS coordinates are given, as well, it displays advertisements on the bottom of the screen which are reloaded each 30 seconds. In addition, the data determined can be stored (button *"Save location"*) and exported (applications menu) into a txt file to create a personal movement profile. Through the button *"Stop update"* the localization stops at which the GPS is not switched off.

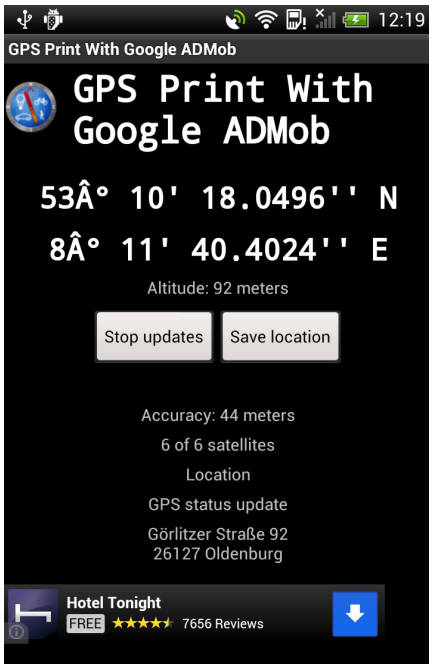


Figure 2.10: GpsPrint

This application is used for the Energy Refactorings *Third-Party Advertisement* and *Binding Resources too early* in Section 4.

GpsStarter

GpsStarter is a simple Android application by Marcel Schröder [Sch13] with 73 LOC. It is needed to start the application GpsPrint automatically after a defined time. In this thesis, GpsPrint is restarted each minute. GpsStarter has no user interface so that no screen shot exists.

This application is needed for the Energy Refactoring *Binding Resource too early* (cf. Section 4) to measure the consumed energy for starting GpsPrint.

TreeGenerator

TreeGenerator is an Android application which is created for this master's thesis with 345 LOC. It was implemented to show possible Energy Code Smells (cf. Section 4) within Android applications. A screenshot of this application is shown in Figure 2.11. TreeGenerator displays each three seconds another name of a tree, maybe its type and a picture of it. The screenshot shows *Blauregen* for a tree and *Strauch* for its type. The applications starts and stops by touching the button at the top of the screen. Also, the time how long the application runs is shown below the type. Under the time a picture of the tree is depict. The trees are chosen by a random function which takes arbitrarily a tree of a list with 50 values. Thereby, `if`- or rather `switch`-statements are used. At the bottom of the screen advertisements are integrated. For the several measurements, the application is modified, e.g. the backlight is changed from withe to black and `if`- is changed to `switch`-statements.

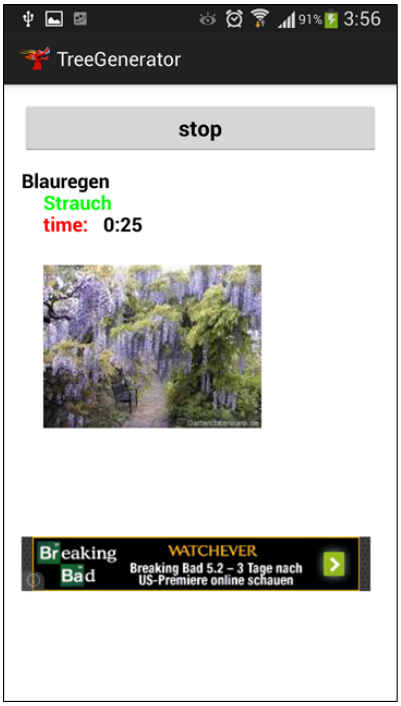


Figure 2.11: TreeGenerator

The application TreeGenerator is used to validate four Energy Refactorings *Third-Party Advertisement*, *Statement Change*, *Backlight*, and *Data Transfer* in Section 4.

3 Energy Refactorings

The energy consumption of mobile devices rises through the increasing number of installed applications. Many applications are realized by hobby programmer, hence, guidelines for energy-aware programming are seldom considered. Due to this, Energy Refactorings can be defined which have the aim to save energy on mobile devices by optimizing applications' source code. Energy Refactorings include Energy Code Smells and their restructuring. Therefore, Energy Code Smells are defined and an approach to remove them is created. This section gives an overview about the meaning of Energy Code Smells and its dependencies. Also, a brief summary is given, how the idea of Energy Code Smells arises. Hereafter, the structure of Energy Refactorings is explained which is used in the next sections for a consistent presentation. This presentation contains a name, a short definition, a motivation, constraints, an example, an analysis and a restructuring approach, and an evaluation.

3.1 Energy Code Smells

Firstly, Energy Code Smells are explained to understand why Energy Refactorings are important. For that purpose, a definition for Energy Code Smells and their dependencies are given. Secondly, the identification and handling of them are briefly described. The identification means, how Energy Code Smells are discovered and chosen. The handling of Energy Code Smells should describe the next step, how they will be removed.

3.1.1 Definition

Energy Code Smells are code parts within applications which consume more energy than an alternative implementation. Hence, these code parts can be reprogrammed in another way to save energy without changing applications' functionality. A similar definition is given by Fowler et al. [FBB+02], who describe Code Smells as code parts which are difficult to understand and maintain, and hence, it should be avoided. Also, refactorings which improve the internal structure of the source code and do not affect the external behavior of the code are presented by Fowler et al. [FBB+02, p. xvi].

Due to the increased energy consumption, Pathak et. al [PCHZ11, p. 1] denote Energy Bugs, aka Energy Code Smells, as types of errors (no functional error) within applications because of the disregarded, high energy consumption. Energy Code Smells do not have an influence on applications' functionality, so that applications do not misbehave or fudge calculations when they contain Energy Code Smells. This makes the detection of Energy Code Smells much more difficult than other programming mistakes [PCHZ11, p. 1]. A hint to an Energy Code Smell after installing an application is the shorter battery duration of mobile devices than expected [PCHZ11, p. 1].

3.1.2 Dependencies

Energy Code Smells are described platform-, software-, and hardware-independently in this master's thesis. But in the filed of mobile devices, many mobile devices use several platforms, softwares, and techniques which have an influence on the energy consumption. Due to the Android openness, the platform Android is here used for the execution and evaluation of Energy Refactorings. This is done to show the complete process of an Energy Refactoring which includes the analysis, the restructuring, and the evaluation. This would be more difficult with other platforms who do not give any information about their system. All vendors use several software enhancements and hardware components for their mobile device. In this work mostly one mobile device, the HTC, is used for the evaluation of Energy Refactorings, so that a comparison with other software and technologies is not possible. But for one Energy Refactoring, it can be shown that differences exist. In Section 4.4 an Energy Refactoring is described which based on hardware, it is shown that several screen technologies consume different amounts of energy.

3.1.3 Identification

The identification and definition of several Energy Code Smells is based on a comprehensive literature study. Initially, the department of Software Engineering at the University Oldenburg starts with a seminar paper in which first information about energy aware programming were collected. This paper was published as an *Early Research Achievements* on the CSMR 2012 [JGJ+12]. After that, a paper with a list of several Energy Code Smells and an example for one refactoring was published on a workshop for energy-aware software (EASED@GI 2012) [GJJW12].

On basis of these two paper, the master's thesis by Mirco Josefiok [Jos12] and the diploma's thesis by Marcel Schröder [Sch13] were written. Mirco Josefiok describes an *Energy Abstraction Layer* which represents a platform-independent approach for a software energy measurement. Thereupon, Marcel Schröder describes several software energy measurement techniques and implements them for Android. The idea of these two theses were published at the workshop for *Energy Aware Software-Engineering and Development* (EASED@BUIS 2013) [BGNW13, p. 17–18]. Accordingly, this master's thesis is written which includes a more detailed description of Energy Code Smells and their refactorings with an implementation, and an evaluation with the measurement techniques which were described in Schröders diploma's thesis.

On this base, authors, such as Pathak [PCHZ12], and Höpfner and Bunse [HB11], were found. These authors describe many code parts within applications which use to much energy, such as hardware resources which are started too early or stopped too late [HB10], inefficient algorithms [BRM09], and third-party advertisements [PCHZ11]. Additionally, they make hardware energy measurements to show that these code parts use more energy than the reprogrammed code. But they do not describe an approach how the source code must be changed to avoid unnecessary energy consumption. Owing to this, Energy Code

Smells and their restructuring are described and adapted to the mobile platform in this work. The mentioned ideas of the authors were described in Section 1.3.

3.1.4 Handling

After identifying Energy Code Smells, the remainder part of Energy Refactorings can be described. This part includes the detection and removing of Energy Code Smells. The term refactoring is defined in Section 2.1 and represents a combination of code analysis and restructuring without changes in its functionality, i.e. Energy Refactorings contain code analysis in which Energy Code Smells are detected by a defined analysis and removed by a defined restructuring which are described in Section 4.

3.2 Template for Energy Refactorings

A consistent presentation of Energy Refactorings is given through a structured pattern which represents the basis for an Energy Refactoring Catalog in the following sections. A part of this pattern is derived from Fowler et. al [FBB+02]. The pattern is subdivided into eight parts: name, definition, motivation, constraints, example, analysis, restructuring, and evaluation, which are described hereafter:

Name: The name of an Energy Refactoring to allow an unique identification.

Definition: A short description of an Energy Code Smell to delimit its influence on applications.

Motivation: This point shows why an Energy Code Smell and the resulting Energy Refactoring are important and in which cases it occurs. Also, references to this Energy Code Smell are given to name the original author who detected and tested it.

Constraints: This point shows problems and limits of Energy Refactorings, such as legal limitations and manual decisions of programmers. Programmers' decisions are constraints because sometimes Energy Refactorings can not be removed when it has influence on applications' behavior.

Example: An example demonstrates how an Energy Code Smell may look like to enhance the reader understanding. Also, it shows how to find Energy Code Smells in practice.

Analysis: An approach to detect Energy Code Smells within applications. This step represents the second step of the approach described in Section 1.2. At this point, several platforms can be discussed, such as Android, iOS, or Windows Phone. However, the detection of Energy Refactorings is described platform-independently, but it is only realized for Android because it is a free operation system and some free applications exist which can be analyzed.

Restructuring: At least one approach to remove an Energy Code Smell by restructuring is demonstrated. These approaches can be the same for several Energy Code Smells. If this

is the case, the second Energy Code Smell in the catalog which uses the same restructuring references to the first.

Evaluation: To demonstrate energy savings an evaluation by an energy measurement is done. For this measurement, mobile device settings are denoted to make the evaluation repeatable. These settings are described in Section 2.5.2. Also, several energy measurement techniques are used which are described in Section 2.5 to assess the Energy Refactoring.

4 Energy Refactoring Catalog

The Energy Refactoring Catalog contains a list of several Energy Code Smells and their restructuring. The looks like the described template in Section 3.2. The Energy Refactoring Catalog contains: *Third-Party Advertisement*, *Binding Resources Too Early*, *Statement Change*, *Backlight*, and *Data Transfer*. All these are described in detail, and their energy consumption before and after restructuring is measured and compared.

4.1 Third-Party Advertisement

Name: *Third-Party Advertisement*

Definition: *Third-Party Advertisements* are integrated code parts within applications which display advertisements during the operation of these applications. Thereby, advertisements do not have an influence on application's functionality but consume much energy through 3G or Wi-Fi connection [PCHZ12, p. 2].

Motivation: Tests have shown that approximately 65 % of energy consumption results from additional functionalities, such as advertisements, wakelocks, and location pinpoints, which are not important for actual applications functionality [PCHZ12, p. 1]. Many popular, free applications, like AngryBirds [Ang] and Leo [leo], include advertisements. Due to their frequent usage, e.g. AngryBird has been installed about 100,000,000 times [Ang], the total energy savings by conducting on Energy Refactoring are higher than by other applications with less downloads. Advertisements consume much energy through the 3G or Wi-Fi connection which updates advertisements every few seconds or minutes. For example, after installing AngryBirds, a 3G or Wi-Fi connection is not required for the game, but for advertisements' update. Hence, energy is saved for this application when the communication with unrequired components stops [PCHZ12, p. 13]. Alternatives for users to remove advertisements are additional applications, like Adblock Plus [PF], which filter the data stream of the 3G or Wi-Fi connection to remove advertisements from the screen [Hei12]. On the one hand, this approach does not prevent the call of methods which are responsible for advertisements, and thus, reduces the energy consumption only a little through less changes on the screen. On the other hand, the energy consumption could increase through the additional application. This approach is checked by an energy measurement in Appendix B.

Constraints: The functionality of applications is not changed, but its behavior, inasmuch as advertisements are not requested, and hence, advertisements are not displayed anymore. Programmers and application vendors use advertisements to finance the development of applications, and hence, advertisements can not be deleted without any consequences. Due to this restructuring, the GNU General Public License (GPL) [GNU] or other licenses in general are often violated. Hence, this Energy Code Smell is tested and evaluated here but not legally permitted without an agreement of programmers or vendors.

Example: The example shows a part of an application's source code which presents this Energy Code Smell and gives an idea to detect it. It is shown in Figure 4.1 and represents a small part of an Android application. This Android application includes advertisements through the Google API `ads` (see lines 1–3). The layout of the advertisement is defined in an XML-file which must be loaded (see line 8). The advertisement is displayed as Banner during the application's runtime (see lines 9–11). After that, advertisements from AdMob [AdM] are loaded and integrated into the layout. This API can be used by Android, iOS, and Windows phone applications [Goob], so that a similarity exists to detect this energy code smell.

```
1  import com.google.ads.AdRequest;
2  import com.google.ads.AdSize;
3  import com.google.ads.AdView;
4  [...]
5
6  public void onCreate(Bundle savedInstanceState) {
7     [...]
8     LinearLayout layout = (LinearLayout)findViewById(R.id.ad);
9     AdView adView = new AdView(this, AdSize.BANNER, "a1516d1a3e604e5");
10    adView.loadAd(new AdRequest());
11    layout.addView(adView);
12    [...]
13 }
```

Figure 4.1: Example of Third-Party Advertisements

The usage of the Google API `ads` within Android applications is only possible, when the `manifest.xml` (see Section 2.3.1) is changed. The part which must be included into the `manifest.xml` is illustrated in Figure 4.2. Firstly, `AdActivity` from the Google API `ads` must be included (see lines 2 – 4) to allow the configuration changes and the usage of it. Additionally, the Android application needs a permission for the network access (see line 6), if Wi-Fi and 3G are not available.

```
1  [...]
2  <activity android:name="com.google.ads.AdActivity"
3    android:configChanges="keyboard|keyboardHidden|orientation|screenLayout|uiMode|
         screenSize|smallestScreenSize">
4  </activity>
5  [...]
6  <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" ></uses-
       permission>
7  [...]
```

Figure 4.2: Ads into manifest.xml

Analysis: This Energy Code Smell can be detected by searching third-party APIs and calls of their methods to display advertisements. Therefore, more information about applications are needed to know which APIs could be used by programmers. This depends on the OS on which applications run. If the OS is known, third-party APIs and their method calls can be detected by querying special imports, initializations, and methods. However, various advertisement APIs have to be known, as well.

Firstly, advertisement imports can be detected through seeking after an `ImportStatement` with a special name. A possible approach is shown in Figure 4.3. In this case, the

wanted `ImportStatement` is called `AdRequest` (see line 2). If this statement detected, all dependent nodes can be deleted.

```
1  from importRequest : V{frontend.java.ImportStatement}
2  with importRequest.element = "AdRequest"
3  report importRequest
4  end
```

Figure 4.3: Third-Party Advertisements Analysis 1

When imports are detected, the search after code which is part of this Energy Code Smell could continue and is depicted in Figure 4.4. Therefore, code like code in Figure 4.2 from line 8 to 11 is seek. In this case, an `Access` node is seek which is also called `AdRequest` (see line 4). A further condition is that the `Access` node is part of a sequence of several nodes (see lines 4–7) which are all integrated in the `MethodType`, named `onCreate` (see line 8). This is necessary because Android allows advertisements only in this method. If a node is detected, it will be returned and can be deleted.

```
1  from adRequestCall : V{frontend.java.Access}, adRCC : V{frontend.java.ConstructorCall},
2       adRCCN : V{frontend.java.NewObject}, caller : V{frontend.java.MethodCall},
3       onCreate : V{frontend.java.MethodType}
4  with adRequestCall.name = "AdRequest" and onCreate.name = "onCreate"
5       and adRequestCall <--{frontend.java.HasOperand} adRCC
6       <--{frontend.java.HasConstructorCall} adRCCN <--{frontend.java.HasOperand} caller
7       <--{frontend.java.HasExpression} <--{frontend.java.BlockContainsStatement}
8       <--{frontend.java.HasMethodBlock} onCreate
9  report adRequestCall
10 end
```

Figure 4.4: Third-Party Advertisements Analysis 2

Further queries are implemented to detect next nodes, such as `AdView` and `AdSize`. These queries look similar to the described one and do not considered in this Section.

Restructuring: Code parts which are responsible for *Third-Party Advertisements* can be removed without any influence to on applications' functionality. But it is important to remove all code parts of third-party APIs, otherwise the source code might contain errors, e.g. when imports of APIs are deleted but not all method calls. Except for the deletion of advertisements request, advertisements can be replaced by an image which is saved on the memory card. Thereby, the image can be monochrome, e.g. black or white, or multi-colored, e.g. a picture. It is possible that several images have a different influence on energy consumption. In addition, advertisements' display can be suppressed by removing the method which adds the advertisements to applications layout (Figure 4.1 line 11). But the energy costs for communication still exist (Figure 4.1 lines 8–10). In this case, the complete removing of advertisements is done to save energy through displaying and communication.

Removing imports is done by delete nodes and edges which are necessary for the import and a part of it is shown in Figure 4.5. Therefore, the nodes are chosen (see lines 2 – 4) and all their edges are saved in an `Iterable`. Edges of each node are deleted before the node is deleted (see lines 6–9) because after removing the node no access to their edges is available.

```
1  public void removeImports(JValue jValue){
2    JValueCollection vertexCollection = jValue.toCollection();
3    JValue requestVertex = vertexCollection.toJValueList().get(0);
4    Vertex vertexAdRequestImport = requestVertex.toVertex();
5    Iterable<Edge> list = vertexAdRequestImport.incidences();
6    deleteEdges(makeCollection(list));
7    deleteNextVertex(vertexAdRequestImport, vertexAdRequestImport
8        .getAttribute("element").toString());
9    vertexAdRequestImport.delete();
10 }
```

Figure 4.5: Restructuring of Imports

Evaluation: A first evaluation with the application GpsPrint shows the result of the energy refactoring *Third-Party Advertisement*. Figure 4.7 depicts the measurements with and without advertisements. Before the measurement results are considered, the mobile device settings are shown in table 4.1.

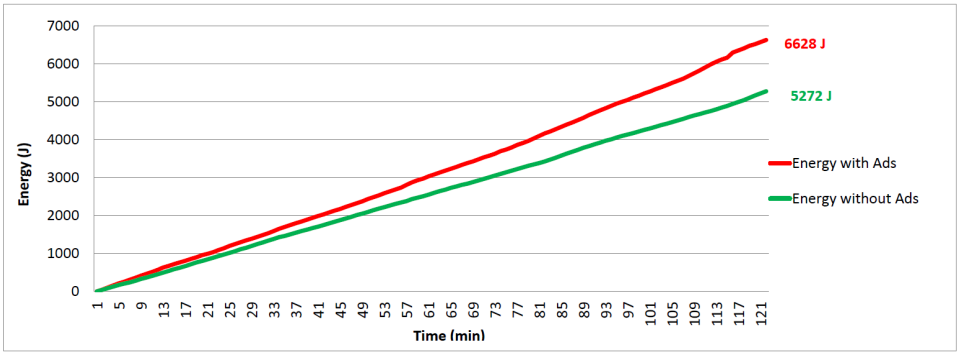
Screen Settings	screen	on
	brightness	lowest
	auto rotation	off
Notification Settings	notifications	off
Applications settings	background applications	Andromedar
	power saver	off
Wireless & network settings	mobile data	off
	blue-tooth	off
	GPS	on
	NFC	off
	Wi-Fi	on
Gesture settings	three finger gestures	off
Miscellaneous	sim card	not installed
	other applications	not installed

Table 4.1: Mobile Device Settings for Third-Party Advertisements (GpsPrint)

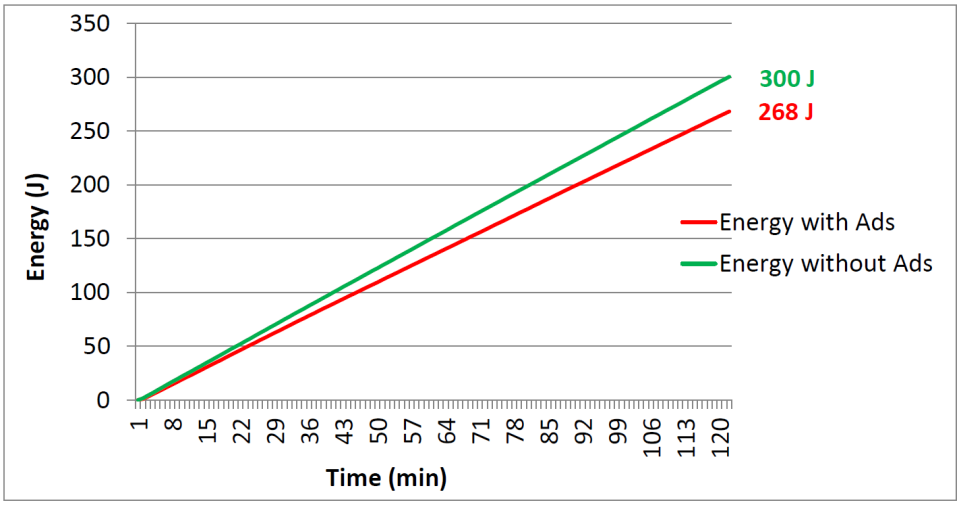
Figure 4.6a shows the *file-based measurement* which results that the energy consumption of GpsPrint with advertisements (red line) is higher than without advertisements (green line). The energy saving amounts 1356 J within two hours. This is a reduction of 21 % of the energy consumption. Due to thus measurements, it is shown that the total energy consumption of applications can be reduced through removing advertisements.

The *Energy Profiling* in Figure 4.6b shows a difference from 32 J in which the application with advertisements consumes less energy. In most cases, removing advertisements can save the Wi-Fi component but here the Wi-Fi is used to check the address of the current location, hence the Wi-Fi works during the whole applications runtime. Only the data traffic of this application is reduced which can not be considered by *Energy Profiling* because the Wi-Fi component can not remain idle when the address data are checked by GpsPrint. Also, the measurement results deviates from the other measurement results because the Power Profile has changed after the last firmware update. A detailed explanation is given in Section 2.5.3.

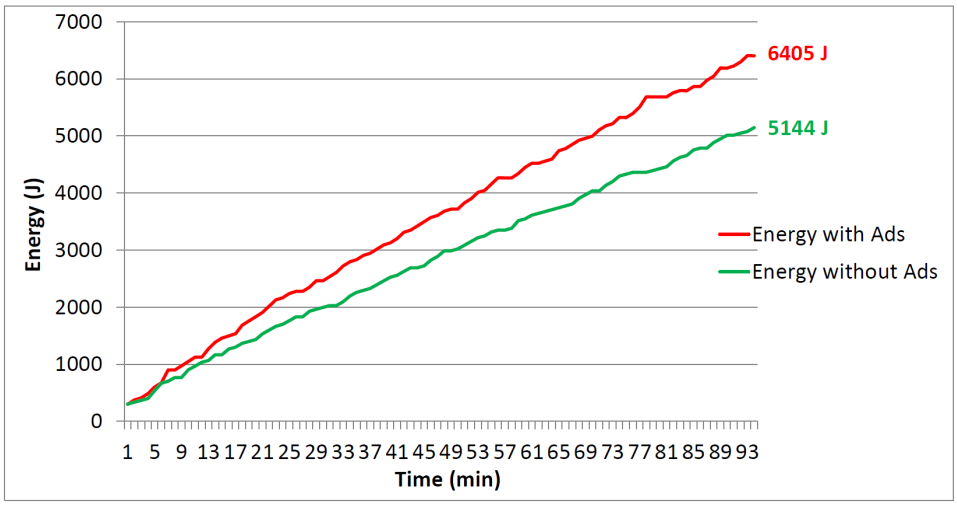
The *delta-B* measurement in Figure 4.6c shows the same trend like the other two. The difference amounts 1261 J, and also, shows that energy saving by removing advertisements is possible.



(a) File-based Measurement



(b) Energy Profiling



(c) Delta-B Measurement

Figure 4.6: Measurement of GPSPrint with and without Ads

A second evaluation is done with the application TreeGenerator (cf. Section 2.6). It is shown in Figure 4.7. In this case, TreeGenerator is modified, so that the Internet access is only needed for advertisements, and hence, the pictures of the trees are deleted to reduce requests via Internet. Therefore, the Wi-Fi component can be switched off for the measurement without advertisements. Hence, three different measurements are made. During two measurements the Wi-Fi component is switched on (red and green lines), and during one measurement it is switched off (blue line). But, advertisements are requested and displayed only during one measurement (red line). The other two measurements are without advertisements (green and blue lines). In Table 4.2 the mobile device settings for these measurements are represented.

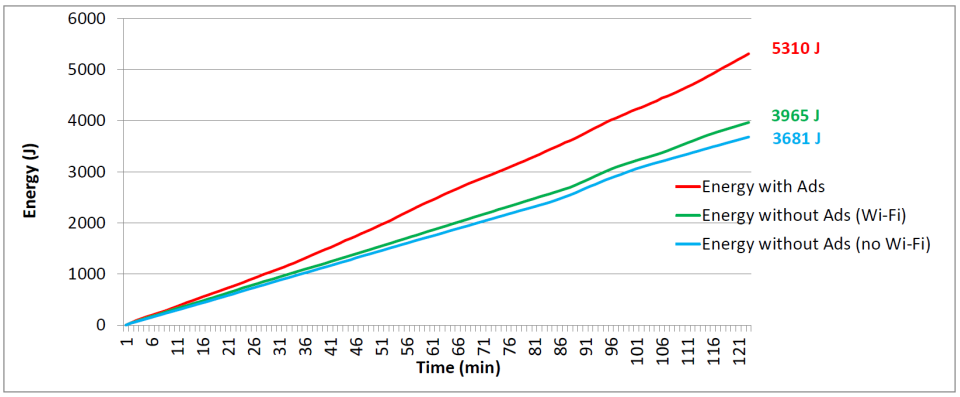
Screen Settings	screen	on
	brightness	lowest
	auto rotation	off
Notification Settings	notifications	off
Applications settings	background applications	Andromedar
	power saver	off
Wireless & network settings	mobile data	off
	blue-tooth	off
	GPS	off
	NFC	off
	Wi-Fi	on / off
Gesture settings	three finger gestures	off
Miscellaneous	sim card	not installed
	other applications	not installed

Table 4.2: Mobile Device Settings for Third-Party Advertisements (TreeGenerator)

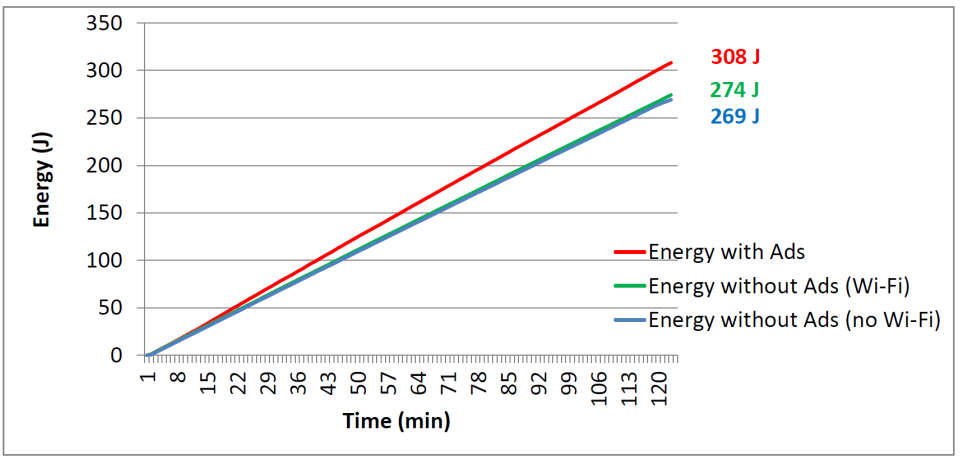
The first graph 4.7a shows the results of the measurement technique *file-based measurement*. TreeGenerator with advertisements consumes significantly more energy than the other two scenarios. The difference with TreeGenerator without advertisements and a switched on Wi-Fi component amounts 1345 J within two hours. The difference with TreeGenerator without advertisements and a switch off Wi-Fi component amounts 1629 J within two hours, so that the difference between switched on and off Wi-Fi component amounts 284 J.

The *Energy Profiling* in Figure 4.7b shows a lesser differences and general lesser values. These values are dependent on the Power Profile which is explained in Section 2.5.3 and shows why the values differ from the other measurements. TreeGenerator with advertisements needs 308 J, without advertisements 274 J are consumed, and without advertisements and switched off Wi-Fi component 269 J are used.

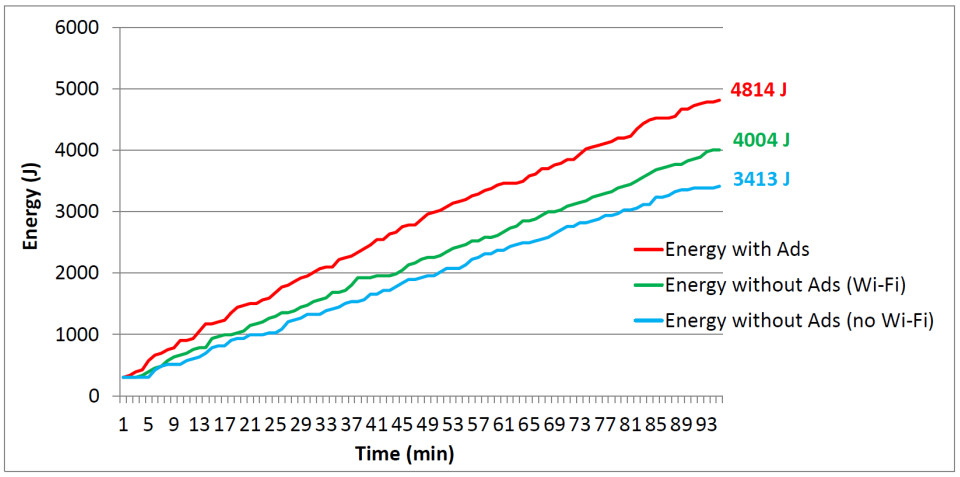
Finally, the *delta-B measurement* in Figure 4.7c is depicted. This measurement shows the highest energy consumption for TreeGenerator. TreeGenerator with advertisements consumes 4814 J and its difference to TreeGenerator without advertisements amounts 810 J and to TreeGenerator without advertisements and Wi-Fi component 1401 J. These measurements have the largest difference between TreeGenerator with Wi-Fi and without, it amounts 591 J, and hence, it shows the energy consumption of the Wi-Fi component.



(a) File-based Measurement



(b) Energy Profiling



(c) Delta-B Measurement

Figure 4.7: Measurement of TreeGenerator with and without Ads

Both evaluations show that removing advertisements within applications saves energy and has no influence on the functionality of applications. It is even possible that the Internet connection can be turned off without influence on applications when advertisements are removed. Thereby, these measurement techniques shows an energy savings of approximately 30 % are possible (cf. Figure 4.7a). This matches the values in Pathak et. al [PCHZ12, p. 1] where still more energy is saved by removing advertisements.

4.2 Binding Resources Too Early

Name: *Binding Resources Too Early* (BRTE)

Definition: *Binding Resources Too Early* describes hardware components, such as Wi-Fi and GPS, which are switched on by applications at an early stage when components are not needed by the application or user [PCHZ11] [GJJW12, p. 6].

Motivation: The energy consumption of several hardware components can be very high [CH10], so that it is important to reduce the runtime and uptime of these components, e.g. reduce the runtime of GPS and reduce the uptime of CPU. One option would be to start these components only when they are really needed. Carroll and Heiser have demonstrated through several measurements that the most energy saving can be realized by the shutdown of unused components, which finally consume energy [CH10, p. 11]. Hence, this Energy Refactoring tries to reduce the runtime of components to produce maximum benefits.

Constraints: *Binding Resources Too Early* can only be detected and removed when the programmer knows the OS-specific structure with regard to the energy efficiency, which is sometimes given by the vendor, e.g. the Android Life Cycle which is described in Section 2.3.2. Furthermore, the programmer must know which notifications or hardware states are typical for the method call of a hardware component, such as `lm.requestLocationUpdate` in Figure 4.8 line 16, in order not to forget to shift code parts which would cause an error after refactoring. So, it must be considered that manual decisions are necessary to execute a successful refactoring and platform-specific knowledge are also necessary.

Example: This is an Android specific example which is structured according to the Android life cycle in Section 2.3.2. The Android life cycle describes several states in which methods, such as `onCreate()` and `onResume()`, are called during switching between the states. These two methods can be seen in Figure 4.8 which presents a source code part of GpsPrint. On the left site the original code of GpsPrint is shown, where the GPS component is started in `onCreate()` in line 16 by `lm.requestLocationUpdates(...)`. On the right site the restructured code is shown. The whole part of starting GPS and saving the component status is shifted to the method `onResume()` which is called later than `onCreate()`. As a result, that the GPS starts when it is needed, in this case when the application is visible for the user.

```
1  public class GpsPrint extends Activity
2      implements OnClickListener,Listener,
3      LocationListener {
4  [...]
5    public void onCreate(Bundle
6        savedInstanceState) {
7  [...]
8      LocationManager lm=(LocationManager)
9          this.getSystemService(Context.
10         LOCATION_SERVICE);
11     if(lm.getAllProviders().contains(
12         LocationManager.GPS_PROVIDER)){
13       if(lm.isProviderEnabled(
14          LocationManager.GPS_PROVIDER)){
15         lm.addGpsStatusListener(this);
16         lm.requestLocationUpdates(
17             LocationManager.GPS_PROVIDER,
                   1000, 0, this);
18         status_view.setText(
19             "GPS service started");
20         } else {
21         status_view.setText(
22             "Please enable GPS");
23         save_location_button.setEnabled(
24             false); }
25 [...] }
26 [...]
27   public void onPause() {
28 [...]
29     lm.removeUpdates(this);
30 [...] }
31   public void onResume() {
32 [...]
33     lm.requestLocationUpdates(
34         LocationManager.GPS_PROVIDER,
35         1000, 0, this);
36 [...] }
37 }                    Before Refactoring
```

```
1  public class GpsPrint extends Activity
2      implements OnClickListener,Listener,
3      LocationListener {
4  [...]
5    public void onCreate(Bundle
6        savedInstanceState) {
7  [...]
8      LocationManager lm=(LocationManager)
9          this.getSystemService(Context.
10         LOCATION_SERVICE);
11         //removed by refactoring
12
13 [...] }
14 [...]
15   public void onPause() {
16 [...]
17     lm.removeUpdates(this);
18 [...] }
19   public void onResume() {
20 [...]
21     if(lm.getAllProviders().contains(
22         LocationManager.GPS_PROVIDER))
23       if(lm.isProviderEnabled(
24          LocationManager.GPS_PROVIDER)){
25         lm.addGpsStatusListener(this);
26         lm.requestLocationUpdates(
27             LocationManager.GPS_PROVIDER,
                   1000, 0, this);
28         status_view.setText(
29             "GPS service started");
30         } else {
31         status_view.setText(
32             "Please enable GPS");
33         save_location_button.setEnabled(
34             false); }
35         }
36 [...] }
37 }                    After Refactoring
```

Figure 4.8: Example of Binding resources too early

Analysis: To detect this energy refactoring, specific method calls for hardware components must be known which can be dependent on the OS. If the method calls for components, such as GPS, Wi-Fi, and Blue-tooth, are known, it will be possible to search for these method calls by querying. An example is shown in Figure 4.9.

The query seeks for a method call which starts a hardware component, in this example the GPS is started through the method `requestLocationUpdates`. If this method is detected and it is called by the node `onCreate` which is a part of any class (here GpsPrint) which extends the class `Activity`, the node `onCreate` will be returned. In addition, a second query proofs whether the method `requestLocationUpdates` is called by the method `onResume` which must be also a part of GpsPrint. If the method `onResume` does not call `requestLocationUpdates`, a third query will be executed which return the method `onResume` anyway, when it is a part of GpsPrint. With these three information the transformation for this Energy Refactoring is done.

```
1  from onCreate, caller : V{frontend.java.MethodType}, "actClass : V{frontend.java.Class},
       superClass, callee : V{frontend.java.DataObject}
2  with onCreate.name = \"onCreate\" and superClass.fullyQualifiedName = \"android.app.
       Activity\"  and callee.name = \"requestLocationUpdates\" and
       callee <--{frontend.java.ext.CallsMethod} caller (<--{frontend.java.
       DataObjectHasType} <--{frontend.java.ext.CallsMethod})* onCreate <--{frontend.java.
       DataObjectHasType}<--{frontend.java.HasMethod} actClass -->{frontend.java.
       HasSuperClass} superClass
3  report onCreate
4  end
```

Figure 4.9: Binding Resources Too Early Analyze

Restructuring: After detecting a method call in a too early state, the call must be shifted into the right place. Therefore, the structure of applications for a specific OS must be known to decide where the code is shifted. This must be tested by a programmer who has semantic program understanding in order not to change applications functionality. In this regard, it should be considered that all code parts are shifted which are connected with the component method call, such as comments and hardware state storages. The realized transformation in Figure 4.10 only shifts the method `requestLocationUpdates` which is responsible for starting the GPS is, the comments must be shifted manually because their structure is to individual.

```
1  if (edge.getOmega().getAttribute("name").equals("requestLocationUpdates")) {
2     if(onResumeV.toCollection().size()==0){
3        edge.delete();
4     } else {
5        [...]
6        edge.setAlpha(onResumeV.toVertex());
7        [...]
8     }
9  }
```

Figure 4.10: Binding Resources Too Early Restructuring

First, it is tested which edge of the node `onCreate` calls the method `requestLocationUpdates`, this is shown in line 1. If this edge is detected, it will be proofed whether the same call exist in `onResume` or not (line 2). If the method call `requestLocationUpdates` does not exist in `onResume`, the method `requestLocationUpdates` (called by `onCreate` before) is shifted to `onResume` (line 6). Otherwise, the edge can be deleted without any dependencies.

Evaluation: After restructuring, an energy measurement is done to check the possible energy saving. The tested application is GpsPrint, and hence, the mobile device settings are depicted in Table 4.3. GPS and Wi-Fi are switched on for both measurements (with and without *Binding Resources Too Early*) to get the coordinate and to localize the mobile device the whole time.

Screen Settings	screen	on
	brightness	lowest
	auto rotation	off
Notification Settings	notifications	off
Applications settings	background applications	Andromedar
	power saver	off
Wireless & network settings	mobile data	off
	blue-tooth	off
	GPS	on
	NFC	off
	Wi-Fi	on
Gesture settings	three finger gestures	off
Miscellaneous	sim card	not installed
	other applications	not installed

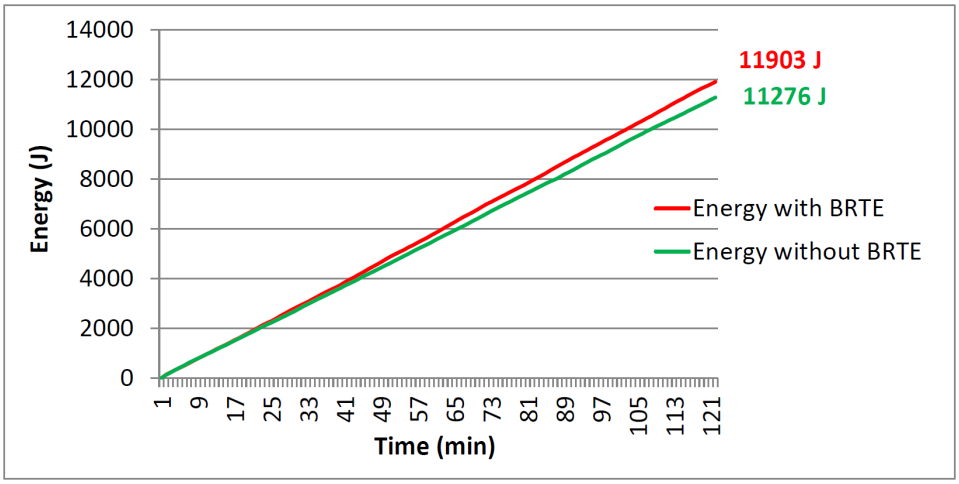
Table 4.3: Mobile Device Settings for Binding Resources Too Early

The result of the *file-based* measurement is shown in Figure 4.11a. Theses graphs illustrate the average of seven measurements to get a significant result. The HTC and the running application GpsPrint with *Binding Resources Too Early* (BRTE) consume 11903 J (red line). After that, GpsPrint without *Binding Resources Too Early* is tested and consumes 11276 J. Hence, the energy saving amounts 627 J within two hours.

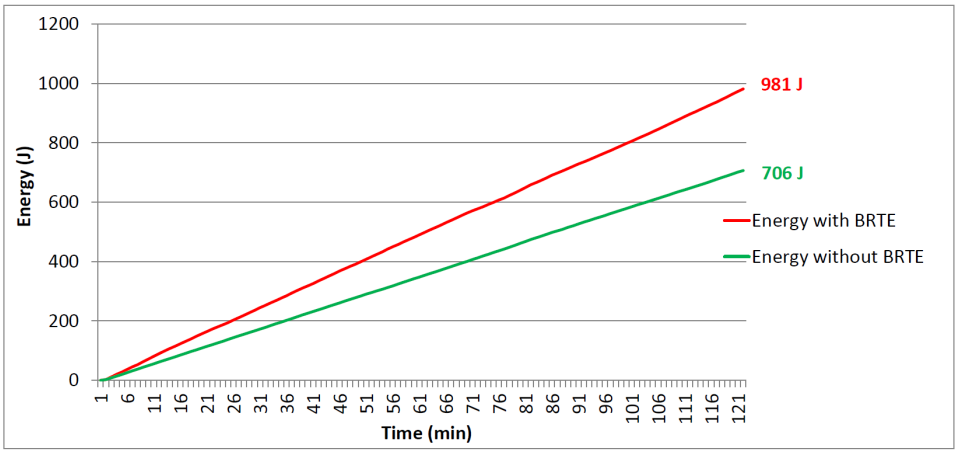
The second measurement result based on *Energy Profiling* and is shown in Figure 4.11b. The difference amounts 275 J within two hours, and hence, it is minimal like the measurement before shows. However, these measurement results are strange in comparison to the other measurements in Figure 4.11a and 4.11c. The difference of the total energy consumption is clearly visible in Figure 4.11a and 4.11b, it amounts about 10922 J. The reason is the updated HTC Power Profile which contains lower current values than before. This is described in Section 2.5.3.

The result of the *delta-B* measurement is depicted in Figure 4.11c. The total energy consumption for the HTC and the application GpsPrint with *Binding Resources Too Early* amounts 19566 J and without *Binding Resources Too Early* 16032 J. This measurement shows the same trend like the *file-based* measurement, i.e. the energy saving amounts 3534 J within two hours.

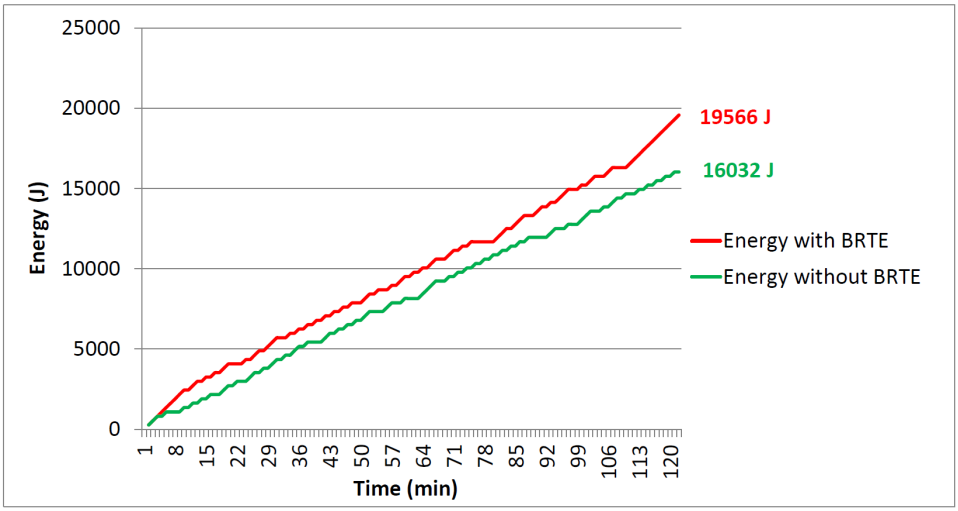
This evaluation shows that energy savings for Android applications are possible when the Android Life Cycle in Section 2.3.2 is followed. The *delta-B measurement* illustrates an energy saving of approximately 18 % and the *Energy Profiling* even shows an energy saving of 28 % in which it is not known how good the Power Profile of it is (cf. Section 2.5.3). This Energy Refactoring is a software-specific one, because the Android Life Cycle is only for the Android OS and can not be transmitted to other OSs. However, similar structure could exist for their life cycle.



(a) File-based Measurement



(b) Energy Profiling



(c) Delta-B Measurement

Figure 4.11: Measurement of GPSPrint with and without BRTE

4.3 Statement Change

Name: *Statement Change*

Definition: *Statement Change* describes two statements, in this case, `if` and `switch` which can be interchanged, because both statements are used to run code parts under certain conditions [Ull11, ch. 2.6].

Motivation: During the *2nd EASED Workshop (Energy-Aware Software-Engineering Development)* [BS13] some ideas about energy-aware programming were discussed. One of these ideas was the interchange of `if`- and `switch`-statements within applications. For Android applications, it is checked in this thesis, because no literature was found which describes or checks this idea by an energy measurement. Currently, `switch`-statements are used for a better code maintenance because a sequence of `switch`-statements is more clear than a sequence of `if`-statements [Ull11, ch. 2.6].

Constraints: `Switch` and `if` are used to make decisions during the program runtime. But they do not work completely similar, e.g. after a match with one `case` in `switch`-statements, all other assignments are executed when no `break` is used (called *Fall-Through* [Ull11, ch. 2.6]). `If`-statements are only executed when the condition is true, and hence, no `break` is needed. Furthermore, `if` can be used for comparisons with all data types and `switch` only with `integer`, `enum`, and `String` (since Java 7). Android SDK 18.0 works with Java 1.6, and hence, a comparison with `Strings` is not allowed. These differences limit the usage of `switch` within Android applications. Also, `switch` works only with constant operators. Hence, an additional method must be created which gets an operator which can be changed during applications runtime (cf. Figure 4.12) [Ull11, ch. 2.6].

Example: Figure 4.12 shows an example for the two statements. The statements `if` (left side of Figure 4.12) and `switch` (right side of Figure 4.12) are chosen. Both code parts have the same functionality and are implemented in TreeGenerator (cf. Section 2.6). Firstly, it should be noted that `switch`-statements need more lines of code to implemented the same functionality like `if`- statements. The reason is, that `switch`-statements are outsourced in new methods (see lines 9, 12, 20, and 34) and an `enum` must be created to define values which can be compared in `switch`-statements (see line 1). Secondly, each `case` ends with a `break` to prevent the *Fall-Through* (see lines 25, 28, 31, 39, and 42). Finally, a `for` loop are needed to compare all `enum` values with the current chosen tree (see lines 11–17), so that the method `switchTreeTypes()` (line 13) runs for them all.

```
// tree generator
class ValueTask extends TimerTask {
   @Override
   public void run() {
      MainActivity.this.runOnUiThread(
          new Runnable() {
         @Override
          public void run() {
             i = (int) (Math.random() *
                 50);
    //tree list
    if(i == 1) {
       value.setText("Nikko-Tanne");
    } else if(i == 2){
       value.setText("Riesen-Tanne");
    } else if(i == 3){
       value.setText("Nordmanntanne");
    }
    [...]
    //tree type list
    if (value.getText().toString().
        equals("Blauregen")){
       type.setText("Strauch");
    } else if (value.getText().toString
        ().equals("Weisstanne")){
          type.setText("Tanne");
    }
            [...]
}
```

```
private enum Tree {Blauregen, Weisstanne
    , Nordmanntanne, Blaufichte,
    Douglasie, Umweltmammutbaum, Spirke,
     Weymouthskiefer, Zirbelkiefer,
    Blauglockenbaum, Sadebaum;}
// tree generator
class ValueTask extends TimerTask {
@Override
public void run() {
   MainActivity.this.runOnUiThread(new
       Runnable() {
   @Override
   public void run() {
      j = (int) (Math.random() * 50);
      switchTree(j);
      for (Tree tree : Tree.values()) {
         if (value.getText().toString().
             equals(tree.toString())) {
             switchTreeType(tree);
             break;
         } else {
               type.setText("not
                   specified");
         }
         [...]
}
// tree list
public void switchTree(int random) {
   switch (random) {
      case 1:
         value.setText("Nikko-Tanne");
         break;
      case 2:
         value.setText("Riesen-Tanne");
         break;
      case 3:
         value.setText("Nordmanntanne");
         break;
      [...]
}
// tree type list
public void switchTreeType(Tree tree) {
   switch (tree) {
      case Blauregen:
         type.setText("Strauch");
         break;
      case Weisstanne:
          type.setText("Tanne");
          break;
       [...]
}
```

Figure 4.12: Example for Statement Change

Analysis: This Energy Refactoring can be detected by a query which seeks `if`-statements with more than three `else`-statements. Three `else`-statements should be the minimum when `if`-statements are replaced through `switch`-statements because the effort is too high when each simple `if`-statement is identified and must be checked by a programmer. Furthermore, the restructuring should have an influence on the energy consumption, and hence, changes should not be too small. The query to find these `if`-statements could look like the GReQL query in Figure 4.13.

```
1  from ifStatement : V{frontend.java.IfStatement}, class : V{frontend.java.Class},
       method : V{frontend.java.DataObject}
2  with ifStatement (<--{frontend.java.HasElseStatement} <--{frontend.java.HasElseStatement
       } <--{frontend.java.HasElseStatement})+ <--{frontend.java.BlockContainsStatement}
       [<--{frontend.java.HasMethodBlock} <--{frontend.java.DataObjectHasType} method
       <--{frontend.java.HasMethod}] class
3  report ifStatement, method, class
4  end
```

Figure 4.13: Statement Change Analyze

The query seeks for three nodes: `ifStatement`, `method`, and `class`, to get information in which method and class `if`-statements with more than three `else`-statements are used. Therefor, `(<-frontend.java.HasElseStatement <-frontend.java.HasElseStatement <-frontend.java.HasElseStatement)+` are requested. This demonstrate that only statements are identified which have at least three other statements before directly. The symbol + denotes that the edges inside the parentheses must be occur once. In any case, statements are a part of a block which is a part of a class or a method, hence, the symbol `[]` is used to represent a method and its call between class and block which is optional.

Restructuring: In this case, an automatic restructuring is very difficult because many changes must be done, e.g. when `enums` are needed because `Strings` are used within `if` conditions. Furthermore, the nodes for the `if`-statement must be changed and shifted into a new method when the condition is not constant. Hence, it is easier to make the change by a programmer who decides whether a manually restructuring is sensible or not. The programmer is supported by the analyze which returns methods with `if`-statements which have more than three `else`-statements.

Evaluation: The energy measurement is done with the application *TreeGenerator* (cf. Section 2.6) but without advertisements and pictures (cf. Section 2.6) to have less components which have an influence on the energy consumption. Also, the frequency of showing tree types is increased to each second instead of each three seconds. The mobile device settings are presented in Table 4.4

For the energy measurement, also the three measurement techniques, *file-based measurement*, *Energy Profiling*, and *delta-B measurement*, are used. The measurement results in Figure 4.14 show that their is not a significant difference concerning the energy consumption between `if`- and `switch`-statements. All results of the several measurement techniques

Screen Settings	screen	on
	brightness	lowest
	auto rotation	off
Notification Settings	notifications	off
Applications settings	background applications	Andromedar
	power saver	off
Wireless & network settings	mobile data	off
	blue-tooth	off
	GPS	off
	NFC	off
	Wi-Fi	on
Gesture settings	three finger gestures	off
Miscellaneous	sim card	not installed
	other applications	not installed

Table 4.4: Mobile Device Settings for Statement Change

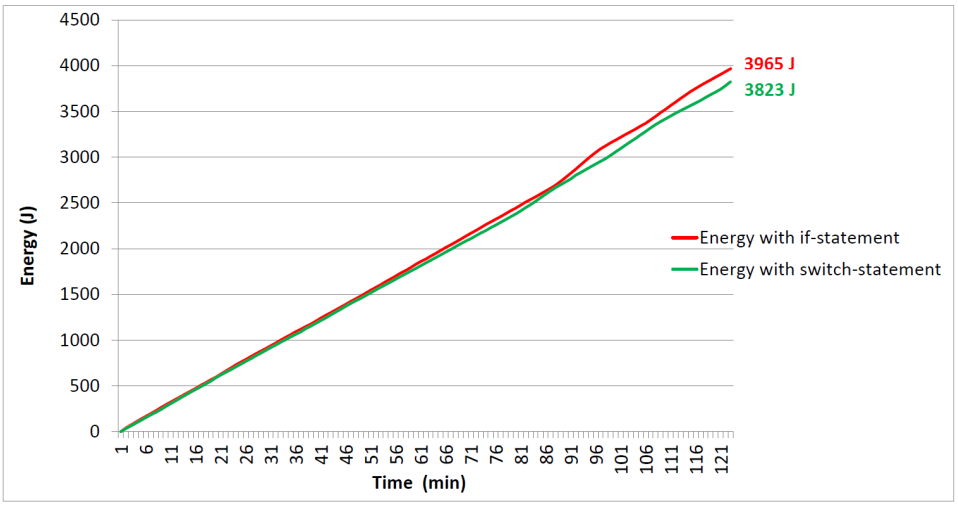
show a very minor difference between `if` and `switch`, and hence, it can now be said that both statements consume the same energy.

The *file-based measurement* in Figure 4.14a shows a difference of 142 J in which the application with `if`-statements need more energy. However, the values of the first part of the measurement are similar for both statements, the second part shows the small difference. The consumed energy for `if`-statements amounts 3965 J and for `switch`-statements it amounts 3823 J within two hours.

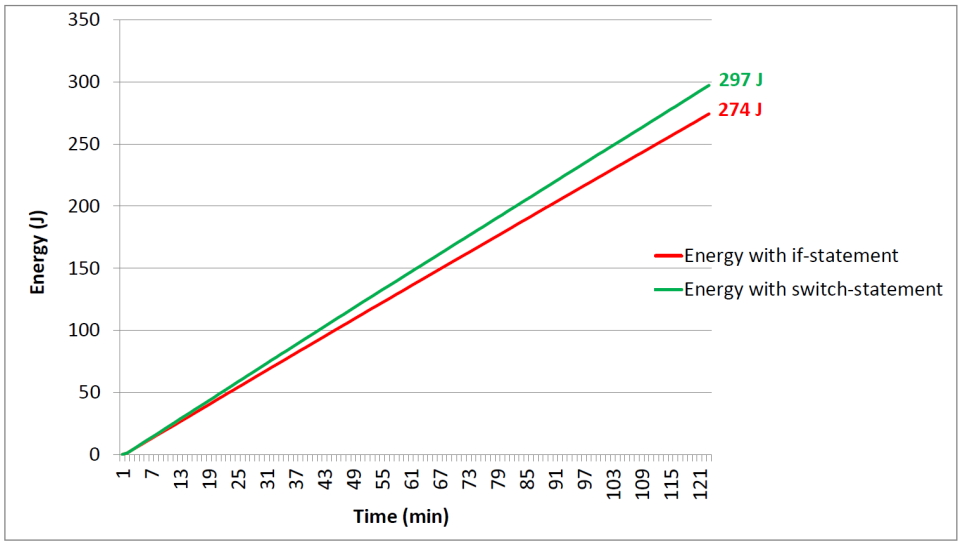
In Figure 4.14b the *Energy Profiling* represents a small difference of 23 J, but in this case, the application with `switch`-statements need more energy. The variant with `switch`-statement consumes 297 J and with `if`-statement it consumes 274 J. The difference to the other two measurements arises from the changed HTC Power Profile (cf. Section 2.5.3).

The *delta-B measurement* in Figure 4.14c shows a difference of 386 J, but here the application with `if`-statements consumes more energy. The energy consumption for the variant with `if`-statements amounts 3768 J and for `switch`-statements it amounts 3382 J.

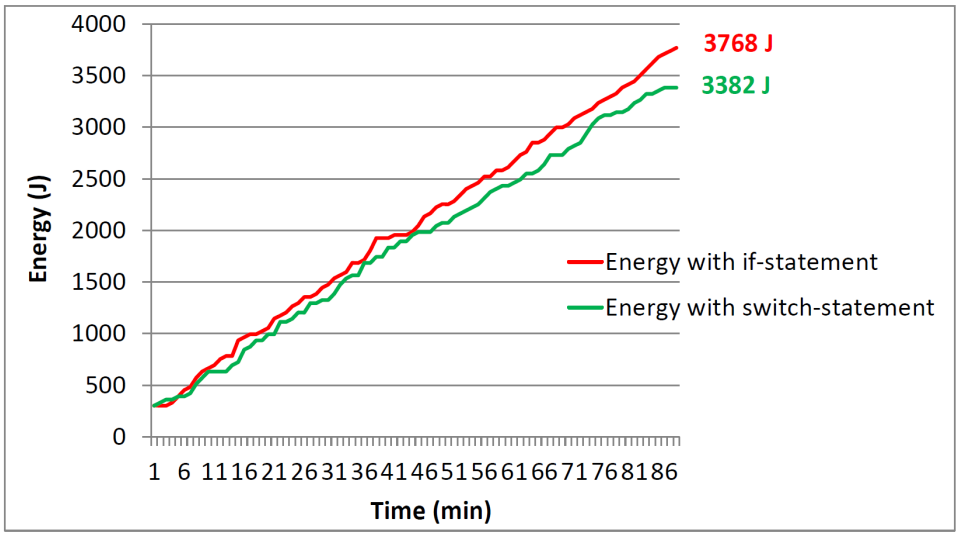
The measurement results show that no major difference between `if`- and `switch`-statements according to the energy consumption exist in this use case. Further use cases exist to proof the energy consumption of `if`- and `switch`-statements, e.g. running an application with a million of `if`s or `switch`s without a timer which influences the CPU. *Energy Profiling* in Figure 4.14b depicts that `switch`-statements consumes a little more energy. The other two measurements show the contrary. But, the difference is so small that the effort is too high to swap `if` to `switch` or otherwise. Generally, the energy consumption must be tested for several cases, maybe `integer` are more energy-efficient for `switch`-statements and `Strings` for `if`-statements, because `switch`-statements map all possible data types on integers also `Strings` [Ull11, ch. 2.6.4].



(a) File-based Measurement



(b) Energy Profiling



(c) Delta-B Measurement

Figure 4.14: Measurement of TreeGenerator with if- and switch-statement

4.4 Backlight

Name: *Backlight*

Definition: *Backlight* means the background color of an application. For different screen technologies (Super LCD and Super AMOLED) the energy consumption could variate for several background colors [CCMF13].

Motivation: Super AMOLED (Active-Matrix Organic Light Emitting Diode) screens have begun to replace LCD (Liquid Crystal Display) screens in smartphones, because AMOLED screens offer a better quality and higher energy efficiency [CCMF13, p. 1]. The energy-efficiency is realized by its unique lighting technique, i.e. one pixel is subdivided into three sub-pixel which represents the color red, green, and blue. The energy consumption of each pixel is dependent on the displayed color, i.e. more intensive colors consume more energy [CCMF13, p. 3]. In addition, each pixel can be switched on or off [Ste12]. A LCD screen always needs backlight to display several colors which are generated by each pixel separately [CSC02, p. 113]. For one LCD screen the energy consumption of the several colors is presented [CSC02, p. 114] and it shows that a black background consumes more energy than a white background. In this master's thesis the HTC and S4 are used to validate this Energy Refactoring, the S4 has a Full HD Super AMOLED screen and the HTC has a Super LCD 2 screen (cf. Section 2.4).

Constraints: The motivation also contains the constraint that several devices use different screen technologies, at which they consume different energy for several colors. Hence, applications must be adapted according to hardware information on the screen. Hence, the Energy Refactoring can base on a strategy pattern [**?**] which decides at runtime which screen is built-in to chose the right Energy Refactoring.

Example: Android applications define the background in `activity_main.xml` where the complete layout of it is described. In Figure 4.15 a small part of this XML is shown. In this case, the background color is defined in line 3 `android:background="@color/black`. The color type `black` is assigned to the attribute `@color`. In another XML (strings.xml) the color type is defined which uses the RGB schema. Further components for the layout are defined in this file, e.g. lines 5–10 show a text filed which is displayed during applications runtime. For this text field several layout parameters are determined, e.g. text size, style, and position. But they are not relevant for this Energy Refactoring.

```
1  <RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
2     android:background="@color/black"
3     [...]>
4     <TextView
5         android:id="@+id/time"
6         android:textSize="18sp"
7         android:textStyle="bold"
8     [...]>
9     [...]
10 </RelativeLayout>
```

Figure 4.15: Backlight in activity_main.xml

Analysis: Information of applications layout are saved in the auto-generated class `R`. If *Backlight* can be detected by the TGraph approach (cf. Section 2.2), the needed information are saved in this class. After analyzing the generated TGraph, it is not possible to get information about the background color of an application because it is only saved in `activity_main.xml` and it is not assigned into the auto-generated class `R`. In `R` several colors are defined which can be used for the background, also some layout information are saved, e.g. horizontal and vertical margins. Hence, this Energy Refactoring cannot be detected by this approach because all information and hints to it are in an XML file which is not considered in the Java TGraph approach in Section 2.2. This is not a general problem of the TGraph approach because it is possible to extend the presented Java meta model in Figure 2.4, so that it contains Android-specific parts which can be represented in TGraphs of parsed Android applications. However, it can be checked whether the color black is used for the application, like the query in Figure 4.16 shows. If black is not used, the Energy Refactoring for AMOLED screens could exist.

```
1  from color : V{frontend.java.DataObject}, class : V{frontend.java.Class}
2  with color.name = \"black\" and class.name = \"color\" and color <--{frontend.java.
       HasField} class
3  report color
4  end
```

Figure 4.16: Backlight Analyze

The query returns one node: `color`. This node contains one value, when the color black is defined in class `Color`, which is queried through `color <-{frontend.java.HasField} class`.

Restructuring: If this Energy Refactoring exists, a new color type must be saved into `strings.xml` and assigned to the attribute `@color` in `activity_main.xml`. In this master's thesis only Java files are considered by the TGraph approach in Section 2.2, hence, this Energy Refactoring cannot be executed through the functionality of JGraLab when only Java files are parsed into TGraphs. Therefor, *Backlight* must be done manually.

Evaluation: The evaluation of this Energy Refactoring is done on two several mobile devices, HTC and S4. For the energy measurements, the application TreeGenerator is modified, so that more pixel displays the black or white background otherwise it is covered by the pictures for the trees and advertisements (cf. Section 2.6). Firstly, the evaluation for the HTC with the LCD screen is executed. In Table 4.5 the mobile device settings for the HTC are depicted. It can be seen that all components expect for the screen are switched off. In Figure 4.17 the results for the measurement are illustrated.

Screen Settings	screen	on
	brightness	lowest
	auto rotation	off
Notification Settings	notifications	off
Applications settings	background applications	Andromedar
	power saver	off
Wireless & network settings	mobile data	off
	blue-tooth	off
	GPS	off
	NFC	off
	Wi-Fi	off
Gesture settings	three finger gestures	off
Miscellaneous	sim card	not installed
	other applications	not installed

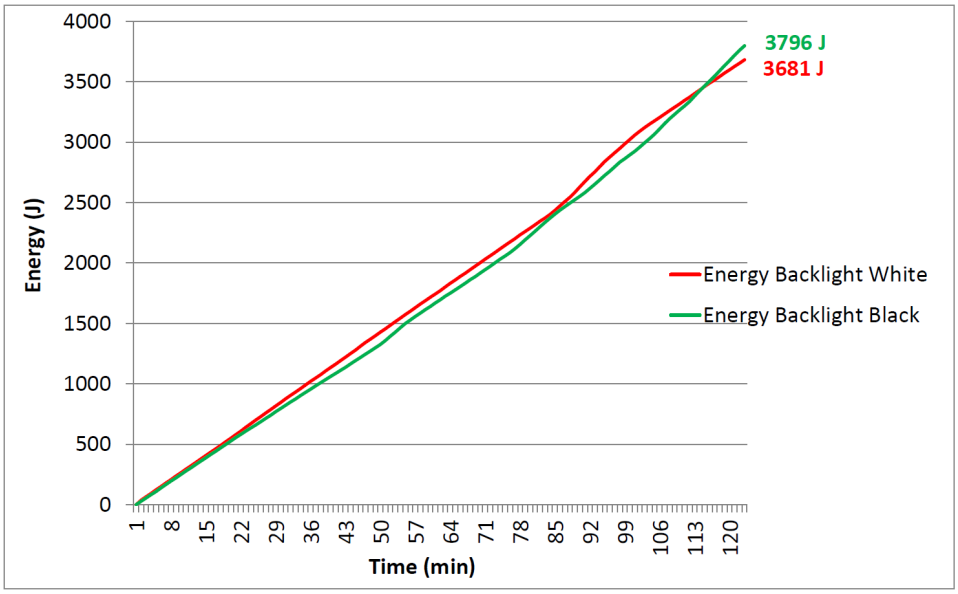
Table 4.5: Mobile Device Settings for Backlight on the HTC

The graph in Figure 4.17a shows the *file-based measurement* for a white and black background color of the application TreeGenerator. The energy consumption for a white background amounts 3681 J and for a black background 3796 J. The difference is 115 J, and hence, it is very little.

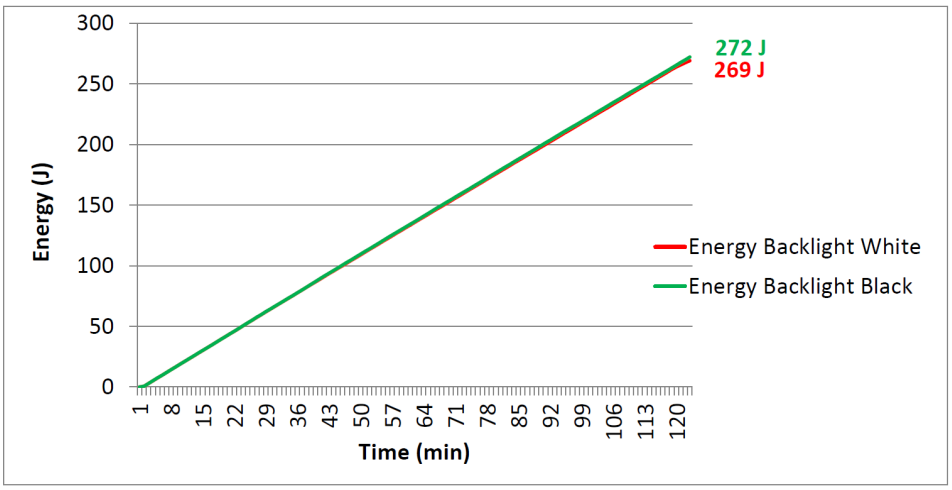
The results for *Backlight* with the *Energy Profiling* are depicted in Figure 4.17b. TreeGenerator with white background consumes 272 J and with black background it consumes 269 J. Hence, the difference amounts 3 J, and hence, it does not constitutes a significant difference. The difference to the other two measurement techniques arises from the changed HTC Power Profile (cf. Section 2.5.3).

In Figure 4.17c the results for the *delta-B measurement* are shown. This measurement shows as well as the other two measurements that TreeGenerator with a black background consumes a little more energy than with a white background. The difference amounts 102 J.

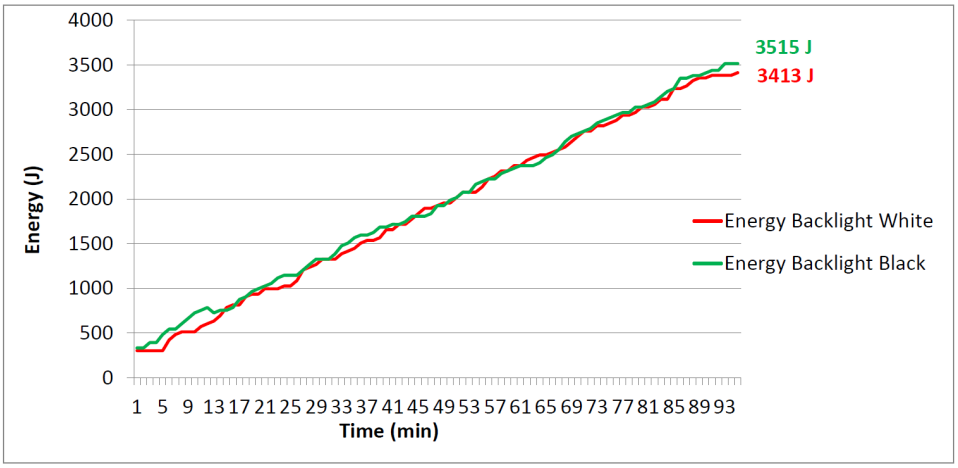
All three measurements show that a black background for the HTC consumes a little more energy than a white background. This agrees with the measurement results in Choi et. al [CSC02, p. 114]. Hence, for saving energy on LCD screen it makes sense to use a white background color into applications.



(a) File-based Measurement



(b) Energy Profiling



(c) Delta-B Measurement

Figure 4.17: Measurement of TreeGenerator with white and black background (HTC)

The second validation with the S4 are illustrated in Figure 4.18 and contains only two measurement techniques *delta-B measurement* and *Energy Profiling* (cf. Section 2.5) because the *file-based measurement* is not support for Samsung devices. The mobile device settings in Table 4.6 differs from the setting in Table 4.5. For this measurement, the screen brightness is set to the highest level to improve the comparability with the measurements in Chen et. al [CCMF13, p. 2] because they also tested mobile devices with AMOLED screen and similar settings. The list of background applications is longer than for the HTC because some applications cannot stop completely without data loss (Google Play Music, Uhr, and AntiVirus). The S4 screen cannot be permanently on because it is not destined by Samsung. Hence, the application *Screen On Toggler* is installed which stops the screen stand-by and lets its on which is necessary to run the application TreeGenerator. The Offline modus is chosen to stop all notifications. Also, further applications are installed but they do not run during the measurements.

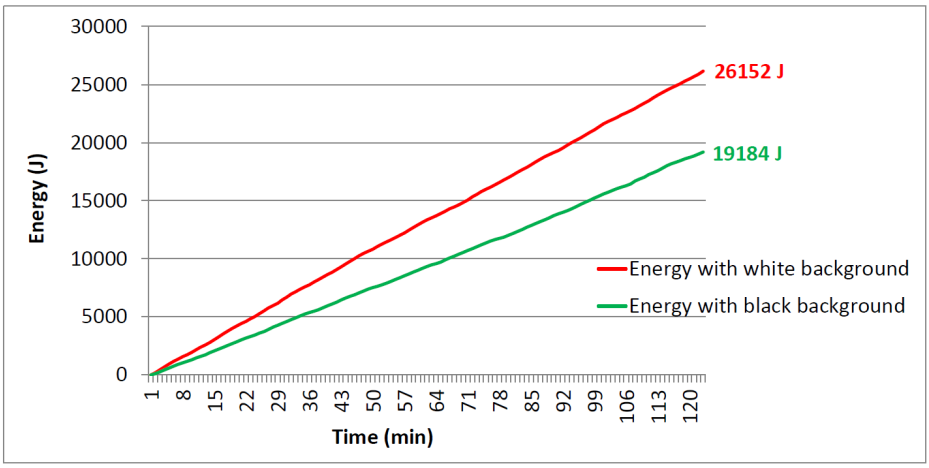
Screen Settings	screen	on
	brightness	highest
	auto rotation	off
Notification Settings	notifications	off
Applications settings	background applications	Andromedar, Screen On Toggler, Offline modus, AntiVirus, Google Play Music
	power saver	off
Wireless & network settings	mobile data	off
	blue-tooth	off
	GPS	off
	NFC	off
	Wi-Fi	off
Gesture settings	three finger gestures	off
Miscellaneous	sim card	not installed
	other applications	installed

Table 4.6: Mobile Device Settings for Backlight on the S4

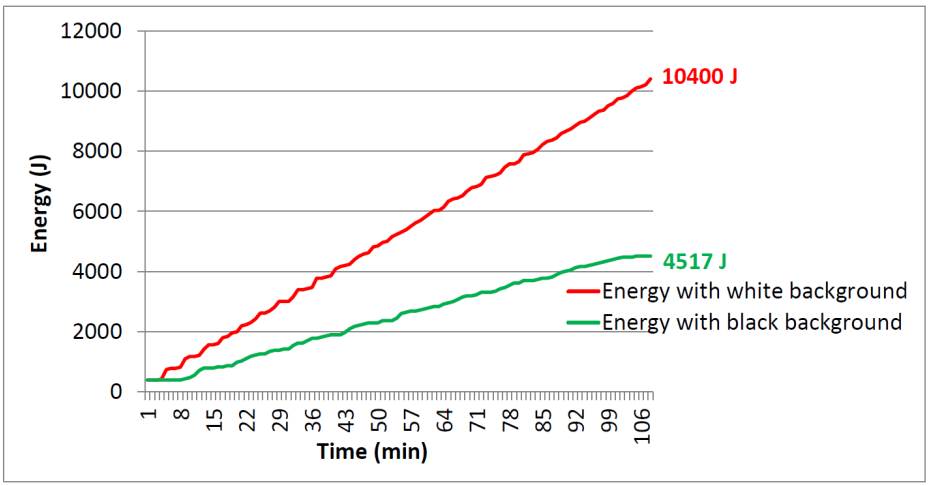
Figure 4.18a shows the results of the energy measurement with *Energy Profiling*. It shows a significant difference between TreeGenerator with a white and black background. For TreeGenerator with white background 26152 J are consumed and 19184 J for TreeGenerator with black background. Hence, the difference amounts 6968 J. This is an energy saving of approximately 30 % when the background color for a Super AMOLED screen is changed.

The *delta-B measurement* is depicted in Figure 4.18b. This measurement shows a deeper differences than *Energy Profiling*. Here, the application consumes 10400 J with a white background and 4517 J with a black background. The difference amounts 5883 J which presents an energy saving of approximately 60 %.

These energy measurements for HTC and S4 show a hardware-dependently Energy Code Smell which can be removed by a hardware-independently restructuring because the source code is for both mobile devices the same. Hence, if the screen technology for mobile devices is known, a restructuring can be done which is aligned to the technology, e.g. if a AMOLED screen is used, it should be checked whether black is used as background.



(a) Energy Profiling



(b) Delta-B

Figure 4.18: Measurement of TreeGenerator with white and black background (S4)

4.5 Data Transfer

Name: *Data Transfer*

Definition: *Data Transfer* means to load data from a server via Wi-Fi during applications runtime instead of reading-out these data from the applications storage.

Motivation: Many applications use data, such as images, videos, and sound effects. Programmers can decide where these data are stored. On the one hand, all data can be stored on applications storage (on memory card) during the installation of applications. On the other hand, all data can be stored on a server and applications are able to access these data after installing, and when an Internet access is available. The measurements in Carroll and Heiser [CH10, p. 6] recommend to save the data on memory card. They show that reading a file from memory card needs approximately 45 mW, whereas loading a file via Internet needs

approximately 710 mW. In both cases, the power for CPU and RAM are nearly the same, but they tested files with different file sizes (in proportion the Wi-Fi also consumes more energy) [CH10, p. 6]. This measurement is hardware-dependent and the measurements by Carroll and Heiser use the Openmoko [CH10, p. 2]. After first measurements, a further possibility exists which depicts a mix of the above described approaches. Data can be loaded from a server, and thereupon, these data can be stored in applications cache. While applications cache is not cleared, the data do not have to be loaded again from the server which saves energy. Hence, this two variants are considered.

Constraints: Saving data on applications storage needs storage space on mobile devices while an application is installed and the whole time when it is installed. So, users should known how big the storage of their memory card is, when they install many applications which save all data on their storage. But, loading all data via Internet generates a high data traffic each time when applications run. And if no Internet access available, applications will not run, except data which are stored in cache. If the data access of an application should be changed from loading data from server to store data on memory a programmer is necessary who loads all data and creates a folder for them within the application. Also, many code parts must be changed what is very difficult for an automatic transformation (cf. Figure 4.19). Also, for both approaches an API is needed which must be known to detect *Data Transfer*.

Example: How the source code of an Android application can look like is demonstrated in Figure 4.19. It shows on the left side the code for loading data from server, and on the right side data are read from memory card. To load data from a server an additional free API `com.androidquery.AQuery` [Andc] (line 4) is needed to request the picture on the server (lines 8 and 11). Also, it must be decided whether the cache is used or not, and an alternative picture can be chosen when the server does not answer, in this case, *ic_launcher* is chosen which is the applications symbol. The decision concerning the usage of the cache is the second form of this Energy Refactoring which was named in the motivation part. The two booleans in line 8 are set to `false`, hence cache is not used, when these values set to `true` cache is used during runtime. This is not depicted in Figure 4.19, but it is considered in the analysis part of this Energy Refactoring. The code on the right side in Figure 4.19 demonstrates the same functionality but the data are read-out from the application storage instead of loading it from the Internet. Therefor, the API `android.content.res.Resources` (line 4) is used as a standard Android API. Data are read by this API (lines 8 and 12) and then it is binded on `pic` which is a component of the Android layout (lines 9 and 13).

```
1  @Override
2  public void run() {
3     i = (int) (Math.random() * 51);
4     AQuery aq = new AQuery(pic);
5     //tree list
6     if(i == 1) {
7        value.setText("Nikko-Tanne");
8        aq.id(pic).image("http://
             mgottschalk.eu/img/bilder/
             nikko.jpg", false, false, 200,
              R.drawable.ic_launcher);
9     } else if(i == 2){
10       value.setText("Riesen-Tanne");
11       aq.id(pic).image("http://
             mgottschalk.eu/img/bilder/
             riesentanne.jpg", false, false
             , 200, R.drawable.ic_launcher)
             ;
12    }
13    [...]
14 }
```

```
1  @Override
2  public void run() {
3     i = (int) (Math.random() * 51);
4     Resources res = getResources();
5     //tree list
6     if(i == 1) {
7        value.setText("Nikko-Tanne");
8        Drawable picture = res.getDrawable
             (R.drawable.nikko);
9        pic.setImageDrawable(picture);
10    } else if(i == 2){
11       value.setText("Riesen-Tanne");
12       Drawable picture = res.getDrawable
             (R.drawable.riesentanne);
13       pic.setImageDrawable(picture);
14    }
15    [...]
16 }
```

Figure 4.19: Example for DataTransfer

Analysis: The analysis shows the case, that data are loaded from Internet without using cache, and hence, the object is to use applications cache. Firstly, to detect this Energy Code Smell the API `com.androidquery.AQuery` should be identified. Secondly, if this API is detected, the other code parts can be seek. The query for the import statement is nearly the same like the import query for advertisements in Figure 4.3, and hence, it is not explained any further. In Figure 4.20 the query to identify the method of `AQuery` for loading data from a server is depicted. Within this method two literals are used to decide whether the devices or the applications cache should be used during applications runtime.

```
1  from cache : V{frontend.java.Literal}, block : V{frontend.java.Block}, image : V{
       frontend.java.Access}, aquery : V{frontend.java.Class}
2  with cache.value = "false" and aquery.name = "AQuery" and
       cache <--{frontend.java.HasOperand} <--{frontend.java.HasExpression} <--{frontend.
       java.BlockContainsStatement} block
       -->{frontend.java.BlockContainsStatement} -->{frontend.java.HasExpression} (-->{
       frontend.java.HasOperand})* -->{frontend.java.HasExpressionType} -->{frontend.java.
       HasNamedType} -->{frontend.java.HasExpressionType} aquery and
       image <--{frontend.java.HasOperand} <--{frontend.java.HasOperand} <--{frontend.java.
       HasExpression} <--{frontend.java.BlockContainsStatement} block
3  report cache
4  end
```

Figure 4.20: DataTransfer Analysis

The query seeks for literals which have the value `false` and belongs to the same `Block` as the method `image`, which is called by an object of the class `AQuery`. Each `block` of an application is checked whether it includes `cache` with the value `false` (2 — 3 lines within the `with` part). Thereafter, it is checked whether the `block` contains an object of the class `AQuery` (4 – 6 lines). At least, it is tested whether `image` is called within this block (7 – 8 lines). All matching literals are returned and can be restructured.

The analysis for the other variant of this Energy Refactoring (storing data on the mobile device instead loading from Internet) would have a different query. The first query for the import statements would be the same, and thereafter, it would only seek for the method call `image` to know which part of the code must be replaced. The replacement could be with the Android API `android.content.res.Resource` which is demonstrated in Figure 4.19 by the example. This Energy Refactoring is not realized, and hence, it is not considered any further.

Restructuring: In this master's thesis the restructuring considers only the case, that the applications cache is not used which should be changed. If data are loaded from server, it can be validated whether data are stored in cache or not. If these data are not stored, the boolean values in lines 8 and 11 of the example in Figure 4.19 must be changed to `true`. This restructuring is illustrated in Figure 4.21. The results of the query in Figure 4.20 are saved as `jValueList`, it contains all detected literals. Figure 4.21 shows that all literals are checked again in line 2 before the value of the literals is changed in line 3.

```
1  for(int i = 0; i < jValueList.size(); i++){
2    if(jValueList.get(i).toVertex().getAttribute("value").toString().equals("false")){
3      jValueList().get(i).toVertex().setAttribute("value", "true")
4    }
5  }
```

Figure 4.21: DataTransfer Restructuring

Evaluation: The energy measurement for this Energy Refactoring contains four several measurements: data transfer with and without using cache (red and green lines), without data transfer with and without Wi-Fi connection (blue and violet lines), which are illustrated in Figure 4.22. For these measurements, the mobile device settings are represented in Table 4.7. The setting for Wi-Fi is *on / off* to represent all settings for all measurements. The first three measurements are with a switched on Wi-Fi and the last one is done with a switched off Wi-Fi.

The *file-based measurement* in Figure 4.22a illustrates that TreeGenerator without using applications cache consumes more energy than the other three possibilities. It amounts 4232 J, and hence, 40 J more than TreeGenerator without data transfer. If TreeGenerator uses its cache, the energy consumption will drop to 4008 J which depicts a difference of 224 J. TreeGenerator without data transfer and a switched off Wi-Fi consumes least energy with 3864 J, but this type of TreeGenerator is only sensible when no other applications run which needed the Wi-Fi. The differences between these measurements are small but show a trend, that using applications cache is sensible when Wi-Fi is used.

The graph in Figure 4.22b shows the result of the *Energy Profiling*. How all other measurements with *Energy Profiling* the results are much lower than the results of the other two measurement techniques. The reason is the changed power profile of the mobile device which is described in Section 2.5.3. The measurement for TreeGenerator with data transfer

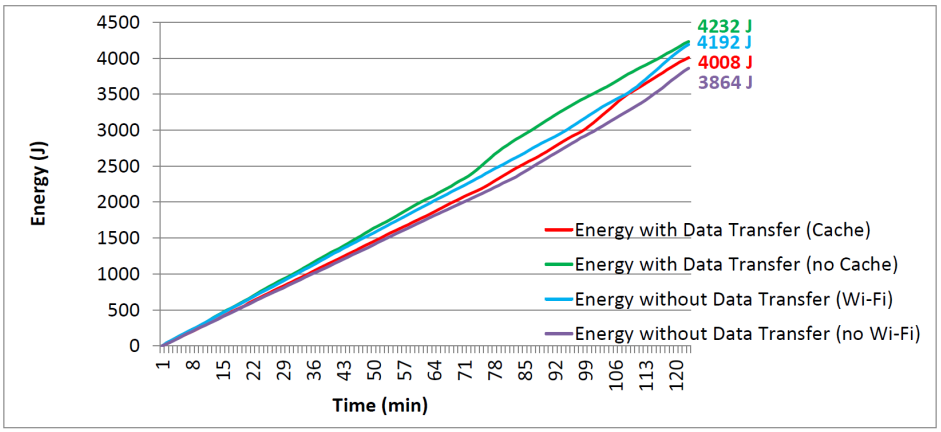
Screen Settings	screen	on
	brightness	lowest
	auto rotation	off
Notification Settings	notifications	off
Applications settings	background applications	Andromedar
	power saver	off
Wireless & network settings	mobile data	off
	blue-tooth	off
	GPS	off
	NFC	off
	Wi-Fi	on / off
Gesture settings	three finger gestures	off
Miscellaneous	sim card	not installed
	other applications	not installed

Table 4.7: Mobile Device Settings for Data Transfer

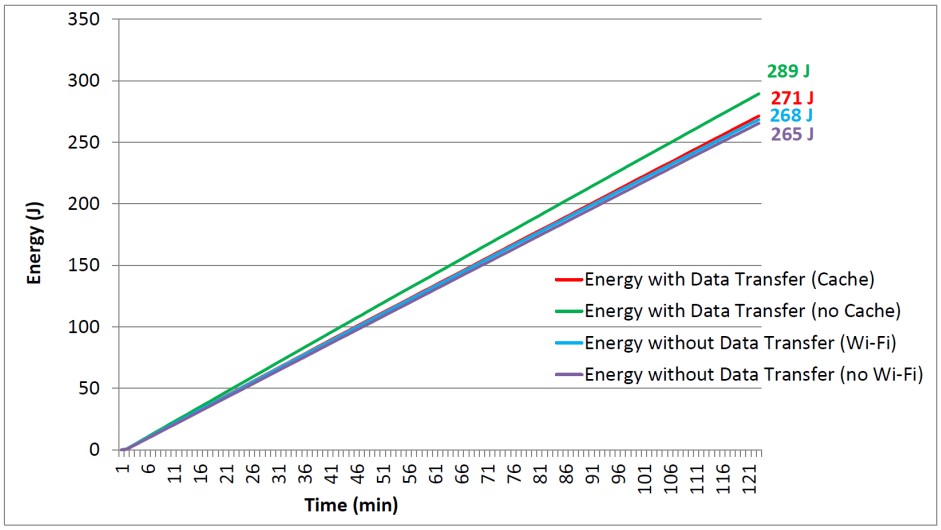
and without using cache consumes 289 J. The other three measurements consumes nearly the same and their energy consumption amounts approximately 268 J. This amounts a difference of approximately 21 J.

The *delta-B measurement* in Figure 4.22c illustrates the same trend like the *file-based measurement* in Figure 4.22a. TreeGenerator with data transfer and without cache consumes 4064 J. The variant without data transfer but with a switched on Wi-Fi consumes 4000 J, and therefore, the difference amounts 64 J to the variant with the highest energy consumption. TreeGenerator with data transfer and cache consumes 3787 J, and hence, the difference amounts 277 J to the first variant. However, the variant without Wi-Fi consumes at least energy with 3497 J.

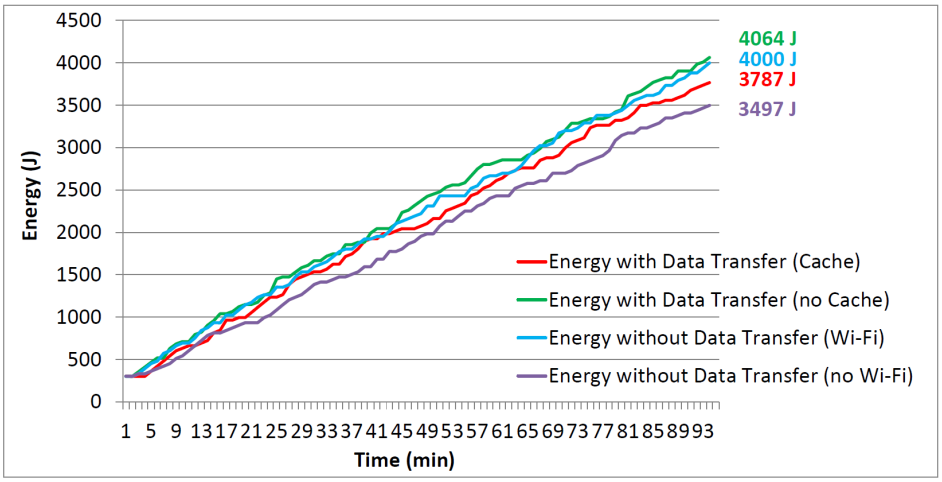
These measurements show that the usage of applications cache is a sensible way to save energy. In most cases, the Wi-Fi component of the mobile device is switched on, so that not much energy is saved when data are stored within the application. Furthermore, mobile devices storage can be saved when the variant with data transfer and with using applications cache is taken. It needs 1.31 MB for the application and approximately 636 KB for the cache. If TreeGenerate does not use data transfer, it needs 2.45 MB storage for the whole application. This Energy Refactoring is only realized and checked for the usage of the AQuery and the Android Resource API. Other APIs could have a different energy consumption, and hence, this Energy Refactoring is software-dependent.



(a) File-based Measurement



(b) Energy Profiling



(c) Delta-B Measurement

Figure 4.22: Measurement of TreeGenerator with and without Data Transfer

5 Further Energy Refactorings

In this part an overview of more Energy Refactorings is given. These Energy Refactorings are described in the same way as before but without an example, implementation, and evaluation. Hence, the Energy Refactorings *Using expensive Resources*, *Dead Code*, *Replace Sorting Algorithm*, *Loop Bug*, *In-Line Method*, *Wake Lock*, *Fowlers' refactorings* [FBB+02], and *Design Pattern* [GHJV93] are represented theoretically. This listing constitutes an outlook for further theses and publications.

5.1 Using expensive Resources

Name: *Using expensive Resources*

Definition: *Using expensive Resources* describes the possibility to swap energy expensive hardware resources for applications against energy-efficient alternatives [SH12, p. 1].

Motivation: Applications often use hardware resources, such as GPS to localize mobile devices, for tasks which can be done by other resource. This approach was validated and shows energy savings by replacing resources in Schirmer and Höpfner [SH12]. They show that the usage of cell towers and Wi-Fi consumes less energy than GPS [SH12, p. 5]. Therefore, the basic energy consumption of a mobile device is compared with a GPS run and a cell tower and Wi-Fi run. The GPS run increases the energy consumption by 74.56 %, and the cell tower and Wi-Fi run increases the energy consumption by 1.5 %. This example shows that *Using expensive Resources* is a further Energy Refactoring.

Constraints: Some hardware resources are more precise than other, e.g. GPS can be replaced with cell tower and Wi-Fi, but they are less precise than GPS [SH12, p. 1]. Another constraint is the availableness of data, e.g. GPS satellite connection or cell tower connection. If the cell tower connection is not available, the GPS connection must be used.

Analysis: Known hardware resources which consume much energy must be seek to replace them. Therefore, vendor information or hardware measurements are needed to decide which resources consume more energy than other resources. In addition, resources must be achieved same tasks.

Restructuring: In this case, the transformation replaces one hardware component against another to create an more energy-efficient application. Due to the difference of supplied data, e.g. GPS sensors give coordinates and the connection with a cell tower gives an id, the id supplies coordinates via Wi-Fi by read-out a database which stores all coordinates for cell towers [SH12, p. 1].

5.2 Dead Code

Name: *Dead Code*

Definition: *Dead Code* are code parts which are never reached during applications runtime. But it is also loaded into memory, and thus, it consumes energy [CGKO97].

Motivation: Sometimes, code parts are implemented which are not used at the end. In this case, the Java compiler will perform dead code elimination [Hav09]. Another variant of *Dead Code* is code which is invoked but whose results are immediately discarded. These type of *Dead Code* is not detected on compiler level, but it can be detected on source code level [GJJW12].

Constraints: Whether results of code parts are not used, will be detected by dynamic code analysis. Due to the dynamic code analysis, the code behavior is validated at runtime.

Analysis: The first variant is easier to detect through static code analysis, only methods which are never actually called must be developed. The second variant is more difficult to detect, here a dynamic analysis is needed to create use cases. The use cases should call all methods which are called under conditions which never arise at runtime [GJJW12].

Restructuring: If *Dead Code* of the first variant is detected, it can be deleted completely. If the second variant is detected, it must be considered, why the results are discarded. Moreover, it must be checked whether no use case exists which uses the method result.

5.3 Replace Sorting Algorithm

Name: *Replace Sorting Algorithm*

Definition: *Replace Sorting Algorithm* means to replace one sorting algorithm with another more energy-efficient sorting algorithm [BRM09].

Motivation: In Bunse et. al [BRM09] a substitution of sorting algorithm is presented and validated concerning their energy consumption. The validation was done for one special hardware platform which shows significant differences between the energy consumption of several sorting algorithms. In this case, *Insertionsort* is the most energy-efficient algorithm and *recursive Quicksort* is the most energy-inefficient algorithm [BRM09, p. 4]. The energy consumption of *recursive Quicksort* is approximately 6 times higher than the consumption of *Insertionsort*. This shows, that it is important to compare several sorting algorithm for different platforms to choose the most energy-efficient one for programming applications.

Constraints: As mentioned above, the energy consumption of sorting algorithms can change on different platforms, so that a validation for each device type is needed.

Analysis: To detect specific sorting algorithms, structured queries are needed which represents the structure of sorting algorithms, in which each algorithm is illustrated by at least one

query. These queries are very complex, and hence, it could be that not all sorting algorithms are identified when several ways are used to program them which cannot be all considered.

Restructuring: The restructuring contains the complete replacement of one sorting algorithm which could be better implemented by a programmer to avoid errors. Hence, the analysis can help to detect energy-inefficient source code but the restructuring should be done manually to get a better result.

5.4 Loop Bug

Name: *Loop Bug*

Definition: *Loop Bug* describes a program behavior wherein applications are repeating the same activity over and over again without reaching intended results [PCHZ11, p. 4].

Motivation: This Energy Refactoring can occur in two several ways which are described in Pathak et. al [PCHZ11]. Firstly, it is possible that a method is called again and again because it does not get any results. One reason therefore could be that a server is not acquirable so that applications stuck in this loop. Secondly, an application calls itself when it is switched off, so that it starts after ending. Both types of *Loop Bug* should be avoided to save energy within applications.

Constraints: The reachability of external services cannot be detected by static source code analysis, hence, a dynamic analysis is necessary, and even then it is rarely detected, e.g. a server failure should not take place for a long period.

Analysis: The first described case can be assumed by seeking of method calls which response an external service and does not have a security query to check its reachability. Due to this, a *Loop Bug* is prevented during runtime. The second described case can be identified by checking the method calls which are executed when applications end. For example, if one of it contains `onRestart()` or `onResume()` (cf. Section 2.3.2), a programming mistake is existent and should be removed.

Restructuring: In the first case, the security query must be added, when it is not existent. In the second case, the method call which restarts the application must be deleted.

5.5 In-Line Method

Name: *In-Line Method*

Definition: *In-Line Method* describes the exchange of a method call with its method body which increases applications performance, as the computational overhead of a method call is avoided [dSB10, p. 2].

Motivation: Da Silva et. al [dSB10] presents the idea to use *In-Line Methods* for saving energy. This was tested with three applications by an energy measurement, and all measurement results show that energy is saved when *In-Line Method* is used [dSB10, p. 3–4]. Hence, this is also a candidate for the here described Energy Refactoring Catalog.

Constraints: Using *In-Line Method* reduces readability and maintainability for programmers [dSB10, p. 2]. Also, methods which are often used while applications runtime are practical for *In-Line Method* to save energy, but therefore, a dynamic analysis is needed.

Analysis: For this Energy Code Smell short methods, such as getter and setter, are practical to replace with their method body, hence, these methods and the method caller must be detected.

Restructuring: Detected method calls are replaced with their body, and their parameter's must be checked and maybe they must be also replaced.

5.6 Wake Lock for Resources

Name: *Wake Lock for Resources*

Definition: *Wake Lock for Resources* is used to prevent the direct shutdown of hardware resources after using them to enable a fast restart [PCHZ12, p. 3].

Motivation: Resources, such as GPS and WiFi, are sometimes equipped with wake locks which prevent the direct shutdown. Due to wake locks [Andh], applications can restart faster than without it because resources are directly ready. But, it consumes more energy when resources are switched on the whole time [BBC]. Pathak et. al say that *Wake Lock for Resources* is the most prominent Energy Code Smell [PCHZ12, p. 3], because wake locks are often used by programmers to make their applications faster. The issue arises when the wake lock for components is not released even after their job is completed, e.g. a maximum wake lock time for components must be defined, in which the energy consumption and applications restart speed are considered.

Constraints: For applications which often switch between fore- and background, wake locks are useful to get a smooth flow between the states (cf. Android Life Cycle in Figure 2.7). Hence, someone must check whether wake locks are useful or only energy consumer.

Analysis: First, wake locks must be identified, e.g. Android uses the method `acquire()` from `PowerManager.WakeLock`. Second, it must be checked whether this wake lock is necessary for applications functionality or not. This decision should be made with the help of dynamic analyses.

Restructuring: If the wake lock is too long or not necessary for applications functionality, the wake lock can be deleted or set to null, so that it takes no time and resources are directly switched off.

5.7 Fowlers' Refactorings

Name: *Fowlers' Refactorings*

Definition: *Fowlers' Refactorings* contain all 72 code refactorings which are described in Fowler et. al [FBB+02]. These refactorings are actually used to create a better readability and maintainability for source code than before [FBB+02, p. xvi].

Motivation: One of these Code Smells and their refactoring is *In-Line Method* which is described in Section 5.5 as further Energy Refactoring. *In-Line Method* is listed as Energy Code Smell because it was tested and it shows energy savings [dSB10, p. 3–4]. The further code refactorings of Fowler et. al are not validated yet. But, it is possible that these refactorings also save energy. Hence, *Fowlers' Refactorings* should be validated concerning their energy consumption to add them to the Energy Refactoring Catalog, like *In-Line Method*.

Constraints: Here no constraints exist because no specific Energy Code Smell is described.

Analysis: The detection of *Fowlers' Refactorings* is difficult because no special import or method call within applications is seek, but a special sequence of calls for each code refactoring, e.g. *Extract Method* [FBB+02, p. 110] seeks for methods which are too long to understand. This analysis is very unspecific, and hence, difficult to realize with the TGraph approach (cf. Section 2.2).

Restructuring: How the restructuring looks like is dependent on the defined Energy Code Smell. For *Extract Method*, a new method must be created and code must be shifted from the old method to the new, and a method call for the new must be added to the old.

5.8 Design Pattern

Name: *Design Pattern*

Definition: *Design Patterns* are defined by Gamma et. al [GHJV93] and they describe solutions for special problems which can be applied a million times over within several applications, i.e. they provide a special vocabulary to reduce software complexity [BS13, p. 1].

Motivation: In Bunse and Stiemer [BS13] some of these patterns are analyzed concerning their energy use. The results show that differences between the implementation with and without *Design Pattern* exist, e.g. the *Decorator* pattern consumes significantly more energy than an alternative programming. Due to this, it is useful to prove further *Design Pattern* to include this in the decision whether *Design Patterns* are used or not.

Constraints: Here no constraints exist because no specific Energy Code Smell is described.

Analysis: The *Design Patterns* are described by Gamma et. al [GHJV93], hence, analyses can be created on the base of it. The detection of these patterns is also difficult, like the detection of *Fowlers' Refactorings*, because a sequence of classes, methods, and calls must be seek, and no import or special method name is available.

Restructuring: If a *Design Pattern* is energy-inefficient, an alternative implementation must be created and realized by a transformation.

6 Implementation of Energy Refactorings

The implementation for Energy Refactorings is done as an Ecplise Project, named *EnergyRefactoring*, with the JGraLab API (cf. Section 2.2). In this section, the execution of the five Energy Refactorings are explained to enable their execution for several applications. Therefor, the Android applications must be parsed into TGraphs which are represented by tg files. The parsing part is done with the parser by pro et con, used in the SOAMIG project [FWE+12], so that the return of the parser can be used for the transformation. First, the software which is used to make the transformation is described. Second, a class diagram shows the structure of the code. Finally, the output of the implementation is illustrated and explained.

6.1 Software

As mentioned above the Energy Refactorings are implemented as a Java project implemented with the Eclipse IDE. Eclipse is used in the version *Indigo Service Release 2*. Before the refactorings can be started, the applications code must be parsed as described in Section 1.2. This is made with the pro et con parser [FWE+12] which generates XML files which can be transformed to TGraphs with the JGraLab API. On TGraphs refactorings are executed, and afterward, TGraphs are saved as XML. The transformed XML files after refactoring are parsed back in Java code, also with the pro et con parser. It should be considered that the validated applications based on the API level 15 which uses Java 6, hence, Java 6 or higher is needed to build the restructured applications. Otherwise, no further software is needed to execute the described Energy Refactorings, and hence, to generate more energy-efficient code.

6.2 Class Diagram of EnergyRefactoring

The class diagram in Figure 6.1 shows the structure of the Eclipse project EnergyRefactoring. It illustrates that the code is divided into three parts: `Main`, `Analyzing`, and `Restructuring`. However, attributes and return values are not displayed, it shows only the structure and methods of EnergyRefactoring.

The `Main` part includes the class `EnergyRefactoring` which represents the main class of the project. This class must be run to execute all Energy Refactorings for one application which occurs as XML file. If `EnergyRefactoring` is executed, the user is prompted to select the XML file which should be analyzed and restructured and a name for the restructured XML file. In addition, the class `Helper` is implemented in `Main`, it contains all methods which are not directly needed for the refactorings, e.g. methods for the transformation from XML to tg and backwards.

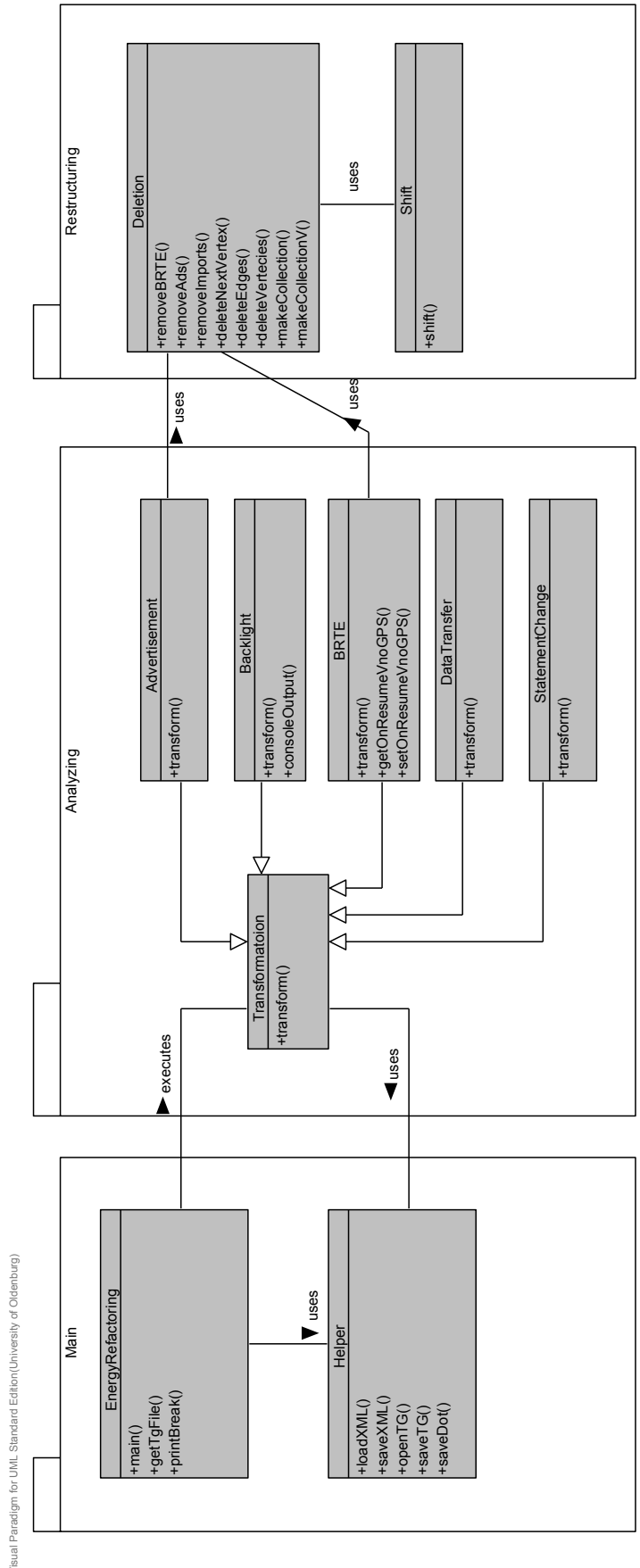


Figure 6.1: Class Diagram for EnergyRefactoring

The `Analyzing` part includes the five analyzes for the five different Energy Refactorings in Section 4. All these classes get relations, attributes, and methods from the class `Transformation` which is connected to the classes `EnergyRefactoring` and `Helper`.

The `Restructuring` part contains the two classes `Deletion` and `Shift`. These both are needed for the analyzes in *Advertisement* and *BRTE*. For both restructurings nodes or edges must be deleted, and `BRTE` also needs `Shift`. The restructuring of the analysis in class `DataTransfer` is realized in the analyzing part because it is only one line long. For the remaining analyzes no restructuring is implemented, but after their analyzes the user gets information whether and maybe where an Energy Code Smell exists, so that, the application can be restructured manual.

6.3 Output of EnergyRefactoring

The output of the EnergyRefactoring is illustrated in the screen shot in Figure 6.2. It shows the eclipse console output. All Energy Refactorings are separated through a line and their name. At the beginning the name of the XML file which should be refactored and the name of the outcome document are defined by the user through simple console inputs. Firstly, *Third-Party Advertisement* is executed, hence, its name is printed, hereon, the number of processing elements is given 24013. GpsPrint contains advertisements, hence, *"Ads were deleted"* is printed and the number of saving elements is 23890, 123 elements were deleted. The second running Energy Refactoring is *Statement Change*, it shows, how many `if`-statements with more than three `else`-statements exit, and in addition, it shows in which method and class the `if`-statements are. In this case, 19 `if`-statements are in the method `run`, which is a part of the class `ValueTask2`. Thirdly, *Backlight* is executed, it only shows whether the colors black and white are used. Hence, the user can imagine whether it includes *Backlight* as Energy Refactoring. In this case, GpsPrint does not use these colors in this form. Fourthly, *Data Transfer* is executed and it shows in the end whether the cache is used for data within the application or not. If the cache is not used, it will be transformed so that it is used. The last running Energy Refactoring is *Binding Resources Too Early*. GpsPrint includes this Energy Refactoring, and hence, the output *"Binding Resources too early is removed"* is given, and 23889 elements were saved.

After executing these Energy Refactorings, the number of elements in GpsPrint have been reduced by 124 elements. The same execution for TreeGenerator with advertisements has a reduction of 173 elements (cf. Appendix D).

6.4 Extensions of EnergyRefactoring

In this section it is brief described how the project EnergyRefactoring can be extended. Firstly, the existing Energy Refactorings can be extended through further queries to detect more Energy Code Smells, e.g. detecting further APIs for advertisements. Therefore, a GReQL query must be integrated within a class in the package `Analyzing`. Also, the

query must be implemented within the method `transform()` and should save its result in a JValue object to use existing restructurings. Secondly, a complete Energy Refactoring can be integrated by creating a new class in `Analyzing` with a query to detect further Energy Code Smells and a new class in `Restructuring` to remove the Energy Code Smell in case no suitable for restructuring exist. In addition, to execute the new Energy Refactoring a call in the class `EnergyRefactoring` in the package `Main` must be added.

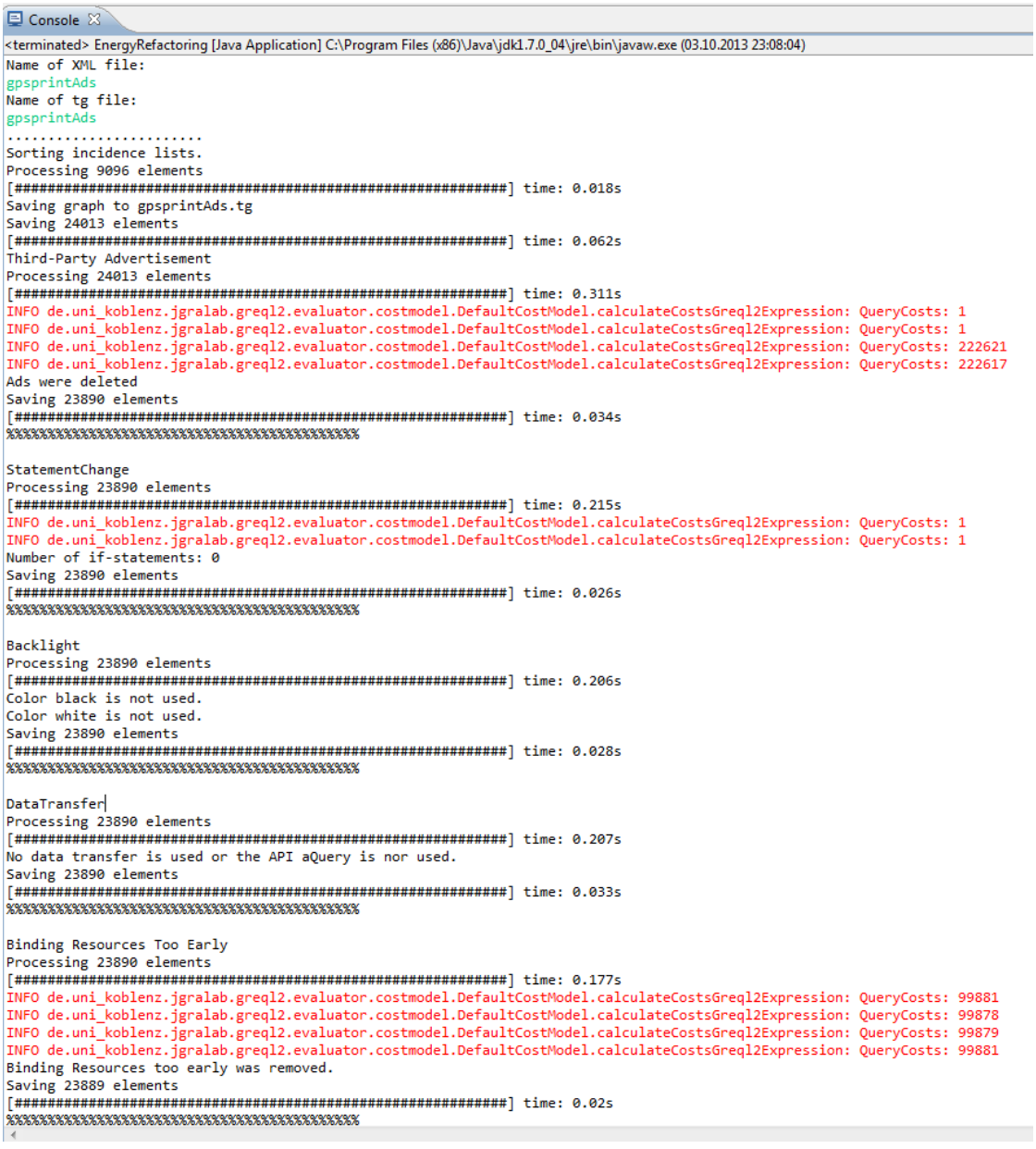


Figure 6.2: Screen shot of the EnergyRefactoring results

7 Conclusion

This chapter concludes this master's thesis with a summary of the evolution results of the Energy Refactorings in Chapter 4. In addition, general awarenesses are described in form of a *Lesson Learned* section. Moreover, an overview about completed work packages are given to show whether the objectives of this work are reached. On the base of the measurement results a general conclusion is written. In addition, an outlook is given to name further possibilities to save energy within applications. Finally, benefits of this work are depicted.

7.1 Energy Refactorings' Results

In Chapter 4 several Energy Refactorings are described and validated. The number of measurement for the validation of the Energy Refactorings amounts ten measurements for each case, hence about 140 measurements with a period of two hours were done. The measurement results are summarized to show possible energy savings for mobile devices on application level. These results are depicted in Table 7.1. The difference in percent for each Energy Refactoring and measurement technique is shown to present the savings at a glance.

Energy Code Smell	Measurement with Energy Code Smell			Measurement without Energy Code Smell			Difference (in %)		
	File Based	Energy Profiling	Delta-B	File Based	Energy Profiling	Delta-B	File Based	Energy Profiling	Delta-B
Third-Party Advertisement "GpsPrint"	6628 J	268 J	6405 J	5272 J	300 J	5144 J	20.5	-10.7	19.7
Third-Party Advertisement "TreeGenerator"	5310 J	308 J	4814 J	3681 J	269 J	3413 J	30.7	12.7	29.1
Binding Resources Too Early	11903 J	981 J	19566 J	11276 J	706 J	16032 J	5.3	28.0	18.1
Statement Change	3965 J	274 J	3768 J	3823 J	297 J	3382 J	3.5	-7.7	10.2
Backlight HTC	3681 J	269 J	3413 J	3796 J	272 J	3768 J	-3.0	-1.1	-2.9
Backlight S4	---	26152 J	10400 J	---	19184 J	4517 J	---	26.6	56.6
Data Transfer	4232 J	289 J	4064 J	3864 J	265 J	3497 J	8.6	8.3	14.0

Table 7.1: Energy Refactoring Results

The values for the energy consumption of each Energy Code Smell are extracted from the evaluation part of the Energy Refactoring Catalog in Chapter 4. In the base of these values, the difference is calculated. The differences in Table 7.1 show that almost all validated Energy Code Smells saves energy when the *file-based* and *delta-B measurements* are used.

The *Energy Profiling* has two negative results for *Third-Party Advertisement "GpsPrint"* and *Statement Change* which declares that the application with the Energy Code Smell consumes less energy ()while the other two measurement techniques show positive results. Also, *Backlight* shows no energy saving for the HTC.

For two Energy Code Smells *Third-Party Advertisement* and *Backlight* two energy measurements are made to show the difference between different applications and mobile devices. These different are clearly visible. Removing *Third-Party Advertisement* saves approximately 20 % energy within GpsPrint and approximately 30 % energy within TreeGenerator, although the same advertisements are used. The difference between these applications is that GpsPrint needs the Internet connection, even if advertisements are not requested. The measurement for TreeGenerator without Energy Code Smell was done without an Internet connection, and hence, the additional energy saving of 10 % is explainable. The Energy Code Smell *Backlight* is validated for two mobile devices, the HTC and the S4, to show the dependencies of different hardware resources, such as the screen. These measurements clearly show that this Energy Code Smell is important for AMOLED screens because the *delta-B measurement* illustrates energy savings up to 56.6 %. For the HTC, it can be said, that the inverse case saves energy (white background instead of black), however the savings are less with approximately 3 %.

All measurement techniques shows energy savings for the Energy Code Smell *Binding Resources Too Early*. However, the results are quite different, due to the modified HTC Power Profile (cf. Section 2.5.3). The *file-based measurement* depicts a saving of approximately 5.3 % and the *Energy Profiling* even of approximately 28 %. The measurement results of *Statement Change* are also very different. The *Energy Profiling* shows no energy saving, i.e. `if`-statements are more energy-efficient than `switch`-statements. The other two measurements illustrates an energy saving between approximately 3.5 % and 10.2 % for `switch`-statements. The last validated Energy Code Smell is *Data Transfer*. Here all measurements depict an energy saving between approximately 8.3 % and 14 %. This means, that it saves energy when data are saved on memory card instead to load them form a server via Internet during applications runtime.

These results show that it is sensible to execute the Energy Refactorings to save energy on mobile devices. The presented Energy Refactorings are only validated for Android applications, but the definition of the Energy Code Smells can be assigned to other platforms. For further platforms, the analysis and restructuring must be implemented. Moreover, the Energy Code Smells must be validated for other platforms to confirm their savings because this master's thesis checked them only on Android mobile devices.

7.2 Lesson Learned

This section summarizes some important points which were identified during the master's thesis and have an influence on the described results. Firstly, the encoding of parsed Android applications chances after parsing with the pro et con parser [FWE+12], so that the

code which is parsed back cannot be read completely. In addition, the method's annotation within Android applications runs off, so that unparsed applications cannot be executed. Furthermore, Energy Code Smells which occur in layout information cannot detected because these information are saved into XML files which are not represented by the Java TGraph approach. This is the case for *Backlight*, the information about the background color are saved into `activity_main.xml` for Android applications. To get these information by this approach, the Java meta model in Figure 2.4 must be extended with additional functions of Android applications. Moreover, the Android application AdBlock Plus was also validated to check whether the energy consumption is influenced. The measurements in Appendix B show that the energy consumption rises because advertisements are partly not displayed but their request is not stopped. Hence, AdBlock Plus is an energy-inefficient application. This was tested as the Energy Code Smell *Third-Party Advertisement* was described.

This master's thesis shows that it is possible to reduce the energy consumption of Android applications by Energy Refactorings. However, less free applications could be found with the here defined Energy Code Smells, so that an own application needed to be written to check the energy consumption of the Energy Code Smells.

7.3 Work Packages for the Thesis

In section 1.4 objectives and work packages for this master's thesis are described. At this point, it is checked whether the described objectives and work packages have been reached. Firstly, the work packages in Section 1.4 are repeated here. Secondly, it is described how these work packages were completed. Finally, the main objective is checked.

The following work packages were described at the beginning of this master's thesis and now it is proofed whether they are realized:

- *Literature study about required techniques and other possibilities to reduce energy consumption on applications level.*
  The parts *Related Work* in Section 1.3, *Basic Techniques* in Section 2, *Energy Refactorings* in Section 3, and the definition of Energy Code Smells in Section 4 and 5 show which literature was used to create this master's thesis.

- *Defining at least five Energy Refactorings (cf. Section 4) for no special platform.*
  In Section 4 five Energy Refactorings were defined completely, and in Section 5 eight further Energy Refactorings were defined but without an example, analysis, and restructuring.

- *Implementing at least three restructurings for the Android platform.*
  In Section 4 three Energy Refactorings (*Third-Party Advertisement*, *Binding Resources Too Early*, and *Data Transfer*) contain an analysis and a restructurings which were implemented with JGraLab. In addition, the other two Energy Refactorings (*Statement Change* and *Backlight*) contain only an analysis.

- *Apply the implementation on different freely available application, like GPSPrinter [Rob12], Standup Timer [Woo11], MyTracks [MyT], etc.*
  The most Energy Refactorings were only applied for one application. The Energy Refactoring *Third-Party Advertisement* was validated for GpsPrint and TreeGenerator, and the Energy Refactoring *Backlight* was validated with two mobile devices. Further validations were not done, because no further source code for applications were found which contain one of the defined Energy Code Smells.

- *Evaluating the Energy Refactorings using the energy measurement tool by Schröder [Sch13] to check their energy consumption.*
  The five Energy Refactorings in Section 4 were validated with the measurement tool by Marcel Schröder. In addition, the energy consumption for the sim card request of the HTC and the application AdBlock Plus were measured.

- *Intended result: The complete process for refactorings (name, definition, motivation, constraints, example, analysis, restructuring, and evaluation) (cf. Section 3.2) must be demonstrated. The number of implemented restructurings can be changed when the implementation needs more time than expected.*
  Three Energy Refactorings were completely realized to demonstrate the defined process in Section 1.2.

The main objective *"Creating an Energy Refactoring Catalog which defines energy-inefficient source code parts and removes them by a semi-automatic transformation."* (p. 2) was reached and the catalog was presented in Section 4. Moreover, further Energy Refactorings were described in Section 5 but not validated, hence, they depict a basis for the Energy Refactoring Catalog.

7.4 Outlook

In this section next steps are described, which can be made after this master's thesis to develop this area further. On the one hand, the described Energy Refactorings in Chapter 5 can be implemented with the JGraLab API and evaluated with the measurement techniques by Josefiok and Schröder [JSW13]. Thus, the Energy Refactoring Catalog in Chapter 4 would be supplemented, and it would be enforced their validity. On the other hand, the validated Energy Refactorings can be checked by further applications to confirm their energy saving. If more measurements exist, a better evidence can be made about the energy saving for each Energy Code Smell. In favor, the already existing project can be extended to facilitate the implementation of new Energy Refactorings.

Furthermore, more Energy Code Smells and their restructuring can be defined and validated. In addition, the Energy Refactorings can be realized for other mobile platforms, such as Windows phone 8 and iOS, to make it interesting for more people and vendors. Therefor, more information about the other platforms are needed to adapt the Energy Code Smells to them or to define new Energy Code Smells which are platform-dependent, such as *Binding Re-*

sources Too Early for Android. To identify and to realize Energy Refactorings for Windows phone 8 and iOS, the catalog in Section 4 constitutes a good basis for new definitions.

To detect further Energy Code Smells, it would make sense to extend the Java TGraph approach. If the Java meta model in Figure 2.4 is extended, the TGraph approach will work for more Energy Code Smells on the Android platform (cf. Section 4.4). Furthermore, new meta models for Windows phone 8 and iOS are needed to apply this approach on other platforms which use different programming languages, such as C, C#, Objective C, etc. (cf. Section 2.3.3).

If further Energy Code Smells are identified, the energy saving for one application might be higher, when all Energy Refactorings are executed. And if the energy saving rises through executing Energy Refactorings on applications, more people would be interested in it to extend the battery duration of their mobile devices.

7.5 Benefits from this Thesis

The definition and implementation of the Energy Refactoring Catalog describes energy-efficient code parts and their restructurings to remove them. This could help programmers to check their applications before applications are adjusted into the applications store, such as Google Play [Gooa]. Moreover, further Energy Refactorings were defined to show further ways to save energy on applications level on mobile devices.

Due to the extensive literature research, several Energy Code Smells and authors, who are also interested in energy-efficient programming, were found. In comparison to this work, the most approaches to save energy on mobile devices by other authors (cf. Section 1.3) includes no restructuring so that their Energy Code Smells must be removed manually. In this thesis a first approach for Android is shown how energy-inefficient code can be removed by a semi-automatic process.

A Energy Measurement for Sim Card Request

In this part the sim card request of the HTC One is measured to identify possible influences on the energy consumption during the energy measurement of Energy Refactorings. This measurement is needed to show a possible energy consumption during the sim card request. The measurement is made with Andromedar which is presented in Section 2.5 and the mobile device settings are described in Table A.1. The result of the measurements are illustrated in Figure A.1 and show that the request has a low influence on the energy consumption of the HTC.

Screen Settings	screen	on
	brightness	lowest
	auto rotation	off
Notification Settings	notifications	off
Applications settings	background applications	Andromedar
	power saver	off
Wireless & network settings	mobile data	off
	blue-tooth	off
	GPS	off
	NFC	off
	Wi-Fi	on
Gesture settings	three finger gestures	off
Miscellaneous	sim card	installed / not installed
	other applications	not installed

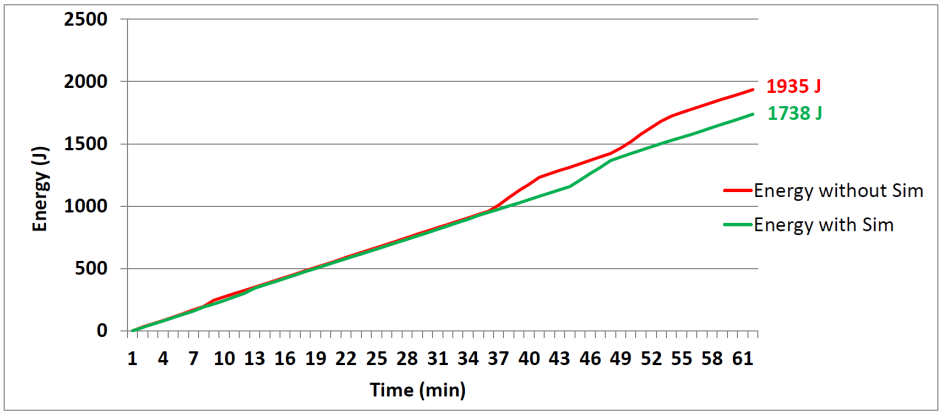
Table A.1: Mobile device settings for Sim Card Request

The first measurement with the *file-based measurement* in Figure A.1a shows a difference of 197 J within one hour. It depicts that the HTC without sim card needs more energy than with an installed sim card.

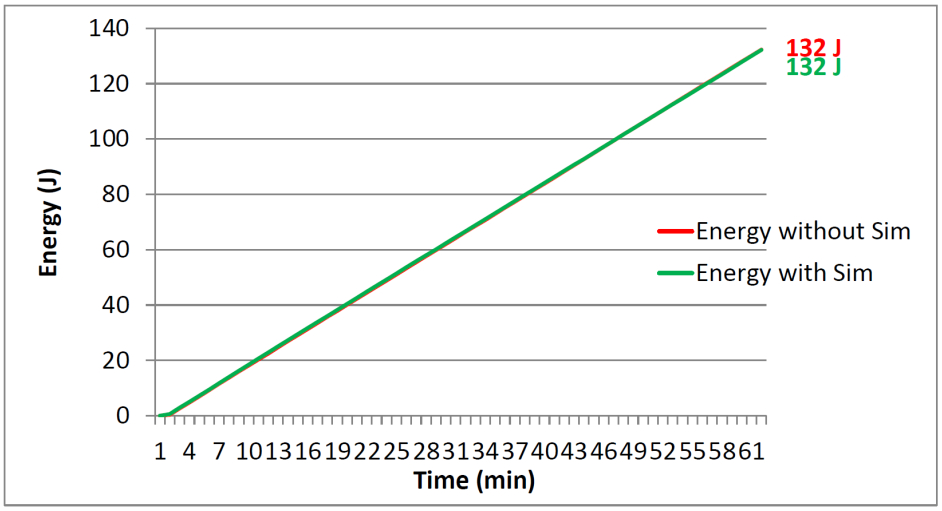
The second measurement illustrates the *Energy Profiling* in Figure A.1b. This measurement technique shows no difference between the measurements with and without sim card. The energy consumption amounts 132 J, its difference to the other two measurements because the HTC Power Profile was changed by a firmware update. This is explained in Section 2.5.3.

In Figure A.1c the *delta-B measurement* is shown. It demonstrates the same trend as the first measurement. The measurements without sim card consumes a little more energy than with an installed sim card. In this case, the difference amounts 30 J.

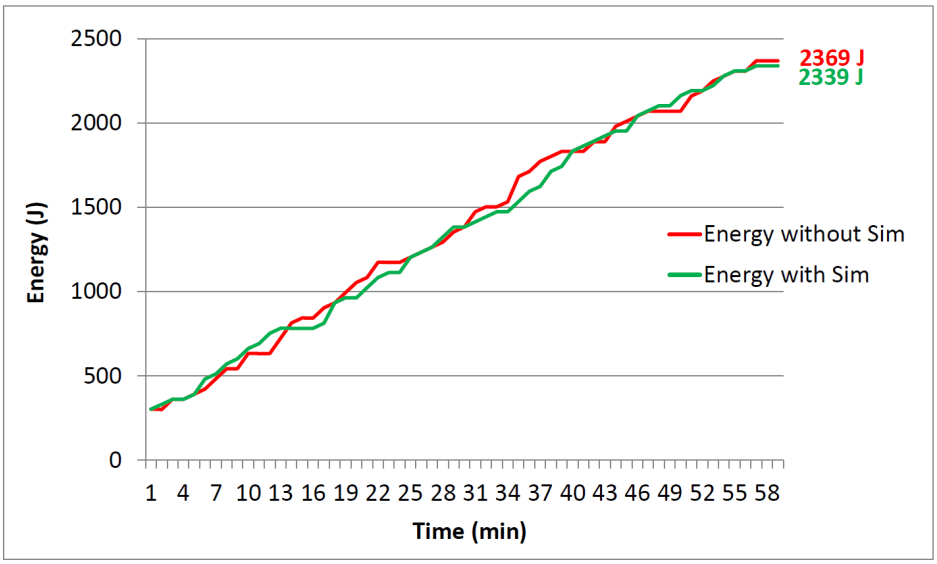
Also ten measurements were done for the validation of the sim card request, and hence, it shows only a trend which says that it has a low influence on other measurements because all measurements are done without sim card.



(a) File-based Measurement



(b) Energy Profiling



(c) Delta-B Measurement

Figure A.1: Measurement of GpsPrint with and without Sim Card

B AdBlock Plus

This section describes an application, called AdBlocker Plus, which is used to eliminate advertisements within applications and browsers for mobile devices. After description, the energy consumption of this application is analyzed.

B.1 General

AdBlock Plus is an Android application which stops displaying advertisements within other applications and browsers. AdBlock Plus runs in background as a service (cf. Section 2.3) and seeks for advertisements by using filters.The filters contain a list with URLs and suffixes which will be blocked, if requests are similar to an URL or ends with one of those suffixes. The lists are adapted for several languages, so that users can chose a list for their user behavior [PF].

However, AdBlock Plus allows some advertisements which adhere to their guidelines and are registered by them. This should help application vendors to get some income without any limitations for users because users can decide which lists they install to block advertisements, so that registered advertisements are also blocked.The guideline is described on their website [AdB], it contains: only static advertisements (e.g. no music, no animations), possible position for advertisements (e.g. placed before or after text, size of advertisements), advertisements should be clearly distinguished from application content (e.g. background color), etc.

Additionally, AdBlock Plus offers more services, such as disable tracking, disable mailware domains, and disable social media button. These services can be enabled and disabled through the user [PF].

B.2 Energy Measurement

Ten energy measurements with this application and GpsPrint with advertisements were done to check the energy consumption. The mobile device settings for the measurement are described in Table B.1 and the measurement results are depicted in Figure B.1. The results without AdBlock Plus are the same like the results for GpsPrint with advertisements in Section 4.1.

In Figure B.1a the results of the *file-based measurement* are depicted. The energy consumption with GpsPrint and AdBlock Plus amounts 7635 J and without AdBlock Plus 6628 J. The difference amounts 1007 J and shows that the use case with AdBlock Plus consumes significant more energy than without it.

The second graph shows the *Energy Profiling* which is depicted in Figure B.1b. It also shows a difference between the measurements with and without AdBlock Plus, it amounts 46 J. The difference of the results to the other two measurement techniques based on the

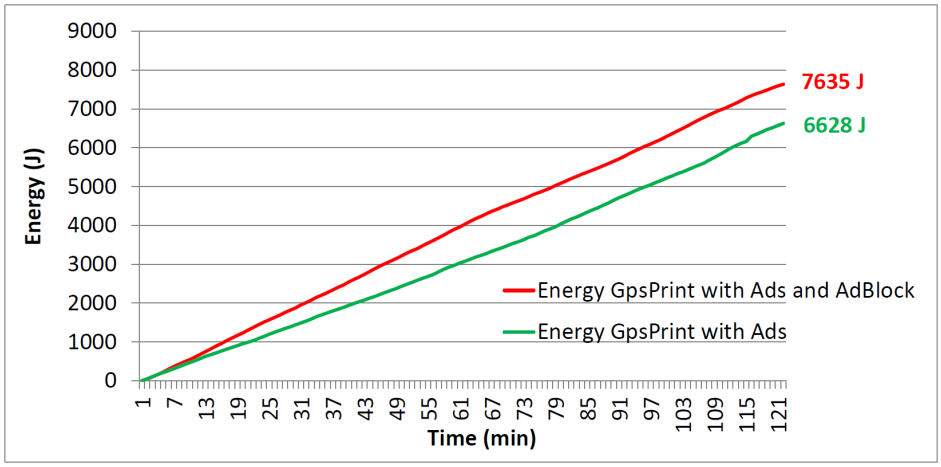
Screen Settings	screen	on
	brightness	lowest
	auto rotation	off
Notification Settings	notifications	off
Applications settings	background applications	Andromedar, AdBlocker Plus
	power saver	off
Wireless & network settings	mobile data	off
	blue-tooth	off
	GPS	on
	NFC	off
	Wi-Fi	on
Gesture settings	three finger gestures	off
Miscellaneous	sim card	not installed
	other applications	GpsPrintWithAdMobs

Table B.1: Mobile device settings for AdBlock Plus

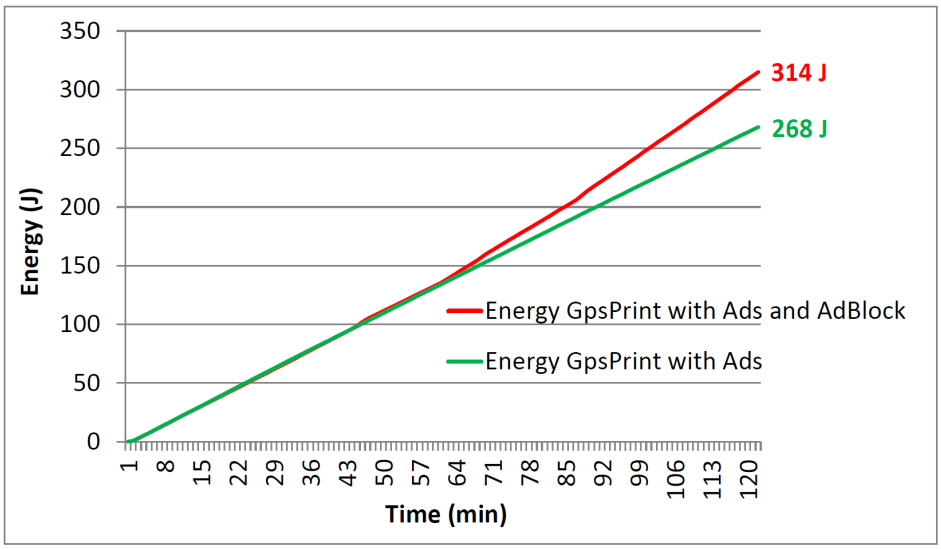
changed HTC Power Profile which is described in Section 2.5.3. But, it shows the same trend like the others.

The last measurement technique in Figure B.1c, the *delta-B measurement*, illustrates the same trend as the *file-based measurement*. It shows a difference of 1565 J between the two measurements. In this case, GpsPrint and AdBlock Plus consume 7970 J and only GpsPrint consumes 6405 J.

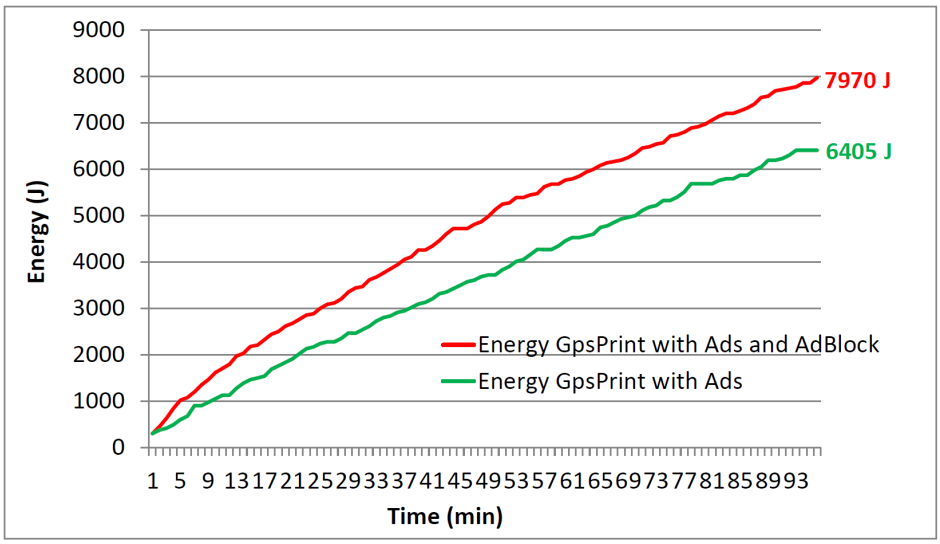
These results show that the usage of AdBlock Plus does not save energy though advertisements are filtered. But, some advertisements were displayed completely or partially during the energy measurement with AdBlock Plus. Advertisements which contain only text messages were always displayed. When the advertisement consists of text and image, only the text was shown, and the image did not exist and was displayed as not available. In both cases, AdBlock Plus does not stop displaying advertisements. It is possible that the advertisements in GpsPrint are conform to the guidelines of AdBlock Plus, i.e. these advertisements are not on the filter lists, because it is only a banner which is positioned after the text of GpsPrint. Hence, advertisements are displayed and an additional application runs which leads to a higher energy consumption as before without AdBlock Plus. Finally, it can be said that AdBlock Plus is an Android application which is energy-inefficient programmed. However, its objective is not saving energy but rather blocking advertisements.



(a) File-based Measurement



(b) Energy Profiling



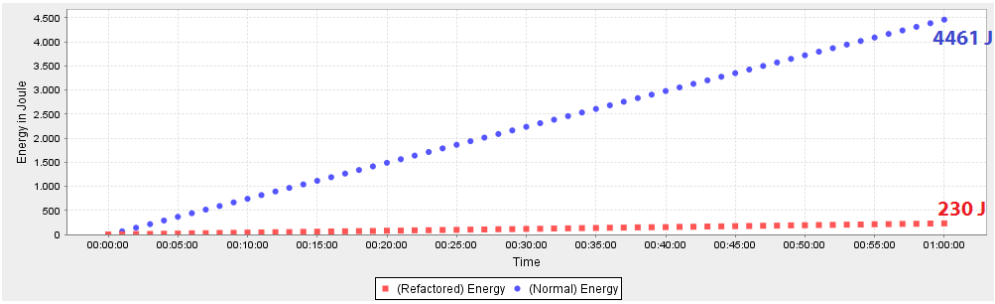
(c) Delta-B Measurement

Figure B.1: Measurement of GpsPrint with and without AdBlock Plus

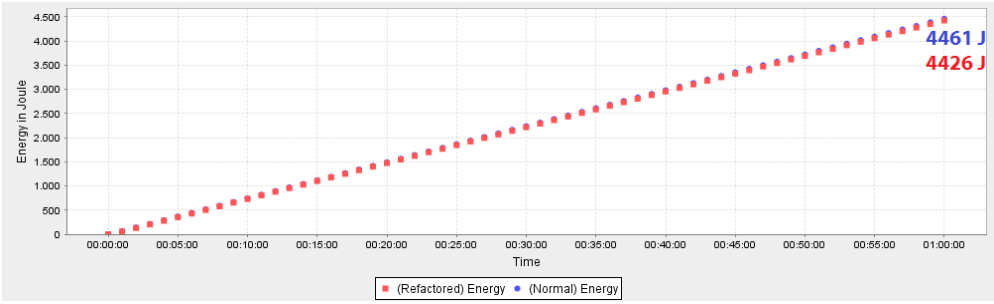
C Modified Power Profile for HTC

The HTC power profile was changed by the last firmware update to the profile on the right side in Table C.1. Hence, the measurement results of the *Energy Profiling* in Section 4 differs from the other two. This was described in Section 2.5.3. In this section, the HTC power profile is changed again to get better measurement results. The modified power profile is on the left side in Table C.1. It contains the values of the first HTC power profile in Table 2.1 on the right side. However, not all values can be adopted because the number of CPU states is different. Hence, the state of *CPU_BS0* is the same, and the state *CPU_BS15* gets the value of the state *CPU_BS11* in Table 2.1. These values describes the scope of the energy consumption of the CPU. The states between are modified and depict nearly the same intervals as the power profile in Table 2.1.

This modified power profile is used for two measurement results to compare them with the standard HTC power profile. First, the results for GpsPrint with advertisements which are described in Section 4.1 are shown with both power profiles in Figure C.1a. It shows a significant difference of 4231 J. The result with the modified power profile is higher, but the other measurement results in Section 4.1 are even higher with an energy consumption of 6628 J or 6405 J.



(a) GpsPrint with advertisements (new and old power profile)



(b) GpsPrint with and without advertisements (new power profile)

Figure C.1: GpsPrint with Advertisements (modified power profile)

However, the modified power profile seems to produce better results, and maybe the manual modifications on the CPU states are not good enough because the state *CPU_BS15* is never reached but the measurements of Marcel Schröder [Sch13] shows that *CPU_BS11* is reached.

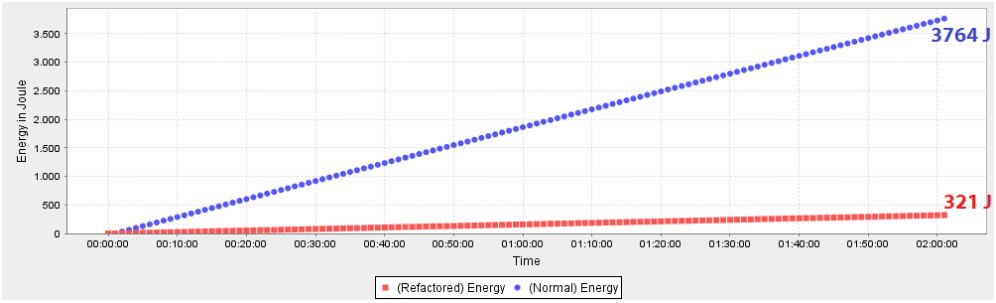
Component	Current in mA
Screen_BSOn	100
Screen_BS0	16
Screen_BS1	48
Screen_BS2	80
Screen_BS3	112
Screen_BS4	144
CPU_BS0	66,6
CPU_BS1	80
CPU_BS2	84,8
CPU_BS3	90
CPU_BS4	99
CPU_BS5	105,5
CPU_BS6	111,3
CPU_BS7	115,6
CPU_BS8	122,5
CPU_BS9	130,1
CPU_BS10	138,5
CPU_BS11	144,7
CPU_BS12	150,3
CPU_BS13	156,6
CPU_BS14	162,5
CPU_BS15	168,4
CPU_FILE_IDLE	2,8
GPS_BS	170
Wifi_BS	4
Wifi_DATASend	120
Wifi_DATASend BYTES	0
Wifi_DATAReceived	120
Wifi_DATAReceived BYTES	0
Mobile_BSOn	300
Mobile_BSScan	0
Mobile_BSRadio0	3
Mobile_BSRadio1	3
Mobile_BSRadio2	3
Mobile_BSRadio3	3
Mobile_BSRadio4	3
Mobile_DATASend	300
Mobile_DATASend BYTES	0
Mobile_DATAReceived	300
Mobile_DATAReceived BYTES	0
Bluetooth_THREAD	0,3
Audio_THREAD	88

Component (OS 4.1.1)	Current in mA
Screen_BSOn	2,19
Screen_BS0	4,861
Screen_BS1	14,583
Screen_BS2	24,305
Screen_BS3	34,027
Screen_BS4	43,749
CPU_BS0	10
CPU_BS1	20
CPU_BS2	30
CPU_BS3	46,9
CPU_BS4	79,4
CPU_BS5	84,4
CPU_BS6	87,8
CPU_BS7	90,6
CPU_BS8	106,1
CPU_BS9	110,3
CPU_BS10	118,3
CPU_BS11	133,3
CPU_BS12	160,6
CPU_BS13	179,2
CPU_BS14	233,6
CPU_BS15	253,3
CPU_FILE_IDLE	0,1
GPS_BS	1
Wifi_BS	0,1
Wifi_DATASend	0,1
Wifi_DATASend BYTES	0
Wifi_DATAReceived	0,1
Wifi_DATAReceived BYTES	0
Mobile_BSOn	1
Mobile_BSScan	0
Mobile_BSRadio0	0,2
Mobile_BSRadio1	0,1
Mobile_BSRadio2	0,1
Mobile_BSRadio3	0,1
Mobile_BSRadio4	0,1
Mobile_DATASend	1
Mobile_DATASend BYTES	0
Mobile_DATAReceived	1
Mobile_DATAReceived BYTES	0
Bluetooth_THREAD	0,1
Audio_THREAD	0,1

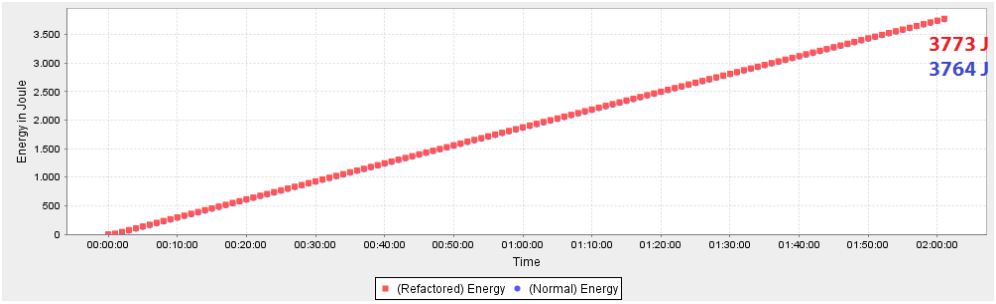
Table C.1: Modified HTC Power Profile

The modified power profile contains approximately the same values of the used power profile in [Sch13], but they are scattered on 15 values instead of 11, and hence the highest energy consumption is never reached in the measurements of this master's thesis, so that the measurement results are lesser than the values of Schröders measurements. The second graph in Figure C.1b depicts the energy consumption of GpsPrint with and without advertisements. It shows that the energy consumption is nearly the same, which results of the similar runtime of components. For both measurements the same components are needed in which the data traffic for GpsPrint with advertisements are higher but not catch by this measurement technique.

The second measurement results which are changed by the modified power profile are the results of the Energy Refactoring *Backlight* in Section 4.4. The difference between the standard power profile and the modified power profile amounts 3443 J and is illustrated in Figure C.2a. TreeGenerator with a white background consumes 3764 J, the other two measurements in Figure 4.17 shown an energy consumption of 3796 J and 3515 J. Hence, the modified power profile provides nearly the same measurement results. The measurements show that mostly the CPU states 0 until 4 are used and the difference between the energy consumption of this states is not so high than in the higher states. Therefore, the modified power profile improves the results of *Energy Profiling* in this case. The second graph in Figure C.2b shows the results of TreeGenerator with a black and white background. Both variants consumes nearly the same energy, the white background consumes 3773 J and the black background consumes 3764 J. This reflects the results in Figure 4.17.



(a) TreeGenerator with black background (new and old power profile)



(b) TreeGenerator with black and white background (new power profile)

Figure C.2: TreeGenerator Backlight measurement

D Console Output for TreeGenerator

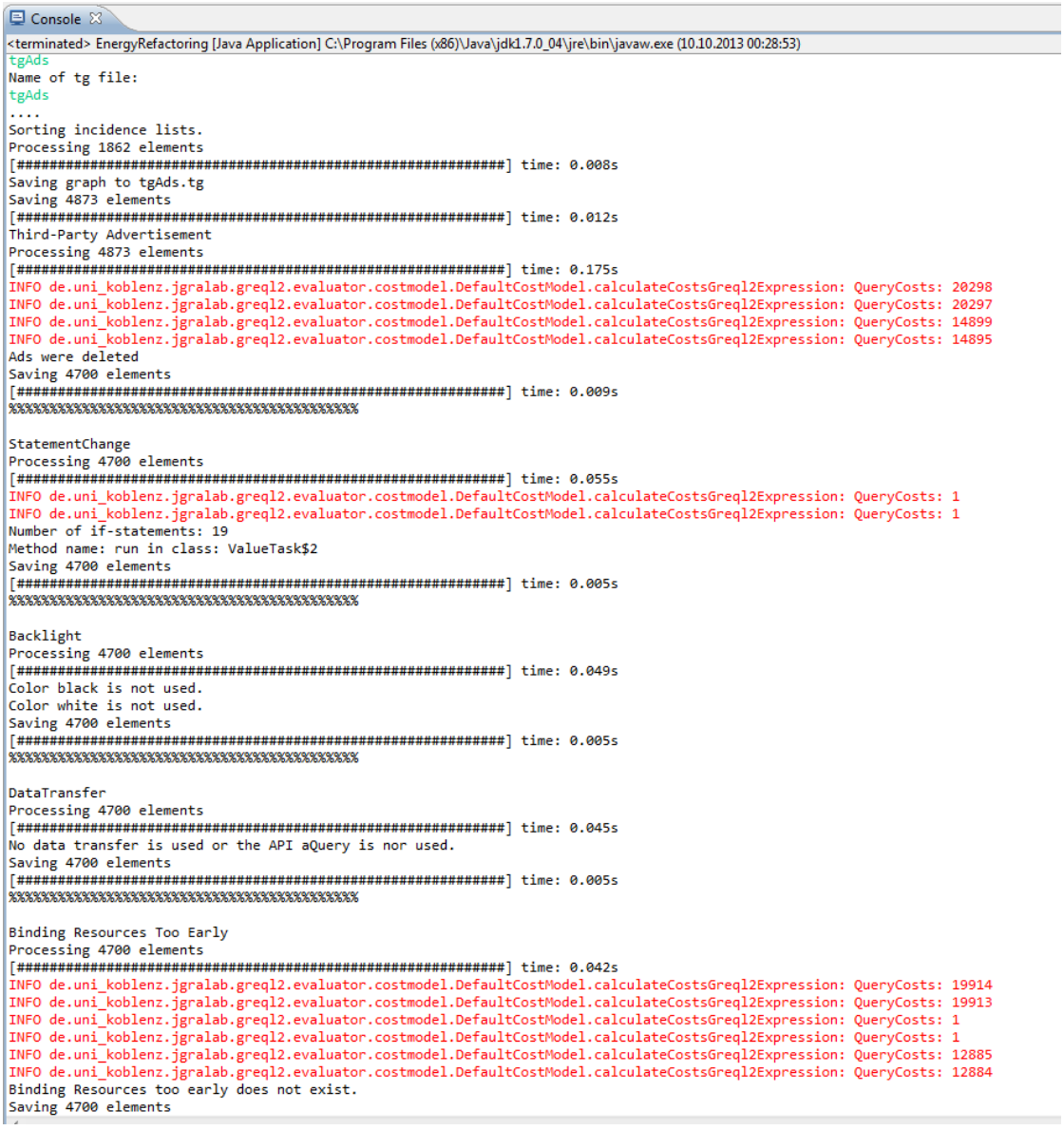


Figure D.1: Screen shot of the EnergyRefactoring results (TreeGenerator)

E CD Content

In addition to the master's thesis a CD with all measurements, graph queries and transformation, applications, and documents was created. The CD contains the following data: *applications*, *code*, *documents*, and *measurements*.

- *applications*: This file contains all Android apk's which were used in this master's thesis. It is divided into two files: *measuredApplications* and *measurementTechnique*. In *measuredApplications* the apk's AdBlock Plus and different forms of GpsPrint and TreeGenerator are saved. The file *measuredTechniques* contains the apk's Andromedar and GPSStarter.
- *code:* This file includes the complete eclipse project which realizes the queries and transformations for the Energy Refactorings. Also, the source code for GpsPrint and TreeGenerator are available.
- *documents:* This file includes the vision and this master's thesis. Also, it contains all diagrams, figures, and tables which are depict in this thesis.
- *measurements:* This file contains all energy measurements which are ordered by the Energy Refactorings and two additional measurements. Hence, the files *ThirdPartyAdvertisements*, *Backlight*, *BindingResourcesTooEarly*, *DataTransfer*, and *StatementChange* for the Energy Refactorings. The additional measurements are in the files *SimCard* and *AdBlockPlus*. All files contain at least twenty measurements and two excel files which represents the average of all measurements. The files *Backlight* and *BindingResourcesTooEarly* are also subdivided into two files. *Backlight* contains twenty measurements for the HTC and the S4, and *BindingResourcesTooEarly* contains twenty measurements for GpsPrint and TreeGenerator.

List of Figures

List of Tables

References

[AdB] AdBlock Plus. Akzeptable Werbung in Adblock Plus zulassen. https://adblockplus.org/de/acceptable-ads Last visit on 19th August 2013.

[AdM] Build a great app business with AdMob. http://www.google.com/ads/admob/ Last visit on 27th April 2013.

[Anda] Android. Optimizing Battery Life. http://developer.android.com/training/monitoring-device-state/index.html Last visit on 11th October 2013.

[Andb] Android Developers. Activity. http://developer.android.com/reference/android/app/Activity.html Last visit on 23rd May 2013.

[Andc] Android Developers. android-query. https://code.google.com/p/android-query/ Last visit on 5th September 2013.

[Andd] Android Developers. Android, the world's most popular mobile platform. http://developer.android.com/about/index.html Last visit on 24th May 2013.

[Ande] Android Developers. Application Fundamentals. http://developer.android.com/guide/components/fundamentals.html Last visit on 24th May 2013.

[Andf] Android Developers. Intent. http://developer.android.com/reference/android/content/Intent.html Last visit 7th August 2013.

[Andg] Android Developers. Monitoring the Battery Level and Charging State. http://developer.android.com/training/monitoring-device-state/battery-monitoring.html Last visit 7th August 2013.

[Andh] Android, Developers. PowerManager.WakeLock. http://developer.android.com/reference/android/os/PowerManager.WakeLock.html. Last visited on 10th June, 2012.

[Ang] AngryBirds. https://play.google.com/store/apps/details?id=com.rovio.angrybirds Last visit on 30th March 2013.

[BBC] BBC. Free mobile apps 'drain battery faster', March. http://www.bbc.co.uk/news/technology-17431109 Last visited on 10th June 2012.

[BGNW13] Christian Bunse, Marion Gottschalk, Stefan Naumann, and Andreas Winter, editors. *2nd Workshop EASED@BUIS 2013 - Energy Aware Software-Engineering and Development*, number 4/2013, Oldenburg, 04 2013. Carl von Ossietzeky University, Oldenburg, Software-Engineering.

[Bin07] D. Binkley. Source Code Analysis: A Road Map. In *Future of Software Engineering*. IEEE, 2007.

[BRM09] Höpfner H. Bunse, C., S. Roychoudhury, and E. Mansour. Choosing the "best" Sorting Algorithm for optimal Energy Consumption. In *IC SOFT 2009: 4th International Confernce on Software and Data Technologies*, 2009.

[BS13] C. Bunse and S. Stiemer. On the Energy Consumption of Design Patterns. In C. Bunse, M. Gottschalk, A. Winter, and S. Naumann, editors, *2nd Workshop EASED@BUIS 2013 - Energy Aware Software-Engineering and Development*, page 22, 2013.

[CC90] E. J. Chikofsky and J. H. Cross. Reverse Engineering and Design Recovery: A Taxonomy. *Software, IEEE*, 7(1), 1990.

[CCMF13] X. Chen, Y Chen, Z. Ma, and F. Fernandes. How is Energy Consumed in Smartphone Display Applications? In *ACM HotMobile'13*, Jekyll Island, Georgia, USA, February 2013.

[CGKO97] Y. Chen, E. R. Ganser, and E. Koutso Os. A C++ Data Model Supporting Reachability Analysis and Dead Code Detection. *In Proc. 6th European Software Engineering Conference and 5th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, April 1997.

[CH10] A. Carroll and G. Heiser. An Analysis of Power Consumption in a Smartphone. In *USENIX: Annual Technical Conference*, 2010.

[CNR90] Y Chen, M. Y. Nishimoto, and C. V. Ramamoorthy. The C Information Abstraction System. In *Transactions on Software Engineering*, pages 325–334. IEEE, March 1990.

[CSC02] I. Choi, H. Shim, and N. Chang. Low-Power Color TFT LCD Display for Hand-Held Embedded Systems. In *ISLPED'02*, August 2002.

[dSB10] W. G. P. da Silva and L. Brisolara. Evaluation of the Impact of Code Refactoring on Embedded Software Efficiency. In *1. Workshop de Sistemas Embarcados*, pages 145–150, 2010.

[EKRW02] J. Ebert, B. Kullbach, V. Riediger, and A. Winter. GUPPRO. Generic Understanding of Programs - An Overview. *Electronic Notes in Theoretical Computer Science*, 72(2), 2002.

[EKW97] J. Ebert, M. Kamp, and A. Winter. A Generic System to Support Multi-Level Understanding of Heterogeneous Software. Technical report, University Koblenz-Landau, June 1997.

[Ern03] M. D. Ernst. Static and dynamic analysis: synergy and duality. In *WODA 2003 ICSE Workshop on Dynamic Analysis*, pages 25–28, 2003.

[ERW08] J. Ebert, V. Riediger, and A. Winter. Graph Technology in Reverse Engineering, The TGraph Approach. In R. Gimnich, U. Kaiser, J. Quante, and A. Winter, editors, *10th Workshop Software Reengineering (WSR 2008)*, pages 67–81, Bonn, 2008. GI.

[FBB+02] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 2002.

[FWE+12] A. Fuhr, A. Winter, U. Erdmenger, T. Horn, U. Kaiser, V. Riediger, and W. Teppe. Model-Driven Software-Migration - Process Model, Tool Support and Application. In A. D. Ionita, M. Litoiu, and G. Lewis, editors, *Migrating Legacy Applications: Challenges in Service Oriented Architecture and Cloud Computing Environment*. IGI Global, Hershey, PA, 2012.

[GHJV93] E. Gamma, R. Helm, R. E. Johnson, and J. M. Vlissides. Design Patterns: Abstraction and reuse of object-oriented design. In *ECOOP*, 1993.

[GJJW12] M. Gottschalk, M. Josefiok, J. Jelschen, and A. Winter. Removing Energy Code Smells with Reengineering Services. In *Lecture Notes in Informatics*. GI, 2012.

[GNU] GNU. Gnu.de. `http://www.gnu.de/` Last visit on 30th March 2013.

[Gooa] Google play. `https://play.google.com` Last visit on 13th October 2013.

[Goob] Google Developers. Google Mobile Ads SDK. `https://developers.google.com/mobile-ads-sdk/?hl=de` Last visit on 7th August 2013.

[GZT] M. Gordon, L. Zhang, and B. Tiwana. A Power Monitor for Android-Based Mobile Platforms. `http://powertutor.org/` Last visit on 30th September 2013.

[Hav09] K. Havelund. Java Coding Standard. Technical report, California Institute of Technology, 2009. p. 17.

[HB10] H. Höpfner and C. Bunse. Towards an Energy-Consumption based Complexity Classification for Resource Substitution Strategies. In W. Balke and C. Lofi, editors, *Proceedings of the 22. Workshop on Foundations of Databases (Grundlagen von Datenbanken)*, Bad Helmstedt, Germany, May 2010.

[HB11] H. Höpfner and C. Bunse. Energy Awareness Needs a Rethinking in Software Development. In *ICSOFT 2011 - Proceedings of the 6th International Conference on Software and Data Technologies*, Seville, Spain, 2011. SciTePress.

[HE11] T. Horn and J. Ebert. The GReTL Transformation Language. In *Theory and Practice of Model Transformations - 4th International Conference, ICMT 2011*, pages 183–197, Zurich, Switzerland, June 2011. Springer Berlin / Heidelberg.

[Hei12] Heise online. Adblock Plus für Android entfernt In-App-Werbung, 2012. http://www.heise.de/newsticker/meldung/Adblock-Plus-fuer-Android-entfernt-In-App-Werbung-1757454.html Last visit on 30th March 2013.

[Hom09] J. Homann. Begrüßungsansprache. In A. Picot and K.-H. Neumann, editors, *E-Energy: Wandel und Chance durch das Internet der Energie*, pages 3–11. Springer, 2009. p. 9.

[Hor10] T. Horn. Model MigrationWith GReTL. In *Proceedings of Transformation and Tool Contest*, May 2010.

[HTCa] HTC Corporation. HTC One X. http://www.htc.com/uk/smartphones/htc-one/ Last visit on 27th August 2013.

[HTCb] HTC Inside. HTC One X erhält Update auf 3.20.401.1, April. http://www.htcinside.de/htc-one-x-erhaelt-update-auf-3-20-401-1/ Last visit on 2nd September 2013.

[JCD02] D. Jin, J. R. Cordy, and T. R. Dean. Where's the Schema? A Taxonomy of Patterns for Software Exchange. In *IWPC*, pages 65–74, 2002.

[JGJ+12] J. Jelschen, M. Gottschalk, M. Josefiok, C. Pitu, and A. Winter. Towards Applying Reengineering Services to Energy-Efficient Applications. In R. Ferenc, T. Mens, and A. Cleve, editors, *Proceedings of the 16th European Conference on Software Maintenance and Reengineering*. IEEE, 2012.

[Jos12] M. Josefiok. An Energy Abstraction Layer for Mobile Computing Devices. Master's thesis, Carl von Ossietzky Universität Oldenburg, 2012.

[JSW13] M. Josefiok, M. Schröder, and A. Winter. An Energy Abstraction Layer for Mobile Computing Devices. In C. Bunse, M. Gottschalk, S. Naumann, and A. Winter, editors, *2nd Workshop EASED@BUIS 2013*, number 04/2013. Oldenburger Lecture Notes on Software Engineering, 2013.

[Kah06] S. Kahle. JGraLab: Konzeption, Entwurf und Implementierung einer Java-Klassenbibliothek für TGraphen. Master's thesis, University Koblenz-Landau, 2006.

[Kam11] T. Kaminski. Intel erfindet den Transistor mit einer 3D-Struktur neu. online, Mai 2011. http://newsroom.intel.com/community/de_de/blog/2011/05/04/intel-erfindet-den-transistor-mit-einer-3d-struktur-neu Last visit on 8th December 2011.

[Khu] K. Khunkham. Mit Apple in acht einfachen Schritten reich werden. http://www.welt.de/wirtschaft/webwelt/article8315044/Mit-Apple-in-acht-einfachen-Schritten-reich-werden.html Last visit on 27th August 2013.

[Kre13] M. Kremp. Blackberry Z10 im Test: Handy mit Nottank, January 2013. http://www.spiegel.de/netzwelt/gadgets/angefasst-der-blackberry-z10-im-test-a-880411.html Last visit on 1st March 2013.

[KWC98] R. Kazman, S. G. Woods, and J. Carriere. Requirements for Integrating Software Architecture and Reengineering Models: CORUM II. In *Working Conference on Reverse Engineering*, 1998.

[leo] LEO Wörterbuch. https://play.google.com/store/apps/details?id=org.leo.android.dict&hl=de Last visit on 7th April 2013.

[Mic] Microsoft. Windows 8: Windows Neu Erfunden. http://www.microsoft.com/germany/msdn/academic/windows-8/infos-fuer-entwickler.aspx Last visit on 27th August 2013.

[MyT] MyTracks. https://code.google.com/p/mytracks/ Last visit on 2nd March 2013.

[OnV12] Neues GPS-Modul Telit Jupiter SE880 ist das kleinste im Markt und nutzt 3D-Embedded- Technologie, 2012. http://www.onvista.de/news/alle-news/artikel/11.10.2012-09:26:00-neues-gps-modul-telit-jupiter-se880-ist-das-kleinste-im-markt-und-nutzt-3d-embedded-technologie Last visit on 20th December 2012.

[PCHZ11] A. Pathak, Y. Charlie Hu, and M. Zhang. Bootstrapping Energy Debugging on Smartphones: A First Look at Energy Bugs in Mobile Devices. In *Hotnets '11*, Cambridge, MA, USA, November 2011. ACM.

[PCHZ12] A. Pathak, Y. Charlie Hu, and M. Zhang. Fine Grained Energy Accounting on Smartphones with Eprof. In *EuroSys'12*, 2012.

[PF] W. Palant and T. Faida. Über Adblock Plus für Android. https://adblockplus.org/de/about Last visit on 7th August 2013.

[Rob12] Robotmafia.org. GPS Print, 2012. https://play.google.com/store/apps/details?id=com.tyfon.gpsprint&hl=en Last visit on 30th March 2012.

[RWE] RWE Power AG. Kernkraftwerk Emsland. https://www.rwe.com/web/cms/de/16646/rwe-power-ag/standorte/kernkraft/kkw-emsland/ Last visit on 11th October 2013.

[Sama] Samsung. Samsung Galaxy S4. http://galaxys4.samsung.de/technik/ Last visit on 9th September 2013.

[Samb] Samsung. Samsung's Guide to Green. http://www.samsung.com/us/guide-page/green/ Last visit on 11th October 2013.

[Sch13] M. Schröder. Erfassung des Energieverbrauchs von Android Apps. Master's thesis, Carl von Ossietzky University, Oldenburg, 2013.

[SH12] M. Schirmer and H. Höpfner. Towards Using Location Poly-Hierarchies for Energy-Efficient Continuous Location Determination. In I. Schmitt, S. Saretz, and M. Zierenberg, editors, *Proceedings of the 24th GI - Workshop on Foundations of Databases*, Lübbenau, Germany, June 2012.

[She07] Findlay Shearer. *Power management in mobile devices*. Newnes, 2007.

[Sin01] A. Sinha. *Energy Efficient Operating Systems and Software*. PhD thesis, Massachusetts Institute of Technology, 2001. pp. 27 and 31-33.

[Sna11] Snapdragon S4 Processors: System on Chip Solutions for a New Mobile Age. White paper, ARM, October 2011.

[Sta] Statista. Absatzprognosen für 2013: Smartphones verkaufen sich am besten [Statistik]. `http://de.statista.com/themen/647/itk-branche/infografik/711/prognosen-zum-weltweiten-absatz-von-itk-geraeten/` Last visit on 18th December 2012.

[Ste12] S. Steimels. Display-Technik von Smartphones einfach erklärt, June 2012. `http://www.pcwelt.de/ratgeber/AMOLED-LCD-Co-Die-Display-Technik-von-Smartphones-einfach-erklaert-5913756.html` Last visit on 7th September 2013.

[Sto08] L. Stobbe. Stromverbrauch von Informations- und Kommunikationstechnik in Deutschland. Technical report, Bundesministerium für Wirtschaft und Technologie (BMWi), November 2008.

[Ull11] C. Ullenboom. *Java ist auch eine Insel*. Galileo Computing, 2011.

[WBM95] C. H. Wells, R. Brand, and L. Markosian. Customized Tools for Software Quality Assurance and Reengineering. In *Proceedings of 2nd Working Conference on Reverse Engineering*, pages 71–77, 1995.

[Woo11] J. Wood. standup-timer, 2011. `https://github.com/jwood/standup-timer` Last visit on 2nd March 2013.

[WRP+12] C. Wilken, S. Richly, G. Püschel, C. Piechnick, S. Götz, and U. Aßmann. Energy Labels for Mobile Applications. In *Lecture Notes in Informatics*, volume 208, pages 412–425, Bonn, 2012.

Declaration

I declare that the work presented here is, to the best of my knowledge and belief, original and the result of my own investigations, except as acknowledged, and has not been submitted, either in part or whole, for a degree at this or any other University.

Formulations and ideas taken from other sources are cited as such. This work has not been published.

Oldenburg, October 13, 2013

Marion Gottschalk