Using bad smell-driven code refactorings in mobile applications to reduce battery usage

Ana Rodriguez[1,2], Mathias Longo[1,2], and Alejandro Zunino[1,2]

[1] ISISTAN Research Institute. UNICEN University. Campus Universitario, Tandil (B7001BBO), Buenos Aires, Argentina. Tel.: +54 (249) 4439682.
[2] Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET)

Abstract. Mobile devices are the most popular kind of computational device in the world. These devices have more limited resources than personal computers, and more importantly, battery consumption is always an issue since mobile devices rely on their battery as energy supply. On the other hand, to date, many applications are developed using the object-oriented (OO) paradigm, which has some inherent features, such as object creation, that inherently consume energy in the context of mobile development. These features at the same time enable for maintainability, flexibility, among other software quality-related advantages. Moreover, known code refactorings driven by bad smells can be applied to mobile applications to produce good OO designs, at the expense of potentially consuming more energy. Then, this paper presents an analysis to evaluate the preliminary trade-off between OO design purity and battery consumption.

Keywords: Smartphones, Refactoring, Bad smell, Android

1 Introduction

Smartphones and tablets represent a very attractive market to develop applications, mostly because they are the commonest kind of computational devices in the world [15]. Despite their ever-growing capabilities, these devices have limited resources compared to personal computers. In particular, devices rely on batteries as their energy supply and, while their computational capacity has increased considerably, battery life-time has increased slowly [11,2]. Then, even when today's devices have more capabilities, their batteries have shorter duration because more powerful hardware needs more energy to run. In this context, an important open research problem is how to reduce energy consumption on these devices, specially when a resource intensive application is running.

There are different levels at which this problem can be addressed, in principle operating system level and application level. This paper chooses to work at the application level taking into account previous works which are focused on certain good object-oriented programming practices for desktop applications, but the environment and battery limitation claim application building should be rethought in the context of energy-efficient application development. Particularly, this paper focus on well-known anti-patterns known as Bad smells. There are several works whose main goal is to demonstrate how removing these Bad smells improves object-oriented application design (in terms of several quality metrics [6,3]), but after doing so class designs are much more complex, which might negatively impact on battery usage [12].

Even though battery usage is an aspect that affect all mobile devices, they are the most popular computational device and, as a consequence, there is a variety of these devices and their operating systems running on these devices. Then, this work is limited to a particular operating system, and on the other hand we target a particular high-level programming language. First, in regard to the targeted mobile operating system, we focus on Android-powered devices because of the widespread nature of the Android platform, which is present in millions of smartphones, tablets, and other devices. Specifically, Android smartphones units were first in the list of best selling OS in the United States in the last years, having an exponential growth. Particularly, their worldwide market share was 78.7% during 2013 and 81.5% during 2014, more than twice as much as iOS, the second most popular operating system[3].

After selecting the operating system, the second step is to analyze the programming language options to make the experiments because the approach focuses on improving battery consumption at the application level. In this way, there are two options to analyze: Java and Android native code which are the commonest choices used for Android application development. Then, Java is the language selected to use in this paper because of its popularity and its interesting features, i.e., it has been recognized as being a useful language to develop and execute both computationally intensive code [8,7,9,16] and user-centered applications[4]. In fact, Java provides and supports object-oriented programming, multi-threading and distributed computing implemented at the language level, platform independence, automatic memory management and exception handling, which make it interesting for general purpose applications.

As a result, the main goal of this paper is measuring the energy impact of refactoring Java source code based on certain Bad smells. This means that the battery consumption of different applications will be evaluated. For each application several versions are produced, an *original* with Bad smells and *refactored* versions without these smells, with different level of refactorings depending on developer preferences. After this, the impact of the refactorings on both battery consumption and class design quality will be evaluated to provide developers a guide about refactorings and their impact on battery consumption. Indeed, a previous work [15] demonstrates that battery consumption is mainly affected by object creation, method invocation and Java exceptions usage, this last including numerous object creations and method invocations. Moreover, removing Bad smells usually imply increasing the amount of objects and method invocations at runtime. For example, in the case of God class, a trivial refactoring is to split a God class in two or more classes, which implies more object creations. This suggests that energy saving trade-offs application design quality and application maintainability. It is necessary to mention that design quality is considered in purist terms, it means that the less bad smells a program has, the better is the design, and in order to remove those bad smells only the refactorings proposed by Fowler were taken into account.

Finally, according to the suggested methodology, it is necessary to use a specialized device to measure energy consumption accurately. Specifically, we used Power Monitor [13], a tool capable of measuring the power of any device that employs a lithium battery at a frequency of 5Khz. This tool comes along with a software application that shows Voltage and Amperage measurements, and allows users to export this information. Using this tool, the measurement of the impact of different refactorings in a Samsung Galaxy SIII smartphone was performed.

[3] www.idc.com
[4] www.tiobe.com

The rest of the paper is organized as follows. Section 2 describes the variety of Bad smells and their associated refactorings. Section 3 describes in detail the experiments that are analized in Section 3.3. Section 4 presents the conclusions and, finally, in Section 5 future works are mentioned.

2 Bad smells, refactorings and their implications on energy usage

This Section presents and explains the Bad smell concept and the different categories of Bad smells that are taken into account for this paper. Additionally, this Section mentions the variety of refactorings proposed in the literature to remove each Bad smell from codes. Finally, it is important to mention not all Bad smells were considered because there are more than twenty categories recognized by the community.

2.1 Bad smell

Designing and developing an application with object-oriented properties is a complex task. Then, when a team works in a project it is common that the object-oriented design finishes with some imperfections and this has negative consequences in the application features. These code imperfections, which are code sections that have not taken into account the use of good object-oriented patterns, are known as 'Code smells' or 'Bad smells'. Since the Bad smell concept was introduced by Fowler [3] in 1999, the community has identified more than twenty different types. Some of them are *Duplicated code*, *God Class*, *Brain Method* and *Data Class* [6], among others. For this paper, the experiments are centered only on the most frequent Bad smells. Additionally, the Bad smells chosen are those whose associated refactoring impact directly in the number of objects created and the number of messages sent between objects. Finally, next paragraphs explains in detail these Bad smells:

- *God Class*: also called *Long Class* [3]. This Bad smell refers to the classes centralizing most of the application functionality. It means that a Good Class is a class that does a great amount of work alone, delegating only low complex tasks. As a consequence, there are numerous classes that only contain data (*Data Class*), which do not have functionality. Additionally, this Bad smell usually leads to duplicate code and long methods (*Brain Methods*). As a result, *God Classes* have a great impact on system design because they affect negatively all quality attributes characteristic of an object-oriented design (reusability, extensibility, etc.), mainly because it is a typical case in which the system is based on a monolithic class.
- *Brain Method*: *Brain methods* are long methods that contain several variables and great complexity. As a consequence, these are methods that are not easy to reuse, understand and test because they have not a defined and particular functionality.
- *No Encapsulated Field*: This Bad smell relates to one of the most important properties of object-oriented programming: information hiding. This bad practice allows external entities to access directly to class attributes.
- *No Self-Encapsulated Field*: even in the same class, it is advisable to access the attributes by methods. That allows more flexibility, because if it is necessary to change some attribute, it is possible to change only the access method. In addition, it is easier to override the way an attribute is accessed in a class hierarchy. Then, accessing to attributes directly is considered a bad practice in object oriented programming.

2.2 Refactorings

The problems that arise due to having applications with Bad smells call for techniques to fix them in source code. To remove Bad smells, developers must modify the problematic code portions, i.e., to refactor them. There are several refactoring actions to solve many Bad smells. In fact, Martin Fowler proposes a catalog[5] with 80 different refactorings to solve these problems.

Extract Method This refactoring proposes removing portions of a long method, and extract them into new, smaller ones. Martin Fowler states that when developers need to insert a comment into a method, it is because there is a possible extraction method which corresponds with the semantics of the comment. That is, it is always better to have short and well appointed methods, so that any class has good semantics and it is simple to understand. This refactoring is an effective mean to attack *Brain Methods* since a *Brain Method* is a long and complex method. Thus, if it is divided into several short and simple methods the length and complexity decrease.

Replace method by an object Another refactoring to solve Brain Methods is the refactoring called Replace method by an object. This refactoring consists in extracting the *Brain methods* in a new class and then apply the refactoring Extract Method in that class. It is important to mention that applying this technique is more aggressive and laborious than only applying Extract Method.

Replace Conditional with Polymorphism While extracting methods is one of the ways of solving the problem of *Brain Method*, sometimes Extract method or Replace method by an object are not enough because long methods usually has several conditionals (*if-then-else* or *switch* statements) that prevent making a good separation and keeping method complexity low. Therefore, it is necessary to find a way to remove such conditionals maintaining the semantic of the operation. The refactoring Replace Conditional with Polymorphism proposes removing these selections taking advantage of one of the most important features that object-oriented programming offers, polymorphism. Then, when a *switch* or an *if-then-else* statement asking for the type of an element and then performing an action is detected, the solution is to make a hierarchy of classes to implement this method.

Move Method Some times there are methods that are used only for a few external classes. In those cases, the functionality of each class is not well defined. One possible solution for this bad practice is to move the method to the class which uses it the most. However, sometimes the section of code that is used by the external class is only a portion of the method, thus in that situation developer must remove only that portion of code, and then apply Move method. Finally, this refactoring can be used to solve *Brain methods*.

Extract class When there are one or more classes affected by the *God Class* bad smell the best solution is to divide the functionality of those classes in more than one class. This refactoring is known as Extract Class and it allows the developer extracting a portion of

[5] refactoring.com/catalog/

functionality of a larger class into two or more classes. With this refactoring, developers decrease the effects of *God Class* but often it is not enough. This means, to remove this bad smell it is necessary to apply several modifications: first, developers have to apply Extract Method of the longest methods that this class contains, then, they can apply Move method and, finally, they have to apply Extract class based on the remaining functionality.

Attribute access Fowler defines two refactorings to solve the problem of direct access to attributes:

1. Encapsulate Field: an attribute that is public becomes private and methods that allow access to that attribute (getter and setter) are defined.
2. Self-Encapsulate Field: This refactoring proposes to remove the shortcut to attributes within the class in which they are defined, and change the shortcuts by access methods.

Both refactorings help the developer to use the Move Method refactoring, since the use of attributes is decoupled from the rest of the class. For example, if a method is being moved to another class and it accesses the attributes that are being encapsulated, then the refactoring will be easier to apply because it does not imply refactoring the access to those attributes of the original class.

2.3 Refactorings and battery consumption

Although the discussed refactorings help developers to improve object-oriented code quality, these refactorings imply some actions that intuitively might affect negatively the battery consumption. Particularly, these refactorings add new object creations and new method calls. To see the impact of these actions, a previous work of our own was taken into account [15], where several individual micro-benchmarks were evaluated. One of these micro-benchmarks suggested that the attribute access through accessor methods would consume more battery than a direct attribute access. For that purpose, two versions of the same micro-benchmark were evaluated, one with direct access and the other one with access through accessor methods, and they were executed until the battery were fully consumed. In the same way, it was measured the object creation consumption, i.e., two versions were made and executed until the battery was with no charge. For the first version, in each step of the micro-benchmark a new object was created, while in the second one each step reused the object previously created.

Table 1 shows the results of the mentioned measurements. In this table readers can observe that under certain circumstances avoiding object creation can save up to 87.43% of energy and avoiding method calls can save up to 89.03% of energy. It is noticeable that these values are high values, then, proper small modifications in a code might have an important impact in the battery consumption of an application. As a consequence, developing a good mobile application in object-oriented terms would mean it is brittle in energy usage terms. In the Table, object reuse means emptying the state of an instance of a class and reusing it later in the code instead of creating a new instance of that class.

3 Experiments

The main goal of these experiments is to evaluate the impact of Bad smell refactorings on battery consumption. This means that next Sections evaluate the trade-off of applying good object-oriented practices versus practices which consume less battery. For this,

Benchmark	Number of executions (Average)	Gain factor	Energy saved (%)	Standard deviation (%)
Attribute access				
Getter-based Access	115,459,467,000	-		2.97
Direct Access	919,194,540,000	7.96	87.43	2.13
Object creation				
On-demand Object Creation	1,691,926,500	-		2.70
Object Reuse	15,433,805,000	9.12	89.03	2.81

Table 1. Micro-benchmarks evaluation

a specific process is used for filtering the applications with Bad smells which are evaluated. After choosing the applications, proper refactorings were applied and, then, the measurements were performed to compare the different versions of the same application w.r.t. battery usage.

Next sections are dedicated to describe the experiments, and to this end these sections present the applications that were used during the experiments (Section 3.1), the mechanism used for taking measures and experimental scenarios (Section 3.2), and the obtained results (Section 3.3).

3.1 Test applications

One of the main goals of this paper is to evaluate the impact of a set of refactorings on battery consumption in real mobile applications. Then, a set of applications that cover all the Bad smells and refactorings presented in Section 2.1 was chosen. Once the set of applications was chosen, a new version of these applications was developed by applying the corresponding refactoring(s). Finally, using a special tool, the battery consumption of each version was measured for comparison purposes. This methodology allows us to objectively analyze the improvements that can be made on an application.

To determine which applications to use, a survey of existing applications in the Android Market was made taking into account which are open-source and could be modified. To cover different types of applications with different features, games and scientific applications were chosen. The resulting applications from the survey were three: Fivestones, Sorter and Apps. The first two are games while the latter is a scientific application used in [15]. Each of these applications was analyzed to observe the amount of Bad smells present in the code, taking into account some of the Bad smells mentioned in the previous Section, i.e., *God Class* and *Brain Method*, which are in practice the most recurrent. Bad smell counting was performed using JDeodorant [4]. Table 2 shows the study of each application, indicating the amount of Bad smells. Some applications are presented twice in the table because there are two independent algorithms implemented within the same application.

3.2 Preferences analysis

Before applying the refactorings it is important to evaluate not only the quantity of Bad smells but also the accumulated time and number of times that the methods which contains Bad smells are executed. This is because a refactoring applied in a Bad smell which

Application	God Class	Brain Method
FiveStones	15	33
Sorter	1	8
Apps - Knapsack	2	3
Apps - Matrix Multiplication	2	3

Table 2. Test applications: Bad smells

is executed 99% of the total time has more impact than numerous refactorings applied in Bad smell that are executed only 1% of the total time. For this, two automatic and specialized tools were necessary: first, JDeodorant was used to detect the Bad smell and the methods that contain them; second, an ad-hoc application (*Tracer*) was used to measure the time execution and the number of invocations of each method. After data collection it was necessary to evaluate which methods will be refactored taking into account the effort necessary and the trade-off between object-oriented design and battery consumption.

Data collection Tables 3 and 4 show an example of the report obtained for the application Sorter. The first Table shows the report obtained via JDeodorant of each of the code smells detected and the affected entity, with their respective ranking ordered by the potential impact on object-oriented quality of its refactoring. In the second Table, all information relevant to the message flow of the application at runtime obtained via the Tracer is shown. In this case, the reader can see, for example, that the *makeShader* method is the most invoked one, with more than 50% as depicted in the "Invocation (%)" column.

Code smell	Entity	Ranking
God Class	eu.danielwhite.sorter.components.SortView	1
BrainMethod	eu.danielwhite.sorter.SortCompare.randomlyPermute(java.lang.Integer[])::void	1
BrainMethod	eu.danielwhite.sorter.algos.SelectionSorter.run()::void	2
BrainMethod	eu.danielwhite.sorter.SortCompare.initStartingState()::void	3
BrainMethod	eu.danielwhite.sorter.SortCompare.onCreate(android.os.Bundle)::void	4
BrainMethod	eu.danielwhite.sorter.components.SortView.onDraw(android.graphics.Canvas)::void	5
BrainMethod	eu.danielwhite.sorter.algos.QuickSorter private.qSort(int, int)::void	6
BrainMethod	eu.danielwhite.sorter.algos.MergeSorter.mergeSubArray(int, int, int, int)::void	7
BrainMethod	eu.danielwhite.sorter.algos.BubbleSorter.run()::void	8
No Self-Encapsulated Field	eu.danielwhite.sorter.algos.Sorter.mFinished	1
No Self-Encapsulated Field	eu.danielwhite.sorter.algos.Sorter.mData	2
...	...	...

Table 3. Bad smell report for the Sorter application

Method	Accumulated invocation time (ms)	Number of invocations (%)	Total invocations (accumulated %)
...	...	...	...
Sorter.fireSorterDataChange(void):void	705	1,1628	4,5885
EventListener. sorterDataChange(SorterEvent<Integer>):void	705	1,1628	5,7513
SortCompare.run(void):void	711	1,1727	6,9240
SortCompare.refreshList(SortView, Sorter<Integer>):void	721	1,1892	8,1131
SortView.setSorter(Sorter<Integer>):void	721	1,1892	9,3023
Sorter.setDataVal(int, T, boolean):void	1357	2,2382	11,5405
Sorter.getData(void):T[]	1405	2,3173	13,8578
SortView.onDraw(Canvas):void	1405	2,3173	16,1752
Sorter.compareData(T, T):int	1580	2,6060	18,7811
Sorter.doSleepDelay(long):void	3074	5,0701	23,8512
Sorter.getDataVal(int):T	4019	6,6287	30,4800
SortView.makeShader(int, float):Shader	42150	69,5200	100,0000

Table 4. Message flow report for the Sorter application

Method selection Once the report with all relevant information to the different applications was obtained, developers have to apply a filter in each of them to determine which of the Bad smells spotted should be refactored. Indeed, in practice, under a scenario where a balance between battery consumption and object-oriented design is desired, developers would refactor code selectively. Moreover, to consider different scenarios, two different broad scenarios with different strategies and preferences were defined:

Object oriented scenario The first considered scenario is the one in which the quality of the object-oriented design is prioritized, eliminating most important Bad smells. Therefore, all *God classes* and *Brain methods* that appear in the 90% of the most invoked methods should be removed. This is because it is considered that these are the Bad smells that influence more in the implementation of methods and generate a direct negative impact on the design.

Battery consumption oriented scenario The second case is the opposite scenario, in which a developer tries to reduce battery consumption of the application as much as possible. Since battery consumption is influenced directly by the message flow and object creations, all Bad smells present in the code were left unsolved under this scenario. Moreover, the accessor methods within the 95% of the most invoked methods already present in the code were removed, i.e., Bad smells, namely No Encapsulated Field and No Self-Encapsulated Field, were introduced to better reflect the impact on battery reduction.

3.3 Results

Using an external tool called Robotium [14] to define different test cases, numerical and exact results were obtained. Robotium is an Android test automation framework that allows developers to write and excecute tests for Android applications. With Robotium the

experiments were executed several times to demonstrate the results. These results allows us to evaluate the effort and trade-off of applying refactorings in the applications. Table 5 shows the consumption measured by the Power Monitor tool for each application. The first column of the table specifies the name application, the second shows the version of it (considering the different scenarios presented in Section 3.2), the third shows the execution time, the fifth column presents the Power consumed in average per unit time, the seventh column shows the average of power consumption by unit of time, the fourth and sixth column shows the standard deviation of time and power measures respectively, and, finally, the last column represents the energy gain of the different versions of the same application w.r.t. the original version. The experiments were performed on a Samsung Galaxy SIII, with the following characteristics: Quad-core 1.4 GHz Cortex-A9 CPU, 2 GB RAM, internal 16GB of storage and lithium ion battery of 2,100 mAh.

To analyze the results it is important to mention that the total battery consumption is calculated with the following formula: $consumption = time * power$, where $time$ refers to the execution time of the test case and $power$ refers to the average of watts sampled per unit of time during the execution. It is worth noting that a sample frequency of 5Khz was considered. Taking into account these values, developers have to know that an application can reduce its battery consumption either by reducing its execution time or its energy consumption per unit of time.

Now, analyzing the results, in general, support the hypothesis presented at the beginning of this paper. First, for the efficient version of FiveStones, in which some access methods are removed, the battery consumption was reduced in a 6.896%. In Sorter this percentage was even larger, i.e. around 35.096%. Moreover, the applications Knapsack and Matrix Multiplication were not significantly modified, then, these applications did not show improvement for this scenario (only for Knapsack the consumption was reduced in a 1.578%). All reductions are relative to the performance of the original versions of the applications.

In the object-oriented scenario the battery consumption increases because there are more objects and interaction between objects. Taking Sorter in consideration, the power increases in a 8.673%. In addition, the results of FiveStones have also an important difference since consumption is increased by 23.725% in the object-oriented scenario. However, the most significant difference resulted in the case of the scientific applications, where the percentage increases up to 70.536%. One of the main reasons for this difference in the percentages between GUI-driven and scientific applications is that the graphical interface is an important component that consumes large amount of battery power [1].

In summary, although not generalizable, the results support the original hypothesis. That is, an object-oriented design tends to carry more battery consumption than a design with less objects and messages between them. Then, when developers try to achieve a better object-oriented design, this may generate an impact on battery consumption, and viceversa.

4 Conclusion

This paper analyzed the trade-off between having a high-quality object-oriented application and having an application that is more energy-efficient. The results presented by different works show that developing a good mobile application is not a trivial task since it is important to evaluate the object-oriented advantages to achieve a good balance with the battery consumption. Basically, this is an important topic since object-oriented programming is one of the most used paradigms in application development, which includes

Application	Version	Time	Standard deviation (%)	Power	Standard deviation (%)	Consumption (J)	Improvement (%)
FiveStones	Battery consumption oriented	24.164	1.675	0.168	2.259	4.066	-6.896
	Original	24.208	2.236	0.179	1.388	4.342	0
	Object-oriented	23.945	2.960	0.237	2.457	5.694	23.725
Sorter	Battery consumption oriented	24.164	1.675	0.168	2.259	4.066	-35.096
	Original	16.166	7.436	0.339	1.726	5.493	0
	Object-oriented	16.137	8.239	0.372	1.785	6.015	8.673
Apps - Knapsack	Battery consumption oriented	4.389	6.291	1.204	2.684	5.281	-1.578
	Original	4.698	5.625	1.143	3.830	5.364	0
	Object-oriented	12.447	3.810	1.161	3.021	14.441	62.852
Apps - Matrix Multiplication	Battery consumption oriented	7.690	0.933	1.680	0.815	12.920	N/A
	Original	7.690	0.933	1.680	0.815	12.920	0
	Object-oriented	26.149	0.370	1.677	0.497	43.854	70.536

Table 5. Battery consumption results

practices that significantly improve the level of modificability, understandability, maintainability, among other attributes of the code. However, a correct use of the paradigm requires the presence of many classes and methods to have a good distribution of functionality, but these features impact negatively on the level of battery consumption of the application, as shown in [15]. The main reason of this evaluation is that battery life is an important feature for mobile users, then, if they install an application that consume a lot of battery probably they will not be confortable with it. For this paper, it was considered that a good object oriented design meant a program without bad smells.

Particularly, the results show that assuring important features of object-oriented design, such as maintainability and flexibility, through removing Bad smells increase battery consumption. As a consequence, developers should evaluate which features can be relaxed to prioritize battery consumption. Even though this paper does not present a methodology to deal with this trade-off, the steps used to demonstrate how some refactorings have an important impact on battery consumption can be repeated and used by developers to design and develop their own applications while considering these issues. The aim of these steps is to give basic hints to the developer to balance the number of classes and methods that applications have to implement to do not consume a lot of battery. In detail, the process behind such steps is divided into several stages, each with a very specific function. In the first stage an analysis of the applications detects the different Bad smells. As explained, these Bad smells are practices that are opposite to the good practices for object-oriented programming. After that, developers have to decide their preferences to allow a "good" balance –according to their needs– between object-oriented design quality and battery consumption. In this case two quite extreme scenarios were presented, but it is possible to define a middle scenario in which for example the God Class and Brain Method bad smells are removed, but accessors are left. That would be the case for example of ensuring the code adheres to the JavaBeans[6] standard. In this line, to what extent OO purity and/or energy efficiency are considered is up to the developer since this decision depends on application requierements. Thirdly, the classes and methods to be changed have to be determined. The last stage is to perform the refactorings. Finally, taking into account the experiments presented, developers can quantify the effort and gains of applying some refactorings in their applications.

5 Future works

By looking at the steps to make refactorings, and considering the obtained results, it is worth mentioning the various research directions that can be followed in future. Firstly, as it was mentioned before, the graphical interface consumes a large amount of energy and because of that the difference between the consumption of the two versions was not very high in the case of the games applications. Therefore, we plan to test this approach in server applications as they have no graphical interface and usually they use the processor in a more intensive way. Then, a small difference in energy consumption could result in a big difference in the whole consumption. This is even more important in cluster environments, since there might be a lot of computers executing the same (unoptimized) code. For that reason, a device has just been purchased [10], which measures the energy consumption of PCs in the same way PowerMonitor does for mobile devices, i.e., directly from the power source. In addition to that, to make it easier to find applications that present all the combination of existing code smells and hence generalize the results,

[6] Java beans specification

we plan to implement a tool that automatically injects different types of code smells following the same techniques implemented by fault injection tools, an approach commonly used in sofware testing research.

Another possible work is to consider a different way to "measure" the level of good object oriented design. For that purpose, traditional OO patterns (e.g., those from the Gang of Four) will be considered to remove bad smells, instead of Fowler's refactorings. We will also analyse if there are patterns that can improve both the object oriented design and the battery consumption. Our motivation is that these patterns not only provide high-quality OO design structures but also in some cases –such as the Flyweight pattern– they might be energy-friendly.

Finally, since we determined that JDeodorant had some mistakes when detecting code smells, another tools for the detection of code smells will be tested. In particular, we plan to use JSpirit [5], which not only provides several kind of smell rankings using different criteria, but also detects many other code smells such as Data Class, Brain Class or Intensive Coupling. What is more, JSpirit is a plugin for Eclipse, so that the installation and use of the tool would be straightforward.

Acknowledgements

We acknowledge the financial support provided by ANPCyT through grant PICT-2012-0045.

References

1. Aaron Carroll and Gernot Heiser. An analysis of power consumption in a smartphone. In *USENIX annual technical conference*, pages 271–285, 2010.
2. B. Flipsen, J. Geraedts, A. Reinders, C. Bakker, I. Dafnomilis, and A. Gudadhe. Environmental sizing of smartphone batteries. In *Electronics Goes Green 2012*, pages 1–9, 2012.
3. Martin Fowler. Refactoring: Improving the design of existing code, 1999.
4. JDeodorant. http://www.jdeodorant.com/.
5. JSpirit. https://sites.google.com/site/santiagoavidal/projects/jspirit.
6. Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice - Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems.* Springer, 2006.
7. Cristian Mateos, Alejandro Zunino, and Marcelo Campo. An approach for non-intrusively adding malleable fork/join parallelism into ordinary JavaBean compliant applications. *Computer Languages, Systems and Structures*, 36(3):288–315, 2010.
8. Cristian Mateos, Alejandro Zunino, and Marcelo Campo. On the evaluation of gridification effort and runtime aspects of JGRIM applications. *Future Generation Computer Systems*, 26(6):797–819, 2010.
9. Cristian Mateos, Alejandro Zunino, Ramiro Trachsel, and Marcelo Campo. A novel mechanism for gridification of compiled java applications. *Computing and Informatics*, 30(6):1259–1285, 2011.
10. Power Meter. http://power-meter.com.ar/rs232.html.
11. J. A. Paradiso and T. Starner. Energy scavenging for mobile and wireless electronics. *IEEE Persvasive Computing*, 4:18–27, 2005.
12. R. Perez-Castillo and M. Piattini. Analyzing the harmful effect of god class refactoring on power consumption. *Software, IEEE*, 31(3):48–54, May 2014.

13. PowerMonitor. https://www.msoon.com/labequipment/powermonitor/.
14. Robotium. https://code.google.com/p/robotium/.
15. Ana Victoria Rodríguez, Cristian Mateos, and Alejandro Zunino. Mobile devices-aware refactorings for scientific computational kernels. In *41 JAIIO - AST 2012*, pages 61–72, 2012.
16. Guillermo L. Taboada, Sabela Ramos, Roberto R. Exposito, Juan Tourino, and Ramon Doallo. Java in the high performance computing arena: Research, practice and experience. *Science of Computer Programming*, vv:pp, 2011.