On Energy Debt

Managing Consumption on Evolving Software

Marco Couto, Daniel Maia, João Saraiva
HasLab/INESC TEC & Universidade do Minho
marco.l.couto@inesctec.pt
a77531@alunos.uminho.pt
saraiva@di.uminho.pt

Rui Pereira
C4 - Centro de Competências em Cloud Computing
(C4-UBI) Universidade da Beira Interior
Rua Marquês d'Ávila e Bolama, Covilhã
ruipereira@di.uminho.pt

ABSTRACT

This paper introduces the concept of energy debt: a new metric, reflecting the implied cost in terms of energy consumption over time, of choosing a flawed implementation of a software system rather than a more robust, yet possibly time consuming, approach. A flawed implementation is considered to contain code smells, known to have a negative influence on the energy consumption.

Similar to technical debt, if energy debt is not properly addressed, it can accumulate an energy "interest". This interest will keep increasing as new versions of the software are released, and eventually reach a point where the interest will be higher than the initial energy debt. Addressing the issues/smells at such a point can remove energy debt, at the cost of having already consumed a significant amount of energy which can translate into high costs. We present all underlying concepts of energy debt, bridging the connection with the existing concept of technical debt and show how to compute the energy debt through a motivational example.

CCS CONCEPTS

• **Software and its engineering** → **Automated static analysis**; **Software performance**.

KEYWORDS

Green Software, Energy Debt, Code Analysis

ACM Reference Format:
Marco Couto, Daniel Maia, João Saraiva and Rui Pereira. 2020. On Energy Debt: Managing Consumption on Evolving Software. In *International Conference on Technical Debt (TechDebt '20), October 8–9, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3387906.3388628

1 INTRODUCTION

Technical Debt (TD) describes the gap between the current state of a software system and the ideal state of that same software. The key idea of technical debt is that software systems may include artifacts which can be hard to understand/maintain/evolve, causing higher costs in the future software development and maintenance activities. These extra costs can be seen as a type of debt that developers owe the software system.

Although technical debt is still a recent area of research, it has gained significant attention over the past years: A recent systematic mapping study [16] identified ten different types of technical debt, namely *requirements*, *architectural*, *design*, *code*, *test*, *build*, *documentation*, *infrastructure*, *versioning*, and *defects* technical debt. In fact, TD is a concern both for researchers and software developers. The current widespread use of non-wired computing devices is also making energy consumption a key aspect not only for hardware manufacturers, but also for researchers and software developers [27]. Indeed, several *energy inefficient* programming practices have been reported in literature, namely, energy patterns for mobile applications [5, 7], the energy impact of code smells [20, 21, 28], energy-greedy API usage patterns [18], energy (inefficient) data structures [24], programming languages [25], etc. which do have significant impact on the energy consumption of software.

All these research works show that energy-greedy programming practices, also called energy smells, do often occur in software systems. These can be attributed to the current lack of knowledge software developers have in order to build energy efficient software, and the lack of supporting tools [27].

This paper defines *energy debt* as the additional estimated energy cost of executing a software system, due to the occurrence of energy smells in the software's source code, when compared to the estimated energy cost of executing the non-energy smelly (*i.e.* energy ideal) version of that same software. To express energy debt we consider a set of energy code smells presented in the current state of the art literature on green software, together with the energy savings reported in the studies where such smells have been presented. Thus, the energy debt of a program is computed after knowing the number of occurrences and their locations in the program's source code: energy smells inside loops/recursion, single statements, or inside dead code do have different debt weights.

This paper is structured as follows: Section 2 thoroughly describes the notion of our novel concept of energy debt, and how it should be expressed/calculated; Section 3 presents the related work; finally, our conclusions and future work are included in Section 4.

2 INTRODUCING ENERGY DEBT CONCEPTS

In this section, we will explain a novel concept called energy debt, which is very much aligned with technical debt in the sense that it presents developers and decision makers with information regarding the evolution of energy inefficiency of their software systems. As presented in a recent ACM communications [27], developers fall into energy-greedy practices and tendencies due to the lack of

TechDebt '20, October 8–9, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7960-1/20/05…$15.00
https://doi.org/10.1145/3387906.3388628

knowledge and the lack of tools to help understand, locate, and optimize energy inefficiencies. Additionally, other practicioners and decision makers also lack the necessary tools to help interpret how energy inefficiencies can impact their product lifecycle, and what they should focus on tackling in order to reduce energy costs [19].

2.1 Concept Overview

Before we present the definition of energy debt, let us recall the metaphor of technical debt. Technical debt reflects the cost arising from performing additional work on a software system, due to developers taking "shortcuts that fall short of best practices" [2]. Hence, this cost can be defined as the technical effort, in working hours, required for fixing all issues associated with bad programming practices, in a given release. The cost keeps increasing, as new versions (with new issues) keep getting released, and if the initial issues are not properly addressed, they accumulate *interest* [4].

Based on the underlying concept of technical debt, we define **Energy Debt** as *the amount of unnecessary energy that a software system uses over time, due to maintaining energy code smells for sustained periods.*

A visual comparison of the two concepts is depicted in Figure 1. The left-hand side of the figure illustrates the well-known representation of technical debt, including the concepts of refactoring and maintenance effort, along with the definition of interest. On the right-hand side we present the definition of energy debt, where we assume that evolving the software (i.e., introducing new features on new releases) will eventually result in the addition of new (energy) code smells, hence the Energy Debt (*ED*) increases per version.

The main difference between technical and energy debt, at this point, is the fact that the former can be presented as a unique cost value expressing how much effort would be necessary to address the issues, whereas the same approach cannot be applied to the latter. The cost of maintaining energy code smells in a software release is always directly proportional to the amount of time that the same release operates. As an example, if two software systems S_1 and S_2 have the exact same energy code smells, the amount of excessive energy consumed by S_1 might be higher than S_2 if it is intended to be used longer, during the same timespan.

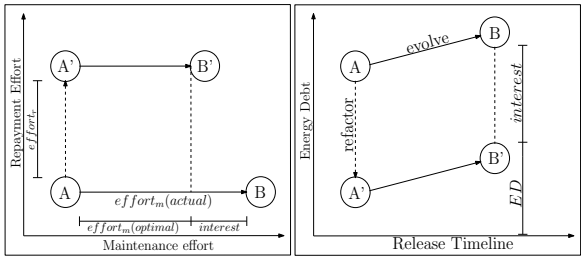

Figure 1: Technical Debt vs Energy Debt Terminology

Given the previous assumptions, we argue that the energy debt *ED* of a software release must be expressed not as a cost value, but as a cost function, which receives, as input, two variables: a software release r, and a usage time t. Equation 1 defines such a function, and it allows us to obtain, for a given release r, its energy debt *ED* after a given usage time of t:

$$ed(r,t) = cost(r) * t \tag{1}$$

The $cost(r)$ function included in the equation represents the energy cost of release r, per unit of time. In other words, it relates to the existing number of energy code smells in that version, and the energy cost (per unit of time) of each one. The definition of that function is expressed as Equation 2:

$$cost(r) = \sum_{i=1}^{N} w_i(r) \times E(i) \tag{2}$$

Here, N is the number of smells included in the considered catalog, while $w_i(r)$ returns a weight value for smell i, which is affected by the number of i smells found in release r and the context in which they were found (we will discuss this with greater detail in Section 2.2). $E(i)$ returns the expected energy debt per time unit of smell i, as defined in the smell catalog.

The formulas presented thus far assume that each considered energy code smell has an associated energy debt value, expressed in function of time units (for instance, per minute). Nevertheless, when studying the energy consumption impact of code smells, researchers often tend to present the potential gains/savings as an interval (i.e., highest and lowest observed energy saving).

The highest/lowest saving approach adds valuable information regarding potential energy savings. Fixing a certain smell can result in savings between, e.g., 150mJ and 3000 mJ per minute. When compared to another smell with savings between 300 mJ and 900 mJ per minute, we know that in a best-case scenario refactoring the first one would result in higher gains, but in a worst-case scenario the second presents better savings. Hence, a developer can use this information to decide how to properly focus their attention when refactoring code smells, depending on the project goals [5].

In accordance with the previous assumption, we decided that our approach for energy debt should consider, for each code smell, two energy values: the highest (E_{max}) and lowest (E_{min}) observed energy savings. Since energy debt must be expressed in a function of the usage time, it is expected that E_{max} will be much higher with the increase of usage time, as depicted in Figure 2.

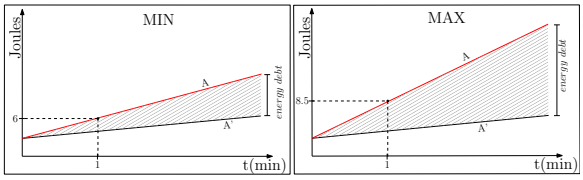

Figure 2: Energy Debt Thresholds Increase Over Time

There are two represented versions of a release in this figure: the optimal version, with all smells removed (A'), and the *energy smelly* version (A). The optimal version already has a constantly increasing energy consumption, as it would be expected. Energy debt can be summed up as the area between the line for A', and the (red) line for A, which becomes much larger when considering the maximum values. This will introduce changes to Equation 1, which will consider two cost values, in the form of two functions:

$$ed(r,t) = \Big(cost_{min}(r) \times t; cost_{max}(r) \times t\Big) \tag{3}$$

The energy debt will therefore always be presented in the form of an interval. Consequently, each of the cost functions will need to refer to the proper energy debt per time unit. In other words, the

$E(i)$ function in Equation 2 will be $E_{min}(i)$ for the lowest savings, and $E_{max}(i)$ for the highest savings.

Finally, as previously stated, an energy smell catalog is necessary in order to estimate energy debt. Such information is already present within state-of-the-art research works [5, 7, 20, 21, 28], which report the energy impact of different smells and the associated maximum and minimum cost (or potential savings) per time unit, which can be mapped directly into the energy debt estimation.

2.2 Counting Expenses & Estimating Debt

The next step towards estimating energy debt is to define a strategy to analyze the occurrence of such smells in a given release. The starting point for this task will be to use a common source code analysis tool capable of detecting code smells. There are several ways to achieve this. For instance, *SonarQube*, which is a widely used tool for technical debt estimation, provides an API for defining detection rules for issues/smells of different languages.

Detecting smell occurrences, however, is a necessary but not the sole requirement to properly analyze its impact on energy debt. A smell can be detected, for instance, inside a block of dead/unreachable code, or it can be placed inside a procedure which may only be executed once in a software lifecycle (eg. an initial setup). On the other hand, a code smell can also be part of a mechanism designed to be re-utilized several times, such as a loop or a thread. These scenarios should be considered when estimating energy debt, and since our approach implies using statical analysis mechanisms, we can follow already defined strategies for static energy analysis.

A very common and well-established approach for these situations is to define weights for smells, depending on the context on which they occur. For instance, Jabbarvand et al. [13] defined a strategy for weighing instructions which might be repeated. First, it extracts the full method *call graph* of a program, and provide for each method an energy score; such a score depends on 3 things: *(i)* how many paths can be taken to reach that node from the root node, *(ii)* whether it is found inside a loop, and if so *(iii)* what is the expected loop's bound; this statically obtained information is then used to increase/decrease the energy score of the node.

Several strategies have been suggested for this task, all of which accepted by the community. This leads us to believe that, although it is important to weight code smells depending on the occurrence context, several factors can influence the decision on what approach to follow (e.g. how much information is extracted from the smell detection tool, or trading off information detail with the analysis time). Hence, we argue that the selected strategy is also context dependent, and can be as simple or as detailed as desired. Nevertheless, whatever approach one follows, an update to Equation 2 is necessary to consider it. As an example, we considered a simplification of the strategy from Jabbarvand et al. [13]:

$$w(i, r) = \sum_{j=1}^{C} paths(j) \times LB \tag{4}$$

In this equation, *(i)* C is the number of i smells found in the release r; *(ii)* $paths(j)$ represents the number of paths in the *call graph* through which the j^{th} occurrence of smell i is reachable; *(iii)* LB will be 1 **if** the j^{th} of the smell is outside a loop, or a constant indicating the loop bound; it can be inferred if possible, or pre-established.

In order to better explain how all these concepts connect with each other, when aiming at estimating the energy debt of different software releases, we have prepared a running example, depicted in Figure 3. In this example, we have a catalog with 3 smells, each one with the energy gains thresholds defined (values are in milliJoules per minute), and 3 releases with the analysis report for each. The report is a list of the detected smells, where for each one there is information regarding *(i)* the number of paths through which the smell is reachable (*paths*), and *(ii)* whether it was found inside a loop ($LB > 1$) or not ($LB = 1$).

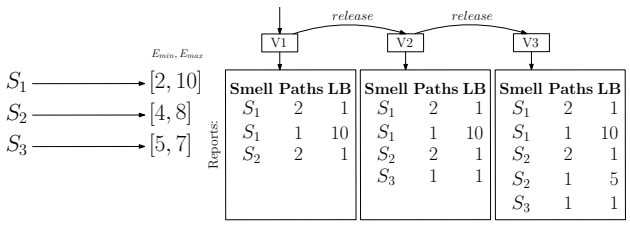

Figure 3: Estimating Energy Debt per Release

Using the formula from Equation 4, we can determine the weight to be applied to each smell. For example, for release $v1$, the weights for smell $s1$ and $s2$ would be:

$$w(s1, v1) = (2 \times 1) + (1 \times 10) = 12$$
$$w(s2, v1) = (2 \times 1) = 2$$

We can apply the computed weights in the formula from Equation 2, to obtain an estimated value for the energy debt of release $v1$. As previously mentioned, our energy debt definition considers two reference values: the lowest and highest estimated energy debt. This means that the *cost* function in Equation 2 must be computed twice: the first using the lowest estimated gains per smell (E_{min}), and the second using the highest (E_{max}). Once again, for release $v1$, we would have the following $cost_{min}$ and $cost_{max}$ values:

$$cost_{min}(v1) = (w(s1, v1) \times E_{min}(s1)) + (w(s2, v1) \times E_{min}(s2))$$
$$\Leftrightarrow cost_{min}(v1) = (12 \times 2) + (2 \times 4) = 32$$

$$cost_{max}(v1) = (w(s1, v1) \times E_{max}(s1)) + (w(s2, v1) \times E_{max}(s2))$$
$$\Leftrightarrow cost_{max}(v1) = (12 \times 10) + (2 \times 8) = 136$$

These two reference values represent the energy debt for release $v1$. This means that energy debt can vary from a minimum of 32 to a maximum of 136 milliJoules per minute. As explained previously, energy debt is expressed as a function of usage time. Therefore, if one wants to know how much debt this release accumulates after being used for, e.g., one hour, this can be estimated as follows:

$$ed(v1, 60min) = \Big(cost_{min}(v1) \times 60; cost_{max}(v1) \times 60\Big)$$
$$\Leftrightarrow ed(v1, 60min) = \Big(1,920mJ; 8,160mJ\Big)$$

The estimated values indicate an energy debt varying between 1.92 and 8.16 Joules per hour. This means that, for every hour that release $v1$ is being used, it could be consuming **at least** $1.92J$ less, and the savings could be **up to** $8.16J$.

Finally, it is important to point out that the accuracy of the estimated thresholds rely on the adequacy/robustness of the analysis components, namely the smells catalog, the code analysis tool, and the weighing function for repeated smell's executions. It is possible

to use our energy debt approach to compute reference values for the energy inefficiency of a release, rather then to produce extremely accurate estimates of the potential energy savings per usage time. It depends on how one wants to apply the concept.

2.3 Paying Interests

The concept of *interest* in technical debt has already been formulated [4], and its practical application has also been studied [1, 3, 30]. The concept is based on the fact that, as a software system evolves (i.e., new versions are released), the cost/effort of adding features to a new release (*maintenance effort*, expressed as working hours) keeps increasing if the task of addressing the technical debt keeps being postponed. Maintaining a release with technical debt requires more effort than to maintain the same release without it; the effort difference between the two is called the *technical debt interest*.

The left-hand side of Figure 1 illustrates the interest concept. There is a software version, A, containing code smells, and therefore technical debt. At this point, a decision can be made on what to prioritize: *(i)* invest effort in fixing the smells (*repayment effort*) and release an optimal version of that release, A', without technical debt, or *(ii)* release the version with the smells. If the priority is *(ii)*, then the evolution effort to a new release B will be higher. This additional effort could be avoided, but the priority was releasing a new version, which can happen for a wide variety of reasons (e.g. client demands, faster market reach, etc.); this resembles the idea of accumulation of debt, and debt needs to be re-payed.

Chatzigeorgiou et al. [4] presented a technique to predict the technical debt *breaking point*, i.e., when the accumulated interest is higher than the initial effort to remove the technical debt (i.e., the *principal*). With this, it is possible to present developers another choice: if technical debt keeps being "ignored", then they have approximately until release number N to properly deal with it; otherwise, from that moment on, the additional *maintenance effort* will always be higher than the effort to deal with the *principal*. In conclusion, at that point they are wasting development time.

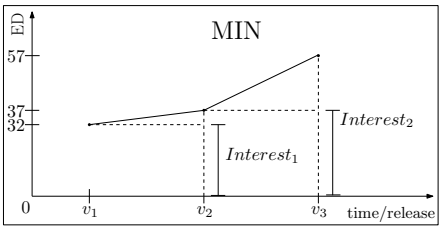

Figure 4: Accumulation of Interest

When considering energy instead of technical debt, the interest concept needs another definition. First, it will not indicate how much more maintenance effort is being applied, since energy debt does not measure effort, but the drainage of a resource. In that sense, *energy debt interest* is the amount of excessive energy consumed over time, that could be avoided if the issues were properly addressed earlier. In simple terms, it is the accumulated energy debt after n releases. This concept complements energy debt in the sense that it can be used to estimate the "real-world" cost of not fixing the smells, which can be monetary (as energy costs money) or uptime related (if the analyzed software is targeted for IoT/mobile devices).

Figure 4 illustrates our perception of energy interest, using our example from Figure 3. Again, 3 software releases are considered: $v1$, $v2$, and $v3$[1]. For this example, we are assuming that, as new versions are released, the issues from previous versions were not addressed. Hence, *energy debt* is always increasing. For $v1$, we consider that no interest was accumulated, due to the fact that energy debt depends on usage time. Hence, at the exact instant when the version was released, it was never used.

For release $v2$, we know that it added a new smell, $s3$. If **all** smells from the previous version ($v1$) were fixed upon release, then this $v2$'s minimum energy debt would be 5. However, since $v1$ contained smells, from the time interval comprised between the two releases, the software was excessively consuming 32 mJ for each minute it was being used (i.e., the energy debt from $v1$). Therefore, for release $v2$, the accumulated debt (i.e., the interest) is 32 mJ per minute.

When considering release $v3$, however, the reasoning to infer the interest requires adjustments. For once, $v3$ has two predecessors, while $v2$ has only one. Between $v1$ and $v2$, the energy debt was 32, and between $v2$ and $v3$ it was 37. To estimate how much debt was accumulated, we should infer a value based on the two. One possible way to tackle this is to compute the average of **all** energy debts from previous releases. In this particular case, the minimum accumulated interest would be $\frac{(32+37)}{2} = 34.5$.

Finally, it is important to interpret interest similarly to how energy debt is interpreted: a minimum/maximum energy being excessively consumed **per usage time**. Hence, we argue that, when considering the interest for the n^{th} release, the expected usage time should be higher than the one for any previous release. Therefore it is guarantee that, even though energy debt is reduced from one release to another, the interest will be inflated for later releases.

3 RELATED WORK

Technical debt is a term which refers to the pitfalls of creating sub-optimal software to fit a shorter interval, introduced by Cunningham [9]. As software evolves, it's liable to take on debt from several sources: "technological obsolescence, change of environment, rapid commercial success, advent of new and better technologies, and so on — in other words, the invisible aspects of natural software aging and evolution." [14]. As already known, allowing technical debt to continuously build up without a level of debt management raises the risk of producing unmanageable and inefficient code, which can hamper the addition of new or updating existing functionalities. Thus, the longer such code goes unattended, the more resources will be needed to correct it and with diminishing returns [4].

One such inefficiency in software is of high energy consumption. In fact, the energy efficiency of software has become a vey active research field. Studies have shown that developers are aware of the energy consumption problem, and often times seek help in solving such issues [27]. Currently, there is a broad range of work done on understanding what aspects in programming languages can contribute to high energy costs such as different data structures [11, 17, 24, 26], languages [25], or design patterns [29]. Specific to the Android ecosystem, there has been research in topics such as the classification of Android applications as being more/less energy efficient [13], identifying energy green APIs [18], estimating energy

[1]The presented values refer to the minimum estimated debt

consumption in code fragments [6], etc. In fact, energy efficiency in Android is a very active area of research [5, 7, 20, 21, 28]. The results of most of these studies are able to quickly translate into our energy smell catalog to be used in the calculation of energy debt.

Additionally, much research has been conducted in providing several approaches to the measurement of energy consumption. For example, for Android energy analysis there is *eCalc* [10], *vLens* [15], *eProf* [22], or *Trepn* [12, 13]. Nevertheless, there is evidence that relying only on profilers and measuring tools are not enough to locate efficiency problems [23]. There is also work in automatic tools to help detect energy greedy code spots [23], refactoring for the most energy efficient data structure [26], or refactoring energy greedy Android patterns [5, 8]. These works, however, do not yet translate their potential gains across a period of time into the actual energy savings a developer or business can have on the software by applying such transformations. It is our belief that, our work closes this gap in not only knowing if an alternative solution is more energy efficient, but by how much can we save (in energy/money) over time if and when we adopt the energy efficient alternative.

4 CONCLUSIONS AND FUTURE WORK

This paper presented the concept of energy debt as the additional energy cost over time of a software system due to the occurrences of energy code smells in its source code. It is expressed as a function considering *(i)* the number of smells, *(ii)* the context in which they were detected, and *(iii)* the expected usage time of the application. Energy debt interest is also expressed as the accumulation of energy debt per release, which could be avoided by eliminating energy smells in previous releases.

Currently, we are concluding the construction of a catalog of reported state-of-the-art energy code smells, and their known energy costs per usage time, which can be considered when calculating energy debt. Additionally, an extension of the concept of energy debt is being developed within the *SonarQube* framework, where it will support the inference of the context and number of detected smells, based on our catalog.

ACKNOWLEDGMENTS

This work is financed by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, within project UIDB/50014/2020. The first author is also financed by FCT grant SFRH/BD/132485/2017. The last author is also supported by operation Centro-01-0145-FEDER-000019 - C4 - Centro de Competências em Cloud Computing, cofinanced by the European Regional Development Fund (ERDF) through the Programa Operacional Regional do Centro (Centro 2020), in the scope of the Sistema de Apoio à Investigação Científica e Tecnológica - Programas Integrados de IC&DT.

REFERENCES

[1] Areti A., A. Ampatzoglou, P. Avgeriou, and A. Chatzigeorgiou. 2015. Establishing a Framework for Managing Interest in Technical Debt. In *Proc. of the 5th Int. Symposium on Business Modeling and Software Design*, Vol. 1. SciTePress, 75–85.

[2] Eric Allman. 2012. *Managing Technical Debt*. Vol. 55. ACM. 50–55 pages.

[3] A. Ampatzoglou, A. Michailidis, C. Sarikyriakidis, A. Ampatzoglou, A. Chatzigeorgiou, and P. Avgeriou. 2018. A Framework for Managing Interest in Technical Debt: An Industrial Validation. In *Proceedings of the 2018 International Conference on Technical Debt (TechDebt '18)*. ACM, 115–124.

[4] A. Chatzigeorgiou, A. Ampatzoglou, A. Ampatzoglou, and T. Amanatidis. 2015. Estimating the breaking point for technical debt. In *2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)*. 53–56.

[5] M. Couto, J. P. Fernandes, and J. Saraiva. 2020. Energy Refactorings for Android in the Large and in the Wild. In *2020 IEEE 27th Int. Conference on Software Analysis, Evolution and Reengineering (SANER)*. London, Ontario, Canada.

[6] M. Couto, Carção T., J. Cunha, J. P. Fernandes, and J. Saraiva. 2014. Detecting Anomalous Energy Consumption in Android Applications. In *Programming Languages*, Fernando M. Quintão Pereira (Ed.). LNCS, Vol. 8771. Springer, 77–91.

[7] Luis Cruz and Rui Abreu. 2017. Performance-based Guidelines for Energy Efficient Mobile Applications. In *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft '17)*. IEEE Press, 46–57.

[8] Luis Cruz and Rui Abreu. 2018. Using Automatic Refactoring to Improve Energy Efficiency of Android Apps. *CoRR* abs/1803.05889 (2018).

[9] Ward Cunningham. 1992. The WyCash Portfolio Management System.

[10] Shuai Hao, D. Li, W. Halfond, and R. Govindan. 2012. Estimating Android Applications' CPU Energy Usage via Bytecode Profiling. In *Proc. of the First Int. Workshop on Green and Sustainable Software (GREENS'12)*. IEEE Press, 1–7.

[11] Samir Hasan, Zachary King, Munawar Hafiz, Mohammed Sayagh, Bram Adams, and Abram Hindle. 2016. Energy profiles of java collections classes. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, 225–236.

[12] M. A. Hoque, M. Siekkinen, K. N. Khan, Y. Xiao, and S. Tarkoma. 2015. Modeling, Profiling, and Debugging the Energy Consumption of Mobile Devices. *ACM Comput. Surv.* 48, 3 (2015), 39:1–39:40.

[13] R. Jabbarvand, A. Sadeghi, J. Garcia, S. Malek, and P. Ammann. 2015. EcoDroid: An Approach for Energy-based Ranking of Android Apps. In *Proc. of 4th Int. Workshop on Green and Sustainable Software (GREENS '15)*. IEEE Press, 8–14.

[14] Phillipe Kruchten, Robert L. Nord, and Ipek Ozkaya. 2012. Technical Debt: From Metaphor to Theory and Practice. https://ieeexplore.ieee.org/document/6336722

[15] D. Li, S. Hao, W. G. J. Halfond, and R. Govindan. 2013. Calculating Source Line Level Energy Information for Android Applications. In *Proc. of 2013 Int. Symposium on Software Testing and Analysis (ISSTA 2013)*. ACM, 78–89.

[16] Z. Li, P. Avgeriou, and P. Liang. 2015. A Systematic Mapping Study on Technical Debt and Its Management. *J. Syst. Softw.* 101, C (March 2015), 193–220.

[17] L. G. Lima, F. Soares-Neto, P. Lieuthier, F. Castor, G. Melfe, and J. P. Fernandes. 2016. Haskell in Green Land: Analyzing the Energy Behavior of a Purely Functional Language. In *2016 IEEE 23rd Int. Conf. on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. 517–528.

[18] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk. 2014. Mining Energy-greedy API Usage Patterns in Android Apps: An Empirical Study. In *Proc. of 11th Working Conf. on Mining Software Repositories (MSR 2014)*. ACM, 2–11.

[19] Irene Manotas, Christian Bird, Rui Zhang, David Shepherd, Ciera Jaspan, Caitlin Sadowski, Lori Pollock, and James Clause. 2016. An empirical study of practitioners' perspectives on green software engineering. In *International Conference on Software Engineering (ICSE), 2016 IEEE/ACM 38th*. IEEE, 237–248.

[20] R. Morales, R. Saborido, F. Khomh, F. Chicano, and G. Antoniol. 2018. EARMO: An Energy-Aware Refactoring Approach for Mobile Apps. *IEEE Transactions on Software Engineering* 44, 12 (Dec 2018), 1176–1206.

[21] F. Palomba, D. Di Nucci, A. Panichella, A. Zaidman, and A. De Lucia. 2019. On the impact of code smells on the energy consumption of mobile applications. *Information and Software Technology* 105 (January 2019), 43–55.

[22] A. Pathak, Y. C. Hu, and M. Zhang. 2012. Where is the Energy Spent Inside My App?: Fine Grained Energy Accounting on Smartphones with Eprof. In *Proc. of 7th ACM European Conf. on Computer Systems (EuroSys '12)*. ACM, 29–42.

[23] R. Pereira, T. Carção, M. Couto, J. Cunha, J. P. Fernandes, and J. Saraiva. 2020. SPELLing out energy leaks: Aiding developers locate energy inefficient code. *Journal of Systems and Software* 161 (2020), 110463.

[24] R. Pereira, M. Couto, J. Cunha, J. P. Fernandes, and J. Saraiva. 2016. The Influence of the Java Collection Framework on Overall Energy Consumption. In *Proc. of 5th Int. Workshop on Green and Sustainable Software (GREENS '16)*. ACM, 15–21.

[25] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. P. Fernandes, and J. Saraiva. 2017. Energy Efficiency Across Programming Languages: How Do Energy, Time, and Memory Relate?. In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering (SLE 2017)*. ACM, 256–267.

[26] R. Pereira, P. Simão, J. Cunha, and J. Saraiva. 2018. jStanley: Placing a Green Thumb on Java Collections. In *Proc. of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, New York, NY, USA, 856–859.

[27] Gustavo Pinto and Fernando Castor. 2017. Energy Efficiency: A New Concern for Application Software Developers. *Commun. ACM* 60, 12 (Nov 2017), 68–75.

[28] Rubén Saborido, Rodrigo Morales, Foutse Khomh, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. 2018. Getting the most from map data structures in Android. *Empirical Software Engineering* 23, 5 (2018), 2829–2864.

[29] C. Sahin, F. Cayci, I. L. M. Gutierrez, J. Clause, F. Kiamilev, L. Pollock, and K. Winbladh. 2012. Initial explorations on design pattern energy usage. In *Green and Sustainable Software (GREENS), 2012 First Int. Workshop on*. IEEE, 55–61.

[30] A. Tsintzira, A. Ampatzoglou, O. Matei, A. Ampatzoglou, A. Chatzigeorgiou, and R. Heb. 2019. Technical Debt Quantification through Metrics: An Industrial Validation. In *15th China-Europe Int. Symp. on Software Engineering Education (CEISEE'19)*. IEEE, –.