E-Debitum: Managing Software Energy Debt

Daniel Maia, Marco Couto, João Saraiva
HASLab/INESC TEC & Universidade do Minho
daniel.maia,marco.l.couto,joao.a.saraiva@inesctec.pt

Rui Pereira
HASLab/INESC Tec, Portugal
ruipereira@di.uminho.pt

ABSTRACT

This paper extends previous work on the concept of a new software energy metric: *Energy Debt*. This metric is a reflection on the implied cost, in terms of energy consumption over time, of choosing an energy flawed software implementation over a more robust and efficient, yet time consuming, approach.

This paper presents the implementation a SonarQube tool called *E-Debitum* which calculates the energy debt of Android applications throughout their versions. This plugin uses a robust, well defined, and extendable smell catalog based on current green software literature, with each smell defining the potential energy savings. To conclude, an experimental validation of *E-Debitum* was executed on 3 popular Android applications with various releases, showing how their energy debt fluctuated throughout releases.

CCS CONCEPTS

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

KEYWORDS

Green Software, Energy Debt, Code Analysis

ACM Reference Format:
Daniel Maia, Marco Couto, João Saraiva and Rui Pereira. 2020. E-Debitum: Managing Software Energy Debt. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Technical Debt (TD) describes the gap between the current state and the ideal state of a software system. The key idea of technical debt is that software systems may include hard to understand/maintain/evolve artefacts, causing higher costs in the future 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 [27] 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 [42]. Indeed, several *energy inefficient* programming practices have been reported in literature, namely, energy patterns for mobile applications [8, 12], the energy impact of code smells [4], energy-greedy API usage patterns [29], energy (in)efficient data structures [33, 38], programming languages [39], 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 [43].

This paper is based off work on energy debt [9], a metric to estimate the 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.

Our contribution in this paper is three-fold: a) We thoroughly define a robust and extendable Android smell catalog, based on reported state-of-the-art Android energy code smells and their known energy costs per usage time, to be usable for an energy debt analysis; b) We present an open-source SonarQube tool called *E-Debitum*, to automatically calculate the energy debt between various versions of an Android application; c) We present the results of the energy debt analysis (energy smell detection and energy debt evolution) on three popular Android applications containing various releases (one having 17 releases).

This paper is structured as follows: Section 2 thoroughly describes the notion of energy debt, and how it should be expressed/-calculated, and introduces our energy debt smells catalog; Section 3 presents our SonarQube tool, E-Debitum, and an experimental validation using E-Debitum on 3 popular Android apps to analyze their energy debt; Section 4 presents related work; finally, our conclusions and future work are included in Section 5.

2 INTRODUCING ENERGY DEBT CONCEPTS

This section will describe the current definition of energy debt by presenting in Section 2.1 the general idea behind the concept. After presenting such concepts, we clarify each one in detail. First, we define how to express our newly introduced smells catalog, i.e., the considered must-fix problems and their associated energy cost, and present an instantiation of such a catalog adapted to the current concept of energy debt (Section 2.2). Afterwards, in Section 2.3 we explain how the energy debt can be estimated for a given software release version, using the occurrences of smells found for a given release. Finally, we discuss how this debt can be be transformed into interest with the appearing of new releases, and how to estimate such an interest (Section 2.4).

2.1 Concept Overview

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" [1]. In other words, 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 software 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* [5]. Based on the underlying concept of technical debt, **Energy Debt** is defined 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. The right-hand side presents 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 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.

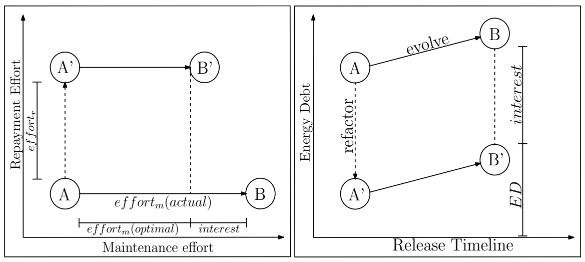

Figure 1: Technical Debt vs. Energy Debt Terminology

Given the previous assumptions, 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 smell 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. $E(i)$ returns the expected energy debt per time unit of smell i, as defined in the smell catalog.

The presented formula assume that each considered energy 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). This is due to the fact that measuring energy is not a completely deterministic task, for example, the CPU/room temperature greatly affects energy consumption.

Following the highest/lowest saving approach adds valuable information regarding potential energy savings. A certain smell can have a maximum savings of, for instance, 3000 mJ per minute, and a minimum of 150 mJ per minute. When compared to another smell with respective maximum and minimum savings of 900 mJ and 300 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 smell 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 [8].

In accordance with the previous assumption, 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 usage time, it is expected that E_{max} will be much higher with increase of usage time, as seen in Figure 2.

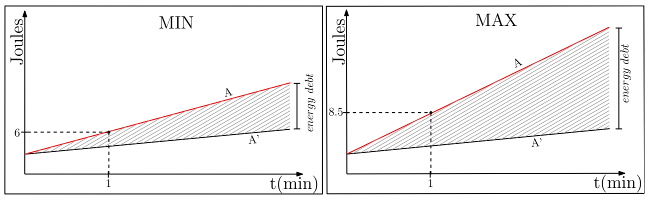

Figure 2: Energy Debt Thresholds Increase Over Time

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:

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

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.

2.2 The Smells Catalogue

In order to report the technical debt of a software system/release, it is necessary to consider a set of issues and their severity in terms of refactoring time. Hence, for the energy debt approach, the first step is to define those issues, and the highest/lowest energy gain which can be obtained from refactoring/removing each one.

The energy code smells contained in our catalogue have all been reported in green software literature and are widely used and studied specifically within the Android ecosystem, where energy consumption is a main software concern. We excluded smells which did not provide a way to infer a highest/lowest energy saving value per minute, either because it was not reported by the authors or because raw result data was not publicly available. For each energy smell, we indicate where its energy impact was studied, a brief description on why it has a negative influence on energy consumption, and the corresponding reported maximum and minimum energy savings in milliJoules per minute (E_{max} and E_{min}, respectively).

DA - `Draw Allocation`.

- E_{max} : 158 mJ per minute
- E_{min} : 36 mJ per minute

This is the first of five smells whose energy impact analysis was included in [8, 12, 13]. The authors aimed at understanding how fixing code patterns detected by Android *lint*[1] can improve energy efficiency. *Lint*’s issues are divided into categories, such as Performance or Security. `Draw Allocation`, as well as the next 4 smells, is a **Performance** issue.

`Draw Allocation` occurs when new objects are allocated along with draw operations, which are very sensitive to performance. In other words, it is a bad practice to create objects inside the `onDraw` method of a class which extends a `View` Android component, as we see in the following snippet:

```
public class CloudMoonView extends View {
  @Override
  protected void onDraw(Canvas canvas) {
    RectF rectF1 = new RectF(); ✘
    ...
    if(!clockwise) {
      rectF1.set(X2-r, Y2-r, X2+r, Y2+r);
      ...
} }
```

The recommended alternative for this smell is to move the allocation of independent objects outside the method, turning it into a static variable, as shown next:

```
public class CloudMoonView extends View {
  RectF rectF1 = new RectF(); ✔
  @Override
  protected void onDraw(Canvas canvas) {
    ...
    if(!clockwise) {
      rectF1.set(X2-r, Y2-r, X2+r, Y2+r);
      ...
} }
```

[1]*Lint* is a code analysis tool, provided by the Android SDK, which reports upon finding issues related with the code structural quality. Website: developer.android.com/studio/write/lint

WL - `Wakelock`.

- E_{max} : 194 mJ per minute
- E_{min} : 10 mJ per minute

`Wakelock` is the second Android *lint* performance issue [8, 12, 13, 34, 50]. Essentially, *lint* detects whenever a wake lock, a mechanism to control the power state of the device and prevent the screen from turning off, is not released, or is used when it is not necessary.

The following snippet shows an example of a wake lock being acquired, but not released when the activity pauses.

```
public class DMFSetTempo extends Fragment {
  PowerManager.WakeLock wakeLock;

  public void onClickBtStart(View view) {
    wakelock.acquire(); ✔
  }

  @Override()
  public void onPause() { super.onPause(); ✘ }
  }
}
```

The alternative here would be to simply add a `release` instruction as shown next:

```
public class DMFSetTempo extends Fragment {
  PowerManager.WakeLock wakeLock;

  public void onClickBtStart(View view) {
    wakelock.acquire();   ✔
  }

  @Override()
  public void onPause() {
    super.onPause();
    if (wakeLock.isHeld()) wakeLock.release(); ✔
  }
}
```

RC - `Recycle`.

- E_{max} : 533 mJ per minute
- E_{min} : 15 mJ per minute

`Recycle` is another Android *lint* performance issue [8, 12, 13]. It detects when collections or database related objects, such as `TypedArrays` or `Cursors`, are not recycled nor closed after being used. When this happens, other objects of the same type cannot efficiently use the same resources.

The following snippet shows a `Cursor` instance being used without being recycled:

```
public Summoner getSummoner(int id) {
  SQLiteDatabase db = this.getReadableDatabase();

  Cursor cursor = db.query(TABLE_FAV, new String[] { ... };
  ...
  return summoner;  ✘
}
```

The alternative in this case would be to include a `close` method call before the method’s return:

```
public Summoner getSummoner(int id) {
  SQLiteDatabase db = this.getReadableDatabase();

  Cursor cursor = db.query(TABLE_FAV, new String[] { ... };
  ...
  c.close();   ✔
  return summoner;
}
```

VH - `View Holder`.

- E_{max} : 2105 mJ per minute
- E_{min} : 892 mJ per minute

`View Holder` is the last Android *lint* performance issue [8, 12, 13], whose alternative intends to make a smoother scroll in *List Views*. The process of drawing all items in a *List View* is costly, since they need to be drawn separately. However, it is possible to make this more efficient by reusing data from already drawn items, which reduces the number of calls to `findViewById()`, known to be energy greedy [29].

In order to better describe this smell, we introduce this snippet:

```
public View getView(int pos, View cView, ViewGroup par) {
  LayoutInflater inflater = (LayoutInflater) context
    .getSystemService(Context.LAYOUT_INFLATER_SERVICE);

  cView = inflater.inflate(R.layout.apps, par, false);
  TextView txt=(TextView) cView.findViewById(R.id.label); ❶
  ImageView img=(ImageView) cView.findViewById(R.id.logo);❷
  return row;
}
```

Every time `getView()` is called, the system searches on all the view components for both the `TextView` with the id "label" (❶) and the `ImageView` with the id "logo" (❷), using the energy greedy method `findViewById()`. The alternative version is to cache the desired view components, with the following approach:

```
static class ViewHolderItem {
  TextView txtView; ImageView imgView;
}

public View getView(int pos, View cView, ViewGroup par) {
  ViewHolderItem hld; LayoutInflater inflater = ...

  if (cView == null) {                              ❸
    cView = inflater.inflate(...);
    hld = new ViewHolderItem();
    hld.txtView = (TextView) cView.findViewById(...);  ❹
    hld.imgView = (ImageView) cView.findViewById(...); ❺
    cView.setTag(hld);
  } else {
    hld = (ViewHolderItem) cView.getTag();          ❻
  }
  TextView txt = hld.txtView;  ImageView img = hld.imgView;
  ...
}
```

Condition ❸ evaluates to true only once, which means instructions ❹ and ❺ execute once, i.e., `findViewById()` executes twice, and its results are stored in the `ViewHolderItem` instance. The following calls to `getView()` will use cached values for the view components `txt` and `img` (❻).

HMU - `HashMap Usage`.

- E_{max} : 229 mJ per minute
- E_{min} : 28 mJ per minute

This smell is related to the usage of the `HashMap` collection [4, 8, 32, 34, 45]. In fact, as stated in the Android documentation page, the usage of `HashMap` is discouraged, since the alternative `ArrayMap` is allegedly more energy-efficient, without decreasing the performance of `map` operations[2].

The alternative is to simply replace the type `HashMap`, whenever it is used, with `ArrayMap`.

EMC - `Excessive Method Calls`.

- E_{max} : 9529 mJ per minute
- E_{min} : 557 mJ per minute

Unnecessarily calling a method can penalize performance, since a call usually involves pushing arguments to the call stack, storing the return value in the appropriate processor's register, and cleaning the stack afterwards. This penalty was explored by [4, 8, 24], showing that the energy consumption in Android applications can be decreased by removing method calls inside loops that can be extracted from them. An example of an extractable method call would be one which receives no arguments, and is accessed by an object that is not altered in any way inside the loop.

The alternative is to replace the call with a variable declared outside the loop and initialised with the return value of that call.

MIM - `Member Ignoring Method`.

- E_{max} : 7844 mJ per minute
- E_{min} : 88 mJ per minute

This smell addresses the issue of having a non-static method inside a class which could be static instead [4, 8, 34], i.e., it does not access any class fields, directly invoke non-static methods, and is not an overriding method. Static methods are stored in a memory block separate from where objects are stored, and no matter how many class instances are created throughout the program's execution, only one instance of such method will be created and used. This mechanism helps in reducing energy consumption.

2.3 Counting Expenses & Estimating Debt

With a defined energy smell catalog, the next step 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*[3], which is a widely used tool for technical debt estimation, provides an API for defining detection rules for issues/smells of different languages.[4]

Detecting the occurrence of a smell, however, is 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 or unreachable code, or it can be placed inside a procedure which may only be executed once in a software lifecycle (e.g. 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, thread, or *Service* (very common in Android). These kind of scenarios should be considered when estimating energy debt, and since the energy debt approach implies the usage of statistical analysis mechanisms, we can follow already defined strategies for static energy analysis.

Several strategies have been suggested for this task, all of which have been accepted by the community with promising results. As advised in previous work on energy debt [9], we will consider a simplified version of the strategy from Jabbarvand et al. [21]:

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

In this equation:

- C is the number of i smells found in the release r;
- $paths(j)$ represents the number of paths in the *call graph* through which the j^{th} occurrence of smell i is reachable;
- 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.

[2] *ArrayMap* documentation: http://bit.ly/32hK0y9.

[3] *SonarQube* webpage: http://www.sonarqube.org.

[4] Adding Coding Rules webpage: https://docs.sonarqube.org/latest/extend/adding-coding-rules/

For more details and examples on the calculation of Energy Debt throughout versions, please refer to [9].

2.4 Paying Interests

The concept of *interest* in technical debt has already been formulated [5], and its application has been studied [2, 3, 49]. The concept is based on the fact that, as software evolves, the cost/effort of adding features to a new release increases if technical debt is not addressed. Maintaining a release with technical debt requires more effort than maintaining one without it; the effort difference between the two is what is called the *technical debt interest.*

The left-hand side of Figure 1 illustrates interest. There is a software version, A, containing code smells, and therefore technical debt. At this point, a decision can be made on what to prioritise. If the priority is releasing the version as it is, without dealing with the smells then the effort of maintaining/evolving to a new release B will be higher than if the priority was investing effort in fixing the smells and releasing an optimal version of that release, A', without technical debt. This resembles the idea of accumulation of debt, and debt needs to be re-payed.

Chatzigeorgiou et al. [5] presented a technique to predict the technical debt *breaking point*, i.e., when will the accumulated interest be higher than the initial effort to remove the technical debt (i.e., from the initial release, which is called *principal*). With this approach, it is possible to present developers with an alternative: if technical debt is not addressed now, then they have approximately until release number N to properly deal with it; otherwise, from then on, the additional *maintenance effort* from not dealing with technical debt, will always be higher than the effort to deal with the *principal.* This ultimately means that, at that point, they are wasting development time.

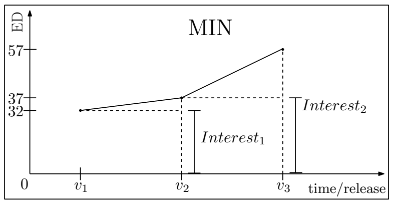

Figure 3: Accumulation of Interest

When considering energy instead of technical debt, the concept of interest needs a different approach to be defined. For starters, it will not tell developers (or project managers) how much additional maintenance effort is being applied. This is due to the fact that energy debt does not measure effort, but the drain of a particular resource. In that sense, the *energy debt interest* is the amount of excessively consumed energy over time, that could be avoided if the issues which caused it were properly addressed earlier. In simple terms, it is the accumulated energy debt after n software 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 or uptime related.

Figure 3 illustrates the perception of interest within energy debt. Three software releases are included here, and the presented values refer to the minimum estimated debt. For this example, we assume that issues from previous versions are not addressed in any way, hence, energy debt is always increasing. For the initial release, $v1$, we consider no interest was accumulated thus far, due to the fact that energy debt is estimated in function of the usage time, which would be null at the exact instant when the version was released.

Starting from $v2$, we can compute interest. 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 32. Likewise, when considering release $v3$, the minimum acumulated interest would be $\frac{(32+37)}{2} = 34.5$.

The presented approach might seem over-simplified, but it is a fair portrayal of debt repayment. Interest is directly proportional to energy debt, which ultimately means that when interest is reduced, the initial debt is being re-payed. It is important to interpret interest similarly to energy debt: an interval describing the minimum/maximum energy being excessively consumed **per usage time**. As such, we argue that, with each new release, the expected usage time should be higher than any previous release. Therefore it is guaranteed that, even though energy debt can be reduced between releases, accumulated interest will be inflated for later releases.

3 E-DEBITUM: ENERGY DEBT ANALYSIS

Although energy smells at large may occur in any programming language, a number of language specific smells exist. With this in mind, this proof of concept was written in Java.

To achieve our goals, the *E-Debitum* tool[5] was implemented through two separate SonarQube plugins, through the aid of templates which will be used to handle the necessary logic involved in integrating the built tool into the platform. The first plugin implements a set of rules corresponding to the aforementioned energy smells. As such, it is dubbed as the rules plugin. It is used during the analysis process to detect and document any and all energy smells present in the code, alongside any other issues actively being searched for by the platform.

The second plugin, dubbed the metrics plugin, is active immediately after the code scan is complete and is responsible for measuring the minimum/maximum estimated values of Energy Debt in the program. It tallies up the number of instances of each given code smell and, using the estimated joule per minute expenditure they cause, stores the total debt value in its best/worst case scenarios.

These functionalities were kept separated into two as to better improve the readability of the plugin's code, as well as to allow SonarLint users to make use of the rules as they write code, as opposed to exclusively discovering the energy smells upon analysis. It is worth noting that in this case, however, they would only see the rules but not fully know their impact as a whole.

3.1 Rules Plugin

To create the first SonarQube plugin, an official Java specific rules template[6] was used to save development time. As such, five files will be created for each smell detecting rule implemented.

For starters, a test file will be written. This will contain code to be used to test the rule. It holds a set of methods in which

[5]Github: https://github.com/e-debitum/E-Debitum-tool
[6]*SonarQube* Java rules template plugin: https://github.com/SonarSource/sonar-custom-rules-examples/tree/master/java-custom-rules

instances of both compliant and non-compliant code is written to show SonarQube what patterns to search for when testing for that given smell and which to eke out. Afterwards, a rule class will be developed, which handles the logic when SonarQube detects a possible instance of non-compliant code. It will check to confirm that it is not a false positive and, if it does, report the issue. Finally, a test class will be created, which simply houses the unit test for the rule. This will verify that instances where the test file and rule class flag code as non-compliant match up when building the plugin.

Additionally, an HTML and JSON file will be made so as to display the details of the rule to the end user within SonarQube. These include a short description of the rule, along with an example case of non-compliant code and its respective fix, as well as the rule's related techical debt and tags.

Once implemented, each rule must be activated within the plugin by adding it to the *RulesList* class baked into the plugin template.

3.2 Metrics plugin

For this plugin, another template was used, making use of the Sonar API[7]. To do this, three Java classes need to be developed. The first of these implements Sonar API's Plugin interface. This class is the entry point for all extensions made to SonarQube. In this case, two extensions will be added, which are the other two necessary classes.

The first extension is an implementation of the MeasureComputer interface. This class is responsible for describing its output to the system and reading which and how many energy smells were detected and sum up the associated maximum and minimum estimated debt. Lastly, the final class necessary and the second extension added is an implementation of the Metrics interface. This class simply defines the metadata for minimum and maximum energy debt as variables in SonarQube.

3.3 Experimental Validation

To test the efficacy of the implemented plugins, a set of Android applications were acquired through searching GitHub's library of open-source projects. From these, a handful of highly downloaded Google apps were selected with *(i)* a fluctuating number of smells per release and *(ii)* a significant number of release across its lifespan. As such, three applications which met this criteria were chosen.

PDF Viewer Plus[8] is an open source PDF viewer which allows the reading and sharing of PDF files, with the ability to customise the UI with a variety of themes, which had 17 separate releases. Table 1 presents the results of the tool analysis.

Despite the high number of releases of the app, the number of smells detected remained relatively low, with only one to two smells being detected across nearly its entire lifespan. Likewise, its energy debt remained stable all throughout, keeping a low range between its best and worst case scenarios at an maximum average of 4.09 J/min and a minimum average of 1.18 J/min. Using SonarQube's Activity tab, we can see the plot of the apps code smells - in this case, exclusively energy smells - along with their maximum and minimum energy debt, as seen in Figure 4.

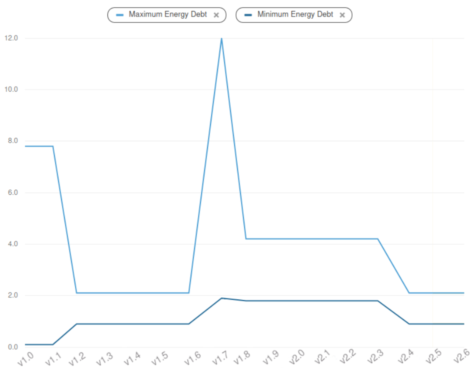

Figure 4: Minimum and Maximum Energy Debt for *PDF Viewer Plus* releases over time

Malse Geluiden is another open source app available in the Google PlayStore. Through it, users can play a set of sounds and animations on their device, as well as share them through *WhatsApp*. This app was tested across 7 explicit releases, which were run through SonarQube. Table 2 presents the four energy smells and its occurrences found throughout the analysis of this app.

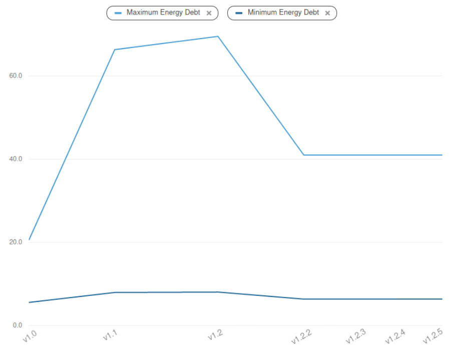

Figure 5: Minimum and Maximum Energy Debt for *MalseGeluiden* releases over time

This app finds an increasing number of energy smells with each release and a rising overall level of debt, as well as a wider difference between its maximum and minimum values, achieving an average of 45.76 J/min and 6.66 J/min, respectively, as seen in Figure 5.

Lastly, *EscapeApp* is a client application used for a virtual reality escape game, which contains three different smells occurrences spread over five releases, as seen in Table 3.

The smell detection performed on *EscapeApp* displays a wide gap of around 100 J/min between its best and worst case, with a maximum and minimum averages of 139.20 J/min and 8,64 J/min, respectively, as seen in Figure 6.

4 RELATED WORK

Technical debt is refers to the pitfalls of creating sub-optimal software to fit a shorter interval, introduced by Cunningham [15]. This is a common practice employed to meet short term development

[7] *SonarQube* custom plugin example plugin: https://github.com/SonarSource/sonar-custom-plugin-example
[8] *PDF Viewer Plus* code: https://play.google.com/store/apps/details?id=com.gsnathan.pdfviewer

	versions																
	1.0	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**	**1.6**	**1.7**	**1.8**	**1.9**	**2.0**	**2.1**	**2.2**	**2.3**	**2.4**	**2.5**	**2.6**
smell	MIM:1	MIM:0 ▼	MIM:0	MIM:0	MIM:0	MIM:0	MIM:0	MIM:0	MIM:0	MIM:0	MIM:0	MIM:0	MIM:0	MIM:0	MIM:0	MIM:0	MIM:0
count			VH:1	VH:1	VH:1	VH:1	VH:1	VH:2 ▲	VH:2	VH:2	VH:2	VH:2	VH:2	VH:2	VH:1 ▼	VH:1	VH:1

Table 1: Detected smells on Pdf Viewer Plus releases

	versions						
	1.0	**1.1**	**1.2**	**1.2.2**	**1.2.3**	**1.2.4**	**1.2.5**
	MIM:1	MIM:2 ▲	MIM:2	MIM:2	MIM:2	MIM:2	MIM:2
smell	VH:6	VH:6	VH:6	VH:6	VH:6	VH:6	VH:6
count		EMC:4	EMC:1 ▼	EMC:1	EMC:1	EMC:1	EMC:1
			RC:6	RC:6	RC:6	RC:6	RC:6

Table 2: Detected smells on *MalseGeluiden* releases

	versions				
	v0.4	**v0.5**	**v0.6**	**v0.7**	**v0.8**
smell	EMC: 1	EMC: 2 ▲	EMC: 1 ▼	EMC: 1	EMC: 1
count			HMU: 8	HMU: 8	HMU: 8
			MIM: 2	MIM: 3 ▲	MIM: 3

Table 3: Detected smells on *EscapeApp* releases

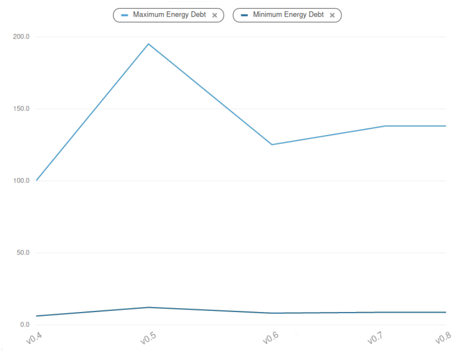

Figure 6: Minimum and Maximum Energy Debt for *EscapeApp* releases over time

deadlines, with the intent of completing it at a later time. As software evolves, it is 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 ageing and evolution." [23]. However, more common issues can also plague software development, born from poor coding practices or general ignorance, accentuating different aspects to technical debt [48].

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. This makes it so the longer such code goes unattended, the more resources will be needed to correct it and with diminishing returns [5].

One such inefficiency in software is of high energy consumption. In fact, the profiling, analysing and improving the energy efficiency of software has become a very active research field. Studies have shown that developers are aware of the energy consumption problem, and often times seek help in solving such issues [30, 42, 43]. 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 [17, 28, 38, 41], languages [10, 39], memoization [44], design patterns [46], code refactoring [47, 51], and even the testing phase [26].

Specific to the Android ecosystem, there has been research in topics such as the classification of Android apps as being more/less energy efficient [21], identifying energy green APIs [29], estimating energy consumption in code fragments [11, 19], etc. In fact, research in the reduction of energy consumption in the Android system is most likely the most explored environment in this research field over the past decade [4, 12–14, 22, 24, 32, 34, 45, 50]. The results of most of these studies are able to quickly translate into our energy smell catalogue 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* [16], *vLens* [25], *eProf* [35], *Trepn*[18, 19, 21], *GreenOracle* [7], *GreenScaler* [6], *COBWEB* [20]. There is also work in automatic tools to help detect energy greedy code spots [36, 37], automatically refactoring for the most energy efficient data structure [31, 40], or automatically refactoring energy greedy Android patterns [4, 8, 13].

The aforementioned works on the different approaches to reducing energy consumption of software systems do not yet translate their potential gains across a period of time into the actual energy savings (or costs) a developer or business can have on his/her software by applying such transformations. It is our belief that, with this work, we have helped close this gap in not only knowing if an alternative solution is more energy efficient, but by how much can we save (in energy consumption or even monetary savings) over time if and when we adopt the energy efficient alternative.

5 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. We have presented a catalog of reported state-of-the-art energy code smells, their known energy costs per usage time, and have expressed energy debt 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.

The automatic detection of the presented energy code smells, and the computation of energy debt and interest, has been achieved using our *E-Debitum* tool. This tool is usable within the SonarQube framework through two plugins. Using *E-Debitum*, we were able to perform an experimental validation on 3 popular Android applications across various releases.

We are currently working on extending the smell catalog, which we consider, to be updated with more energy code smells. Additionally, we are preparing a large scale study on the energy debt of hundreds of open-source Android applications.

REFERENCES

[1] Eric Allman. 2012. Managing Technical Debt. *Commun. ACM* 55, 5 (May 2012).

[2] Areti Ampatzoglou, Apostolos Ampatzoglou, Paris Avgeriou, and Alexander Chatzigeorgiou. 2015. Establishing a Framework for Managing Interest in Technical Debt. In *Proceedings of the Fifth International Symposium on Business Modeling and Software Design - Volume 1: BMSD,*. INSTICC, SciTePress, 75–85.

[3] Areti Ampatzoglou, Alexandros Michailidis, Christos Sarikyriakidis, Apostolos Ampatzoglou, Alexander Chatzigeorgiou, and Paris 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)*. Association for Computing Machinery, 115–124. https://doi.org/10.1145/3194164.3194175

[4] A. Carette, M. A. A. Younes, G. Hecht, N. Moha, and R. Rouvoy. 2017. Investigating the energy impact of Android smells. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 115–126.

[5] 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.

[6] Shaiful Chowdhury, Stephanie Borle, Stephen Romansky, and Abram Hindle. 2018. GreenScaler: training software energy models with automatic test generation. *Empirical Software Engineering* 24 (07 2018). https://doi.org/10.1007/s10664-018-9640-7

[7] Shaiful Chowdhury and Abram Hindle. 2016. GreenOracle: estimating software energy consumption with energy measurement corpora. 49–60. https://doi.org/10.1145/2901739.2901763

[8] Marco Couto, João Paulo Fernandes, and João Saraiva. 2020. Energy Refactorings for Android in the Large and in the Wild. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. London, Ontario, Canada.

[9] Marco Couto, Rui Pereira, Daniel Maia, and João Saraiva. 2020. On Energy Debt: Managing Consumption on Evolving Software. In *2020 IEEE/ACM International Conference on Technical Debt (TechDebt)*. IEEE.

[10] Marco Couto, Rui Pereira, Francisco Ribeiro, Rui Rua, and João Saraiva. 2017. Towards a Green Ranking for Programming Languages. In *Proceedings of the 21st Brazilian Symposium on Programming Languages (SBLP 2017)*. ACM, 7:1–7:8.

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

[12] 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.

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

[14] Luis Cruz and Rui Abreu. 2019. Catalog of energy patterns for mobile applications. *Empirical Software Engineering* 24, 4 (01 Aug 2019), 2209–2235.

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

[16] S. Hao, D. Li, W.G.J. Halfond, and R. Govindan. 2012. Estimating Android applications' CPU energy usage via bytecode profiling. In *Green and Sustainable Software (GREENS), 2012 First Int. Workshop on*. 1–7.

[17] 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.

[18] Mohammad Ashraful Hoque, Matti Siekkinen, Kashif Nizam Khan, Yu Xiao, and Sasu Tarkoma. 2015. Modeling, Profiling, and Debugging the Energy Consumption of Mobile Devices. *ACM Comput. Surv.* 48, 3 (2015), 39:1–39:40.

[19] Yan Hu, Jiwei Yan, Dong Yan, Qiong Lu, and Jun Yan. 2018. Lightweight energy consumption analysis and prediction for Android applications. *Science of Computer Programming* (2018), 132–147. Special Issue on TASE 2016.

[20] R. Jabbarvand, J. Lin, and S. Malek. 2019. Search-Based Energy Testing of Android. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 1119–1130.

[21] 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.

[22] Hao Jiang, Hongli Yang, Shengchao Qin, Zhendong Su, Jian Zhang, and Jun Yan. 2017. Detecting Energy Bugs in Android Apps Using Static Analysis. In *Formal Methods and Software Engineering*, Zhenhua Duan and Luke Ong (Eds.). Springer International Publishing, 192–208.

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

[24] D. Li and W. G. J. Halfond. 2014. An Investigation into Energy-saving Programming Practices for Android Smartphone App Development. In *Proc. of 3rd Int. Workshop on Green and Sustainable Software (GREENS 2014)*. ACM, 46–53.

[25] 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.

[26] D. Li, Y. Jin, C. Sahin, J. Clause, and W. G. J. Halfond. 2014. Integrated Energy-directed Test Suite Optimization. In *Proc. of 2014 Int. Symposium on Software Testing and Analysis (ISSTA 2014)*. ACM, 339–350.

[27] Zengyang Li, Paris Avgeriou, and Peng Liang. 2015. A Systematic Mapping Study on Technical Debt and Its Management. *J. Syst. Softw.* 101, C (March 2015).

[28] 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.

[29] 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.

[30] 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.

[31] Irene Manotas, Lori Pollock, and James Clause. 2014. SEEDS: A Software Engineer's Energy-Optimization Decision Support Framework. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 503–514.

[32] 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.

[33] W. Oliveira, R. Oliveira, F. Castor, B. Fernandes, and G. Pinto. 2019. Recommending Energy-Efficient Java Collections. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. 160–170.

[34] Fabio Palomba, Dario Di Nucci, Annibale Panichella, Andy Zaidman, and Andrea De Lucia. 2019. On the impact of code smells on the energy consumption of mobile applications. *Information and Software Technology* 105 (January 2019).

[35] 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.

[36] R. Pereira, T. Carção, M. Couto, J. Cunha, J. P. Fernandes, and J. Saraiva. 2017. Helping Programmers Improve the Energy Efficiency of Source Code. In *Proc. of the 39th International Conference on Soft. Eng. Companion (ICSE-C 2017)*. ACM.

[37] Rui Pereira, Tiago Carção, Marco Couto, Jácome Cunha, João Paulo Fernandes, and João Saraiva. 2020. SPELLing out energy leaks: Aiding developers locate energy inefficient code. *Journal of Systems and Software* 161 (2020), 110463. https://doi.org/10.1016/j.jss.2019.110463

[38] 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.

[39] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João 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.

[40] Rui Pereira, Pedro Simão, Jácome Cunha, and João Saraiva. 2018. jStanley: Placing a Green Thumb on Java Collections. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, New York.

[41] G. Pinto and F. Castor. 2014. Characterizing the Energy Efficiency of Java's Thread-Safe Collections in a Multi-Core Environment. In *Proc. of SPLASH'2014 workshop on Software Engineering for Parallel Systems (SEPS), SEPS*, Vol. 14.

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

[43] Gustavo Pinto, Fernando Castor, and Yu David Liu. 2014. Mining Questions about Software Energy Consumption. In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR 2014)*. Association for Computing Machinery.

[44] Rui Rua, Marco Couto, Adriano Pinto, Jácome Cunha, and João Saraiva. 2019. Towards using Memoization for Saving Energy in Android. In *Proceedings of the XXII Iberoamerican Conference on Software Engineering (CIbSE)*. 279–292.

[45] 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.

[46] 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.

[47] C. Sahin, L. Pollock, and J. Clause. 2014. How Do Code Refactorings Affect Energy Usage?. In *Proc. of 8th ACM/IEEE Int. Symposium on Empirical Software Engineering and Measurement (ESEM '14)*. ACM, 36:1–36:10.

[48] Edith Tom, Aybüke Aurum, and Richard Vidgen. 2013. An exploration of technical debt.

[49] Angeliki Tsintzira, Apostolos Ampatzoglou, Oliviu Matei, Areti Ampatzoglou, Alexander Chatzigeorgiou, and Robert Heb. 2019. Technical Debt Quantification through Metrics: An Industrial Validation. In *15th China-Europe International Symposium on Software Engineering Education (CEISEE' 19)*. IEEE, –.

[50] Panagiotis Vekris, Ranjit Jhala, Sorin Lerner, and Yuvraj Agarwal. 2012. Towards Verifying Android Apps for the Absence of No-sleep Energy Bugs. In *Proceedings of the 2012 USENIX Conference on Power-Aware Computing and Systems (HotPower'12)*. USENIX Association.

[51] Roberto Verdecchia, Rene Aparicio Saez, Giuseppe Procaccianti, and Patricia Lago. 2018. Empirical Evaluation of the Energy Impact of Refactoring Code Smells. In *ICT4S2018. 5th International Conference on Information and Communication Technology for Sustainability (EPiC Series in Computing)*, Birgit Penzenstadler, Steve Easterbrook, Colin Venters, and Syed Ishtiaque Ahmed (Eds.), Vol. 52.