

# Object Metrics for Green Software

Mourad Chabane Oussalah\*, Romain Brohan, Ossama Moustafa

LS2N, University of Nantes - Faculty of Science and Technology (FST) Building 34. 2 Chemin de la Housinière, BP 92208, 44322 Nantes Cedex 3, Loire-Atlantique, France.

\*Corresponding author. Tel.: +33 2 51 12 58 47; email: mourad.oussalah@univ-nantes.fr

Manuscript submitted January 10, 2021; accepted March 8, 2021.

doi: 10.17706/jsw.16.6.285-305

---

**Abstract:** Today, the energy consumption of computers represents a significant part of the overall consumption. The purpose of this article is to apply object and architectural metrics to observe the impact on application consumption. This article focuses on the most common object applications to date, and their architectures that are already useful to optimize the reusability, composability or dynamicity of these applications. To do this, consumption must be evaluated and compared according to the variations of object and architectural metrics. These observations help to determine how effective these metrics could be.

**Key words:** Metrics, software consumption, object application, software architecture.

---

## 1. Introduction

In today's world, the need to think about ecology is more and more present. Technological development raises new questions about the impact of new technologies on the ecosystem [1], with the deployment of more and more data centers [2], different tools and applications. To give an idea, if the Internet were a country, it would be the 7th largest emitter of CO<sub>2</sub> in the world [3]. Reducing the power consumption of these data centers requires an optimization of the software concerned.

In this article, we present an approach to help address the problem of reducing the consumption of devices and data centers in general, by focusing on the code and architecture [4] of the applications used by machines. We measure the energy consumption of an application, first without applying any metrics to it, then by applying them, to show that the use of object metrics in the field of object-oriented programming (chosen for its widespread use [5]) and the right choice of architecture lead to a reduction in application consumption and therefore a reduction in CO<sub>2</sub> emissions from devices and data centers.

Several studies have already been carried out in this field, notably within the framework of the TEEC project [6] whose aim is to develop a complete software consumption measurement tool, or the previous work of our team [7] on the state of the art of Green Software. Other projects have also focused on comparing consumption between different programming languages [8].

In the rest of this article we look at the work already existing on the subject. This section will present different object and architectural metrics, followed by tools for measuring object metrics and software architecture tools, and finally a presentation of several consumption measurement tools. In the second section entitled Conceptual Framework, we get to the heart of the matter by presenting in more detail our equipment

and the path planned for carrying out the experiments. This part will also consist in selecting first the different object and then architectural metrics that we will use from the one discovered in the Related Works part, followed by the same selection for the object, architectural and software consumption measurement tools. Then finally, we will present our experiments, in order to prove our hypotheses, which will be followed by a synthesis reviewing the purpose of this article. Finally, we will end with the practical framework and experimental evaluation part in which we will first carry out experiments on the different selected metrics using the chosen tools, followed by a discussion on each of the experiments carried out, and we will conclude.

## 2. Related Works

First of all, we are interested in existing metrics. Metrics [9] are used to evaluate the quality of an application in different aspects: for example, to assess its maintainability, comprehension, or performance. Several studies [10]-[13] have proposed these metrics so that they can be applied to all types of applications, which allows, for example, to compare several versions of the same application, in order to know which is the best one according to the chosen aspects. Here, we want to know if these metrics can also influence the consumption of an application and which are the most relevant.

To carry out our research, we must first choose and implement different criteria for the different points of interest to us, namely selecting the best criteria to have a good object code, and a quality software architecture, because it already allows to optimize other domains: indeed, there are object type metrics for the source code, and architectural metrics, which will have an impact on the application structure. These criteria allow us to compare the different metrics, in order to find those that would be most interesting for our case. Our goal is to observe the variation in consumption according to these metrics.

### 2.1. Object Metrics

Let's start with the criteria for classifying metrics. Our research has led us to several classifications already suggested in the works [14], [15], which complement each other. We therefore summarize here the criteria found:

- Package: Metrics performing measurements at the package level.
- Class: Metrics performing measurements on classes.
- Method: Metrics measuring everything related to methods.
- Relationship: Metrics measuring the coupling relationships and relationships between classes.
- Inheritance: Metrics measuring inheritance relationships.

These criteria give us a good overview of the concepts of object-oriented programming.

Following the work already done [16], here is what we have retained (the numbers in brackets indicate the number of quotations from the original paper of the metric):

- Weighted method per class (WMC) /McCabe Cyclomatic Complexity (6394/ 2196): The score of this metric corresponds to the sum of the complexities of the methods in a class.
- Depth Inheritance Tree (DIT) (6394): Indicates the depth of a class in its inheritance tree.
- Number Of Children (NOC) (6394): Indicates the number of direct children in a class.
- Response for a class (RFC) (6394): gives the number of methods that can be executed in response to a message received by an object in the class concerned by this score.
- Coupling Between Objects(CBO) (6394) : Gives a coupling score between two classes. Calculated by counting the other classes that use the methods and attributes of the class, plus the number of classes used by the class in question (via method or attribute).
- Lack of Coherence of Methods (LCOM or LOCOM) (6394): Gives a cohesion score per method of a class.
- Instability(I) (291): Allows to measure the relative susceptibility of the class to changes.

- Number Of Packages(NoP): counts the number of packages used by the selected element.

A large part of these metrics come from well-known metric sets: Chidamber & Kemerer in 1991 and Robert Martin in 1994.

## 2.2. Object Metric Tools

In order to carry out our tests successfully, we are looking for, in addition to metrics, a tool that allows us to measure the future selected metrics. To do this, we establish several criteria based on Kayarvizhy's research in 2016[17] :

- Maintained tool: if the tool is still maintained, then it can be assumed that it is up to date and functional.
- Metric coverage: Ideally, the tool should calculate several metrics. In Table 1, the Coverage column gives the number of metrics covered by the tool. We consider that a tool has a good coverage from 20 metrics, between 10 and 20 it is considered as average and below 10 its coverage is not enough to be interesting. The choice will also depend on the relevance and effectiveness of the measured metrics, as discussed later.

Table 1. List of Tools for Measuring Object Metrics (Based on Kayarvizhy's Work)  
NA: No Response Received from the Scientific Community

Tool name	Language	Automatic	Free	Validation	Coverage	Author/Team	Location	Date
SD Metrics [18]	UML	yes	no	NA	131	Jürgen Wüst	Allemagne	2012
RSM [19]	C++, Java, C#	yes	no	NA	100	M Square Technologies	Floride	1998
JHawk [20]	Java	yes	no	25+	115	Virtual Machinery	Ireland	1999
QMOOD++ [21]	C++	yes	yes	None	30+ 00 metrics	Jagdish Bansiya and Carl Davis [27]	Alabama	1997
Ckjm [22]	Java	no	yes	25+	8	Diomidis Spinellis	Grèce	2005
JMT [23]	Java	yes	yes	5	19	Ingo Patett	Allemagne	2002
JDepend [24]	Java	no	yes	25+	7	Mark Clark	France	2008
Eclipse Metrics [25]	Java	yes	yes	25+	29	State Of Flow	Angleterre	2006

Test-Well CMT-Java [26]	Java	yes	no	NA	20	Testwell Oy	Finlande	2012
CodeMR [27]	Java, C++, Scala	yes	Yes and no	NA	40	CodeMR Team	Angleterre	2018

- Language: Language supported by the tool. Important if the user is subject to constraints on the programming language.
- Automatic: do you need any special handling to install and use the tool? The difficulty of getting started is an important criterion for a tool, if it is not affordable by everyone, it will not be used much.
- Free: is the tool free or not? Important depending on the user's budget.
- Validation: has the tool been validated by the scientific community? This criterion is interesting for determining the reliability of the tool. In Table 1, NA means that there was no response from the scientific community on the tool, while 25+ indicates that there were more than 25 responses regarding the validity of the tool.
- Author: Person who created the tool (informative criterion)
- Location: Region where the tool was developed (informative criterion)
- Date : Date de réalisation de l'outil (critère informatif)

Some more recent tools were then added to complete the list, such as CodeMR [9] for example. [See Table 1]

### 2.3. Architectural Metrics

There are various architectural metrics [28], useful for assessing the architectural quality of software. In a previous work [29], we developed different architectural metrics, including loose coupling, abstraction of communications, expressive power, evolutionary power, proprietary responsibility, and package depth, which we detail below.

The loose coupling metric [30] evaluates the independence of classes, because the more a class is coupled, the more it depends on other classes. The explicit architecture metric evaluates the clarity of the application structure: i.e. the names assigned to the classes/packages, so that it is more understandable. The communication abstraction metric allows to evaluate the simplification of complex communication channels, in order to better understand a conversion system, or a heterogeneous system. For the power metrics, expressive and evolutionary, they respectively assess the ability of the paradigm to create an understandable concept, and the possibility of updating or improving this system under good conditions. In the long term, scalability can also have an impact on the consumption of application maintenance, which is different from the end-user consumption we see here: a better architecture will require less effort from developers when updating the application, knowing that the latter represent a considerable part of the work [31] of IT companies, and therefore of their consumption. Finally, the property liability metric aims to observe the freedom granted to the user by the developer: the more the end user has the possibility of modulating the application as he wishes, the higher the metric will be; and the package depth metric observes the package hierarchy: the deeper the child subpackages are, the more the metric will evaluate a good architecture structure.

All metrics can be grouped into three viewpoints, but with different weights depending on the viewpoint

used. These three points of view are used to evaluate an architecture: reusability (object architecture), composability (component architecture), and dynamicity (service architecture).

Table 2. Architectural Metrics and Extraction Tools

Architectural metrics	Concerns the relationships between classes	Is observable through the structure	Can be evaluated to compare	Will reduce consumption during maintenance
<b>Loose coupling</b>	Yes, in proportion to the number of couplings.	Yes, the coupling is represented by a link.	Yes, the more coupling there is, the lower the value decreases.	Yes, less coupling facilitates replacement and composability
<b>Abstraction of communication</b>	Yes, simplification of communication relationships.	Yes, abstract communication is simplified in the structure.	Yes, the more complex the communications are, the more the metric decreases.	Yes, allows the developer to spend less time on communications
<b>Expressive power</b>	No	No	No	Yes, allows you to understand the architecture more quickly
<b>Evolutionary power</b>	Yes, reducing the number of relationships facilitates evolution.	No	Yes, the more possibilities there are to replace or improve components, the more the metric increases.	Yes, allows components to be easily upgraded rather than creating new ones
<b>Owner's responsibility</b>	No	Yes, the parts that can be adjusted by the end user must be distinguishable.	Yes, the more freedom the user has, the more the metric increases.	No, giving more freedom to the user may require more work from developers
<b>Depth of packages</b>	No	Yes, the package encapsulation is visible in the structure.	Yes, the greater the average depth of the packages, the greater the value increases.	Yes, a better structure directs the developers' choices

In order to compare these metrics [Table 2], seeing if they imply relationships between classes, if they are seen in the structure of the application, and if it applies a value (if not, it only lists their variety).

We have to find out if a metric has an influence on consumption by the end user, for example the expressive power only serves to explain the architecture from the developer's point of view (class names, package names...), so it cannot have any directly in the short term by the end user; however, it can have an influence on another energy consumption: that by the developers, who will work less when maintaining the application if it has good expressive power. We must not only take coupling metrics, but we must diversify our tests in

order to better explore this field. And we need a valuation from the metric, otherwise comparisons would be impossible.

Some of these architectural metrics can be observed through the application architecture. However, there may be several possible architectures for the same application, and the architecture of an application may be non-existent or eroded. This is why we are then interested in architecture extraction, in order to obtain a new architecture from an application. Our goal is to find a link between this extraction and some of the architectural metrics discussed in this section.

## 2.4. Architectural Tools

We have a choice of three types [32] of extraction tools: by correspondence, by groupings, or by conciliation. Correspondence means corresponding the structure of the application to a created conceptual architecture, grouping means forming groups of classes according to their relationships, in order to form groups with strong cohesion with weak coupling between the groups; finally, conciliation consists in first grouping and then reconciling these groups to a conceptual architecture. We then compare these categories according to the metrics mentioned [Table 3].

An example of a grouping tool is Bunch, developed by Mitchell [33], [34]. The latter can use three clustering algorithms: optimal, sub-optimal, or genetic. The optimal algorithm has the problem that it does not scale up, the sub-optimal algorithm has a local optimum problem, while the genetic algorithm does not have these problems. This tool then makes it possible to concretize a process of groupings, and to apply it to the source code of an application in order to extract an architecture. Basically, the Bunch tool performs groupings according to the cohesion/coupling criterion, but it is possible to modify these criteria, based on other metrics, which would be more interesting for our project.

An example of a mapping tool is the Reflexion Model [35], which compares the source model with the conceptual architecture. It was developed in 1995 by Murphy, Notkin and Sullivan in the United States. It has already proven itself on large-scale applications, such as a Unix operating system, or Microsoft Excel. It then displays all the differences: the parts that exist in the source model but not in the conceptual architecture, and vice versa. This tool exists both as a full-fledged application, and as an eclipse plug-in [36].

An example of a conciliation tool is the extension of KNIME, by Mira Abboud [37]. It applies only to Java code, and allows the architect to choose the number of groups to be produced. It is based on KDD[38] (Data Extraction from Knowledge), specializing it: the extracted data is the final architecture, and the knowledge is both the source of the application and the high level knowledge of the architect; this tool starts by extracting the source model, and performs groupings according to its entities, as well as the relationships between them. It is not simply a grouping process, because it takes into account the architect's knowledge, which can influence the final architecture: for example, on the number of groups that must be obtained.

Table 3. Architecture Tool Criteria

Tool Metric	Groupings (example: Bunch)	Correspondence (example: Reflexion Model)	Conciliation (example: Knime)
Coupling	Yes, minimizes coupling between groups.	If the architect decides to group in such a way as to reduce the coupling.	Yes, minimizes coupling between groups.

<b>Abstraction of communication</b>	No	If the architect decides to simplify a complex communication protocol.	If the architect decides to simplify a complex communication protocol.
<b>Expressive power</b>	No	Depending on the name chosen by the architect.	Depending on the name chosen by the architect.
<b>Evolutionary power</b>	Yes, make the groups independent.	If the architect independence of the groups that will need to evolve.	Yes, make the groups independent.
<b>Owner's responsibility</b>	No	If the architect gives the user freedom.	If the architect gives the user freedom.
<b>Depth of packages</b>	Yes, encapsulates groups with strong internal cohesion.	According to the hierarchy of packages chosen by the architect.	Yes, encapsulates groups with strong internal cohesion.

## 2.5. Consumption Measurement Tools

In order to measure the consumption of the applications running, we are looking for a tool that can measure the power consumption of software in real time. To do this, we once again choose selection criteria, which are in fact the same as those used in a previous work of our team, which we felt were relevant. We add to this the current state of development of the tool, which gives us the following criteria:

- **Granularity:** Indicates the lowest level measurable by the application. This is useful if we want to apply our measures to a particular level of application.
- **Coverage:** Indicates which elements of the physical machine are covered by the energy consumption measurement. Important if you want a tool that can measure with high accuracy or not, or only the CPU or memory for example.
- **Development:** Location of the development site of the tool (informative criterion).
- **Year:** Gives the year of publication of the tool, useful if you are looking for a recent tool.
- **Industry/Research:** Informs whether the tool comes from the research field or from the industrial field.
- **Dev status:** Current status of tool development: is it complete? maintained? private?

We thus present the list of the following tools (see [Table 4]).

Table 4. List of Software Consumption Measurement Tools

<b>Name</b>	<b>Granularity</b>	<b>Coverage</b>	<b>Development</b>	<b>Year</b>	<b>Industry/Research</b>	<b>Dev status</b>
jRAPL[39]	Source Code	CPU-RAM-"Uncore"	USA/Brazil	2015	Research	Completed
TEEC[40]	Source Code	CPU, RAM, disque, réseau	France	2015	Research	On Going/private



PowerAPI[41]	Software	CPU	France	2012	Research	On Go- ing/private
Greenspector [42]	Source Code	Application Mobile	Nantes	2011	Industry	On Go- ing/private
Jalen[43]	Source Code	CPU, Disque, network	Lille	2013- 2014	Research	Completed

### 3. Conceptual Framework

#### 3.1. Metrics and Target Element

During these experiments we try to cover all aspects of object programming, based on the criteria announced in the Related Work section. We therefore target the following object elements: Package, Class, Method, Relationship and Inheritance.

Then, we have to select the metrics we will use from those we will record. For this purpose we establish several criteria:

a complete coverage of the different aspects of object-oriented programming via the chosen metrics: These different aspects are: Package, Class, Method, Relationship and Inheritance. Each selected metric measures at least one of these aspects.

Metrics whose measurements are relevant to our research: i.e. metrics that impact the effort of the code when applied, such as time complexity, or the number of operations performed by the application.

The popularity of metrics: This criterion is based on the fact that if metrics are known, then they are used and accepted by the scientific community. For this purpose we use the number of citations of the original paper of the metric, we consider that the metric is recognized and strongly used when its paper exceeds 6000 citations.

We now use these criteria to select the metrics whose application has demonstrated the final quality of the code:

- Weighted method per class (WMC) / McCabe Cyclomatic Complexity: This is exactly what we are looking for since these metrics allow us to detect abnormal values of complexity in certain methods and classes.
- Depth Inheritance Tree (DIT): Allows the analysis of the complexity of the application architecture.
- Response for a class (RFC): The impact of a call can affect consumption if, for example, the call of a method results in very high successive calls of methods, therefore relevant metrics.
- Coupling Between Objects (CBO): Coupling can be a source of complexity.
- Package Depth (PD): Determines the quality of the architecture structure according to the average depth of the packages. The greater the depth, the more well structured the architecture is considered.

Quoted as such, the metrics are estimated on an equal footing in terms of impact on consumption. However, it is possible that several metrics may prove to have a greater impact than others once a certain threshold is reached by the metric. We can therefore determine for some metrics, an optimal weight and threshold:

Optimal threshold: The threshold would correspond to the optimal value of the metric (obtained by measurement with a tool or by hand) for which the application's consumption is the lowest. If this criterion is feasible, it will be very useful for programmers to choose which metric they intend to influence, for example if the optimal threshold of one metric cannot be reached for them for particular reasons, they may seek to fall back on another metric whose optimal threshold is less distant. This criterion also provides a "goal to



achieve" for programmers who want their application to consume as little as possible.

**Weight:** Represents the degree of influence and impact of the metric on the variation in consumption of the application. This criterion, if feasible, would allow a classification of the most influential object-oriented metrics in the energy field, which would help programmers make the right choices for consumption.

To observe architectures, we use an architecture extraction tool, which is the Bunch tool[44], because it allows us to evaluate the coupling, and to minimize it: it allows us to group classes so that groups have a strong internal cohesion and a weak inter-class coupling. This tool is useful to us because it uses interesting architectural metrics. The goal is to find an architecture of the application with as little coupling as possible. Another possible architectural target is the depth of the packages: through our observations through the tool, we must prioritize the application's packages.

Below (see [Table 5]) is represented the coverage of all the selected metrics in relation to our chosen criteria. We can see that all criteria are covered by the selected metrics.

Table 5. Metrics Coverage Table Chosen by Metric Category:

\* estimable weight      \*\* estimable threshold

Measured item Name of the metric	Package	Class	Method	Relation	Heritage
Depth Inheritance Tree* **		X			X
Weighted Methods per Class* **		X	X		
Response for a Class* **		X	X		
Coupling Between Objects* **		X		X	X
McCabe Cyclomatic Complexity**			X		
Package Depth* **	X				X

After these measurements, the next step is to combine architectural metrics and objects. Depending on the results, optimizations should accumulate. For each new version of the application, the consumption must be re-measured, to observe if a change has occurred. Combinations are then possible between architectural and object metrics, but also with metrics that are both architectural and object.

### 3.2. Selected Object Metric Tool

In order to monitor the variation of the metric values we must now choose the tool we will use. To do this, we review the criteria used and the tools found during our research, and we analyze the list to see which tools best meet our needs.

Overall, all the tools found have a large metric coverage, except ckjm which only measures Chidamber and Kemerer metrics, and JDepend which does not measure any metrics in the source code[45], so we leave them out. Then, we are looking for an ideally free tool, so we can remove TestWell CMTJava, SD Metric, JHawk and RSM. So we still have QMOOD+++, Eclipse Metrics and CodeMR. We can already remove QMOOD+++ since it does not cover Java. Eclipse Metrics and CodeMR respect all the conditions and are in addition easy to access

since they can be integrated into Eclipse. We are therefore mainly interested in these two tools.

To separate these two tools and choose which one we will use, we decide to compare them according to the established criteria. To do this, we compare the metrics they measure with the ones we have selected. Eclipse Metrics covers WMC, LCOM, DIT, NOC, Instability and Number of Package. CodeMR covers WMC, LCOM, DIT, NOC, Instability, Number of Package and the last two that interest us, namely RFC and CBO. As a result, CodeMR covers all the criteria we have selected, which makes it the most interesting tool for us and it is the one we will use later.

### **3.3. Selected Architectural Tool**

With regard to low coupling architectural metrics, the extraction of architecture by grouping is interesting to us: the common goal is to reduce to a minimum the relationships between the different groups, which constitute packages. This also makes it possible to observe groups with very strong cohesions, which could be reunited, or on the contrary, groups with weak cohesion, which should be split.

Unlike correspondence and conciliation tools, which use a conceptual architecture, handmade by the architect. These cannot automate any metrics, and require full control of the architecture over the choices considered.

A tool for extracting by architecture that we find is Bunch : it allows different ways to find these groups, by exhaustiveness (tests all possible combinations), or by genetic algorithm (tests a population, and performs mutations). First of all, use the Chava tool[46], which allows you to extract an mdg file. Then, you must give this file to the Bunch software, then choose a grouping method. Once the grouping is complete, Bunch outputs a dot file, representing the groupings made in diagram form. We choose this tool because it allows us to evaluate the weak coupling of an architecture, which is one of the metrics we have listed.

### **3.4. Selected Consumption Tool**

As we can see from the results of our research, several tools have a wider coverage of measured components than the others, it is first of all towards them that we will go. Among them we have jRAPL, Tool to Estimate Energy Consumption and Jalen. Unfortunately, Jalen has not been maintained since 2014, so there is no evidence of its functioning, and TEEC [47], the tool that seems most interesting, is developed privately and not distributed. This leaves jRAPL, a tool that works at the source code level, which allows you to frame the part of the code you want to measure with two tags. It is a precise tool that can be used in conjunction with another tool to compare the consistency and reliability of results.

This leaves PowerAPI, GreenSpector and PeTRa [48]. Unfortunately, again, we are mainly looking for a tool that can measure an application locally, but PeTRa and GreenSpector focus on mobile applications, which limits their usefulness in our case. Finally, although PowerAPI only measures the processor, it is known that the majority of an application's consumption comes from this element, and the tool is relatively widespread and used by several groups. However, despite its use, the results it provides are variable, which requires several measurements and an average to obtain a result. We have therefore been led in parallel to take a more precise tool which is jRAPL.

## **4. Practical Framework and Experimental Evaluation**

### **4.1. Experimentation**

For our experiments, we implement the following methodology: starting from an object-oriented application, we first create a non-optimized version of the code by adding the score of the metric we want to test. We then measure the consumption of this version. Then we create an optimized version of this code, always based on the score of the measured metric. We then measure this second version, and finally compare the

two measurements with each other.

To highlight the impact of the metrics on consumption, we started looking for a test set that was representative and close to real conditions. This test game is an application implementing a Tower Defense game, made during the Cobresun Fall Game Jam 2017[49], [50]. The goal of the game is to protect the White House from terrorists who approach along a specific route, hidden among tourists. As the experiments progress, we modify this application by adding classes and/or complexity in the form of method calls and adding complex calculations to vary the metric scores. We measure the application's consumption before and after the modification. We remind you that to measure the values of the metrics we use CodeMR, and that the consumption used is an average of several dozen measurements of the same test, measurements that are performed by PowerAPI and jRAPL, for more accuracy. The results shown are rounded to the nearest ten Ws, given the uncertainty imposed by the instability of the measurements. All these measurements were performed on the same machine (equipped with an i7-6500u processor), in order to compare the results. In the application, the characters arrive in waves, and in order to homogenize all our tests, we apply them all on the first wave.

Also to structure our tests, we removed the random variables from the application and then identified the part of the code that would be most interesting to modify. We mainly modify the *move()* method of the Citizen class, which allows characters to move forward, since the latter is the most used method of the application, which allows us to observe the impact of metrics on the application's performance. It is also important when carrying out the test sets that they are comparable with each other and produce the same result. Here, we will ensure that each test set performs the same number of times each a precise and complex calculation. We put this calculation a little complex mainly to force the application to make an effort and facilitate the observation of the results.

For the variations in the scores of the metrics measured on the different test sets, we use the thresholds already established by CodeMR, i.e. we compare two versions of the code each time:

- A version with a low metric score
- A version with a high metric score, based on the thresholds estimated by CodeMR

In order to observe the impact of applying several metrics, we start by measuring with a single metric, then with two metrics, then three and finally four. Note that we have made measurements on each of the metrics individually but that we will only show here the most relevant in terms of consumption, namely WMC. We use several different combinations of metrics to maximize results.

#### 4.1.1. Experimentation: WMC

To carry out this experiment, we tried to increase the score of the application's WMC metric. To do this, we add 1000 methods in the class that contains the *move()* method and modify the code so that each *move()*'s call causes the 1000 methods to be called one after the other. We implement them in the same class as *move()* to increase only the complexity and not the coupling between the classes, in order to have precise results, without interference from another metric.

We choose to implement 1,000 of them for two reasons: the first is due to the threshold of the WMC metric according to CodeMR. Beyond a WMC of 120, the class is considered much too complex, and with 1,000 methods we far exceed this threshold. The second reason comes from the results of our other internal tests, 100 methods were not enough to observe a difference on our processor, so we go to 1,000 for this test.

Each method then performs the same complex calculation as the others. We therefore have 1,000 times the same calculation performed per consecutive method call for each call of the *move()* method. To be able to compare the results [Table 6] with a less complex code, we perform another test set, in which we replace the 1000 method calls located in the *move()* method by a for loop of 1,000 iterations, which performs the same complex calculation as the methods at each loop, which again makes 1,000 times the same calculation with

one for per *move()* execution. A single for loop does not increase the WMC score or only slightly.

Table 6. Results of Experiments on WMC

Test set version : WMC	1,000 calculations as a consecutive call of methods (WMC: 1000 in Citizen)	1,000 calculations with a for (WMC: 20)
Consumption	<b>jRAPL: 140 Ws</b>  PowerAPI: 200 Ws	<b>jRAPL: 120 Ws</b>  PowerAPI: 160 Ws

#### 4.1.2. Experimentation: WMC & CBO

To complete our previous experiments, we now combine two metrics which are CBO and WMC. To do this, we carry out four test sets based on those already carried out. In any case, the test set [Table 7] contains 1030 classes, with 1,000 classes coupled to each other, and 1,000 methods implemented in the Citizen class which contains the *move()* method. Here is the more precise definition of the four sets used (see [Table 7]):

- Without WMC metrics and CBOs: The program is not supervised by the metrics at all, so this game is the worst case. We have here 1,000 method calls in the *move()* method to raise the WMC score, and each method calls the *d.coupling1000.method()* method which is the method that causes the coupling chain. The last method in the chain then performs the same complex calculation as used in previous experiments. There are therefore 1000 calculations performed by move execution, via high complexity and high coupling.
- Without WMC metric and with CBO: Here we apply the coupling metric, so we reduce the coupling of the program, but we keep the complexity high. To do this, we simply use the same test set as before but the methods perform the calculation directly, instead of doing it through the coupling string using the *d.coupling1000.method()* method.
- With WMC metric and without CBO: This time, we apply the WMC metric, so we reduce complexity by replacing the 1,000 method declarations and calls by a for loop placed in the *move()* method. However, we leave a strong coupling, so the for executes the method *d.coupling1000.method()* at each iteration, which makes 1,000 calculations per *move()* again.
- With metric WMC and with CBO: Finally, for the last set, we apply both metrics, WMC and Coupling. We reduce their score by merging the two techniques seen above: the 1,000 methods are replaced by a for at 1000 iterations in *move()*, and instead of calling the *d.coupling1000.method()* method to perform the complex calculation, the loop does the complex calculation directly at each iteration, without going through a method call. This avoids complexity and strong coupling.

The columns and rows "Without metrics" therefore correspond to a raw version, without supervision of the values of the program's metrics. The "With metric" columns and rows correspond to a more optimized version of the code.

#### 4.1.3 Experimentation: WMC & DIT

For this experiment we created a test set mixing high complexity and large heritage tree. To do this, we have 20 mother classes in the Citizen class inheritance tree, to raise its DIT to 21, and we have declared the 1000 methods used to increase WMC in the Heritage1 class, which is the highest class placed in the created inheritance tree. This allows us to make sure that the 20 classes added to the tree are taken into account and that java does not jump directly to Citizen by skipping the first 20 empty classes of the tree. The 1000 methods

are then called in the *move()* method of the Citizen class. We compare this test set with the version without depth or complexity, which corresponds to a for of 1000 iterations in the *move()* method. Both the for and the 1000 methods perform the same complex calculation as the other tests. The results of this test can be observed in [Table 8].

Table 7. Results of Experiments on CBO and WMC

Metrics Objects Complexity & Coupling 1000 classes	Without CBO metric	With CBO metric
Without WMC metric	Number of coupled classes: 1000/1000 1000 method calls <b>JRAPL: 160 Ws</b> PowerAPI: 230 Ws	Number of coupled classes: 30/1000 1000 method calls <b>JRAPL: 150 Ws</b> PowerAPI: 220 Ws
With WMC metric	Number of linked classes: 1000/1000 1000 calls per loop for <b>JRAPL: 150 Ws</b> PowerAPI: 220 Ws	Number of coupled classes: 30/1000 1,000 calls per loop for <b>JRAPL: 120 Ws</b> PowerAPI: 160 Ws

Table 8. Results of the Experiments on WMC and DIT

Test set version: WMC & DIT	1,000 calculations in the form of consecutive method calls (WMC: 1000 in Citizen Citizen's DIT: 21)	1,000 calculations with a for (WMC: 20 Citizen's DIT: 1)
Consumption	<b>jRAPL : 140 Ws</b> PowerAPI : 210 Ws	<b>jRAPL : 120 Ws</b> PowerAPI : 160 Ws

#### 4.1.4. Experimentation: WMC & PD

In order to test a purely object metric (WMC) with a purely architectural metric (PD), we then combine the latter two. We then start from the basic application, with a high WMC value, and a low PD value. Then we apply [Table 9] apart from the two metrics, in order to lower the WMC value, and increase the PD value. Then, finally, we combine these two metric applications.

Table 9. Results of the Experiments on WMC and PD

Test set version: WMC & PD	1,000 calculations in the form of consecutive method calls (WMC: 1000 in Citizen)	1,000 calculations with a for (WMC: 20)
-------------------------------	--	--

Without PD metric (Average PD: 1)	<b>jRAPL: 140 Ws</b> PowerAPI: 200 Ws	<b>jRAPL: 120 Ws</b> PowerAPI: 160 Ws
With PD metric (Average PD: 4)	<b>jRAPL: 140 Ws</b> PowerAPI: 200 Ws	<b>jRAPL: 120 Ws</b> PowerAPI: 160 Ws

#### 4.1.5. Experimentation: WMC & CBO & DIT

For this experiment we have combined the three metrics WMC, CBO and DIT. To do this, we have resumed the previous test set and added 20 mother classes to the Citizen class, which places it at a depth of 21 in its inheritance tree. We chose to add 20 classes because the threshold indicated in CodeMR indicates a very high metric score above 20 in depth. The second version with a low inheritance score is again made with a for, so it is the same version as for previous tests. You will find the results of our tests in the table below (see [Table 10]).

Table 10. Results of Experiments on WMC, CBO and DIT

Test set version: WMC & CBO & DIT	1,000 calculations in the form of consecutive method calls (WMC: 1000 in Citizen CBO: 1000 coupled classes Citizen's DIT: 21)	1,000 calculations with a for (WMC: 20 CBO: 30 coupled classes Citizen's DIT: 1)
Consumption	<b>jRAPL: 160 Ws</b> PowerAPI: 210 Ws	<b>jRAPL: 120 Ws</b> PowerAPI: 160 Ws

#### 4.1.6. Experimentation: WMC & CBO & DIT & PD

In order to add a fourth metric, we then add the PD metric: Package Depth; that is, the depth of the packages, which is an architectural metric. The application having a very shallow average depth of the packages, we then apply this metric to increase this average. Basically, the application has an average depth of 1, because all packages are combined at the same level. This practice is bad for large applications in the long term when developers update it, because too many packages at the same level makes the task more complicated for those who have to maintain it, and therefore will make them consume more energy. While if the package depth is higher, it will simplify future evolutions of this application. Here is the result of the application of this metric in [Table 11].

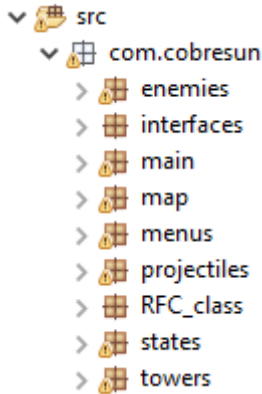
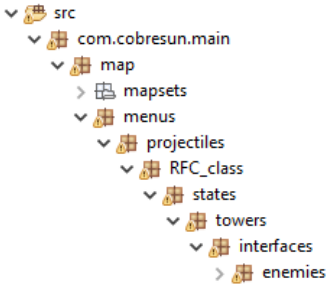
Among the selected metrics, another metric that interested us is RFC. We have made several combinations of this metric with the others, but the results obtained were not convincing enough and we do not consider it necessary to treat them here.

### 4.2. Discussions

We will now discuss the results obtained during the previous experiments. To do this, we will mainly observe the difference in consumption between the tests and not the difference in consumption within the same

test between jRAPL and PowerAPI. Indeed, it seems that PowerAPI and jRAPL do not measure consumption in the same way, which creates a difference in the result between the two tools for an identical test set. However, we observe that, even if the results for a test set are different, the variation between the tests remains the same: if the consumption of jRAPL increases from one test to another, so will that of PowerAPI. During these discussions we will take the jRAPL measurements as the main reference since it is the most accurate tool, and we will indicate the PowerAPI measurements in brackets.

Table 11. Results of Experiments on WMC, CBO, DIT and PD

Test set version: WMC & CBO & DIT & PD	1,000 calculations in form Consecutive call of methods (WMC: 1000 in Citizen CBO: 1000 coupled classes DIT of Citizen: 21 Average PD: 1)	1,000 calculations with a for (WMC: 20 CBO: 30 linked classes DIT from Citizen: 1 Average PD: 4)
Consumption	<b>jRAPL: 160 Ws</b> PowerAPI: 210 Ws 	<b>jRAPL: 120 Ws</b> PowerAPI: 160 Ws 

#### 4.2.1. WMC: Discussion of results

Let's move on to the complexity test on the WMC metric. Comparing the two test sets, we notice that according to jRAPL, the implementation with the 1000 methods consumes 10 Watts-second (40 Ws for PowerAPI) more than the version with the for. The two measures taken with the two tools are consistent and it can be deduced that the version with the highest complexity consumes more than the version with the lowest complexity.

#### 4.2.2. WMC & CBO: Discussion of results

Let us now turn to the analysis of the two metrics CBO and WMC, applied together to the program. If no metrics are applied, the program consumes 160 Watts-seconds according to jRAPL (230 PowerAPI). If we now apply only the WMC metric (thus reducing complexity), the consumption drops to 150 Watts-second for jRAPL (220 Watts-second PowerAPI). This result shows that reducing the complexity of the program as we did has led to a reduction in consumption. If, instead of applying the complexity metric, the coupling metric is now applied, we obtain 150 Watts-second according to jRAPL (220 Watts-second PowerAPI). Again, consumption decreased as a result of the application of the coupling metric, but less than as a result of the application of WMC. We can therefore deduce that a strong coupling has an impact on consumption, but that it



remains less important than that induced by a high complexity.

Finally, the last case includes the application of both metrics. jRAPL announces a consumption of 120 Watts-second (160 Watts-second PowerAPI). Consumption is even lower than when applying a single metric. We can therefore deduce that the reduction in consumption due to the application of each metric adds up, and that it is good to apply both a low coupling and a low complexity for a better consumption of the application.

#### **4.2.3. WMC & DIT: Discussion of the results**

From the results of the test combining WMC & DIT metrics, we can see that the application of the metrics decreases the software consumption from 140 Watt-second (210 Watt-second for PowerAPI) to 120 Watt-second (160 Watt-second for PowerAPI). This reduction is greater than when applying WMC or DIT alone, so there is an additional reduction when applying these two metrics at the same time.

#### **4.2.4. WMC & PD: Discussion of the results**

As we can see from these results, they are identical to the WMC test set. Indeed, the application of the PD metric, whether alone or in combination with WMC, did not change the consumption values. We can then think that the PD metric does not seem to have a direct influence on energy consumption.

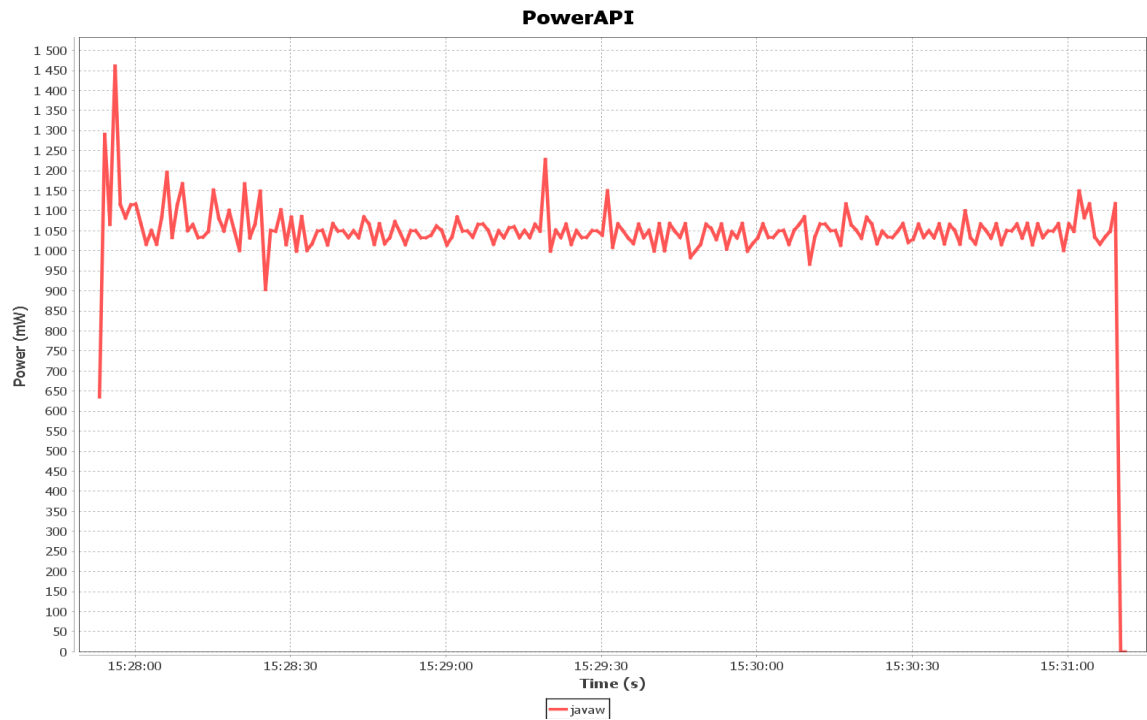
#### **4.2.5. WMC & CBO & DIT: Discussion of results**

During this test set, the version with high coupling, complexity and inheritance depth consumes 160 Watts per second (210 Watts per second PowerAPI) according to jRAPL. This represents 40 Watt-second (50 Watt-second PowerAPI) more than the version of the test set performed with a for, but as many Watts-second according to jRAPL as the previous test. From what we can see here, adding depth and increasing the DIT score does not seem to affect the consumption of the program.

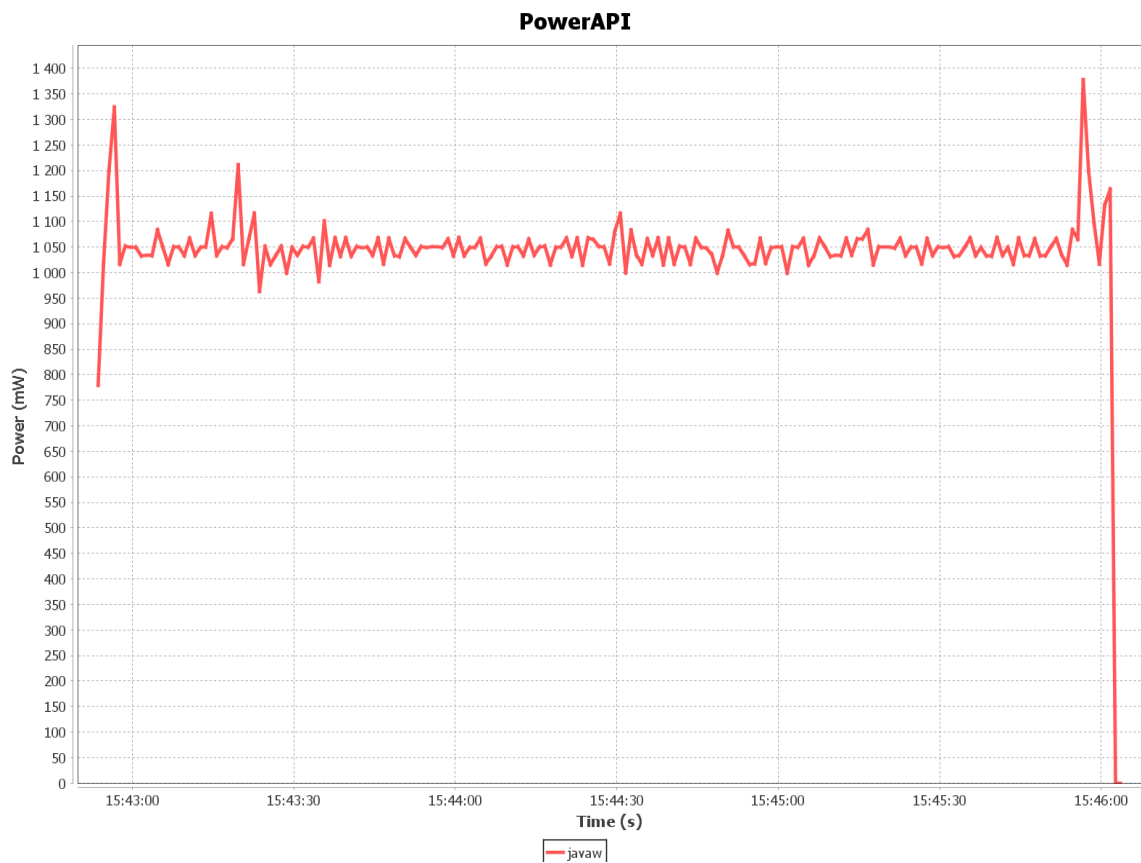
#### **4.2.6. WMC & CBO & DIT & PD: Discussion of results**

Observing this experiment then gives us a stagnation of the results: by applying these metrics, the consumption drops to 120 Watt-second for jRAPL, and 160 Watt-second for PowerAPI, which was already the case without applying the package depth metric. We therefore deduce from this that this 4th metric is architectural and does not affect consumption in these cases. However, it should be remembered that it affects consumption in the long term, during future maintenance of this application.

Finally, here is a table (see [Table 12]) summarizing our results, which shows the percentage reduction in consumption when applying one or more metrics, depending on the combinations we have made and the number of metrics applied. We also have the [Figure 1] which allows us to visualize the difference in execution time according to the application or not of the metrics. Here we present a screenshot of PowerAPI on the four metrics test set (WMC, CBO, DIT, and PD), since powerAPI offers a good graphical representation of consumption in real time. Although we mainly use jRAPL values as indicated earlier, the percentage difference in consumption according to powerAPI gives the same result to the nearest percent, i. e. a 25% reduction in consumption[see Table 11 and 12].



Ordinate: Consumption in mW; Abscissa: time



Ordinate: Consumption in mW; Abscissa: time

Fig. 1. Comparison of powerAPI results.

First Graph: Without Metric (3 Minutes 20 Seconds)

Second graph: with metrics WMC/CBO (3 minutes)

Table 12. Maximum Reduction of the Program's Consumption in Percentage According to jRAPL and PowerAPI When Applying Metrics, Depending on the Combination of Metrics and the Number of Metrics Applied

number of applied metrics	one metric	two metrics			three metrics	four metrics
metrics name	WMC	WMC & CBO	WMC & DIT	WMC & PD	WMC & CBO & DIT	WMC & CBO & DIT & PD
jRAPL	-15%	-25%	-15%	-15%	-25%	-25%
PowerAPI	-20%	-31%	-24%	-20%	-24%	-24%

### 4.3. Conclusion

Our initial conjecture based on the fact that the quality of an object-oriented program, and therefore respecting the recommended metrics, could reduce its energy consumption. Our experiments have confirmed this hypothesis, to the point where this reduction in energy consumption could be around 30%. This experimentation can be continued by performing tests on metrics that have not yet been tested. By considering all the metrics, and by giving weight to each of them, this would make it possible to offer a guide to good behavior to be observed by any developer wishing to optimize the consumption of his programs.

If there is therefore one result to remember when developing a program, like the 5 fruits and vegetables in the field of health, it is to respect and apply at least one metric among the 4 RFC, WMW, DIT and CBO metrics (in decreasing order of impact), better yet combining 2 or 3 of them can then lead to a 28% reduction in energy consumption.

From these results, we can draw some recommendation to reduce application consumption:

- It is preferable to reduce the number of methods (executed) within a single class (RFC)
- It is also advisable to avoid coupling (CBO) between classes when possible, to reduce the complexity of writing a method (WMC) and to reduce the depth of an inheritance graph (DIT)..
- It is recommended to apply the combination of 2 or 3 metrics which would significantly reduce energy consumption
- Finally, as an application evolves, adhering to these rules would also reduce long-term energy consumption.

In conclusion, the use of these "good programming practices" dictated by object-oriented and architectural metrics would significantly affect the power consumption of applications.

### Conflict of Interest

The authors declare no conflict of interest.

### Author Contributions

Mourad Chabane Oussalah conducted the research; Romain Brohan, Ossama Mostafa analyzed the data; Romain Brohan, Ossama Mostafa and Mourad Chabane Oussalah wrote the paper; all authors had approved the final version.

### References

- [1] Fabrice, F., Michelle, D., & Marion, M. (2014). La face cachée du numérique. L'impact environnemental des nouvelles technologies, Montreuil, L'Échappée, coll. *Pour En finir Avec*.
- [2] Roger, C. (2017). 8,6 millions de datacenters dans le monde en 2017. *Hébergement-et-Infrastructure*.

- [3] Gerald, T. (2018). Impact énergétique des Data Center. Retrieved from: <http://www.influenceursduweb.org/impact-energetique-des-datacenters/>
- [4] Wikipedia. (2019). Software architecture. Retrieved from: [https://en.wikipedia.org/wiki/Software\\_architecture](https://en.wikipedia.org/wiki/Software_architecture)
- [5] Wikipedia. (2019). Object-oriented programming. Retrieved from: [https://en.wikipedia.org/wiki/Object-oriented\\_programming#History](https://en.wikipedia.org/wiki/Object-oriented_programming#History)
- [6] Hayri, A., Gülfem, A., Jean, P. G., & Parisa, G. (Sep 2016). TEEC: Improving power consumption estimation of software. *EnviroInfo 2016*, Berlin, Germany. Hal-01496262.
- [7] Araya, M. C., & Bégaudeau, S. B. (2017). Green software. *Sujet de Capstone 2017*.
- [8] Rui, P., Marco, C., Francisco, R., Rui, R., Jácome, C., João, P. F., & João, S. (2017). Energy efficiency across programming languages. *Proceedings of the SLE'17*.
- [9] Wikipedia. (2017). Métrique (logiciel). Retrieved from: [https://fr.wikipedia.org/wiki/M%C3%A9trique\\_\(logiciel\)](https://fr.wikipedia.org/wiki/M%C3%A9trique_(logiciel))
- [10] Riad, B. (2014). Contribution à l'automatisation et à l'évaluation des architectures logicielles ouvertes. *Génie Logiciel*, Université de Nantes.
- [11] Chidamber, S. R., & Kemerer, C. (1991). A metric suite for object oriented design, du livre. *A Metrics Suite for Object Oriented Design*.
- [12] Thomas, J. M. C. (1976). A complexity measure. *IEEE Transactions on Software Engineering*, 2(4).
- [13] Robert, M. (1994). OO design quality metrics an analysis of dependencies. *Proceedings of the Workshop Pragmatic and Theoretical Directions in Object-Oriented Software Metrics*.
- [14] Mark, S. (1999). A practical guide to object-oriented metrics. *IT Professional*.
- [15] Tahvildari, et al. (2000). Categorization of object-oriented software metrics. *Proceedings of the 2000 Canadian Conference on Electrical and Computer Engineering. Conference Proceedings. Navigating to a New Era*.
- [16] Fernando, B. A. W. M. (1996). *Evaluating the Impact of Object-Oriented Design on Software Quality*.
- [17] Kayarvizhy, N. (2016). Systematic review of object oriented metric tools. *International Journal of Computer Applications, Foundation of Computer Science (FCS)*, NY, USA.
- [18] Jürgen, W. (2012-2019). SD Metrics. metrics.
- [19] Squared, M. (1998). List of measured metrics. Retrieved from: <https://msquaredtechnologies.com/index.html>
- [20] Virtual machinery. (1999). Jhawk. Retrieved from: <http://www.virtualmachinery.com/jhawkprod.htm>
- [21] Jagdish, B., et al. (1997). Automated metrics and object-oriented development. Retrieved from: <http://www.drdoobs.com>
- [22] Spinellis, D. D., (2005). Ckjm chidamber and kemerer metrics Software, technical report. *Athens University of Economics and Business*.
- [23] Ingo, P. (2002). Java measurement tool. Retrieved from: <https://www2.informatik.hu-berlin.de/swt/intkoop/jcse/tools/jmt.html#list%20metrics>
- [24] Mark, C. (2008). Metrics list of JDepend. JDepend. Retrieved from: <http://www-igm.univ-mlv.fr/~dr/XPOSE2005/JDepend/presentation.php>
- [25] Eclipse metrics. Retrieved from: <https://github.com/qxo/eclipse-metrics-plugin>
- [26] Testwel, O. (2012). TestWell CMTJava, metrics 'list. Retrieved from <http://www.testwell.fi/cmtjdesc.html>
- [27] CodeMR Team. (2018). Retrieved from: <https://www.codemr.co.uk>
- [28] Théo, C., Maxence, D., William, M., & Fabio, P. (2019). Software architecture metrics: A literature review. Retrieved from: <https://arxiv.org/abs/1901.09050>
- [29] Jagdish, B., et al. (1997). Automated metrics and object-oriented development. Retrieved from:

<http://www.drdobbs.com>

- [30] Offutt, A. J. (1992). Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 1(1), 5-20.
- [31] Wikipedia. (2019). Software maintenance. Retrieved from: [https://en.wikipedia.org/wiki/Software\\_maintenance](https://en.wikipedia.org/wiki/Software_maintenance)
- [32] Mourad, O. (2014). Software Architecture. Wiley Iste
- [33] Mitchell, B. S., & Mancoridis, S. (2006). On the automatic modularization of software systems using the bunch tool. *IEEE Transactions on Software Engineering*.
- [34] Mitchell, B. S., & Mancoridis, S. (2007). On the evaluation of the bunch search-based software modularization algorithm. *Soft Computing - A Fusion of Foundations, Methodologies and Applications*.
- [35] Murphy, G. C., Notkhin, D., & Sullivan, K. (1995). Software reflexion models: Bridging the gap between source and high-level models. *ACM SIGSOFT Software Engineering Notes (Software Eng Notes)*.
- [36] Eclipse, F. (2019) Eclipse. Retrieved from: <https://www.eclipse.org/>
- [37] Mira, A. (2017). Software architecture extraction: Meta-model, model and tool. *Génie Logiciel*. Université de Nantes.
- [38] Usama, F., Gregory, P. S., & Padhraic, S. (1996). From data mining to knowledge discovery in databases. *AI Magazine*.
- [39] Kenan, L., Gustavo, P., & Yu, D. L. (2015). jRAPL. Retrieved from: <http://klu20.github.io/jRAPL/>
- [40] Hayri, A., Gülfem, A., Jean, P. G., & Parisa, G. (Sep 2016). TEEC: Improving power consumption estimation of software. *EnviroInfo 2016*, Berlin, Germany. Hal-01496262.
- [41] PowerAPI. Retrieved from: <https://github.com/powerapi-ng/powerapi-scala>
- [42] Greenspector. (2011). Retrieved from: <https://greenspector.com/fr/>
- [43] Adel, N. (2013-2014). Jalen. Retrieved from: <https://github.com/adelnoureddine/jalen>
- [44] Bunch. Retrieved from: <https://github.com/ArchitectingSoftware/Bunch>
- [45] Mark, C. (2008). Metrics list of JDepend. JDepend. Retrieved from: <http://www-igm.univ-mlv.fr/~dr/XPOSE2005/JDepend/conclusion.php#limitations>
- [46] Korn, J., Chen, Y. F., & Koutsofios, E. (1999) Chava: Reverse engineering and tracking of Java applets. *Sixth Working Conference on Reverse Engineering (Cat. No.PR00303)*.
- [47] Hayri, A., Gülfem, A., Gelas, J. P., & Parisa, G. (2016). TEEC: Improving power consumption estimation of software. *EnviroInfo 2016, Sep 2016*, Berlin, Germany.
- [48] Di, N., Dario; Palomba, Fabio; Prota, Antonio; Panichella, Annibale; Zaidman, Andy; De Lucia, Andrea. (2017). PETra: A software-based tool for estimating the energy profile of android applications. *Figshare. Dataset*.
- [49] Cole, A. (2017). Protect the president. Cobresun Fall Game Jam 2017. Retrieved from <https://github.com/cole-adams/protect-the-president>
- [50] Kiyoshi, S., Kosuke, K., Yu, M., et al. (2021). Advances in computer entertainment. *Localizing Global Game Jam: Designing Game Development for Collaborative Learning in the Social Context*.

Copyright © 2021 by the authors. This is an open access article distributed under the Creative Commons Attribution License which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited ([CC BY 4.0](https://creativecommons.org/licenses/by/4.0/))



**Mourad Chabane Oussalah** received the B.Sc. degree in mathematics in 1983, and Habilitation thesis of computer science from the University of Montpellier- France- in 1992. He is currently a full professor of computer science at the University of Nantes – France- and the chief of the software architecture modeling Team. His research concerns software architecture, object architecture and their evolution. He worked on several European projects. He is (and was) the leader of national project (France-Telecom,Bouygues-telecom, Aker-STX,...), and was member of more than 200 PC.



**Romain Brohan** received the master of computer science degree from the Faculty of Sciences of Nantes, France in 2019. He is currently developer / solution builder for Sopra Steria company in Nantes.



**Ossama Moustafa** received the master of computer science degree from the Faculty of Sciences of Nantes, France in 2019. He is currently developer / solution builder for Savoye company in Nantes. France.