**A dynamic metric to measure the time required to execute mentally a program**

**Abstract**

An important part of this maintenance cost of a software is due to the time needed to debug it. It is probable that this time is strongly correlated to the time needed to understand it. As a complex code is a priori more difficult to understand than a simpler one, we may think that the existing metrics which claim to measure the complexity of comprehension of code would help us to predict the time needed to fix the bugs of a given program.

Unfortunately, several scientific studies recently demonstrated that all these metrics are (at best) weakly correlated with the time needed by a human being to find the returned value of a given function. These results may lead us to doubt the validity of these metrics. At a minimum, it is highly probable that they will not help us to predict the time to debug a program.

In this article, we will first suggest some reasons which could explain why this correlation is so weak. Second, we will define a new metric which is much better correlated with the results of the experiments. To this end, we will introduce the concept of dynamic metric.

Finally, we will demonstrate that a dynamic metric could predict with high precision the time that developers will need to find the returned values of given functions.

This promising result encourages us to continue to work in this direction, hoping to be able in the near future to predict the time needed to debug a real program.

1. Introduction

For now, most of the complexity metrics are analyzing statically the programs. They simply “read” the code, calculate a complexity score for each line, and then return the sum of these scores. We will see why this approach may be relevant to calculate the global complexity of code snippets, but not to estimate the time T needed to find the returned value of a given function.

Hereafter, we will note Tf(i) the average time needed to find the returned value of a given function f for a given input i. More generally, we will note T the function which associates to each function f the function Tf : i -> Tf(i)

Then, we will use the results published by Janet Siemund et al. in 2012 to define a dynamic metric dyn highly correlated with T. In mathematical terms, we will define a metric which gives a score linearly dependent on T with a Pearson coefficient near to 1. For example, if dyn(f) = 2 \* dyn(g), we will predict that the average time to find the returned value of f will be approximately equal to twice the average time to find the returned value of g (i.e. T(f) = 2 \* T(g)).

2. Prior and related work

During the last decades, dozens of metrics were defined in the aim to measure the complexity of the code. Some of them are very simple: count the number of lines of code (LOC), count the number of identifiers, etc. Others are more sophisticated, like Halstead metrics or McCabe metric (also called “cyclomatic complexity”).

In 2017, the software SonarQube defined an extension of the McCabe metric in the aim to explicitly measure the cognitive complexity of the code, i.e. the difficulty for a human being to understand it.

Later, in 2020, the software Genese Cpx defined a new metric which fills multiple gaps of the SonarQube’s metric, and thus returns probably more relevant results.

Of all these metrics, which are correctly correlated with the difficulty to understand the code ? Many scientific publications tried to reply to this question.

In 2017, Scalabrino et al. reached a surprising result: none of the existing metrics seemed to be really correlated with the code comprehension.

 In 2020, Wyrich et al. demonstrated that the SonarQube metric (not studied by Scalabrino et al.) is probably the only one which has a significant correlation with cognitive complexity.

In 2021, Peitek et al. extended the work of Siegmund by using Functional Magnetic Resonance Imaging (FMRI) in the aim to measure the intensity of the activation of the Broca’s areas of developers which were asked to find the returned value of given functions. Like Scalabrino et al., they found no or weak correlation between 41 metrics and the measure of the activation of the Broca’s areas. Even the SonarQube’s metric was not (or weakly) correlated with their measures.

Is it so surprising ? Maybe not. For example, the aim of SonarQube and Genese Cpx metrics is to give a score which represents the difficulty to “understand” code snippets, which could be defined as “understand the specs of the function, and verify if its implementation corresponds to its specs”.  This is very different from “find the returned value of a given function”. Thus, there is a priori no reason to find a correct correlation between these metrics and the time needed to find the returned value of a given function.

2. The goal of the actual metrics

During the last decades, dozens of metrics were defined in the aim to measure the complexity of the code. Some of them are very simple: count the number of lines of code (LOC), count the number of identifiers, etc. Others are more sophisticated, like Halstead metrics or McCabe metric (also called “cyclomatic complexity”).

Later, SonarQube and Genese Cpx extended the McCabe metric in the aim to explicitly measure the cognitive complexity of the code, i.e. the difficulty for a human being to understand it. Hereafter, we will call this kind of metrics the cognition metrics.

As we can see, there are many different kinds of “complexity”. That’s why the expression “the complexity of the code” is a non-sense. We should always specify which kind of complexity we are talking about, and eventually specify if this complexity should be in relation with something happening in the real world.

For example, the metric “number of lines of code” (LOC) simply counts the number of lines of code snippets, and is not intended to measure something else. There is nothing in the real world that this metric should measure.

Inversely, SonarQube and Genese Cpx were developed in the aim to estimate something existing in the real world: the difficulty for a human being to understand a code snippet. That’s why, unlike LOC, we could experimentally demonstrate if these metrics are correctly correlated with what they claim to measure.

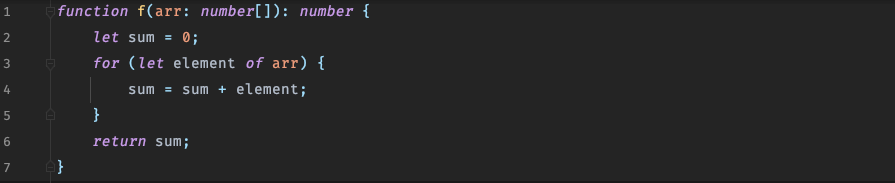
Unfortunately, as far as I know, no experiments  were ever realised in the aim to measure the difficulty of comprehension of the code, defined as “the time needed to understand the specs of the code snippet, and to verify if the implementation of the code corresponds to its specs”. Until we have organised experiments specifically designed to measure these two elements, it will be impossible to affirm that these metrics are correctly measuring what they claim to measure.

3. Why we should not use static metrics when we ask developers to find the value returned by a function

3.1 Loops

Most of the actual metrics statically analyze the code snippets. It simply means that they define some rules to calculate the complexity of each line of code, and then return the sum of these values.

Example :



The complexity of the function f is :

Cpx(f) = Cpx(line1) + Cpx(line2) + ... + Cpx(line7)

This score represents the complexity of the function in its globality. For cognition metrics like SonarQube or Genese Cpx, this score should be correlated with the time needed for a human being to understand what f is doing. This score depends only on the implementation of f.

Remark:

Please note that the lack of comments and the absence of semantics do not let us know what f should do.

First, assume that we ask some developers to find the value returned by f for the input arr = [2, 3]. Now, assume that instead of [2, 3], we gave the input arr = [11, 23, -5, 42, 17, 128, 97, -79]. What would happen ?

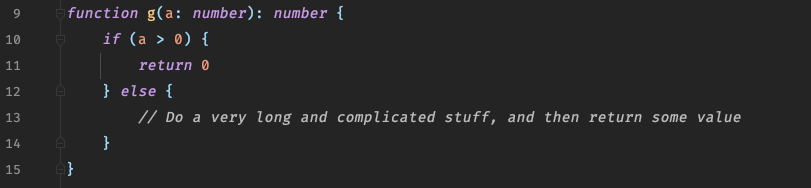
It is highly probable that the average time to find the returned value will be much higher than with arr = [2, 3] (and with more errors). Thus, for the same developers and the same function, the results of the experiment would be completely different by changing the value of the input.

Consequently, why should static metrics be correlated with the results of this kind of experiment ? Their goal is to provide one and only one complexity score for a given function, not multiple scores depending on the values of the inputs.

It is an important reason which explains why we should try to verify the validity of static metrics with this kind of experiment.

3.2 Conditionals

Let’s look at the function g :

Example :

For a cognition metric like SonarQube or Genese Cpx, the complexity of the function g should be very high, because of the complexity of the *else* case.

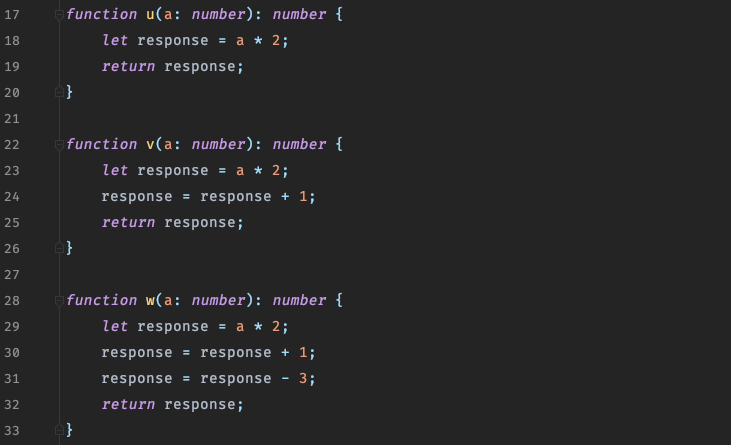
Assume that we do the same experience as above, with the input a = 1. The average time to find the returned value will be very short, because the developers don’t need to read the code which is in the else case. Inversely, if a = -1, it is probable that the developers will need a very long time to find the result..

Like for the previous example with loops, there is absolutely no reason to expect that a static metric will be correlated with the results of this kind of experiment. Moreover, this affirmation is true for every kind of value measured by these experiments: time, intensity of activation of Broca’s areas, etc. We should not expect any kind of correlation between static metrics and results of experiments asking developers to execute mentally a code snippet.

3.3 The illusion of correlation of static metrics

In the above chapter, we saw that we should not expect to find a correlation between the scores provided by static metrics and the results of experiments depending on the mental execution of code snippets. However, some previous studies seem to demonstrate the opposite. It seems that, for particular experiments, some static metrics are really correlated (weakly or strongly) with the results of these experiments. Why ?

In reality, multiple biases exist which are falsing the conclusions. For example, let’s use the LOC metric, and let’s use the set of code snippets below:



 The complexity scores of u, v, and w are :

LOC(u) = 4

LOC(v) = 5

LOC(w) = 6

Let’s call T(a) the time needed by developers to find u(a), v(a) and w(a).

It seems to be clear that for the same input a, the time needed to find v(a) is higher than for u(a), and higher for w(a) than for v(a). Consequently, we might deduce from this result that there is a positive correlation between the LOC metric and the time needed to find the returned value of a given function.

However, this positive correlation is an illusion: it happens because we chose the same value a for the functions u, v and w. Imagine that we used a = 1296,985 for u, 257 for v, and 1 for w; with this choice, we would have obtained the inverse result : the time needed to find v(a) is lower than for u(a), and lower for w(a) than for v(a). Thus,our conclusion would be inverted too.

If the values of the inputs were always chosen randomly, we would not be misled: we would clearly see that no correlation exists. Unfortunately, we can’t randomly choose the inputs, because of the nature of the experiments: we can’t give inputs which induce calculations which are too complex to be done manually by the developers. Each code snippet must not be too long, not too hard to understand, each line must have a low number of identifiers to remember, and the operations between the different identifiers must be simple enough. When the code snippet contains a loop, we must choose an input which implies to execute the loop only a few times. When we write an if-else, we never write a long and complex else block if the developers will never read it because the execution process will never enter in this block.

Consequently, it is highly probable that the main part of the correlations found in previous studies is due to the multiple biases related to the choice of the functions and their corresponding inputs.

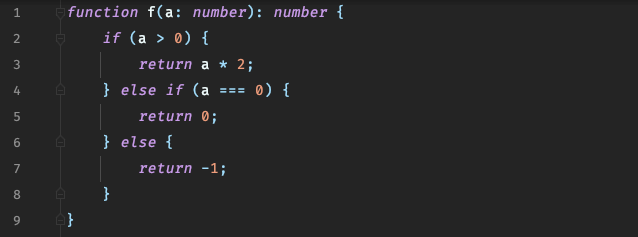
4. Dynamic metrics

4.1 Definition

In the previous chapter, we explained why we should not expect to find a correlation between static complexity metrics and the results of experiments consisting of asking developers to mentally execute a program. However, we could expect it with dynamic metrics.

In this article, I call dynamic metric a complexity metric which depends only on the mental processes that a developer needs to execute a program.

Example 4.1.a

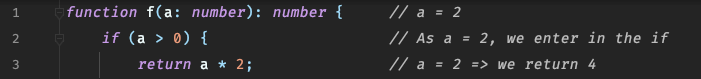


We saw sooner that for a static metric, we simply have

Cpx(f) = Cpx(line 1) + Cpx(line 2) + ... + Cpx(line 9)

Now, assume that we perform an experiment where we ask the developers to find the value returned by f for a = 2. What will be their mental process to find the solution?

Let’s try to represent this process by only writing the lines which are mentally executed by the developer a = 2 :



For a dynamic metric, we will have

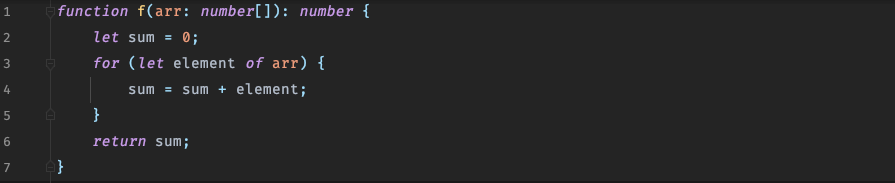
Cpx(f) = Cpx(line 1) + Cpx(line 2) +Cpx(line 3)

Caution: now, the complexity of each line depends on the initial value of a ! For example, with a good dynamic metric, we should have Cpx(f) lower for a = 2 than for a = 8997, because if a = 8997, the multiplication in line 3 takes much more time to calculate.

4.2 Loops and dynamic metrics

Now, let’s look at the usage of dynamic metrics with loops. If the developer needs to read 2 times the content of the loop, a good dynamic metric should usually find a lower value than if the developer needed to read it 15 times.

Example :



Assume that arr = [2, 3]. Let’s try to watch in details the mental process of the developer which tries to find the value returned by f :

Une image contenant texte

Description générée automatiquement

As we can see, the developer needs to read two times the loop, and thus to “manually execute” two times its content. A good dynamic metric should take it into account.

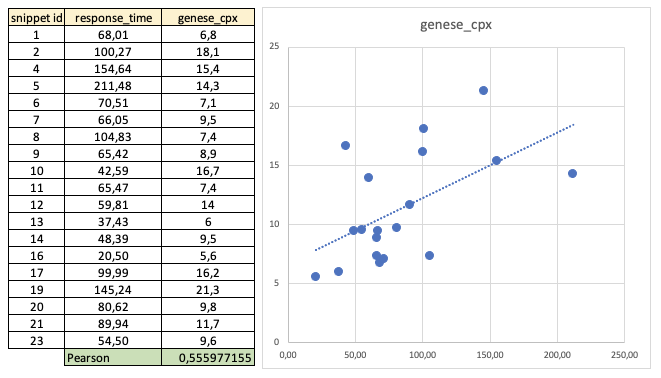
Other remark: when you try by yourself to find the returned value, you notice that you need time because of all the data that you need to store in your memory, and because of the operations that you need to do with them. This time seems to be independent from the control flow (the fors, ifs, switches, etc.). We will use this intuition in the next chapter.

5. Dynamic metric construction

5.1 xxx

In this chapter, we will try to construct a dynamic metric which would be strongly correlated with the results of a real experiment. For that, we will use the datasets published in open-source by Janet Siegmund et al. for their article published in 2012. In their experiment, they asked 41 second-year students from a software-engineering course at the University of Passau to find the values returned by 23 different functions, and they noted the time they needed to do this task (it was not the aim goal of their experience, but it is what we need for now).

First, let’s see what is the correlation between these results and the Genese Cpx metric :



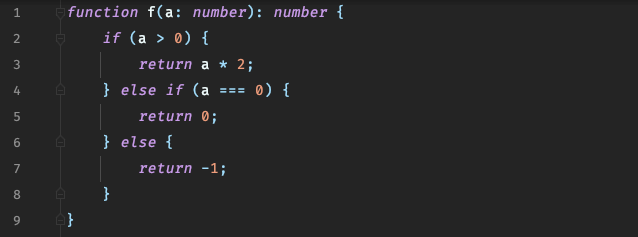
As we can see, there is a clear correlation between the time needed to give the correct response, and the complexity score given by Genese Cpx. The Pearson coefficient is approximately equal to 0,556, which is not so bad (the linear correlation is strong when this coefficient is near to 1, and weak when it is near to 0). However, as we said above, this correlation is probably correct because of multiple biases.

Now, let’s try to define a dynamic metric which would be much better correlated to the results of this experiment. With this new metric, we would be able to predict approximately the time needed to find the returned value of any similar function (in the same experiment with the same developers).

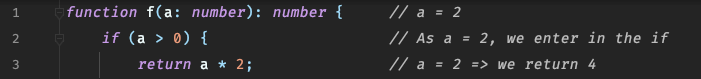
5.2 Conditions

First, we will follow our intuition by using a dynamic metric *dyn* calculating the complexity of each line by counting the number of identifiers used in an operation located in this line (i.e. the weight of each identifier is equal to one).

Let’s remember the example xxx :



Assume that the developers are asked to find the value returned by f for a = 2. The lines which are mentally executed are :



Now, let’s calculate the score of f for our dynamic metric dyn :

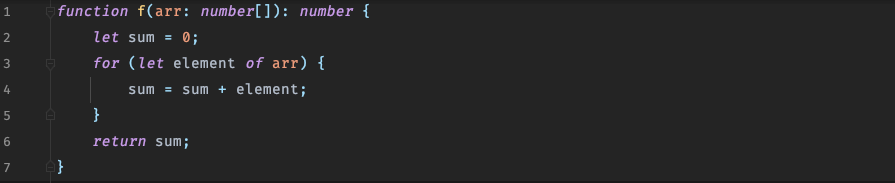
Une image contenant texte

Description générée automatiquement

We find the score dyn(f) = 2.

5.3 Loops

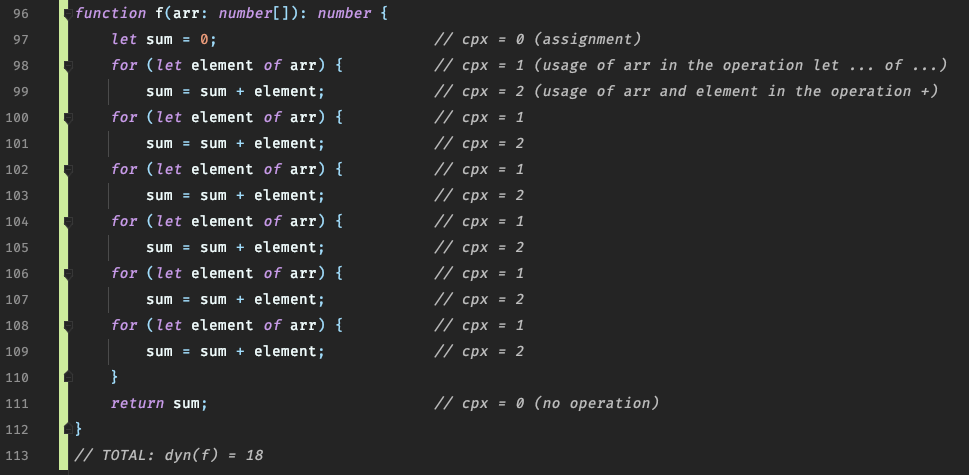
Now, assume that we perform an experiment where we ask the developers to find the value returned by the f of the example xxx for arr = [1, 2, 4, 5, 7, 8].

What will be the mental process of the developers to find the solution?

Une image contenant texte

Description générée automatiquement

Let’s calculate dyn(f) (please note that we count 0 for assignments) :



We find a complexity equal to 18, which seems to be extremely high for a function which is simply adding the numbers of a given array. Where is the problem ?

Intuitively, when you enter in the loop for the first time, you will read it normally. The second time, you will remember what you must do (an addition). And the third time (maybe sooner), you will understand that this function simply adds the numbers of a given array. Thus, you will stop to read the loop, and you will simply mentally add all the other elements of the array.

More generally, we may suppose that each time that we read the same line, we need less effort to understand it. Step after step, we need less time for the same line.

To avoid this problem, we will add a new rule: each time that we will read the same line, we will multiply its complexity score by a coefficient c lower than 1.

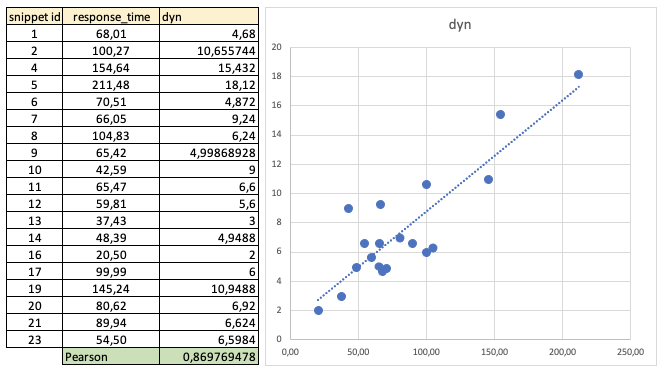
Let’s try with c = 0.5 :

Une image contenant texte

Description générée automatiquement

With c = 0.5, we find a score equal to 5.90625, which seems to be more reasonable.

Now, let’s try our new metric to the results of the experiment :



As we can see, the linear correlation is already much better than with the static metric of Genese Cpx (and even more for the other metrics that we tried). The Pearson coefficient was approximately equal to 0.556, now is approximately equal to 0.87.

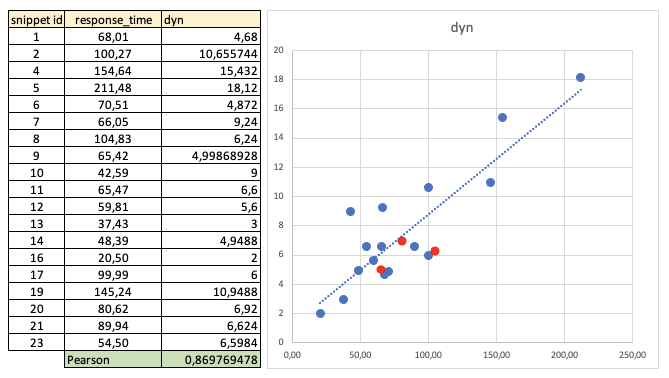
Remark:

After successive tries, we optimized the value of the coefficient c. The best approximation of c in this experiment is approximately equal to 0.4.

5.4 Modulos

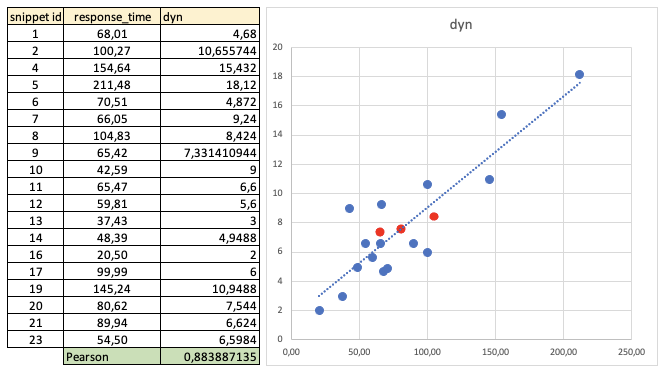
When we tried to reproduce the experiment by ourselves, we noticed that some operators were more difficult to use than others. For example, the operator “modulo” (%) is clearly more difficult to use than the operators + or \*. Our dynamic metric doesn’t take this fact into account. Consequently, the functions containing the modulo operator should have a score abnormally low in our calculations.

We verified this intuition by colouring the dots corresponding to the 3 functions containing modulos in red :



The supposition seems to be correct: the 3 dots are (in average) below the linear regression line.

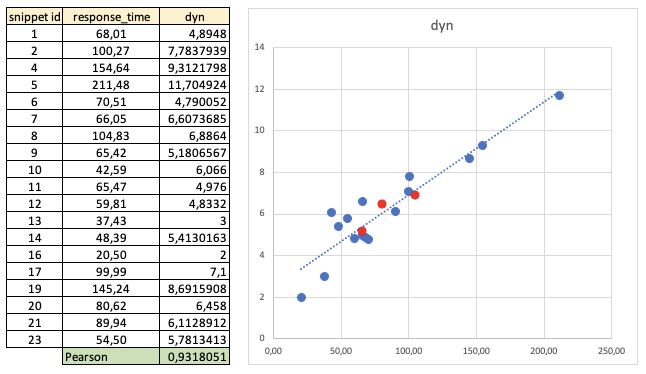
That’s why we decided to improve the performance of our metric by adding a weight to the modulos. After successive approximations, the best value for this weight was equal to 1.4.



The correlation is now a little better : the Pearson coefficient grows from 0.87 to 0.884.

5.5 Reuse of same values

Finally, we remarked that when we used multiple times the same variable with the same value, we needed each time less effort to remember this value. That’s why, like for loops, we used a coefficient c decreasing the weight of the variables each time that we read them when they were keeping the same value. The best approximation of c seems to be approximately equal to 0.7.



Now, the correlation is excellent, with a Pearson coefficient approximately equal to 0.932.

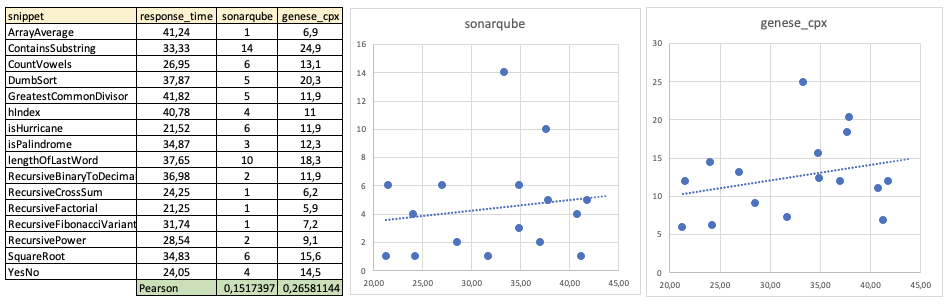
6. Application of the dynamic metric to another experiment

6.1 Application to the Peitek’s experiment

In the previous chapter, we intuitively detected some parameters which seem to have a significant influence on the time needed by the developers to execute some classic mental processes. We added a weight to each identifier used in mental calculations, we used a coefficient decreasing exponentially the score of the lines which were read multiple times, we added a specific weight for the modulo operators, and finally we added a new coefficient decreasing the score of the identifiers which were used successively without value changing.

Naturally, these weights and coefficients were perfectly adapted for this specific experiment, but it is highly probable that it will be very different for other experiments. We tried to verify it by applying our metric to the results of another experiment, realized by Norman Peitek in 2021. The principle of the experiment was the same as in Siegmund's experiment, but the functions and the developers (19 participants) were different.

First, let’s look at the correlation of SonarQube et Genese Cpx static metrics with the results of the Peitek’s experiment :

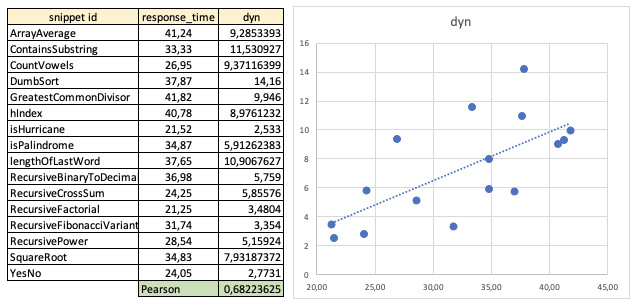


As we can see, in this experiment, the linear correlation of the results with the SonarQube is very weak, with a Pearson’s coefficient approximately equal to 0.152. The correlation with Genese Cpx is significantly better, with a Pearson coefficient approximately equal to 0.266, but we can still consider that the correlation is weak.

As we said in chapter 3.2, it is completely normal to find a weak correlation for static metrics. In the previous experiment, the correlation was much better, but it was probably a result of multiple biases. In Peitek’s experiment, the functions are maybe less similar between them, or maybe the skills of the participants were less homogeneous. It is difficult to identify which are the most important factors which are explaining this difference.

Since the weak correlation between static metrics and Peitek’s results, we expected to find a weak correlation for our dynamic metric too.

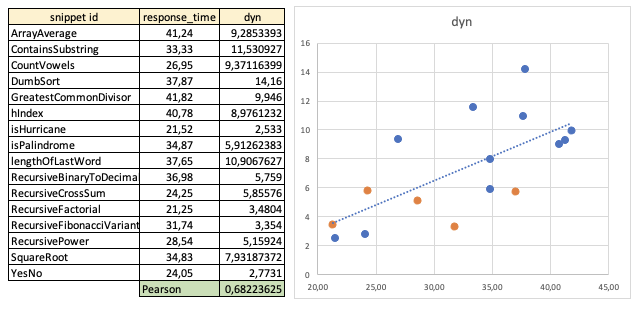
Here are the results :



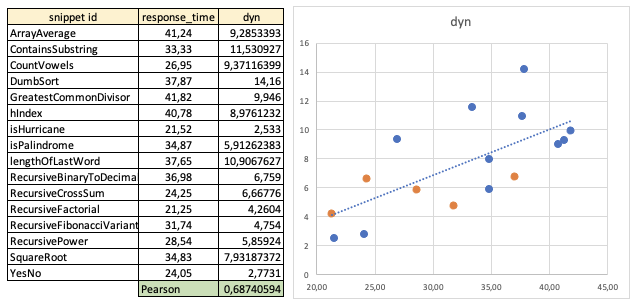
The linear correlation is better than expected, with a Pearson coefficient approximately equal to 0,682. Naturally, it is less than the 0.932 of the previous experiment, but it is still a good score, which is much better than every other static metric that we tried during our verifications.

6.2 Recursion

We tried to identify which could be the code structures present in the functions of the Peitek’s dataset and absent from the Siegmund’s dataset. We identified one: the recursions. Intuitively, the recursivity is a non-trivial operation, which takes time to understand and to use correctly. We coloured in orange the dots corresponding to the functions containing recursions, and we expected that these dots should be located (in average) below the linear regression line.



As we expected, the orange dots are, on average, below the line. Thus, like for modulo operators, we added a weight to the recursions. After multiple tries, the best weight seems to be near 0.5.



As we can see, the correlation is slightly better, but not significantly. Maybe because of the heterogeneity of the code snippets ? For now, it is impossible to conclude.

6.3 Synthesis of the two experiments

We can now do a synthesis of the results of the two experiments :

Une image contenant table

Description générée automatiquement

On average, our dynamic metric has a Pearson correlation with the results of the two combined experiments which is approximately equal to 0.82, which may be considered as a good linear correlation.

7. Conclusion

The lectures of the publications of Scalabrino, Wyrich and Peitek inspired us many questions. Why, for most of the known metrics, the correlation with the code comprehension seems to be so weak, or nonexistent ? Why, for some experiments, the correlation seems to be much better ? The first goal of this article was to reply to these two questions.

To that end, we saw that some of the experiments used in their studies consist of asking developers to find the values returned by different functions. Then, we gave in chapters 3.1 and 3.2 two reasons which may explain why we should not expect to find a correlation between the scores of static complexity metrics and the results of this kind of experiment. In chapter 3.3, we gave other reasons which could explain why, in some experiments, some of these metrics seem to be correlated with the results of the experiments, and why we should not take it into account because of multiple biases.

Later, in chapter 4, we imagined a new way to define a metric which would be strongly correlated with the results of these experiments: the dynamic metrics. Our main goal was to be able to predict with good precision the average time that, for the same experiment, the developers will need to find the values returned by other (but similar) functions.

To that end, we reused the results published by Siegmund in 2012 and Peitek in 2021. By adjusting different weights and coefficients, we were able to define a dynamic metric with a very good correlation with the results of Siegmund’s experiment, and a good correlation with the results of Peitek’s experiment.

We estimate that our dynamic metric is precise enough to make predictions with a correct error margin, but that we need to add more parameters or better optimize our weights and coefficients to be able to make predictions with a low error margin.

 8. Future work

As we said in conclusion, we may try first to optimize our weights and coefficients to improve the correlation between our dynamic metric and the results of these two experiments.

Second, we would like to use other datasets from other studies, in the aim to take into account more parameters, and avoid more biases. The main goal of this work would be to define a metric which would be able to make correct predictions for most of the past and future experiments.

Then, we would like to extend our metric to real projects, in the aim to be able to predict the average time needed by developers to debug the code, which is one of the most fastidious (and costly) tasks for developers. To do that, we will need to analyze the results of more sophisticated experiments, with more complex problematics. For example, we would like to analyze the results of experiments using functions using objects instead of primitives.

Finally, we would like to mix static and dynamic metrics, in the aim to develop a software which would be able to provide information about the global complexity of the functions (with static metrics), and the difficulty to debug a program.

9. Acknowledgments

Thanks to Janet Siegmund, Simone Scalabrino, Martin Wyrich and Norman Peitek for their amazing job, and for publishing their data in open-source. They inspired my work.

[1] S. Scalabrino, G. Bavota, C. Vendome, M. Linares-Vasquez, D. Poshyvanyk, and R. Oliveto, « Automatically Assessing Code Understandability: How Far Are We ? » in Proc. Int’l Conf. Automated Software Engineering (ASE). IEEE, 2017, pp. 417–427.

[2] M. M. Baron, M. Wyrich, and S. Wagner, « An Empirical Validation of Cognitive Complexity as a Measure of Source Code Understandability » in Proc. Int’l Symp. Empirical Software Engineering and Measurement (ESEM), 2020, p. 1-12.

[3] N. Peitek, S. Appel, C. Parnin, A. Brechmann, J. Siegmund, « Program Comprehension and Code Complexity Metrics: An fMRI Study », in Proc. Int’l Conf. Software Engineering (ICSE) IEEE, 2021, pp 524-536

In the example above, a static metric will calculate a score for the if(), a score for the else(), and then will add them. But if your problem is “what is the value returned by this function for a = 3 ?”, you don’t need to read the code of the else() block. The time that you will need to find the returned value does not depend on the complexity of the else() block. That’s one of the reasons which explain why the actual metrics are weakly or non-correlated with the results of most of the scientific experiments, which usually consist of asking developers to give the returned value of a given function for a given input. Many other biases exist, but this one is important enough to explain why we should not use static metrics to measure the time needed to calculate the output of a function.

However, there is a significant difference between “calculate the complexity of a function” and “calculate the average time to find the returned value value for a given input”. In the first case, you try to measure the difficulty to understand the algorithm and/or the specs of a function. In the second case, you take a specific input, and you measure how long it takes to find the correct output. In the first case, you analyze the function in its globality. In the second one, you analyze it for only one input.

3. Dynamic metrics

If we want to estimate the time needed to find the returned value of a given function, we must know its inputs (the values of its parameters). To be able to do that, we must define a dynamic metric, which takes as parameter the inputs of this function. For each different input, the

A first remark: the time needed to check if the returned value is correct is highly dependent on the value of the inputs. For example, if a function does some operations on each element of a given array, the time needed to find the result of the function will be very different if the array has 2 elements, or if it has 23 elements. For this reason, we need another simplification: we will always suppose that the inputs are simple enough to be able to find by hand the returned value. Again, we expect that finding correct answers on simple cases will help us to find later correct answers on complex cases.