Guiding the Evolution of the Software Architecture of Two Large Scale Production Systems

Miguel Pires
IST (ULisboa) and INESC-ID
Lisbon, Portugal

Lucio Ferrão
OutSystems and INESC-ID
Lisbon, Portugal

Rodrigo Rodrigues
IST (ULisboa) and INESC-ID
Lisbon, Portugal

Rui Abreu
FEUP (UPorto) and INESC-ID
Porto, Portugal

***Abstract*—In large-scale production systems, maintaining the quality of the software architecture can be daunting, as the pressure to meet deadlines causes the delivery of new functionality to take priority over thinking through this high-level design. In this paper, we report our efforts at OutSystems, a market leader in low-code platforms, to address this tension in a systematic way. In particular, we developed AGRO, a tool for finding a useful refactoring of components in a software project. The initial design of AGRO is based on a set of existing metrics from the literature, which assess the quality of a software architecture. Our experience in applying these metrics to guide the software architecture of two real-world large-scale software projects led us to conclude that these are not particularly effective in our context. As such, in this paper, we propose two new metrics and report the results and experience of applying them at OutSystems. Our experience showed some preliminary and promising results, to the point where AGRO is now being adapted to become used in production systems.**

***Index Terms*—Software Architecture, Software Metrics, Software Quality**

I. Introduction

In a real-world, large-scale software project, high-level software architecture decisions can have a tremendous impact on the productivity of the development process and, consequently, on the business success of the project [1].

In an ideal scenario, a project's architecture would be carefully thought out in the early stages and remain (mostly) stable during its entire lifetime. However, this is often not possible due to the dynamic nature of this process, and it is fairly frequent that architectural changes done later in the software development life cycle are not done with the same care as in the planning phase [2], [3].

Consequently, the careful maintenance and revision of the software architecture of a project is put at risk by the perception that the more attention is put to this task, the less time developers have to actually program the system itself [4], [5].

Our experience at OutSystems[1], a global software company that is one of the market leaders in low-code application platforms, attests to this tension between planning and maintaining a software architecture, and developing the software itself. The time and energy that is devoted to developing rather than thinking through the quality of the architecture often leads to increasing the time that is spent managing and implementing the necessary changes to cope with the evolution of the software [7], often at a stage when the system has grown to a considerable dimension [6].

In this paper, we report our efforts to address this tension by using (quality) metrics to guide the software architecture of two real-world, large-scale software projects at OutSystems.

In more detail, our first contribution is a tool for guiding the software architecture of projects at OutSystems, called AGRO[2], which is based on a search that efficiently and automatically reorganizes the different components in a system, by trying to maximize a set of metrics that work as a heuristic for the aforementioned search. These metrics must take into account two important properties for a software architecture: (i) the distribution of the sizes of the components that comprise the system and (ii) the number of external dependencies between them. In the literature, this is referred to as high cohesion and low coupling [9], [37]. The purpose of the heuristic is to try to balance the trade-off between these two characteristics, as sometimes minimizing external dependencies can lead to a highly skewed distribution of component sizes, and, conversely, shifting towards a uniform distribution of component sizes can lead to a highly coupled system.

When first approaching this problem, we tried to use an existing metric from the literature, named Decoupling Level [31], and applied it in the context of a large-scale production software project. The second contribution of this paper is that we report on our experience with using an initial version of AGRO based on state-of-the-art metrics, and namely the observation that they fail to meet our requirements of leading to proposed changes that are both useful and practical to system developers and maintainers.

Based on this negative result, the third contribution of our work is that we introduce two novel metrics that were integrated into AGRO. The intuition behind these metrics is that they must consider that this search process must be able to find a small set of movements that can easily be made (i.e., these affect the smallest number of components possible), and yet should be able to achieve a significant impact when it comes to enhancing the system's maintainability. In other words, it is an attempt to answer the question: "what are the best movements one can make in this architecture, when it

[1]https://www.outsystems.com/

[2]AGRO is an acronym for Architecture Guidance and Refactoring at OutSystems

comes to having the least possible cost and highest possible impact in maintainability?"

Our final contribution is an evaluation of applying AGRO and the newly proposed metrics to two large-scale production software projects. The architectural changes that resulted from this were individually analyzed so that one could understand why these were performed; subsequently, we obtained feedback from two lead developers of these systems about the practicality and impact of the proposed changes.

The main conclusions from this study are that, even though our first proposed metric did not achieve attractive results, our second metric has shown some promising results that we believe to be indications that this tool can become useful in production systems. However, before achieving this, our second metric requires some adjustments that we also discuss.

II. Background

A. Architecture Maintainability

A healthy software architecture is the basis for a faster development process, as it leads to a more maintainable system, i.e., its source code is more easily understood and can more easily be modified without introducing new bugs [8]. Several design principles have been established for providing guidance towards a healthier, more organized software architecture. We considered three fundamental maintenance principles [9], [10]for the organization of different components/modules in a system:

a) Cohesion: Measures how strongly the elements inside a module belong together. High cohesion leads to reusability [11]; by allocating elements related to the same concern in a single module, we are making it easy to reuse it, as this module does not need to interact with many other modules.

b) Coupling: Measures how connected distinct modules are. The larger the number of connections between modules, the more likely it is that changes propagate between them [12]. Having changes made in a module propagating to another is a big loss in modifiability, as more time would be needed to fix these changes.

c) Separation of concerns: This principle states that during design, we should divide a software into several modules with the fewest concerns overlapping between them, having, ideally, one concern per module. [13]

B. OutSystems Context

OutSystems is a global software company and the owner of the OutSystems Platform. This platform gives its users the ability to develop large software factories, composed of mobile web and enterprise web applications, through a full-stack, "low-code" visual development process. The OutSystems Platform is considered one of the market leaders [14] in low-code development systems.

An OutSystems Software Factory can be seen through different levels of abstraction: Domains, Services, Modules and Elements. Each domain is composed of services, each service of modules and each module of software elements. Elements can be entities responsible for persisting information in the

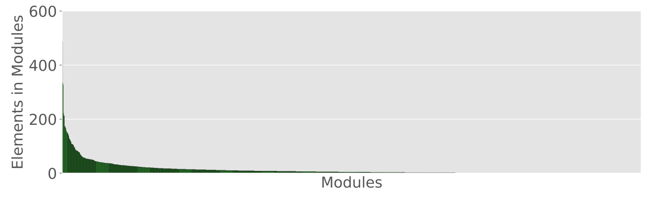


Fig. 1: Distribution of size of the modules in the first project

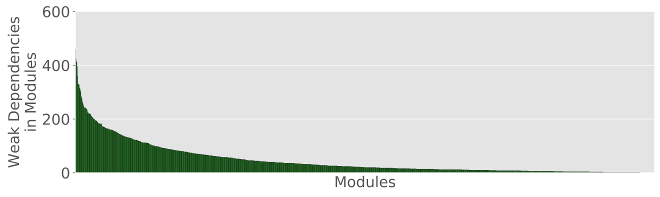


Fig. 2: Distribution of weak dependencies per module in the first project

database, web screens responsible for the user interface or server actions run business logic on the server side. Elements inside a module can be made public, so that other modules can use them. This leads to dependencies between modules and elements, that can be either weak or strong (unlike the former, strong dependencies require importing code).

As a low code platform, many of the developers using OutSystems tend to have little experience in developing and little to no knowledge about design patterns or good practices. In addition, some of the projects developed in OutSystems grow rapidly, to a point where a project that started with small teams of two or three developers can reach a size that may need thirty times the number of developers.

C. Projects analyzed in this work

To guide our design choices and test their effectiveness, we consider two OutSystems projects (for confidentiality reasons, we refer to the projects as first and second). The first one is a particularly large project, with a high number of elements, modules, services, and domains.

A noteworthy aspect of our study is that we could only have access to the dependencies between modules and not the ones between elements. Therefore, even though we will be considering the number of elements within each module, our analysis of the dependencies considers only a two-level hierarchy of services comprised of modules.

Also note that we anonymized all data that could provide hints about the service domain and the details of the solution. In particular, any name used to refer to any element, module or service is replaced with an alias, with the purpose of hiding its true identity.

1) First Project: This project features a total of 19412 elements, 1548 modules, 448 services and 20 domains. Figures 1, 2, and 3 show, respectively, the distribution of sizes (measured in number of elements), weak, and strong dependencies amongst the modules in the system.

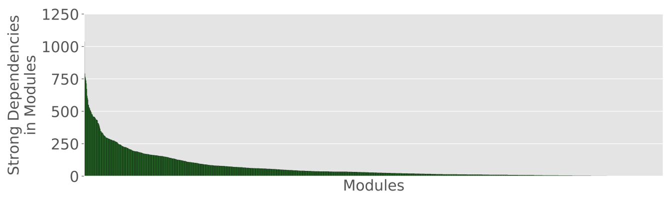


Fig. 3: Distribution of strong dependencies per module in the first project

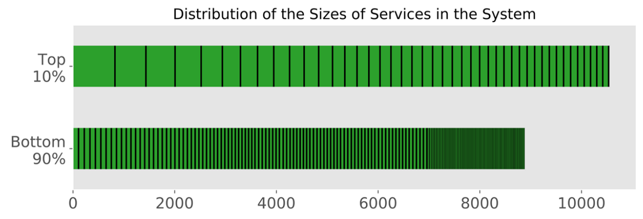


Fig. 4: Number of elements per service of the first project

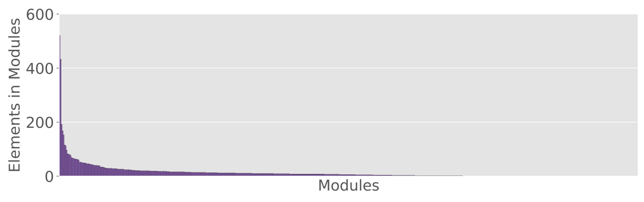


Fig. 5: Distribution in size of the modules in the second project

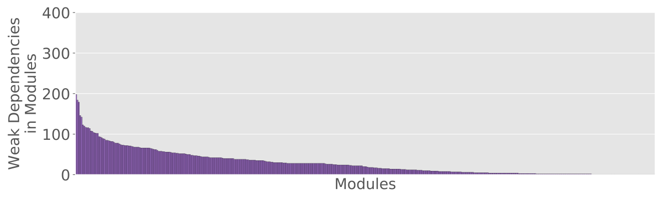


Fig. 6: Distribution of weak Dependencies per Module in the second project

Figure 4 shows a stacked bar chart with two bars with the size of the services of the system. The first bar depicts the stacked size of the top 10% largest services in the system, whereas the second one shows the stacked size of the remaining 90% services.

2) Second Project: As for the second project, as previously mentioned, it is a much smaller project when compared with the first one. This project lacks the domain hierarchy, thus it is comprised of 7539 elements, 581 modules and 264 services.

Figures 5, 6 and 7 show, respectively, the distribution of sizes (number of elements), weak and strong dependencies among the modules in the system.

Again, we have a stacked bar chart divided in two, displaying the size of the services of the system in Figure 8. The top bar represents the size of the top 10% largest services and the bottom one the size of the remaining ones.

III. State-of-the-art Metrics on Healthy Architectures

Several authors have contributed with research proposals directed towards the definition of a healthy architecture or the improvement of an unhealthy one.

Mo & Kazman proposed the Decoupling Level [31], a metric well aligned with Baldwin & Clark's design rule theory [32]. Munialo et al. [15] identified several size-related metrics for service-oriented architectures; Qian et al. [16] also proposed a suite of metrics related to service-oriented architecture systems, however, these were instead based on the coupling that exists between services. The U.S. Air Force proposed DSQI [17], an evaluation on a program's design structure; Bhatia & Singh proposed a way to measure the modularity of a system [18] and Praditwong [19] used an approach based on genetic algorithms, using a metric that measured the quality of the modularization of a system as fitness function. Menzies [20] considered several static code attributes in order to obtain a software defect prediction model. Jureczko & Spinellis [21] took a similar approach, through the use of several metrics gathered from the Chidamber and Kemerer (C & K) [22] and Quality Model for Object Oriented Design (QMOOD) [23] metric suites, and from the work by Tang et al. [24] and McCabe [25]. Sethi et al. proposed a metric based on the percentage of independent modules in a system [26] and McCormack et al. proposed a metric that measures the coupling between files [27], through the use of a square matrix [28]. Finally, Abreu and Goulão created the Modularization Merit Factor [29], [30], which is computed based on the average number of classes per module and the percentage of intramodular coupling instances.

After studying these various metrics, we considered the Decoupling Level [31] to be the most promising. Our choice was justified by the fact that this metric seemed to correctly capture the relevant cost of having strong dependencies and highly coupled components in a system.

A. Decoupling Level (DL)

The main idea behind the Decoupling Level metric (as stated by its authors) is that the smaller the modules, the easier it is to replace them by better versions of them; furthermore, the more independent these modules are, the easier it is to prevent bugs from spreading across modules.

As such, the decoupling level metric measures how well a software is decoupled into modules, considering the two code quality attributes: the size of a module and how dependent the modules are.

In order to properly use this metric, a system must have a Hierarchical Layered Structure[3] [33]–[35]. To compute the

[3]Also called layered architecture, tiered architecture, or n-tier architecture. A layered software architecture consists of various layers, each of which corresponds to a different service or integration. Because each layer is separate, making changes to each layer is easier than having to tackle the entire architecture.

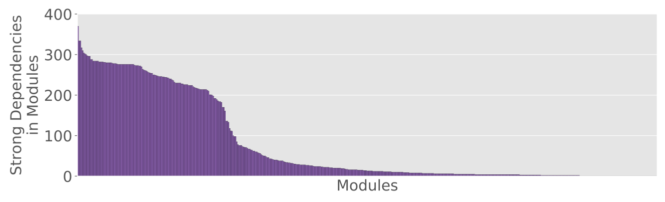


Fig. 7: Distribution of strong Dependencies per Module in the second project

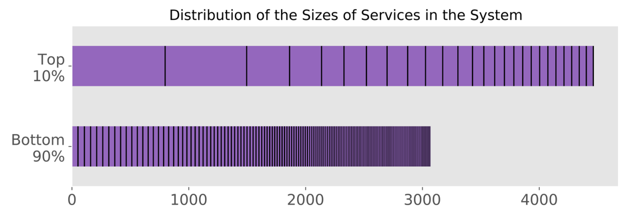


Fig. 8: Number of elements per service of the second project

decoupling level of the system, we must first compute the decoupling level of each layer and sum these results:

$$DL_{System} = \sum_{Li=1}^{k} DL_{Li}$$

Then, for the higher layers, namely for every layer whose modules have depending modules in lower layers, the Decoupling Level is computed in the following manner:

$$DL_{Li} = \sum_{i=1}^{k} [\frac{\#Files(M_j)}{\#AllFiles} * (1 - \frac{\#Deps(M_j)}{\#LowerLayerFiles})]$$

The sum operation represents the summation for every module (M_j) located in Layer i. As for the other variables used:

- $\#Files(M_j)$ - the number of files that compose the module j
- $\#AllFiles$ - the total number of files in the system
- $\#Deps(M_j)$ - the number of files that depend on Module j, directly or indirectly
- $\#LowerLayerFiles$ - the total number of files in lower layers

Finally, the decoupling level of the last layer is obtained by the summation of the size factor of each module that belongs to it. In order to penalize larger modules, a weight is introduced when a module surpasses the threshold size of 5 files (set by the authors of this metric). In particular, when a module has no more than 5 files, its size factor is obtained as:

$$SizeFactor(M_j) = \frac{\#Files(M_j)}{\#AllFiles}$$

whereas, when a module has over 5 files, this value is multiplied by the previously mentioned penalty:

$$SizeFactor(M_j) = \frac{\#Files(M_j)}{\#AllFiles} * log_5(\#Files(M_j))^{-1}$$

IV. AGRO Overview

Our architecture guidance tool (AGRO) conducts a search process by using a blind search algorithm that is guided by a software maintenance metric. This search process attempts to explore the search space of architectural changes considering a set of strategies (or classes of modifications to the system). Before explaining these, we will start by describing the search algorithm that we use to explore the search space.

A. *Search Algorithms*

As search algorithms we have considered both A* [36] and Greedy Best First Search [36], as these are well known and efficient search algorithms. Given that the A* search usually requires more memory, we initially considered the possibility of adopting Greedy Best First Search, which can lead to a solution that is close to optimal, but using only a fraction of the memory used by A*. However, we realized that, in practice, the amount of memory used by A* is far from the limit of the hardware we had access to, and therefore we decided to use A* for our search process.

The next design question we needed to answer was about the nature of the search space. In the next sections, we discuss different strategies in terms of architectural changes, where AGRO can be configured to explore one of these strategies at a time.

B. *Placement of Modules*

The reasoning behind this strategy is to test different placements for each distinct module within the various different services of the architecture, hence obtaining several architectures; then, the architecture with the most favorable value according to the used metric would be considered as the best one.

If we look at the search space of our problem, each node holds a state that is represented by the placement of the modules in services. In one *iteration* of our search algorithms (i.e., when expanding a node), we consider the movement of every module to every service that it is connected to, starting from the state that was held in the node being expanded. Each of these movements represents a visited node.

As we are considering every possible movement when expanding a node, we attempted to make the search process efficient, namely setting a bound on the number of expanded nodes and keeping track of the nodes that were already expanded and visited.

C. *Division of Services*

An alternative strategy employed by AGRO is dividing services, i.e., splitting a target service by converting it into two newly created services, each with a partition of the modules of the original service.

To be exhaustive, this step of the search process must be applied both to every service in the system, and also to every possible split of that service into two separate partitions. Then, our search will gauge how much this division would reduce the value of the metric that is being employed, undo the split, and repeat the process with another partition or another service.

Once the entire search space is explored or an imposed limit for the search is reached, this iteration ends by returning the state that was found with the best value for the metric.

D. *Merging Services*

Another strategy for the search is merging, which is simplified by the fact that this has a local impact: we will only be looking at the final size of the merged service and the loss in external dependencies that such merging leads to.

The algorithm for this merge starts by removing every module inside one of these services, and placing them inside the other service. After this, the service that was left with no modules inside is removed from the system, ending this process with only one service containing every module that was inside the original two services.

In each iteration of this search, every possible coalescing of two services is tested. In the end of this iteration, the search process selects the two services whose merging benefits the selected metric the most and proceeds to the next iteration.

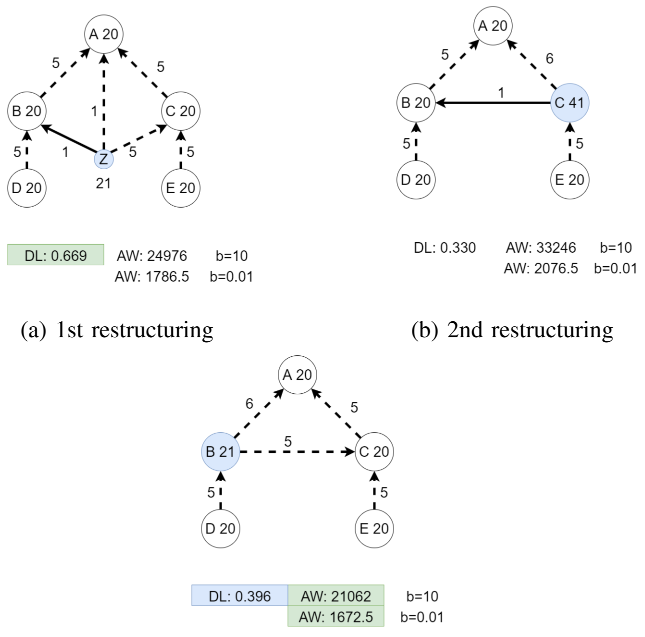
V. Limitations of Decoupling Level

When we first applied AGRO to the codebase of our two projects from Outsystems, we attempted to use the tool with the decoupling level (DL) metric only. However, as we will elaborate in our experimental results, this metric did not yield satisfactory results. In this section, we illustrate the limitations we found in this metric by means of a set of simple examples that help pinpointing those shortcomings.

Figures 9–11 present a set of three small architectures, each of them comprising a group of services (each identified by a letter, and containing a set of modules whose size is displayed next to this identifier) and a distinct recently created module (module z in the figures, marked in blue), initially instantiated as a separate service. In this scenario, the software engineer (or the AGRO tool) wants to place this incoming module among the set of existing services, and therefore each possible placement must be evaluated according to the metrics we are considering.

In these figures, arrows that connect the services represent dependencies. Dashed-line arrows represent weak dependencies, and solid-line arrows represent strong ones (as explained in Section II-B). The number next to these arrows is the number of dependencies of that kind that exist between those services. In our examples, we considered that the size of a service with a strong dependency is the sum of its original size with the size of every service that it strongly depends on, directly or indirectly.

In the first proposed placement for the example architectures, shown in Figures 9a, 10a and 11a, the recently added module (z) remains isolated from the existing services. Below the graph of dependencies, we show the score of the DL metric for the proposed architecture. (We additionally display



(a) 1st restructuring

(b) 2nd restructuring

(c) 3rd restructuring

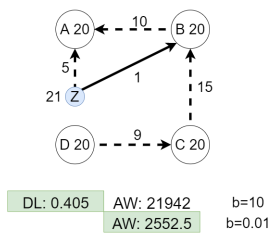
Fig. 9: Example architecture #1

the score of another metric, AW, introduced in the next section.) The remaining subfigures (labeled b–d) represent the placement of module z as part of each of the already existent services of each architecture.

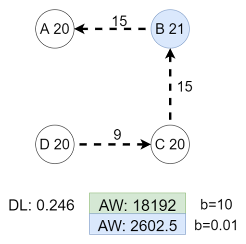
The scores are displayed following a 3-color scale: white, blue and green. If the score of a metric below a placement is green, it means that, according to that metric, the placement is the best one considering all placement possibilities (including leaving the recently added module isolated). A blue score represents that, according to that metric, the placement is the best one, excluding the possibility of isolating the new module.

The results for the DL metric highlight that its effectiveness vary from system to system. In particular, in Figure 9, DL identified the first proposal as the best one, which intuitively corresponds to the most modularized one. However, in Figure 10 (and excluding the possibility to isolate the module), DL scores the solution found in subfigure 10d as the best one, which has the shortcoming of having a less balanced size distribution and higher coupling than some of its alternatives (for instance, the one depicted in figure 10b). Finally, another shortcoming of DL is highlighted in Figure 11, which is that both architectures represented in subfigures 11b and 11c are equally scored according to this metric, even though one has a higher number of dependencies than the other. Thus, these synthetic examples highlight the following limitations of DL (that were also observed in practice):

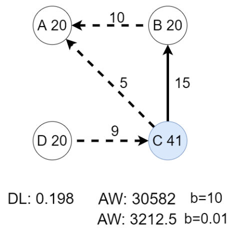
- Assigning higher scores to architectures with a less balanced distribution of sizes and higher coupling.
- Not creating a distinction between architectures with a different number of external dependencies



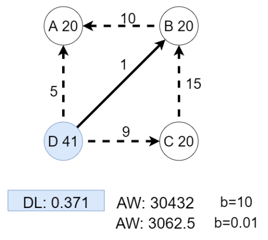
(a) 1st restructuring



(b) 2nd restructuring

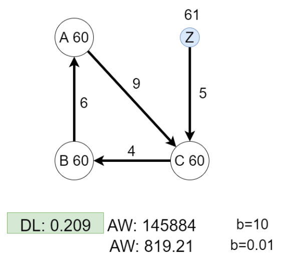


(c) 3rd restructuring

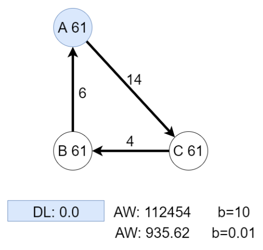


(d) 4th restructuring

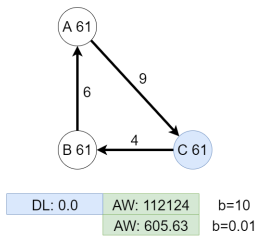
Fig. 10: Example architecture #2



(a) 1st restructuring



(b) 2nd restructuring



(c) 3rd restructuring

Fig. 11: Example architecture #3

Before discussing results with real systems, we next reuse these synthetic architectures to motivate our new proposal.

VI. Proposed metrics

To address these shortcomings, this section presents two new metrics that can serve as an alternative to DL.

A. Architectural Weight (AW)

We start by proposing Architectural Weight (AW), which improves on DL by focusing exclusively on two architectural properties: the distribution of service sizes and the number of dependencies between services.

The idea behind this metric is for its value to grow quadratically as a function of these two properties: the larger and/or more coupled a service becomes, the higher the metric value. (Note that is in the opposite direction of the previous metric, since a lower value implies a healthier architecture.) The reason why we chose a quadratic growth is that, without it, when moving a module between two services, the size of one would increase at the same proportion as the other one decreases, and the aggregate value of the metric would remain the same.

Finally, to generalize this concept to an entire architecture, the AW value of a system is defined as the sum of the AW value of every service in the system.

Given this intuition, for a system with k modules, Architectural Weight is computed as follows:

$$AW(System) = \sum_{i=1}^{k} AW(Service_i)$$

and

$$AW(Service_i) = (\#Deps_i)^2 + b * (Size_i)^2$$

where:

- $\#Deps_i$ is the number of dependencies in $Service_i$ (fan-in + fan-out)
- $Size_i$ is the number of elements in $Service_i$

"b" parameter: This parameter is meant to balance the weight of the dependencies with the weight of the sizes in the metric. During our empirical evaluation, we observed that the unweighted metric can be biased towards either the reduction of dependencies or the distribution of services sizes. We can avoid each of these situations by increasing and decreasing the value held by this parameter, respectively.

Default value for "b": In order to find a default value for this parameter, we experimented the assessment of the artificial architectures in section V with $b = 0.01$ and $b = 10$. We observed that, e.g., in the comparison depicted in subfigure 10c, using $b = 10$, this value led to choosing the architecture with the most services (i.e., the most modularized one) as the best one. This intuitively tells us that this value is giving a fair amount of attention towards the distribution of the module's sizes, which is desirable. For that reason, we will be using 10 as the default value for this parameter.

AW results in the synthetic architectures: As mentioned in section V, when applying the DL metric to assess the architectures in figs. 9 to 11, some architectures with a high number of external dependencies or unevenly distributed component sizes (figs. 10 and 11) were highly ranked. In contrast, by switching to the AW metric, these undesirable properties become penalized. In particular, in Figures 10 and 11, we can see that AW now successfully identifies as the best architectures the ones with the lowest number of dependencies and the more even distribution of module sizes.

B. Distribution Weight (DW)

Next we propose an additional metric, coined Distribution Weight (DW). The rationale behind having another metric is to place more weight on the largest services of the system (which, intuitively, may have grown organically beyond a manageable size), thus guiding potential changes towards modifying them. This metric was conceived with the purpose of evaluating how well a service can be divided, in order to try and maintain the service size distribution as even as possible. DW is quantified using the following expression:

$$\frac{\sum_{services}^{10\%\ largest} Size}{\sum_{service}^{every} Size} + b * \frac{\sum_{services}^{external\ to} Dependencies}{\sum_{service}^{every} Dependencies}$$

This metric was designed so that the modules with little to no dependencies are moved to the newly created service until one of the following happens:

- The modules inside the services to be divided are too coupled to be removed, or
- The newly created service is in the top 10% largest services and does not receive any more modules (because doing so would not affect the value of the metric), or
- The service to be divided has left the top 10% and does not move any more modules (because doing it so would not affect the value of the metric)

One can see that considering only the sizes of the top 10% largest services makes our metric penalize badly distributed systems. Furthermore, this metric also considers the fraction of dependencies that are external, i.e., that span across services, thus making less coupled architectures architectures more valuable according to this metric. Just like our previous AW metric, the higher the value obtained for DW, the less maintainable is the architecture.

"b" parameter: As in the previous metric, DW also has a parameter "b", meant to balance the weight given to the dependencies and sizes.

Default value for "b": We tested several values for the "b" parameter, namely when applying it to divide the service used in Section VII-B1. The value of 0.001 successfully avoided increasing the number of external dependencies; thus, it became the default value for this parameter.

VII. Results

We conducted an experimental evaluation to compare the proposed metrics using production code at OutSystems. Our evaluation is structured according to the strategy of modification of the architecture: for each strategy listed in Section IV, we evaluate the effectiveness of applying different metrics to our real-world codebases to guide that strategy.

For the Placement of Modules strategy, we used our smaller system, detailed in Section II-C2, because this strategy requires a substantial search space. For the other strategies — the Division and Union of services — the larger system, detailed in Section II-C1, was used as the search space for these strategies is significantly smaller in comparison to the Placement of Modules strategy. After obtaining the results, we perform a detailed analysis in section VIII.

A noteworthy technical aspect of our evaluation is that, since the OutSystems architecture is comprised of services holding modules, the natural choice for plugging in values to the formulas of the metrics is to use OutSystems services as modules and OutSystems modules as files.

A. Module Placement Search Strategy

In this experiment, we tested both the DL and the AW metric, with $b = 10$. The DW metric was not tested with this strategy because it is designed to find opportunities to break up modules, and not to change their placement. Given the high cost of evaluating each placement identified by AGRO, we restricted our search to 10 iterations.

1) Architectural Weight Metric Results: When analyzing the set of movements produced by this metric, our measurements showed that, as far as the number of dependencies that became internal are concerned, these ranged between 118 and 477. In particular, the two largest values were 294 and 477, and all remaining movements had the number of external dependencies reduced between 118 and 147.

As for the size of the source and target services, 7 of these movements implied the movement of a module from a smaller service to a larger one. In 8 of the 10 movements, the service where the module originally was had a size lower than the service that module was going to, with all of these 8 original services having either 2 or 3 modules inside.

Throughout these 10 iterations, the modules were moved to one of the same two services. These two services are consumed by most modules and act as infrastructures for the system. Initially, both services were comprised of 2 modules – one with 497 external dependencies, and another with 312.

Analyzing these results, we can point out that these movements significantly reduced the number of dependencies; however, the distribution of the sizes of the various services became even more imbalanced. To gain a deeper understanding, we tried to determine why these movements were the ones chosen by our search process. It turns out that this is due to the fact that the AW metric tries to minimize a factor that is quadratic in the number of dependencies. Consequently, one way to minimize this factor is to perform the change that leads to the most significant decrease in external dependencies. This happened in two of the iterations, where the two modules that were moved were placed in the services they were most coupled to, leading to a significant reduction in external dependencies. In addition, another way to significantly reduce the number of dependencies between modules is to move modules that have a low coupling to the service they are in, which was the case in the other 8 movements. In particular, since the modules moved in these 8 movements were highly coupled to a service that was not the one they were initially placed in, moving them to this service led to significant gains.

In summary, a positive outcome of the proposed changes is that there was a significant amount of dependencies that were lost in the new architectures. However, this led to worsening the imbalance in the distribution of service sizes, especially due to the fact that the same two large services hosted every moved module in these iterations.

2) Decoupling Level Metric Results: Applying the DL metric led to distinct results from AW in terms of changes to the dependencies, since 7 of these movements did not remove more than 7 external dependencies. The remaining three movements removed 19, 32 and 147 external dependencies

As for the sizes of the services involved in these iterations, 8 iterations lead to moving a module from a larger service to a smaller one. The remaining two iterations moved one module from a service with 14 modules to one with 22, and another module from a service with 22 modules to one with 25.

Analyzing the results, we observe that the distribution of the sizes of services became more uniform, leading us to believe that most of these movements were made as a way to even the sizes of the services in the system. When it comes to the dependencies, we did not witness a significant number of dependencies lost in any iteration performed.

B. Division of Services Search Strategy

Based on the observation from the previous section that the DL metric had limited success when applied to our architectures, we decided to drop DL from the set of metrics we use in the remaining set of experiments, and instead focus on comparing AW to the Division Weight metric presented in Section VI-B (which we can now apply since it was created with the purpose of measuring the impact of service divisions).

In this set of experiments we were able to search through 100 iterations, and we set $b = 10$.

1) Architectural Weight Metric: To evaluate the AW metric with the division of services, we selected a service from our first project that we believed could be easily divided without creating new external dependencies, as 20 of its 42 modules have no internal dependencies. The division of this service using $b = 10$ (the original default value) lead to 8 external dependencies between the resulting services. We tested several values for the "b" parameter, including the value of 0 and every tested value returned this solution.

Analyzing these results, we observe that while the original goal of the AW metric was reach a low number dependencies between the resulting services, since it contain a quadratic factor in the number of external dependencies, the search ended up steering towards an even distribution of a not very small number of dependencies. We can thus conclude that the AW metric is not well suited to the strategy of dividing or joining services.

2) Distribution Weight Metric: Using the default value of 0.001 (as justified in Section VI-B), we tested the Division of Services strategy in the larger OutSystems project from Section II-C1. Recall that this project has a total of 448 services, and its top 10% largest services comprise 44 services. Furthermore, the smallest of these has 116 elements, thus, for a service to reach the top 10%, it must have over 116 elements.

Our results showed that the 1st division created a service with 93 elements and another with 115 elements. In the 2nd division, one service was comprised of 86 elements and the other one of 115. In both the 3rd and 4th iteration, only the largest module was moved: in the 3rd iteration, with its 106 elements, it left the other service with 81 elements; in turn, in the 4th iteration, the module that was moved to a new service had 124 and left the other service with 69 elements. In our last iteration, a total of 48 elements were moved to a new service, leaving the other service with 270 elements.

None of these iterations led to the creation of new dependencies, as every module that was moved to the new service had no internal dependencies to its original service. Furthermore, every divided service belonged to the top 10% largest services. We therefore conclude that this metric succeeded in dividing some of the system's largest services without increasing the amount of external dependencies in the system.

C. Union of Services Search Strategy

For our last strategy (union of services), we used Distribution of Weight Metric with the same parameter as in the previous section ($b = 0.001$), using again the larger OutSystems project from Section II-C1.

The 5 iterations of this search using our Distribution Weight Metric coalesced 8 different services into 3.

The first of these 3 resulting services was obtained from the 1st and 4th iterations. Joining 3 of the original 8 services led to a service with 111 elements and 515 internal dependencies. A total of 467 dependencies became internal.

The second resulting service happened in the 2nd iteration and led to a service with 113 elements and 232 internal dependencies. A total of 126 external dependencies were turned into internal ones.

Finally, the last service resulted from the 3rd and 5th iterations. It comprised 83 elements and 248 internal ones; with this decision, 222 external dependencies became internal.

When analyzing these results, a noteworthy observation is that none of the newly created services entered the top 10% that is being considered by our metric; furthermore, our search process managed to achieve this while reducing a total of 815 external dependencies in the system.

VIII. Discussion

Having selected several metrics and performed several tests and case studies in both real and synthetic architectures, we conducted a critical assessment of the observed results. To help us with this task, we asked one of the engineers involved in the second system (from Section II-C2) to validate the results obtained regarding the movement of modules strategy in that same architecture. As for the Services Division and Union strategies, we asked the software architect responsible the first system (from Section II-C1) to share insight on the results of applying these strategies on this system through a survey.

A. Movement of Modules

We start by analyzing the results from Section VII-A, corresponding to the strategy of reorganizing the first architecture through the movement of modules.

1) Decoupling Level Metric: In terms of the effectiveness of this strategy with the DL metric, we observed that movements obtained with this metric have the same issues as the ones obtained with the AW metric. For instance, 7 of these movements were made on modules that held core functionalities of the service or that had a large number of dependencies, which, as discussed, is not a recommended action. Besides this, there was one iteration that moved a module from a smaller service to a larger one.

Small impact movements: In 6 of the 10 movements, less than 7 dependencies were hidden. In a system with thousands of connections, hiding such a small number of dependencies has no significant impact on the overall architecture of the system. Furthermore, if we consider that we restricted our search to the 10 movements that bring the highest positive impact on the system, this result comes as unsatisfactory.

2) Architectural Weight Metric: As for the performance of this search strategy with this metric, after analyzing the changes that occurred in the architecture, we could see that several of the decisions taken by our process differ from recommended practices when trying to reorganize an architecture:

a) Moving the main module of a service: Every Service has a set of modules that hold a very important role within it, as these are consumed by or depend on many of the other modules in the service. These modules are usually the ones with the highest number of dependencies inside a service, thus, moving them is not a decision that is often recommended, as there are more modules that will be affected by this change.

b) Imbalance in the Sizes of Services: We want to reduce coupling between modules and one way to obtain this is by moving a module that is highly coupled to a service into that same service. However, this decision may compromise the balance between sizes of services, and outweigh the benefits of the reduction of coupling between them.

c) Placement in infrastructural services: Many modules are consumed by most services in the system as they hold basic and essential code. Many of these modules are placed together in a large infrastructural service, whose main purpose is to be consumed by other services. For this reason, it is essential that only modules with essential code for the system are placed there, as otherwise we would have code relative to a specific feature or function being imported by every module. For this reason, we should avoid placing modules in these services unless these modules are really consumed by most services.

B. Division of Services

We will now discuss our survey results, regarding the division of services using the Distribution Weight metric.

1) Distribution Weight Metric: The feedback obtained in our survey tells us that 4 out of the 5 iterations did not contribute towards a more maintainable system and the service that was chosen to be divided was not a good one. Throughout the evolution of the system, no suggested division was performed, as none would make the system more maintainable.

There was however, one iteration that was approved by the engineer taking the survey, stating that this iteration made sense as it separated different concerns that were in the same service. When analyzing the survey results and the proposed divisions, we can highlight the following common situations:

a) Default Libraries: Two of these iterations divided default OutSystems libraries. These have a small number of dependencies and require little maintenance from the developers. We can argue that these divisions come as irrelevant, since there are other services requiring more maintenance. Furthermore, our enquired engineer added that if we were to split these large libraries, these splits should be more modular than the ones obtained by our metric.

b) Separating concerns: Two iterations proposed to divide a service that was related to the same concern and was being managed by the same team. Doing this would lead to this concern being present in two services, which is undesirable.

After this feedback, we believe that it would be useful to extend this metric with a term related to the maintenance cost, namely to ignore modules with minimal maintenance.

C. Union of Services

In Section VII-C, we performed a search process on the Architecture from Section II-C1, using our Distribution Weight Metric. Next, we present the feedback we obtained from one of the engineers involved in the design of this architecture.

1) Distribution Weight Metric: These 5 iterations performed with this metric joined 8 services into 3. The feedback we obtained in our survey tells us that 2 of these 3 new services did not contribute towards a more maintainable system, as these were the wrong services to be joined. Again, none of these suggestions were performed, as most would not make the system more maintainable. Only one suggestion was accepted by our survey taker, who stated that even though complex, the union appeared to make sense.

a) Testing Services: The 2 resulting services that did not contribute towards a more maintainable system resulted from the union of services that belong to the system's testing space. This space is not part of the production environment, as such, the services inside it require little maintenance.

Overall, our experiments with the Distribution Weight metric showed promising results, as in both considered exercises, the main weakness identified in this metric was not being able to determine the services that required the most maintenance. For this reason, we believe that AGRO can become useful in practical scenarios, if we complement our DW metric with a maintenance cost term.

IX. Conclusion

Maintaining the quality of the software architecture is a challenging task, as the pressure to meet deadlines causes the delivery of new functionality to take priority over thinking through this high-level design. In this paper, we reported our efforts at OutSystems, a market leader in low-code platforms, to address this tension in a systematic way. In particular, we proposed AGRO, a tool for finding a useful refactoring of components in a software project.

After applying a set of metrics from the literature to guide the software architecture of two large-scale software projects, we conclude that these are not particularly effective in our context. We proposed two new metrics and report the results and experience of applying them at OutSystems.

The observations from our empirical study shed light on the ability and effectiveness of the proposed metrics to guide changes to the software architecture in order to maintain its quality throughout the life-cycle of software development. Our experiments showed some promising results, and AGRO is now being adapted to become used in production systems.

In the future, we think it would be beneficial to complete AGRO by testing other search algorithms, such as the Genetic Algorithm, Differential Evolution, Ant Colony Optimization and Simulated annealing; or to combine the current metric with maintenance metrics, such as the number of changes each module has been subject in the last months, the impact it has on the system or the proximity between the name of the modules and the name of the service it belongs to. Furthermore, we would like to strengthen the explainability of AGRO, so that every suggested architectural decision would be supported by an automatically generated justification.

References

[1] Bass, L., Clements, P., & Kazman, R. (2003). Software architecture in practice. Addison-Wesley Professional.

[2] Williams, B. J., Carver, J., & Vaughn, R. B. (2006). Change Risk Assessment: Understanding Risks Involved in Changing Software Requirements. In Software Engineering Research and Practice, 966-971.

[3] Heijstek, W., & Chaudron, M. R. (2010). The impact of model driven development on the software architecture process. In 2010 36th EUROMICRO Conference on Software Engineering and Advanced Applications, 333-341.

[4] Baskerville, R. L. (2006). Artful planning. European Journal of Information Systems, 15(2), 113-115.

[5] Waterman, M., Noble, J., & Allan, G. The Effect of Complexity and Value on Architecture Planning in Agile Software Development. Agile Processes in Software Engineering and Extreme Programming, 238-252.

[6] Hoch, D. J., Roeding, C., Lindner, S. K., & Purkert, G. (2000). Secrets of software success. Boston Harvard Business School Press.

[7] Ropponen, J., & Lyytinen, K. (2000). Components of software development risk: How to address them? A project manager survey. IEEE transactions on software engineering, 26(2), 98-112.

[8] Visser, J., Rigal, S., Wijnholds, G., van Eck, P., & van der Leek, R. (2016). Building Maintainable Software, C# Edition: Ten Guidelines for Future-Proof Code. O'Reilly Media, Inc.

[9] Kulesza, U., Sant'Anna, C., Garcia, A., Coelho, R., Von Staa, A., & Lucena, C. (2006). Quantifying the effects of aspect-oriented programming: A maintenance study. In 2006 22nd IEEE International Conference on Software Maintenance, 223-233.

[10] Du Bois, B., Demeyer, S., & Verelst, J. (2004). Refactoring-improving coupling and cohesion of existing code. In 11th working conference on reverse engineering, 144-151.

[11] Gui, G., & Scott, P. D. (2006). Coupling and cohesion measures for evaluation of component reusability. In Proceedings of the 2006 international workshop on Mining software repositories, 18-21.

[12] Taube-Schock, C., Walker, R. J., & Witten, I. H. (2011). Can we avoid high coupling?. In European Conference on Object-Oriented Programming, 204-228.

[13] Tekinerdogan, B., Scholten, F., Hofmann, C., & Aksit, M. (2009). Concern-oriented analysis and refactoring of software architectures using dependency structure matrices. In Proceedings of the 15th workshop on Early aspects, 13-18.

[14] Vincent, P., Lijima, K., Driver, M., Wong, J., & Natis, Y. (2019). Magic Quadrant for Enterprise Low-Code Application Platforms. Retrieved December, 18, 2019.

[15] Munialo, S. W., Muketha, G. M., & Omieno, K. K. (2019). Size Metrics for Service-Oriented Architecture. International Journal of Software Engineering & Applications (IJSEA), 10(2), 66-82.

[16] Qian, K., Liu, J., & Tsui, F. (2006). Decoupling metrics for services composition. In 5th IEEE/ACIS International Conference on Computer and Information Science and 1st IEEE/ACIS International Workshop on Component-Based Software Engineering, Software Architecture and Reuse (ICIS-COMSAR'06), 44-47.

[17] Puntambekar, A.A. (2010) Software Engineering And Quality Assurance. Technical Publications.

[18] Bhatia, P., & Singh, Y. (2006). Quantification Criteria for Optimization of Modules in OO Design. In Software Engineering Research and Practice, 972-979.

[19] Praditwong, K. (2011). Solving software module clustering problem by evolutionary algorithms. In 2011 Eighth International Joint Conference on Computer Science and Software Engineering (JCSSE), 154-159.

[20] Menzies, T., Greenwald, J., & Frank, A. (2006). Data mining static code attributes to learn defect predictors. IEEE transactions on software engineering, 33(1), 2-13.

[21] Jureczko, M., & Spinellis, D. (2010). Using object-oriented design metrics to predict software defects. Models and Methods of System Dependability. Oficyna Wydawnicza Politechniki Wrocławskiej, 69-81.

[22] Chidamber, S. R., & Kemerer, C. F. (1994). A metrics suite for object oriented design. IEEE Transactions on software engineering, 20(6), 476-493.

[23] Bansiya, J., & Davis, C. G. (2002). A hierarchical model for object-oriented design quality assessment. IEEE Transactions on software engineering, 28(1), 4-17.

[24] Tang, M. H., Kao, M. H., & Chen, M. H. (1999). An empirical study on object-oriented metrics. In Proceedings sixth international software metrics symposium (Cat. No. PR00403), 242-249.

[25] McCabe, T. J. (1976). A complexity measure. IEEE Transactions on software Engineering, (4), 308-320.

[26] Sethi, K., Cai, Y., Wong, S., Garcia, A., & Sant'Anna, C. (2009). From retrospect to prospect: Assessing modularity and stability from software architecture. In 2009 Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture, 269-272.

[27] MacCormack, A., Rusnak, J., & Baldwin, C. Y. (2006). Exploring the structure of complex software designs: An empirical study of open source and proprietary code. Management Science, 52(7), 1015-1030.

[28] Steward, D. V. (1981). The design structure system: A method for managing the design of complex systems. IEEE transactions on Engineering Management, (3), 71-74.

[29] Abreu, F. B., & Goulão, M. (2001). Coupling and cohesion as modularization drivers: Are we being over-persuaded?. In Proceedings Fifth European Conference on Software Maintenance and Reengineering, 47-57.

[30] Abreu, F. B., & Goulão, M. (2001). A merit factor driven approach to the modularization of object-oriented systems. L'Objet, 7(4), 455-476.

[31] Mo, R., Cai, Y., Kazman, R., Xiao, L., & Feng, Q. (2016). Decoupling level: a new metric for architectural maintenance complexity. In 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), 499-510.

[32] Baldwin, C. Y., Clark, K. B., & Clark, K. B. (2000). Design rules: The power of modularity (Vol. 1). MIT press.

[33] Wong, S., Cai, Y., Valetto, G., Simeonov, G., & Sethi, K. (2009). Design rule hierarchies and parallelism in software development tasks. In 2009 IEEE/ACM International Conference on Automated Software Engineering, 197-208.

[34] Huynh, S., Cai, Y., & Sethi, K. (2008). Design rule hierarchy and analytical decision model transformation. Technical Report, Drexel University.

[35] Cai, Y., & Sullivan, K. J. (2006). Modularity in design: Formal modeling and automated analysis. Technical Report, University of Virginia.

[36] Russel, S., & Norvig, P. (2013). Artificial intelligence: a modern approach. Pearson Education Limited.

[37] Praditwong, K., Harman, M., & Yao, X. (2010). Software module clustering as a multi-objective search problem. IEEE Transactions on Software Engineering, 37(2), 264-282.