On What Kind of Applications Can Clustering Be Used for Inferring MVC Architectural Layers?

Dragoş Dobrean
Babes-Bolyai University
Cluj-Napoca, Romania *dragos.dobrean@ubbcluj.ro*

Laura Dioşan
Babes-Bolyai University
Cluj-Napoca, Romania *laura.diosan@ubbcluj.ro*

Abstract

Mobile applications are one of the most used pieces of software nowadays, as they continue to expand, the architecture of those software systems becomes more important. In the fast-paced domain of the mobile world, the applications need to be developed rapidly and they need to work on a wide range of devices. Moreover, those applications need to be maintained for long periods and they need to be flexible enough to work and interact with new hardware. Model View Controller (MVC) is one of the most widely used architectural patterns for building those kinds of applications. In this paper, we are analysing how an ML technique, in fact clustering, can be used for detecting autonomously the conformance of various mobile codebases to the MVC pattern. With our method *CARL*, we pave the way for creating a tool that automatically validates a mobile codebase from an architectural point of view. We have analyzed *CARL*'s performance on 8 iOS codebases distributed into 3 different classes based on their size (small, medium, large) and it has an accuracy of **81%**, an average Mean Silhouette coefficient of **0.81**, and an average Precision computed for each layer of **83%**.
Keywords: Software Architecture Recovery, Mobile SDK, Clustering.

1. Introduction and Context

Mobile applications are one of the most important software products nowadays used by hundreds of millions of people [16]. These applications need to be flexible to change as the entire domain is fast-paced, there is a constant flux of new hardware and software updates that these applications need to take advantage of, to stay relevant. The software architecture used for building such products plays a major role in their success, as its heavily related to the flexibility and maintainability of these software product [3, 22, 23]. By respecting an architectural pattern the codebase becomes more testable and more extensible in the areas which are important for the business. In addition to this, having a well-defined architecture in place helps the new or inexperienced developers write new code more easily by having clear architectural guidelines in place. Moreover if implemented correctly the architectural pattern makes the entire codebase more open and accommodating to changes (hardware and software).

With this study, we pave the way for building an architecture checker system (that highlights architectural issues, early in the development phase) and examining if there is a correlation between the number of components, size of the codebase, and the performance of our approach. An automatic system that detects architectural layers and issues in the mobile codebase can be integrated into CI / CD pipelines for improving the architectural health of those projects or used as a quality gate. In a previous work [11], the architectural layers of mobile codebases were detected by some heuristics, but they have quite a few downsides. It needs to have the correct heuristics in place to yield good results. However, those might vary from one codebase to another and needs developer interaction (for configuring the heuristics), that could also in-

troduce some bias. Furthermore, the detection approach will have to be adjusted every time the codebase evolves to another architectural pattern or when a new architectural layer / sublayer is introduced.

Thus, arose the need for an automatic and non-deterministic approach, in which machine learning techniques are used for automatically distributing the components of a mobile codebase into architectural layers. *CARL* is using information from the mobile SDKs as they contain more types of components, than web SDKs for instance and the majority of the mobile applications use them for building the UI interfaces. The proposed approach could also work on other platforms such as desktop applications. However, for this study, we are continuing to analyze mobile SDKs as they are richer and the vast majority of the mobile projects are using them. In this paper, we strengthen our findings regarding *CARL*, which is an unsupervised approach, involving a clustering step. Related work in applying clustering techniques for detecting architectural layers already exists [15, 19]. However, none of those approaches takes into consideration the SDK information or is specifically designed for mobile applications.

We are extending our findings by better validating our approach and checking on what types of mobile applications does this method work better. We have enriched the validation phase by adding more codebases (we have added 5 new projects) and split them into 3 categories (small, medium, and large applications, based on the number of components). Moreover, we are enhancing the analysis mechanism by adding external metrics (Silhouette and Davies-Bouldin indexes for measuring the clustering performance and homogeneity, and completeness scores for measuring the integrity of our approach). Furthermore, we address the evaluation of our approach as well as its applicability and efficiency on different types of applications with new research questions. Last but not least, we look at the correlation between the number of components in a codebase and the performance of our approach, to identify classes of applications in which our method works best.

The main **research challenges** are: assesing the performance of our detection method *CARL* on different-sized codebases and investigating if there is a correlation between the size of the codebase and the performance of *CARL*.

The following Section presents the background for our work using text mining and unsupervised ML methods in architecture reconstruction and detection as well as insights about the analyzed architectural pattern Model View Controller (MVC). Section 3 talks about our approach *CARL*, the evaluation process is outlined in Section 4 together with the conducted experiments. In Section 5 we present the downsides of our approach, and at the end of the paper (Section 6) we talk about our conclusions and some directions for further work.

2. Background

For the purpose of this work, our focus was on one of the most prolific presentational architectural patterns, MVC [9], one of the most widely and commonly used architectural pattern that represents the foundation for more specialised architectural patterns such as Model View Presenter or Model View View Model. MVC is a software architecture frequently used in client applications that separates the elements of the codebase in 3 layers: Model (business logic), View (user input / output) and Controller (mediator between the Model and the View). MVC continues to be one of the most used architectural patterns on mobile platforms, a survey conducted in 2019 and filled by over 2000 developers showed that over 66% of the developers use MVC as an architectural pattern, being the most popular of all the architectural choices [32].

Architecture reconstruction methods are two-folded: identification of architectural modules (by clustering) and identification of architectural rules among modules. Several approaches were proposed to support the architecture reconstruction process based on static analysis. A part of these approaches exploit the structural information extracted from the codebase ([7], [21]), another part exploit the lexical information ([8]), while some recent approaches exploit both of

them ([15], [19]). Furthermore, few of them consider the architectural style of the system under analysis. Because our empirical validation is performed in the context of three iOS applications and Apple's flavour of MVC which can be viewed as a linear layered architectural pattern, we describe in what follows several approaches that take into account the particularities of the layered-based architecture. Similar to the general systems case, the approaches developed for analysing the layer-style projects take into account either the structural features of the codebase ([18], [34]) or the hybrid (structural and lexical) features ([4], [26]).

Even if these approaches are able to identify the codebase's components that belong to each layer, in the case of mobile applications they cannot be properly applied. For instance, the methods developed in [31] produce clusters with a nested structure that does not fit the MVC architecture that can be viewed as a layered-architecture. To our knowledge, there are not any approaches that use the information from SDK for inferring the architectural layers of a codebase, hence why we could not compare our method against another proposal. Tools such as ARCADE [6] are not developed for mobile platforms and they are not fit for purpose when applying them to mobile development languages such as Swift or Kotlin.

Vocabulary The following definitions are used throughout the remainder of the paper: **component** – the building block of software architectures and patterns [17]; it can be a programming language structure such as class, struct, another system or subsystem (eq. CarBookingViewController - class, User - model entity, struct); **layer** – a set of components which perform the same role in the software system [27] (eq. Model layer from MVC - responsible for the business logic of the application, Coordinator layer - responsible for keeping the state of the application and deciding when and how to alter it); **module** – a set of components that are related, fulfill the same general purpose; **class** (of applications) – a category of applications based on the number of components.

3. CARL - A Smart Framework for Automatic Detection of Architectural Layers in Mobile Codebases

Mobile applications are usually client – presentational applications, they commonly use monolithic architectures and are frequently self-contained. The purpose of *CARL* is to identify the architectural layers in those codebases based on the scope of the composing components. Such a system could be a real help for both the developers which would be constrained to write better code as well as for the management team which could see in real time the architectural health of the codebase.

The applicability of *CARL* was studied and explained in one of our previous articles [13], our research has shown that over 60% of the students struggle with architectural issues, and over 94% of them make those mistakes, unknowingly in the projects they hand in (according to their instructors). In terms of professional developers, all the responders to our survey stated they encounter architectural issues at least once a month. Over 90% of the participants in our study agreed that a tool like *CARL* would help them write better code.

We base our research on the idea that components from the same architectural layer should have similarities (the same purpose, the same interface, or they achieve the same system goal). In order to detect those similarities and not to restrict our research to a single platform or a single programming language, *CARL* performs a static code analysis and groups the elements from the codebase on layers by exploring a multi-modal knowledge extracted from: API contracts, public / private method and properties definition, inheritance mechanism.

CARL's mechanism involves more stages. In the first phase, *CARL* identifies the program elements (classes, procedures, data structures, etc.) contained in the source code and constructs the abstract representation that reflects the dependencies between these elements. This phase can be done with certain tools, specific for different programming languages and development environments (such as SourceKitten for Swift, or kotlinx.ast for Kotlin). The second stage of

the process is to extract information related to the source code architecture. A directed graph is constructed based on the result from the previous stage. This graph reflects the relationships between the program elements. The first two steps of *CARL* are similar to those of *mACS* [11] and more details about them can be found in [12].

The next step, that of architecture's identification, uses an unsupervised learning technique, in which structural and lexical information associated with different program elements is used and which results in a clustering of the elements into the basic categories of an architectural model. ML algorithms are one of the most powerful tools we have for finding patterns in data. Since every project is different we have decided to use an unsupervised and autonomous algorithm for finding relations between the elements of the codebase. Unsupervised denotes from an ML perspective that the algorithm needs no prior knowledge for finding patterns in data, while autonomous means there is no developer interaction needed while using the proposed approach. These two attributes (unsupervised and autonomous) enhance *CARL* to be a smart system / framework.

In terms of Machine Learning, the investigated association component-layer is considered a clusterisation problem: grouping components into clusters without an *a-priori* knowledge of the category they belong to. The clustering process follows three important steps:

- extract relevant information (features) from the raw data source; in our case, each codebase represents an instance of the dataset;
- use all the features or just some of them to analyse the similarities among components and to build a clustering model; the output generated in this step is, in fact, a division of the data;
- validate the obtained clusters by using evaluation standards.

For the clustering part of our approach, we have discovered using a trial and error approach that Agglomerative Clustering [5, 25] works best for splitting the components of the codebase into architectural layers, and this is what *CARL* uses.

For pilotating our approach, we have conducted a preliminary study where, based on the available information extracted from the codebase, we searched the combination of features which would yield the best results in terms of correctly splitting the codebase into architectural layers [12]. For this part of the investigation, we have used a validation application from which different sets of features were been extracted. This case study is actually an iOS E-Commerce application which has over 20.000 lines of code and uses MVC architecture (this application was also included in the experiments we conducted in the current paper). Two senior iOS developers with over 5 years of development experience had constructed the ground truth by manually labelling the components of the validation application. They tagged the components of the codebase separately and afterwards, they cross checked the differences and agreed on the correct architectural layer where a component should be placed. In their analysis the definition of a codebase component (interface, inheritance, name) had a higher importance than the body of the functions since the code written in a component can present various design and architectural smells, since the purpose of *CARL* is to place the components in the right architectural layers not to highlight architectural drift.

The preliminary study [12] resulted in 5 different approaches which incrementally improved the accuracy of the clusterization process on the validation application as seen on Table 1:

- ***CARL*** F_1 (*Number of dependencies*): how many dependencies a component has with each of the other codebase's components; we use $F_1(c_i)$, for all components c_i ($i \in \{1, 2, \ldots, n\}$) of the codebase;
- ***CARL*** F_2 (*Presence of dependencies*): the type of dependencies that it has with each of the other codebase's components;

Table 1. Analysis of all five versions of *CARL* on the validation application

	Model		View		Controller		Accu-
	Precision	Recall	Precision	Recall	Precision	Recall	racy
CARL-F_1	0.50	0.01	0.22	1,00	1,00	0.10	0.24
CARL-F_2	0.49	0.93	0.17	0.09	1,00	0.08	0.46
CARL-F_3	0.62	0.75	0.33	0.53	0.65	0.22	0.52
CARL-F_4	0.70	0.93	0.84	0.83	0.99	0.56	0.78
CARL-F_5	0.76	0.99	1.00	1.00	0.99	0.57	0.85

- ***CARL*** F_3 (*Name distance*): how many dependencies it has with each of the other codebase's components and the distances between the name of the current component and the names of the other codebase's components; the name distances are computed by using specific text mining methods [20].
- ***CARL*** F_4 (*Keywords presence*): the features F_3 are enriched by the keyword-based features;
- ***CARL*** F_5 (*SDK inheritance*): the features F_4 are enriched by the SDK's inheritance-based features.

For all these feature subsets, the same agglomerative clustering algorithm is applied in order to detect the clusters which encode architectural layers. In the initial study the focus was on analysing MVC architectures; ergo, the number of clusters were set to 3 (Model, View and Controller). After the clustering process is completed, *CARL* assigns responsibilities to the layers based on the types of their components and their inheritance (eq. the cluster with the most items that inherits from UIViewController is marked as Controller layer).

The best subset of features identified in the preliminary study is used in a second more complex one, when eight application are investigated (see the details in Section 4.3). Finally, the findings are analysed and possible improvements are suggested. For the rest of the study we are focusing only on the iOS platform and we are analysing Swift codebases. However, our proposed approach can be easily extrapolated to other platforms which use SDKs for building user interfaces and are presentational software products.

4. Numerical Experiments

We validate our approach by applying it to various MVC iOS codebases. MVC was studied by both practitioners and academia [9] and paves the way for analyzing more specialized architectural patterns that descend from MVC (MVVM, MVP, etc.). Our analysis was focused on the iOS platform; however, the same process can be applied to any other platform which uses MVC and SDKs for building UI interfaces. We extend our previous study [12] with new research questions. In this study we are interested in the effectiveness of our approach on more different sized codebases, we examine the performance of the clustering process and inspect its applicability and efficiency on 3 classes of applications (small, medium, large) of the process by answering the following research questions:

- RQ1 - How effective is the proposed categorization method compared to manual inspections? [1]
- RQ2 - What is the clustering process performance when using the proposed approach? [2]

[1] Metrics like accuracy, precision, and recall will be used for this purpose

[2] The performance can be evaluated through metrics like homogeneity, completeness, Silhouette Coefficient score, Davies-Bouldin index

- RQ3 - On what class of applications does this method work best?
- RQ4 - What is the reliability of *CARL* when compared to *mACS*?

4.1. Analysed Codebases

We have enriched our analysis with five new codebases: Firefox, Game, Stock, Education and Apple's Demo for AR / VR applications. The analysis was conducted on applications from different domains, different sizes, different development styles (private, open-source, Apple's example). For this study, we were interested in analyzing mobile codebases that follow the rules of the MVC architecture, as it one of the most used architectural patterns of those types of software products [32]. To our knowledge, there does not exist a selection of repositories used for analyzing iOS applications. iOS was chosen as opposed to Android as it implements MVC more consistently and Apple encourages the developers to be aware and respect architectural patterns, especially MVC [1].

Table 2 presents the characteristics of the codebases: blank – refers to empty lines, comment – represents comments in the code, code states the number of code lines, while components represent the total number of components in the codebase. In addition to this, we've also split the codebases into 3 different classes, small (Demo and Game), medium (Stock, Education, Wikipedia, and Trust), and large (E-Commerce, Firefox). We've split the codebases into classes based on the number of components because we are interested in whether or not *CARL* is able to correctly place all the components in the right architectural layers, not on the complexity of the components and how big they are (number of lines).

Table 2. Description of investigated applications together with the split by number of components

Application	Blank	Comment	Code	#comp	Class
Demo [2]	785	424	3364	27	Small
Game [private]	839	331	2113	37	Small
Stock [private]	1539	751	5502	96	Medium
Education [private]	1868	922	4764	105	Medium
Wikipedia [33]	6933	1473	35640	253	Medium
Trust [30],	4772	3809	23919	403	Large
E-Commerce [private]	7861	3169	20525	433	Large
Firefox [24]	23392	18648	100111	514	Large

4.2. Evaluation Metrics

During the validation, we are interested in both the correctness and the integrity of the categorization process. Three metrics are of interest in the correctness validation: accuracy, precision, and recall [14], while the integrity is measured through some specific clustering scores: homogeneity and completeness [28]. Besides, we were also interested in the performance of the clustering process, and we have used ML-specific metrics for analyzing this Silhouette Coefficient (Silh. Coef.) score [29] and Davies-Bouldin Index (Davies) [10].

The Silhouette Coefficient score indicates how well are the components placed, while the Davies-Bouldin Index expresses whether or not the layers were correctly constructed. The homogeneity score denotes if a layer contains **only** members that are correctly placed, while the completeness score expresses the degree to which all the members of a layer are assigned by the categorization method to **the same** layer.

The accuracy, precision, and recall metrics are calculated for every architectural layer. For instance, if a codebase has 100 components in the Model layer (in the ground-truth) and *CARL* manages to correctly identify X, then the accuracy is X%. In the same manner, we calculate the precision and recall against the number of elements in each layer.

The ground truth was constructed by manually inspecting each component of the codebases by two senior iOS developers (with over 5 years experience on the iOS platform) who reached a consensus regarding the type of each component. Moreover, for the private projects, we also had developer documentation and internal architectural guidelines to aid the process, since those projects were developed by a mobile specialized software company.

4.3. Empirical Evaluation

After the experiments were ran we analysed the data and answered the research questions based on the results obtained. This subsection presents our findings.

RQ1 - How effective is the proposed categorisation method compared to manual inspections?

Using *CARL-F_5* approach we have obtained an average accuracy of **81,17%** on all the analysed codebases. We have observed that on one of the most complex and largest projects — Firefox — we have obtained an accuracy of **91,17%**. In the case of the worst performing codebases analysed with our method, we have found out that the elements did not have a consistent naming convention. They did not contain many elements which had similarities between names or contained one of the used keywords.

Our method works better in the cases where the codebase is consistent in respects to naming conventions for each architectural layer. Table 3 presents the results of the proposed method on all the analysed codebases.

Table 3. *CARL-F_5* results in terms of detection quality

Codebase	Model		View		Controller		Accuracy
	Precision	Recall	Precision	Recall	Precision	Recall	
Firefox	0.92	0.95	1.00	0.99	0.73	0.64	0.91
Wikipedia	0.78	0.83	1.00	0.54	0.83	0.98	0.82
Trust	0.79	0.69	0.38	0.66	0.62	0.57	0.66
E-comm	0.76	0.99	1.00	1.00	0.99	0.57	0.85
Game	0.87	0.95	0.75	1.00	1.00	0.75	0.88
Stock	0.64	0.98	1.00	0.59	1.00	0.61	0.76
Education	0.55	0.98	0.50	0.05	0.95	0.44	0.62
Demo	0.96	1.00	1.00	0.75	1.00	1.00	0.96

From Tables 2 and 3 we can easily deduce that proposed approach works on both large projects as well as smaller ones: we have good accuracies for large projects (Firefox) as well as for the smaller ones (Game). Our approach however, is greatly impacted by the coding standards and the consistencies of the project. The Education codebase had a recall of only *0.05* which is extremely low. We have analysed the codebase to find the cause and we have discovered that the components did not have a coding standard and a naming convention in place. Also the number of elements in the codebase was small – ergo the clustering algorithm has problems in correctly splitting the elements. The scores chosen for the features detection in the preliminary study [12] might need to be adjusted for some of the analysed codebases, in order for those to yield better results. A better scoring mechanism should be implemented in order to remove the variability of the results.

RQ2 - What is the clustering process performance when using the proposed approach?

Apple's AR / VR example (Demo project) codebase performed best from a clustering performance perspective when we applied the proposed method. Table 4 shows the clustering performance on all the codebases, and we can see that Apple's application scored nearly perfect on the Mean Silhouette Coefficient as well as in the case of the Davis-Bouldin Index. The Mean Silhouette Coefficient was better for small codebases in which there were fewer types of components as the distinction between the three clusters (corresponding to the Model, View

Table 4. *CARL*-F_5 results in terms of cohesion and coupling of identified clusters. Homogeneity & Completeness of the analysed codebases.

Codebase	Mean Silhouette Coefficient	Davies-Bouldin Index	Homogeneity score	Completeness score
Firefox	0.78	0.44	0.60	0.62
Wikipedia	0.74	0.43	0.50	0.56
Trust	0.73	0.42	0.19	0.17
E-comm	0.78	0.32	0.66	0.73
Game	0.89	0.32	0.73	0.78
Stock	0.82	0.37	0.40	0.49
Education	0.78	0.32	0.16	0.30
Demo	0.95	0.05	0.80	0.90

and Controller layer) was more pronounced. In the case of larger codebases where we had many components that fulfilled various micro purposes in each layer, the performance of the clustering algorithm decreased. The Davies-Bouldin Index was also worst in the case of large codebases compared to the smaller ones. This is mainly due to the fact that in small codebases, just like in the case of the Mean Silhouette Coefficient, the clustering algorithm created better clusters (which have a higher density and a larger distance from the other clusters).

CARL achieved good homogeneity and completeness scores on both small (Demo, Game) codebases as well as on larger ones (Firefox, E-comm). The results are in-sync with the ones from Table 3 where accuracy, precision, recall were computed. The homogeneity and completeness score of the worst performing codebases (Trust, Education) are also caused by the naming conventions or the lack of conventions used in the codebase. Those had fewer components that had similarities between the names and did sometimes contain multiple keywords for the same element.

RQ3 - On what class of applications does this method work best?

CARL works well on all classes of codebases, when compared to the manual inspection of the codebases the best accuracy was obtained on one of the smallest codebases (Demo 0.96) followed closely by one of the largest codebases (Firefox 0.91) however similarly good values were also obtained for smaller codebases (Demo, Game 0.88).

Table 5 shows the average of the results on each class of codebases, from an accuracy point of view, the method works best on small and large-sized applications. Those results are in sync with the ones obtained at a layer level (precision and recall) and with Homogeneity and Completeness scores. While the method might separate better codebases that have fewer elements, as the differences between them are more prominent, it's important to notice that *CARL* also works well on large-sized codebases. The class of applications with the worst performance is the medium one, in this case, the separation of concerns is not that well defined, and coding standards are not always in place, hence our feature selection does not yield good results on those. If the coding standards are not well implemented in the codebase, especially in one of a medium-size where the clustering doesn't have a lot of components to work with, *CARL* may not achieve the best results.

In terms of ML metrics, the Adjusted Rand Index (ARI), Davies Bouldin index, and Mean Shillouete Coefficient, those results (see Table 5) are in sync with the ones obtained when compared the results with the ground-truth. The clustering algorithm performed best on the small class of applications, followed by the large one. This is mainly because in smaller codebases the clusters are more pronounced, the components from a cluster have many more particularities than the items in the other clusters.

The results of our approach *CARL* are promising. However, for achieving a better clusterization performance on large and especially medium-sized codebases and other architectural patterns the proposed approach needs to be enriched with more features for the clusterization

Table 5. Average (on applications classes) precision, recall, accuracy, Homogeneity, Completeness, Adjusted Rand Index, Mean Silhouette Coefficient, and Davies Bouldin Index. of *CARL* F_5 on the analyzed codebases against the ground truth.

Size	Average		Acc	Homog	Compl	ARI	Silh	Davies
	Precision	Recall						
Small	0.93 ± 0.11	0.90 ± 0.13	0.93	0.77	0.84	0.80	0.92	0.18
Medium	0.81 ± 0.15	0.67 ± 0.24	0.74	0.35	0.45	0.33	0.78	0.37
Large	0.80 ± 0.16	0.78 ± 0.15	0.81	0.48	0.51	0.50	0.76	0.40

process and analyze them with respect to the entropy of the analyzed codebase. *CARL* can be efficient and scalable on both small and large projects, as long as the naming conventions and the coding standards are respected all over the codebase.

For measuring the correlation between the number of components and the performance of our proposed approach, we've used Pearson's Correlation Coefficient.

In terms of the correlation between accuracy and the number of components, as shown in Table 6, there is a perfect negative relationship in the case of small codebases, and a high positive relationship in the case of the large-sized applications. Those results are in sync with the ones obtained for precision and recall at a layer level. While in the case of large-sized codebases the results are not perfect, the values obtained indicate good performance for those types of applications. In the case of large applications, the results might not be perfect as the number of components is much higher and their roles are not as well defined as in the case of small applications. In addition to this, in the case of large codebases, the coding standards and naming conventions are often not consistent throughout the entire codebase which makes the detection mechanism less accurate, nevertheless, we have discovered strong correlations between the number of components and the accuracy of the method which indicates that our method is well suited for those kinds of codebases as well.

Table 6. Pearson's Correlation Coefficient between the number of components and the detection quality of *CARL* F_5

Size	Model		View		Controller		Accuracy
	Precision	Recall	Precision	Recall	Precision	Recall	
All	0.16	-0.35	0.02	0.37	-0.69	-0.31	-0.43
Small	-1.00	-1.00	-1.00	1.00	0.00	-1.00	-1.00
Medium	0.90	-1.00	0.46	0.38	-0.97	0.93	0.70
Large	0.90	0.52	0.71	0.69	0.03	0.97	0.85

In terms of correlation between the number of components and the precision for the Controller, the layer was the least valuable metric as the values indicate that there is no correlation between those two.

RQ4 - What is the reliability of *CARL* when compared to *mACS*?

Table 7 presents the topological structure of the codebases after applying *CARL*. The table contains the number of components in each layer, together with the total number of relationships between the layers (#ExtDepends) and the number of unique ones (#DiffExtDepend). *CARL* managed to split the codebase in a way that there are fewer architectural violations for the Firefox, Wikipedia, Demo, and Game codebases when compared to *mACS* CoordCateg. approach. In the comparison with *mACS* CoordCateg, we have noticed an increase in the number of external dependencies (#CompleteExtDepend) for the Model layer and a decrease for the View and Controller layers. That means that the components inferred as Model by *CARL* had more exter-

Table 7. Analysis of codebases dependencies - *CARL*

#ExtDepends / #DiffExtDepend								
Dependency	Firefox	Wiki.	Trust	E-comm.	Game	Stock	Educ.	Demo
View-Model	27 / 9	7 / 3	160 / 27	72 / 27	1 / 1	0 / 0	0 / 0	1 / 1
View-Ctrl	0 / 0	0 / 0	2 / 1	0 / 0	0 / 0	0 / 0	0 / 0	0 / 0
Model-View	22 / 10	0 / 0	16 / 10	1 / 1	0 / 0	17 / 3	5 / 1	0 / 0
Model-Ctrl	66 / 10	6 / 3	120 / 29	418 / 64	0 / 0	64 / 14	46 / 10	0 / 0
Ctrl-Model	290 / 44	86 / 33	124 / 23	292 / 63	38 / 6	51 / 14	71 / 20	9 / 6
Ctrl-View	126 / 26	31 / 13	77 / 48	148 / 29	2 / 2	41 / 10	0 / 0	5 / 3
#CompleteExtDepends								
Model	3281	1077	1332	1927	122	442	471	278
View	752	182	903	337	42	114	20	13
Ctrl	1897	3077	1066	1826	204	475	264	106

nal dependencies with the other layers. In addition to this, the components from the View layers also exhibited an increase in the number of external dependencies. Judging by the number of external dependencies and the relationship between the number of dependencies among the layers, *CARL* concluded that the components of the Model and View are more coupled than in the case of *mACS*. When compared to *mACS* SimpleCateg. approach, *CARL* identified much fewer violations of the architectural rules, while the results for the number of external dependencies have the same distribution and are roughly the same for both approaches, with small differences between the codebases and the layers.

5. Threats to Validity

After the analysis we have found out that our proposed method presents the following threats of validity.

Internal: we discovered the features sets based on a trial and error approach; however, a different set of features which was not tested might yield better results. In order to have more details about this aspect, entropy could be used for measuring the importance of a feature.

External: the experiments were ran on the iOS platform and on the Swift language, there might be other SDKs and languages which have particularities which we have not addressed in this paper. Moreover, we have focused this preliminary research only on the MVC pattern without taking much into consideration more complex architectural patterns and their particularities.

Conclusion: the analysed codebases might also be responsible for some bias and more experiments should be exectured.

6. Conclusion & Further Work

With our study, we have increased the confidence in applying AI techniques for the detection of software architectures on mobile devices. Our proposed approach *CARL* works well on codebases that respect coding standards and development best practices, in its current state, small and large-sized codebases. *CARL* is an unsupervised method that needs no prior knowledge to work with a system and is fully autonomous. It paves the way for identifying architectural issues in the codebase by taking care of one of the most important aspects, the mapping between the codebase elements and the architectural layer they reside in.

We plan to further increase the accuracy of the system by running more experiments to find better-suited features that can be feed to the clusterization process as well as trying to leverage

the behavioral aspects of the analyzed architectures. Later, we want to test the approach on more specialized architectures, which have more than three layers, and see in what other kinds of software architectures could *CARL* be applied. In the end, we intend to use this system for improving the architectural health of the mobile codebases by highlighting architectural issues early in the development phase *mobile ArchCheckSys* [11]. Furthermore, such a system can be successfully used for educational purposes, as it can aid beginners to write better-architectured code.

Aknowledgement

This work was supported by the 2021 Development Fund of Babeş-Bolyai University.

References

1. Apple: Model-View-Controller. https://developer.apple.com/library/archive/documentation/General/Conceptual/CocoaEncyclopedia/Model-View-Controller/Model-View-Controller.html (2012)
2. Apple: Placing objects and handling 3D interaction. https://apple.co/3eHS164 (2019)
3. Avgeriou, P., Kruchten, P., Ozkaya, I., Seaman, C.: Managing technical debt in software engineering. Dagstuhl Reports **6**(4) (2016)
4. Belle, A.B., El Boussaidi, G., Kpodjedo, S.: Combining lexical and structural information to reconstruct software layers. IST **74**, 1–16 (2016)
5. Bishop, C.M.: Pattern Recognition and Machine Learning. Springer (2006)
6. Boudali, H., et al.: Arcade-a formal, extensible, model-based dependability evaluation framework. In: 13th IEEE ICECCS. pp. 243–248. IEEE (2008)
7. Cai, Y., et al.: Leveraging design rules to improve SA recovery. In: ACM Sigsoft Conference on Quality of Software Architectures. pp. 133–142. ACM (2013)
8. Corazza, A., Di Martino, S., Maggio, V., Scanniello, G.: Investigating the use of lexical information for software system clustering. In: 2011 15th European Conference on Software Maintenance and Reengineering. pp. 35–44. IEEE (2011)
9. Daoudi, A., et al.: An exploratory study of MVC-based architectural patterns in android apps. In: ACM/SIGAPP SAC. pp. 1711–1720. ACM (2019)
10. Davies, D.L., Bouldin, D.W.: A cluster separation measure. IEEE Transactions on Pattern Analysis and Machine Intelligence **1**(2), 224–227 (1979)
11. Dobrean, D., Dioşan, L.: An analysis system for mobile applications MVC software architectures. In: ICSOFT. pp. 178–185. INSTICC, SciTePress (2019)
12. Dobrean, D., Dioşan, L.: Detecting Model View Controller architectural layers using clustering in mobile codebases. In: ICSOFT. pp. 1–6 (2020)
13. Dobrean, D., Dioşan, L.: Importance of software architectures in mobile projects. In: 2021 IEEE 15th International Symposium on Applied Computational Intelligence and Informatics (SACI). pp. 000281–000286. IEEE (2021)
14. Fawcett, T.: An introduction to ROC analysis. Pattern Recognition Letters **27**(8), 861–874 (2006)
15. Garcia, J., Ivkovic, I., Medvidovic, N.: A comparative analysis of software architecture recovery techniques. In: ICASE. pp. 486–496. IEEE Press (2013)
16. Intelligence, G.: 2019 report. https://www.gsmaintelligence.com (2019)
17. Lakos, J.: Large-scale c++ software design. Reading, MA **173**, 217–271 (1996)
18. Laval, J., Anquetil, N., Bhatti, U., Ducasse, S.: Ozone: Layer identification in the presence of cyclic dependencies. SCP **78**(8), 1055–1072 (2013)
19. Le, D.M.: Architectural evolution and decay in software systems. Ph.D. thesis, University of Southern California (2018)

20. Levenshtein, V.I.: Binary codes capable of correcting deletions, insertions, and reversals. Soviet physics doklady **10**(8), 707–710 (1966)
21. Lutellier, T., et al.: Comparing SA recovery techniques using accurate dependencies. In: ICSE. vol. 2, pp. 69–78. IEEE (2015)
22. Martini, A., Bosch, J.: The danger of architectural technical debt: Contagious debt and vicious circles. In: 2015 12th Working IEEE/IFIP Conference on Software Architecture. pp. 1–10. IEEE (2015)
23. Martini, A., Bosch, J., Chaudron, M.: Investigating architectural technical debt accumulation and refactoring over time: A multiple-case study. Information and Software Technology **67**, 237–253 (2015)
24. Mozilla: Firefox iOS application. https://github.com/mozilla-mobile/firefox-ios (2018)
25. Murtagh, F.: A survey of recent advances in hierarchical clustering algorithms. The Computer Journal **26**(4), 354–359 (1983)
26. Rathee, A., Chhabra, J.K.: Software remodularization by estimating structural and conceptual relations among classes and using hierarchical clustering. In: ICAICR. pp. 94–106. Springer (2017)
27. Richards, M.: Software architecture patterns. O'Reilly Media, Incorporated (2015)
28. Rosenberg, A., Hirschberg, J.: V-measure: A conditional entropy-based external cluster evaluation measure. In: EMNLP-CoNLL. pp. 410–420 (2007)
29. Rousseeuw, P.J.: Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. Journal of Computational and Applied Mathematics **20**, 53–65 (1987)
30. Trust: Trust wallet iOS application. https://github.com/TrustWallet/trust-wallet-ios (2018)
31. Tzerpos, V., Holt, R.C.: ACCD: an algorithm for comprehension-driven clustering. In: Proc. 7th Working Conf. on Reverse Engineering. pp. 258–267. IEEE (2000)
32. Verwer, D.: The iOS developer community survey. https://iosdevsurvey.com/2019/01-apple-platform-development/ (2020)
33. Wikimedia: Wikipedia iOS application. https://github.com/wikimedia/wikipedia-ios/tree/master (2018)
34. Zapalowski, V., Nunes, I., Nunes, D.J.: Revealing the relationship between architectural elements and source code characteristics. In: Proceedings of the 22nd International Conference on Program Comprehension. pp. 14–25. ACM (2014)