A Hybrid Approach to MVC Architectural Layers Analysis

Dragoş Dobrean[a] and Laura Dioşan[b]
Computer Science Department, Babes Bolyai University, Cluj Napoca, Romania

Keywords: Mobile Applications Software Architecture Analyser, Automatic Analysis of Software Architectures, Structural and Lexical Information, Software Clustering, Hybrid Approach.

Abstract: Mobile applications have become one of the most important means of interacting with businesses, getting information, or accessing entertainment and news for the vast majority of the people, especially for the young generations. How those applications are being built, heavily influences their lifecycle, costs, and product roadmap, that is why software architecture plays a very important role as it affects the maintainability and extensibility of those products.
We are presenting a novel automatic approach for detecting MVC architectural layers from mobile codebases that combines an unsupervised Machine Learning algorithm and a classic static analysis. Our proposal does not require any prior training stage or datasets since it does not rely on *apriori* annotated codebases. As another key of novelty, it uses the information obtained from the mobile SDKs for enhancing the detection process.
The validation of our proposal is done in eight different sized codebases that operate in various domains and come from either open-source projects as well as closed-source ones. The performance of the detection quality is measured by the accuracy of the system, as we compared to a manually constructed ground truth, achieving an average accuracy of **85%** on all the analysed codebases.
Our proposal provides a viable hybrid approach for detecting architectural layers from mobile codebases achieving good results by providing the accurate detection of the layers using a deterministic step and great flexibility for being used on other architectural patterns via the non-deterministic step. Furthermore, we consider our approach as being valuable to students or beginners because it could provide insightful information on how the code should be structured and help them to respect architectural guidelines in real-world projects.

1 INTRODUCTION AND CONTEXT

Mobile phones have become one of the most personal devices, they are fully packed with applications that help people manage their finances, health, exercises, and social life. Those applications need to be flexible to change, easily maintainable and they need to run on a large variety of devices and OSs. With this study, we are furthering our work (Dobrean, 2019) towards creating an autonomous system for improving the architectural health of those projects since a large number of the mobile codebases do not have a well defined architectural pattern in place or even if they do, it is not consistently implemented all over the codebase (DeLong, 2017).

Furthermore, identifying and examining the implemented software architecture of mobile applications codebases could help the inexperienced developers or the students to better grasp or perceive the architecture importance, an important premise in software engineering education. These beginners could benefit from using an automatic detection system and to advance in the difficult task of learning architecting (Galster and Angelov, 2016), (Li, 2020).

Such a system would also provide valuable insights to teachers and mentors regarding the mistake their students make by examining the history of the analysis performed by the system on the student's codebase.

In (Dobrean and Dioşan, 2019a) we have presented a deterministic approach for automatically detecting architectural layers. We have furthered our research in (Dobrean and Dioşan, 2020) where we have approached the same problem from a different perspective, we have used Machine Learning algorithms for detecting those layers by considering the detection as a clustering problem (each layer representing a cluster).

[a] https://orcid.org/0000-0001-7521-7552
[b] https://orcid.org/0000-0002-6339-1622

Dobrean, D. and Dioşan, L.
A Hybrid Approach to MVC Architectural Layers Analysis.
DOI: 10.5220/0010326700360046
In *Proceedings of the 16th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2021)*, pages 36-46
ISBN: 978-989-758-508-1

Our newest proposal is a hybrid one, where we use a combination of deterministic and non-deterministic (Machine Learning) methods for solving the same problem. By using this new approach, we are paving the way for more specialized architecture detection (such as MVVM, MVP, VIPER, etc.), we call this approach *HyDe* (Hybrid Detection).

Analyzing mobile codebases using a purely deterministic approach can not yield great results on a large variety of codebase, the main reason for this being the fact that projects have their specific particularities, such as coding standards, naming conventions, codebase split (how external libraries and tools are being integrated), data flows, etc.. When using a Machine Learning approach, some of the elements might be wrongly categorized due to the features of the components. Besides, there is not a lot of information on the particularities of the layers that need to be detected.

Both developed detection systems have strong points as well: the deterministic one is able to accurately detect codebase elements and place them in the right architectural layers, while the non-deterministic one allows more flexibility for the analysis part and it can recognize clusters in more types of codebases. With these thoughts in mind, we joined both ideas into a hybrid one (*HyDe*) and we have used it to separate the architectural layers from a codebase using a combined approach. In this study, we are searching for a viable way of combining the deterministic methods (such as SDK inheritance) with the non-deterministic ones (such as clustering) into a hybrid approach for processing mobile codebases.

Research Challenges. The main challenges of these research topics are:

- architecture detection using a combined approach (deterministic + non-deterministic);
- a clustering process that works while combined with the information obtained from the deterministic process;
- automatically inferring the architectural layers of an MVC codebase that uses an SDK for building the user interfaces — as those SDKs usually contain more types of items than web SDKs.

Contributions. The main contributions of this study are listed below:

- a new workflow for automatic detection of architectural layers using a hybrid system;
- a novel approach at combining a deterministic method that uses SDK and lexical information for detecting architectural layers in a codebase with a non-deterministic one that uses a wide range of features extracted from the codebase (such as the number of common methods and properties and components name similarities) for solving the same problem;
- an innovative way of leveraging the information from the deterministic step to aid the non-deterministic one to increase the accuracy.

The paper is structed as follows. The next section introduces the formalisation of the scientific problem and presents details about other related work. Section 3 introduces our approach *HyDe*. The numerical experiments and analysis of the method are presented in sections 4 and 5. The end of the paper is composed by Section 6 which presents the threats to validity, followed by conclusions and some directions for further work in Section 7.

2 SCIENTIFIC PROBLEM

A mobile codebase is made of components (classes, structures, protocols, and extensions); we represent a codebase in a formal matter by using the following notation $A = \{a_1, a_2, \ldots, a_n\}$ where a_i, $i \in \{1, 2, \ldots, n\}$ denotes a component.

The purpose of this study is to find a way for splitting the codebase into architectural layers. An architectural layer is represented by a partition P of the components $P = \{P_1, P_2, \ldots, P_m\}$ in the codebase that satisfies the following conditions:

- $1 \leq m \leq n$;
- P_j is a non-empty subset of A, $\forall j \in \{1, 2, \ldots, m\}$;
- $A = \cup_{j=1}^{m} P_j$, $P_{j_1} \cap P_{j_2} = \emptyset$, $\forall j_1, j_2 \in \{1, 2, \ldots, m\}$ with $j_1 \neq j_2$.

In this formulation, a P_j ($j \in \{1, 2, \ldots, m\}$) subset of A represents an architectural layer.

Since we are using a hybrid approach, the detected architectural layers (partitions) P_1, P_2, ... P_m are determined by applying a set of deterministic rules and a clusterization process on some of the output partitions.

In order to apply a clustering algorithm, each component a_i ($i \in \{1, 2, \ldots, n\}$) has to be represented by one or more features or characteristics. We suppose to have k features and we denote them by $F(a_i) = [F_i^1, F_i^2, \ldots, F_i^k]$.

The purpose of the hybrid approach is to output partitions of components that match as best as possible the ground truth (the partitions provided by human experts).

The **Model View Controller** is one of the most widespread presentational architectural patterns. It is used extensively in all sorts of client applications, web, desktop and mobile. It provides a simple separation of concerns between the components of a codebase in 3 layers:

- Model: responsible for business logic.
- View: responsible for the user's input and the output of the application.
- Controller: keeps the state of the application, acts like a mediator between the Model and the View layer.

There are many flavours of MVC in which the dataflow is different, but they all share the same model of separation. MVC is also the precursor of more specialised presentational architectural patterns such as Model View View Model (MVVM) or Model View Presenter (MVP) (Dobrean and Dioşan, 2019b), (Daoudi et al., 2019).

In this study, the focus is on analyzing MVC architectures, therefore, the number of partitions $m = 3$. For a better understanding of the concepts, we are going to substitute the notation of partitions $P = \{P_1, P_2, P_3\}$, with $P = \{M, V, C\}$, as this notation better reflects the partition and the output layers we are interested in finding, M representing the Model layer, V representing the View layer, and C being the Controller layer. For the rest of the paper, this notation will be used to refer to the partitions for architectural layers.

Other studies have focused on architecture recovery by clusterization. For instance, Mancoridi (Mancoridis et al., 1999), Mitchell (Mitchell and Mancoridis, 2008) or Lutellier (Lutellier et al., 2015) from a structural point of view, Anquetil (Anquetil and Lethbridge, 1999) or Corazza (Corazza et al., 2016) from a lexical point of view and Garcia (Garcia et al., 2011) or Rathee (Rathee and Chhabra, 2017) from a structural and lexical point of view. However, none of those approaches specifically focused on mobile codebase or codebases that use SDKs for building their UI interfaces.

HyDe combines our previous work, regarding splitting a mobile codebase into architectural layers, a deterministic approach where information from the mobile SDKs is being used for detecting the architectural layers for a certain component using a set of rules (Dobrean and Dioşan, 2019a) with a non-deterministic one for which we previously paved the way with a study where we have applied Machine Learning techniques for solving the same problem (Dobrean and Dioşan, 2020).

3 *HyDe*: A TWO-STAGE AUTOMATIC DETECTION OF ARCHITECTURAL LAYERS IN MOBILE CODEBASES

Mobile applications are clients, usually, they are built as monoliths as they are self-contained.

In this study, we are interested in automatically categorizing the component of a codebase into architectural layers, based on the software architecture used. We are focusing on MVC, as is one of the most widely used architectural patterns, and is the precursor of other, more specialized architectures. For the inferring information from the codebases, we are doing a static analysis of its components.

By component, in this work we understand, a class, a struct, or a protocol, as building blocks of the codebase. We are interested in all the public and private properties, methods, and inherited types (if any) of those components. Our interest does not revolve around the body of the functions; we are only interested in their signature (naming and parameters), nor are we interested in the dependencies between the components or how they interact.

When developing these kinds of applications, SDKs are used for displaying information on the screen, interacting with the user, and using the hardware of those devices (camera, sensors, etc.). The codebase of those products is split into architectural layers, and some of those are closely related to the SDK defined elements (the View and Controller layer from MVC).

Our proposal is categorized by two important features, unsupervised — which means there is no prior knowledge needed before analyzing a codebase — and autonomous — no developer involvement needed after the process is started. Furthermore, *HyDe* involves a two-step process: firstly, we apply the deterministic approach on the entire codebase, and secondly, on some of the components we run a clustering algorithm to enhance the categorization of the components in architectural layers.

3.1 Pre-processing

To obtain information regarding the components of a codebase, a precursor step needs to be made, the pre-processing, where we extract the information required as input by the proposed approach. This is done by statically analyzing all the source files of the project; we are only interested in code source files, so the resources are ignored (a comprehensively presentation of the pre-processing particularities can be found in

(Dobrean and Dioşan, 2019a)).

As an outcome of this step, we have the set of components $A = \{a_1, a_2, \ldots, a_n\}$ where every a_i, $i \in \{1, 2, \ldots, n\}$ is characterised by:

- name
- type (class, struct, protocol)
- path
- inherited types
- all the private, static, and non-static methods and properties

3.2 Deterministic Step

In this part of the study, *HyDe* determines in which layer should a component reside by looking at the relationship between the components and the SDK. The mechanism behind this deterministic step is actually of *MaCS* (Dobrean, 2019): constructing the topological structure of the codebase, analysing the relationships among components and assigning them to the architectural layers.

3.2.1 Extraction

After the codebase has been preprocessed, *MaCS*, firstly, creates the topological structure of the codebase (a directed graph) in which every node corresponds to a codebase component. The links between the nodes of the graph represent dependencies between the components, for instance, if component A has a property of type B, then we are going to have a dependency, a link from the component A to the component B. Unlike a typical graph, the topological structure created could have multiple links between (eg. if component A has multiple properties or methods that have as parameters components of type B).

3.2.2 Categorization

In the categorization part, the codebase is split into architectural layers: every component is analyzed and placed into one of the 3 layers (Model, View, Controller) – since we are only focusing on MVC.

We have applied the CoordController approach (Dobrean, 2019), in which we also detect the Coordinating Controller layer, because it paves the way for analyzing more complex architectures and codebases. The Coordinating Controller layer is responsible for keeping the state of the application and deciding what is the flow of the application, it is a controller of the state of the application. Those type of elements have no direct correspondent in the development SDKs, however, those should also reside in the Controller layer of an MVC architecture (Dobrean, 2019).

The architectural layers are constructed in a deterministic fashion by using the following rules:

- All Controller layer items should inherit from Controller classes defined in the used SDK;
- All View layer items should inherit from a UI classes defined in the used SDK;
- All the remaining items are treated as Model layer items.

In addition to these base rules, we have also applied the one meant to detect the Coordinating Controller elements:

- All the components which have properties or methods that manipulate Controller components (including other Coordinating Controller components, also) are marked as Coordinating Controllers.

All the coordinating controller components are also placed into the Controller layer.

The output of *MaCS* represents a partition $P = \{M, V, C\}$, where the M, V, and C subsets are constructed by applying the above rules over $A = \{a_1, a_2, \ldots, a_n\}$.

To be able to formalise these decision rules, we involve the concept of predicate as following: a predicate of type X over two components as $pred_X(el_1, el_2)$ is True if we can apply the action X over el_1 and we obtain el_2. The proposed checker system defines the following predicates:

- $pred_{instanceOf}(component_a, Type) = True$ when $component_a$ is a variable of type $Type$;
- $pred_{inheritance}(component_a, component_b) = True$ when $component_a$ (directly) inherits $component_b$;
- $pred_{using}(component_a, component_b)$
- $pred_{using}(component_a, listComponent_b)$
- $pred_{inheritance}(component_a, listComponent_b) = True$

With these definitions in place we can represent the layers using the above stated rules as sets:

$$C_{simple} = \{a_i | pred_X(a_i, \text{SDK's Controller}) = True, \text{ where } X \in \{instanceOf, inheritance\}, a_i \in A\} \quad (1)$$

$$Coordinators = \{a_i | a_i \in A, \exists v \in a_i.properties \text{ and } c \in C \text{ such as } pred_{instanceOf}(v, c) = True \text{ or } \exists m \in a_i.methods \text{ and } c \in C \text{ such as } pred_{using}(m, c) = True\} \quad (2)$$

$$C = C_{simple} \cup Coordinators \tag{3}$$

$$V = \{a_i | pred_X(a_i, \text{SDK's View}) = True, \text{ where } X \in \{instanceOf, inheritance\}, a_i \in A\} \tag{4}$$

$$M = A \setminus (V \cup C) \tag{5}$$

After the categorization process is finished, the system yielded the first assignment of the components into architectural layers. By using *MaCS* approach, the View layer is determined with high accuracy, the Controller layer has a good accuracy as well, as the elements which inherit from an SDK defined Controller are being detected.

The major downside of a pure deterministic detection is the fact that the heuristics used for determining the Model and Coordinating Controller components do not always yield great results; it can happen that certain elements from either the Controller or the Model to be wrongly categorized.

3.3 Non-deterministic Step

Trying to improve the detection accuracy, *HyDe* involves a second stage when a clustering algorithm is run onto the output layers of the first step in order to better categorize the elements of the codebase.

We have already conducted some experiments on a clustering approach only (from scratch) for solving the same problem (splitting a codebase into architectural layers) in (Dobrean and Dioşan, 2020). While the clustering algorithm is the same, the process used in this study is rather different as we are focusing on only a part of the codebase for applying the clustering algorithm and one of the most important parts in a clustering process, the feature detection is completely new and different than our previous approach *CARL* (Dobrean and Dioşan, 2020).

Clustering is one of the most commonly used Machine Learning techniques for finding similarities in collections and splitting the items into groups that have similar features.To better split the codebase into architectural layers, a clustering process is applied on the layers in which the accuracy obtained in the first step was not satisfactory – the Model and the Controller layer.

What we try to achieve with this step of the process is filtering better the elements from the Model and Controller layer, to lower the percentage of wrongly categorized components. We know that Controller elements and Model elements have certain particularities that were not caught using the previously described rules. However, a clustering algorithm can find particularities in data that are less obvious.

HyDe is applying the clustering process only to the Model and Controller layer (M and C partitions) obtained from the deterministic approach, the View (V) being ignored. The View layer is ignored as the detection precision for this layer is 100% (Dobrean and Dioşan, 2019a), see Table 2. Therefore the set of analysed elements is represented by

$E = M \cup C = \{a_{i_1}, a_{i_2}, \ldots, a_{i_k}\}$,

where a_{i_j} is a component from set A (see problem definition) and denotes a component that was identified as either as a Model or Controller element by the Deterministic step, and k represents the number of components in E.

3.3.1 Feature Extraction

Feature extraction from raw input data is an essential stage before applying the clustering algorithm. Based on the chosen feature set, a clustering algorithm can yield good or bad results, even if it is applied to the same data collection.

From the pre-processing stage, we have information regarding every component from the codebase. In addition, due to the deterministic process, we also have information regarding the initial categorization of the components in architectural layers.

We have applied *HyDe* to one of the mid-sized codebases (E-Commerce) for which we had the ground-truth constructed and we have analyzed multiple features of the codebase such as:

- Levenshtein distance between the components name
- Whether a component inherits from a Model component (from the output of the deterministic step)
- Whether a component is using a Controller or Model component (it has properties or methods that use Controller or Model items from the deterministic step output)
- Whether a component is using a Controller component (it has properties or methods that use Controller items from the deterministic step output)
- Whether a component is a class or another type of programming language structure (such as structs, protocols, extensions)
- Whether a component is included in the Controller components (from the output of the deterministic step)
- Number of common methods between the analyzed components
- Number of common properties between the analyzed components

We have applied a trial and error approach to the set of the features above mentioned to find out which configuration yields the best results on the benchmark application. After different tries, we have concluded that the best results on the benchmark application come from the following set of features.

Two features take into account the already obtained assignments in the first step of our proposed system:

- Whether a component is included in the set of Controller components (from the output of the deterministic step). We associate a score θ if the component is included in the Controller layer and 0 otherwise. The score value has no influence over the detection process because, by normalisation, a Boolean feature is created.

$$F_1(a_i) = \begin{cases} \theta, & \text{if } a_i \in C \\ 0, & \text{otherwise} \end{cases} \tag{6}$$

- Whether a component is using a Controller component (it has properties or methods that used Controller items from the deterministic step output). We associate a score of 1 if the component uses another component from the Controller layer and 0 otherwise.

$$F_2(a_i) = \begin{cases} 1, & \text{if } pred_{using}(a_i, C) = \text{True} \\ 0, & \text{otherwise} \end{cases} \tag{7}$$

In addition to those two rules, for every pair of components from the analyzed set (E) we compute the following values for the feature set of a component:

- a feature that emphasises the similarity of two components' names

$$\begin{aligned} F_3(a_i) = [&NameDist(a_i, a_{i_1}), \\ &NameDist(a_i, a_{i_2}), \\ &\dots, NameDist(a_i, a_{i_k})] \end{aligned} \tag{8}$$

  where $NameDist(a_{i_1}, a_{i_2})$ is the Levenshtein distance (Levenshtein, 1966) between the names of component a_{i_1} and component a_{i_2}.

- a feature based on how many properties two components have in common

$$\begin{aligned} F_4(a_i) = [&CommProp(a_i, a_{i_1}), \\ &CommProp(a_i, a_{i_2}), \dots, CommProp(a_i, a_{i_k})] \end{aligned} \tag{9}$$

  where $CommProp(a_{i_1}, a_{i_2})$ is the number of common properties (name ant type) of component a_{i_1} and component a_{i_2}.

- a feature based on how many methods two components have in common

$$\begin{aligned} F_5(a_i) = [&CommMeth(a_i, a_{i_1}), \\ &CommMeth(a_i, a_{i_2}), \dots, CommMeth(a_i, a_{i_k})] \end{aligned} \tag{10}$$

  where $CommProp(a_{i_1}, a_{i_2})$ is the number of common methods (signature, name, and parameters) of component a_{i_1} and component a_{i_2}.

The output of this feature extraction phase is represented by the matrix that encodes all five feature sets, $M = F_1 \cup F_2 \cup F_3 \cup F_4 \cup F_5$. In fact, a $(3 \times k + 2) \times k$ matrix, that contains the feature set for every component in the codebase is constructed.

3.3.2 Clusterization

For the clusterization algorithm, we have found out that the Agglomerative Clustering (Murtagh, 1983) works best for this type of problem – a hierarchical, bottom-up approach. Since we are analyzing only the Model and the Controller layer, the number of clusters used for this step is set to two. One of the most important features of a clustering algorithm is the measurement of similarities between the clusters, our approach uses the Euclidian distance (Huang, 2008). The five computed features, described in the previous section, are normalized and then they are fed to the clustering algorithm.

The last step of the process is assigning responsibilities to the output clusters. *HyDe* does that by looking at the output clusters and deciding which one is the Controller cluster based on the number of components that inherit from an SDK defined Controller component.

With this final step, *HyDe* has categorized the codebase in the final architectural layers. The View layer is identical to the one obtained from the deterministic step of the proposal, the Controller one is determined in the last phase after the clustering has been applied and the group which contained the most elements that inherited from a Controller type was identified, while the Model is represented by the other cluster from the clusterization process output. A sketch of our system is depicted on Figure 1.

In case of more specialised architectures, where there are multiple other architectural layers, some heuristics should be used for assigning architectural layers to the output clusters. Those heuristics are closely related to the particularities of the analysed software architecture as well as the purpose of its composing architectural layers.

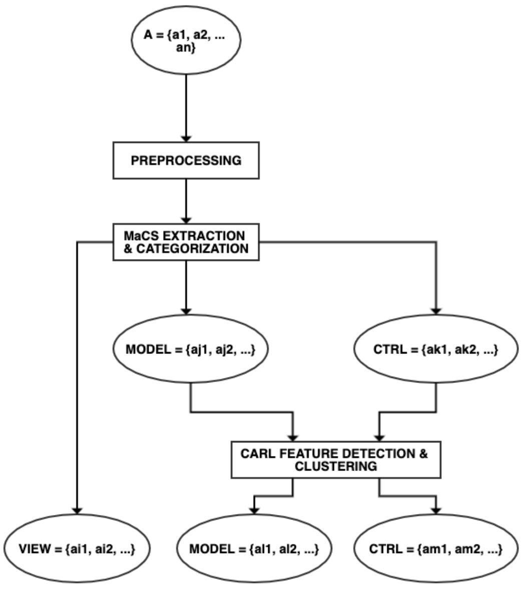

Figure 1: Overview of the *HyDe* workflow.

4 PERFORMANCE EVALUATION

To validate the performance of our approach, we are using the following metrics for analyzing the result of the *HyDe* against a ground truth obtained by manually inspecting the codebase by two senior developers (with a development experience of over five years):

- accuracy: $Acc = \frac{N^{AllLayers}_{DetectedCorrectly}}{N_{allComponents}}$
- precision for the layer X: $P_X = \frac{N^X_{DetectedCorrectly}}{N^X_{TotalDetected}}$
- recall for the layer X: $R_X = \frac{N^X_{DetectedCorrectly}}{N^X_{GroundTruth}}$,

where:

- $N^X_{DetectedCorrectly}$: is the number of component detected by the system which belong to the X layer and are found in the ground truth for that layer;
- $N^X_{TotalDetected}$: is the number of component detected by the system as belonging to the X layer;
- $N^X_{GroundTruth}$: is the number of component which belong to the X layer in the ground truth.

In addition to those metrics, we are also interested in the performance of the process not only from a software engineering point of view but also from a Machine Learning perspective. To analyze the ML performance of our approach we have used the following metrics:

- Silhouette Coefficient score (Rousseeuw, 1987) and
- Davies-Bouldin Index (Davies and Bouldin, 1979).

The Silhouette Coefficient measures how similar is a component of a cluster compared to the other elements which reside in the same cluster compared to the other clusters. The range of values for this score is **[-1;1]** where a high value means that the component is well matched in its own cluster and dissimilar when compared to other clusters. We have computed the mean Silhouette Coefficient score over all components of each codebase.

Davies-Bouldin Index is defined as the average similarity measure of each cluster with its most similar cluster. The similarity is represented by the ratio of within-cluster distances to between-cluster distances. The Davies-Bouldin Index indicates how well was the clustering performed; the minimum value for this metric is **0**, the lower the value the better.

Both of these metrics provide insightful information regarding the clustering performance: the Silhouette Coefficient score indicates how well are the components placed, while the Davies-Bouldin Index expresses whether or not the clusters were correctly constructed.

The Machine Learning perspective was applied to the output data obtained from the entire *HyDe* process, when computing these metrics we looked at the final result, and we've viewed the output architectural layers as clusters, even some of them might come from a deterministic step (View layer in our particular case). By computing these metrics, we can also understand the structural health of the codebase, how related are the components between them, and how high is the degree of differences between architectural layers.

5 NUMERICAL EXPERIMENTS

Our analysis focuses on MVC, a widely used architectural pattern (Vewer, 2019) for mobile development. Our experiments are run on the iOS platform; however, they can be replicated on other platforms that use SDKs for building their user interfaces.

For validating our proposal, we have compiled a list of questions for which we search answers with our experiments:

- **RQ1:** How can the deterministic approach and the non-deterministic one be combined?
- **RQ2:** How effective and performant is the *HyDe* approach?
- **RQ3:** What are the downsides of using a hybrid approach for architectural layers detection?

5.1 Analysed Codebases

We have conducted experiments on eight different codebases, both open-source and private, of different sizes:

- Firefox: the public mobile Web browser (Mozilla, 2018),
- Wikipedia: the public information application (Wikimedia, 2018),
- Trust: the public cryptocurrency wallet (Trust, 2018),
- E-Commerce: a private application,
- Game: a private multiplayer game.
- Stock: a private trading application,
- Education: a private education application for parents,
- Demo: the Apple's example for AR/VR (Apple, 2019)

We were interested in MVC codebases, as this is one of the most popular software architecture used on client applications. iOS applications implement the MVC pattern more consistently than Android, as Apple encourages developers to use it through examples and documentation (Apple, 2012). To our knowledge, there is not a selection of repositories used for analyzing iOS applications, that is why we have manually selected the codebases. The selection was performed to include small, medium, and large codebases. In addition, we have included both private source and open source codebases as there might be differences in how a development company and the community write code. Companies respect internal coding standards and styles that might not work or are different than the ones used by open-source projects.

Table 1: Short description of investigated applications.

Application	Blank	Comment	Code	No. of components
Firefox	23392	18648	100111	514
Wikipedia	6933	1473	35640	253
Trust	4772	3809	23919	403
E-Commerce	7861	3169	20525	433
Game	839	331	2113	37
Stock	1539	751	5502	96
Education	1868	922	4764	105
Demo	785	424	3364	27

Table 1 presents the sizes of the codebase, blank – refers to empty lines, comment – represents comments in the code, code states the number of code lines, while the number of components represents the total number of components in the codebase.

5.2 Empirical Evaluation

After the experiments were conducted on all of the codebases, we have analyzed the results based on the 3 questions which represent the base for this study.

5.2.1 RQ1: How Can the Deterministic Approach and the Non-deterministic One be Combined?

For constructing a system that would yield good results and involve a hybrid approach, it is clear that the only way to do it is to apply the non-deterministic approach after the deterministic one. These approaches can work well together if we leverage the best parts from both of them and try to improve the methods where they lacked.

From (Dobrean and Dioşan, 2019a) it is clear that the deterministic approach works really well for the layers for which there are rules strong enough to determine the components with high accuracy. In the case of the current study, *MaCS* was able to identify with high accuracy the View layer. When analyzing custom architectures, a deterministic approach might yield great results on other architectural layers as well, but for the purpose of this study, we have focused only on MVC.

The non-deterministic approach which was inspired by (Dobrean and Dioşan, 2020) works well for finding similarities between components without using heuristics.

With those ideas in mind, we have concluded that the best way for those approaches to function well together and achieve good results is to apply the deterministic method for identifying the layers, and afterwards to apply the non-deterministic approach to those layers for which the deterministic approach did not work that well. The second stage of *HyDe* could be considered as a filter, enhancing the detection results.

To summarise, we have applied the deterministic approach first and obtained the components split into architectural layers based on the rules used by *MaCS*, afterward we have applied the non-deterministic method for the layers in which *MaCS* results were not confident – the Model and Controller layers.

5.2.2 RQ2: How Effective and Performant is the *HyDe* Approach?

We have applied the *HyDe* approach to 8 codebases of different sizes, to cover multiple types of applications from a size perspective, but also from functionality and the domain they operate in perspectives.

The proposed approach achieved good results on the analyzed codebases with three codebases that achieved over **90%** accuracy with the highest one at **97%**. The average accuracy for the analyzed set of applications was **85%** as seen in Table 2.

The layer which came from the deterministic approach and was left unaltered by the non-deterministic step, the View layer, has achieved a perfect precision score on all the analyzed codebase, indicating that *HyDe* does not produce false positives (it does not label as View components those that, conform the ground-truth, they belong to other layers). In respects to recall, the same layer achieved lower scores on some of the codebases, this was mainly caused by the fact that those codebases used external libraries for building their UI interfaces, and they did not rely solely on the SDK for implementing those features.

In the case of the other layers which were altered by the deterministic step, the precision and recall were heavily influenced by the way the codebase was structured, naming conventions, and coding standards.

The proposed method worked best for the Game codebase which was a medium-sized application. For larger codebase which had higher entropy due to their dimensions and used external libraries the method did not work as well.

For some of the smallest codebases, the method achieved the worst results, this is because the non-deterministic step did not have enough data to make accurate assumptions as the codebases were rather small.

From a View layer precision point of view, *HyDe* achieves perfect results, all the detected View layer elements are indeed views, there are no false positives. In respect to the recall of the View layer, the results are not perfect, there are some false negatives, this is because the analyzed codebases use external libraries for implementing certain UI elements and those libraries were not included in the analysis process.

In respect to performance, we have computed the Mean Silhouette Coefficient, the Davies-Bouldin Index as well as the Homogeneity and Completeness scores.

The Mean Silhouette Coefficient had the best values in the case of medium-sized codebases where the distinction between the 3 output layers was more pronounced as seen in Table 3. In the case of the large codebases, the accuracy was worse as we had many types of components in each architectural layer. The value for the smallest codebase was also poor, this is because it had small number of components.

As seen in Table 3, the results for the Davies-Bouldin Index were hand in hand with the ones for the Mean Silhouette Coefficient: the best performing codedbases were the medium-sized ones, the largest and smallest ones achieved worst results due to the same reasons that affected Silhouette metric.

In case of Homogeneity and Completeness, Table 3, *HyDe* did achieve good results for the medium-sized and small-sized codebase. The scores were better for the codebases which had a naming convention and coding standards in place.

5.2.3 RQ3: What Are the Downsides of using a Hybrid Approach for Architectural Layers and Components Detection?

The most important downside of using a hybrid approach is that based on the analyzed codebase and the architecture it implements, the workflow might need to be adjusted in two places:

- the rules for the deterministic part of the process;
- the feature selection for the non-deterministic step.

In addition to those, an analysis would have to be conducted on the output of the deterministic step to identify the layers which should be feed to the non-deterministic step.

Another downside would be that this method only works if the deterministic step yields really good results for at least some of the layers, otherwise this part of the process becomes irrelevant.

From a computational performance point of view, the proposed approach is also heavier on the processing part as a simple singular process as it is composed of two separate steps; this also applies to the run time, as this is increased due to the same reasons.

In respect to the results, our proposal's main downside is the fact that for the output clusters from the non-deterministic step a manual analysis of those might be needed to match a cluster to an architectural layer if no heuristics can be found in the case of more specialized or custom architectural patterns.

Our proposed approach remains automatic in the case of more specialized software architectures, as it does not need the ground truth of the codebase. The feature extraction process needs to be enriched with information regarding the particularities of the analyzed architecture, in order for the process to yield good results.

6 THREATS TO VALIDITY

Once the experiments were run and we've analyzed the entire process we have discovered the following threats to validity:

Table 2: Results of the process in terms of detection quality.

Codebase	Model precision	Model recall	View precision	View recall	Ctrl precision	Ctr recall	Accuracy
Firefox	0,97	0,77	1,00	1,00	0,47	0,91	82,71
Wikipedia	0,79	0,74	1,00	0,66	0,74	0,95	80,00
Trust	0,86	0,89	1,00	0,70	0,66	0,72	82,86
E-commerce	1,00	0,80	1,00	1,00	0,81	1,00	90,97
Game	0,95	1,00	1,00	1,00	1,00	0,92	97,22
Stock	0,80	0,77	1,00	0,59	0,78	0,93	79,09
Education	0,87	0,95	1,00	1,00	0,92	0,90	93,60
Demo	0,91	0,77	1,00	1,00	0,25	0,67	78,79

Table 3: Results in terms of cohesion and coupling.

Codebase	Mean Silhouette Coef.	Davies-Bouldin Index
Firefox	0,64	0,61
Wikipedia	0,67	0,51
Trust	0,68	0,52
E-commerce	0,63	0,58
Game	0,86	0,20
Stock	0,81	0,28
Education	0,80	0,29
Demo	0,52	1,55

Table 4: Homogeneity and Completeness on the analyzed codebases.

Codebase	Homogeneity score	Completeness score
Firefox	0,60	0,63
Wikipedia	0,50	0,56
Trust	0,20	0,18
E-commerce	0,66	0,73
Game	0,74	0,79
Stock	0,40	0,50
Education	0,17	0,31
Demo	0,80	0,90

- **Internal:** the selection of features for the non-deterministic step was found using a trial and error approach, there might be another set of features that yield a better result. The selection of features can be improved by measuring the entropy of a feature for finding out its importance. Another reason for concern is the fact that in this approach we are only looking at the codebase, ignoring the external libraries used, and that can lead to wrongly detected components. This can be improved by also running the process on the libraries used by the analyzed codebase. We've also applied ML specific metrics to the results, including the View layer which was an output of the deterministic step. This might represent a threat to validity in respect to the results for the Silhouette Coefficient and Davies-Bouldin Index.

- **External:** Our study has focused only on Swift codebases on the iOS platform; other platforms and programming languages might come with their particularities which we have not encountered in the current environment. Furthermore, this study focuses on MVC alone. In the case of more specialized architectures, the rules from the deterministic step might need to be adjusted as well as the features selection involved in the non-deterministic phase.

- **Conclusions:** The experiments were run on a small number of applications that might have some bias, more experiments should be conducted to strengthen the results.

7 CONCLUSIONS AND FURTHER WORK

With *HyDe* we have shown that a hybrid approach between a deterministic method and a non-deterministic one can be successfully applied in the domain of software architecture recognition with great results on medium-sized applications. Not only that it works well on mobile codebases, but this work can also be applied to other platforms and types of applications which use SDKs for building UI interfaces.

HyDe leads the way of automatically splitting codebase components into architectural layers by only analyzing syntactical aspects of the codebase. It represents the major piece into a software architecture checker system, that can highlight the architectural issues early in the development phase. This proposal combines successfully a deterministic approach that can detect certain architectural layers with a high degree of confidence with a non-deterministic approach that offers it flexibility for analyzing more specialized architectural patterns or even custom ones.

We believe that the accuracy of the system can be greatly improved by also taking into consideration the external libraries, especially for large projects which use external libraries developed by the same teams using the same coding standards and conventions, as well as configuring the heuristics and features with respects to a certain type of mobile application (for instance medium-sized applications that are clients for other backend services). It will be also investigated if the detection performance could be improved by considering some features that reflect the specificities of the body components, as for the current approach, *HyDe* only took into consideration the signature of the methods, properties, and inherited types.

We plan to integrate *HyDe* into a software architecture workflow that could be used by developers as well as managers to have a detailed status of the architectural health of a mobile codebase. In additions, *HyDe* system would also be valuable to students or beginners as it could provide insightful information on how the code should be structured and help them to respect architectural guidelines in real-world projects.

REFERENCES

Anquetil, N. and Lethbridge, T. C. (1999). Recovering software architecture from the names of source files. *Journal of Software Maintenance: Research and Practice*, 11(3):201–221.

Apple (2012). Model-view-controller. https://apple.co/3a5Aox9link.

Apple (2019). Placing objects and handling 3d interaction. https://apple.co/3tJw8v2link.

Corazza, A., Di Martino, S., Maggio, V., and Scanniello, G. (2016). Weighing lexical information for software clustering in the context of architecture recovery. *Empirical Software Engineering*, 21(1):72–103.

Daoudi, A., ElBoussaidi, G., Moha, N., and Kpodjedo, S. (2019). An exploratory study of mvc-based architectural patterns in android apps. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, pages 1711–1720. ACM.

Davies, D. L. and Bouldin, D. W. (1979). A cluster separation measure. *IEEE transactions on pattern analysis and machine intelligence*, (2):224–227.

DeLong, D. (2017). A better MVC. https://davedelong.com/blog/2017/11/06/a-better-mvc-part-1-the-problems/link.

Dobrean, D. (2019). Automatic examining of software architectures on mobile applications codebases. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 595–599. IEEE.

Dobrean, D. and Dioşan, L. (2019a). An analysis system for mobile applications MVC software architectures. pages 178–185. INSTICC, SciTePress.

Dobrean, D. and Dioşan, L. (2019b). Model View Controller in ios mobile applications development. pages 547–552. KSI Research Inc. and Knowledge Systems Institute Graduate School.

Dobrean, D. and Dioşan, L. (2020). Detecting model view controller architectural layers using clustering in mobile codebases. pages 196–203. INSTICC.

Galster, M. and Angelov, S. (2016). What makes teaching software architecture difficult? In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pages 356–359. IEEE.

Garcia, J., Popescu, D., Mattmann, C., Medvidovic, N., and Cai, Y. (2011). Enhancing architectural recovery using concerns. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 552–555. IEEE.

Huang, A. (2008). Similarity measures for text document clustering. In *Proceedings of the sixth new zealand computer science research student conference (NZCSRSC2008), Christchurch, New Zealand*, volume 4, pages 9–56.

Levenshtein, V. I. (1966). Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710.

Li, Z. (2020). Using public and free platform-as-a-service (paas) based lightweight projects for software architecture education. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering Education and Training*, pages 1–11.

Lutellier, T., Chollak, D., Garcia, J., Tan, L., Rayside, D., Medvidovic, N., and Kroeger, R. (2015). Comparing software architecture recovery techniques using accurate dependencies. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE Int. Conf. on*, volume 2, pages 69–78. IEEE.

Mancoridis, S., Mitchell, B. S., Chen, Y., and Gansner, E. R. (1999). Bunch: A clustering tool for the recovery and maintenance of software system structures. In *Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM'99).'Software Maintenance for Business Change'(Cat. No. 99CB36360)*, pages 50–59. IEEE.

Mitchell, B. S. and Mancoridis, S. (2008). On the evaluation of the bunch search-based software modularization algorithm. *Soft Computing*, 12(1):77–93.

Mozilla (2018). Firefox iOS application. https://github.com/mozilla-mobile/firefox-ioslink.

Murtagh, F. (1983). A survey of recent advances in hierarchical clustering algorithms. *The Computer Journal*, 26(4):354–359.

Rathee, A. and Chhabra, J. K. (2017). Software remodularization by estimating structural and conceptual relations among classes and using hierarchical clustering. In *International Conference on Advanced Informatics for Computing Research*, pages 94–106. Springer.

Rousseeuw, P. J. (1987). Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *Journal of computational and applied mathematics*, 20:53–65.

Trust (2018). Trust wallet iOS application. https://github.com/TrustWallet/trust-wallet-ioslink.

Vewer, D. (2019). 2019 raport. https://iosdevsurvey.com/2019/link.

Wikimedia (2018). Wikipedia ios application. https://github.com/wikimedia/wikipedia-ios/tree/masterlink.