An Analysis System for Mobile Applications MVC Software Architectures

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

Keywords: Mobile Applications Software Architecture, Automatic Static Analysis, Model View Controller.

Abstract: Mobile applications are software systems that are highly used by all modern people; a vast majority of those are intricate systems. Due to their increase in complexity, the architectural pattern used plays a significant role in their lifecycle. Architectural patterns can not be enforced on a codebase without the aid of an external tool; with this idea in mind, the current paper describes a novel technique for an automatically analysis of Model View Controller mobile application codebases from an architectural point of view. The analysis takes into account the constraints imposed by this layered architecture and offers insightful metrics regarding the architectural health of the codebase, while also highlighting the architectural issues. Both open source and private codebases have been analysed by the proposed approach and the results indicate an average accuracy of 89.6% of the evaluation process.

1 INTRODUCTION

Nowadays there are many companies which are built around their mobile applications (Instagram, Whatsapp, Uber, etc.) which have large teams of people working on those projects. These kind of projects need to be maintained over a long period of time and they need to be flexible to new Software Development Kit (SDK) and hardware features. In order for a project to be extensible, maintainable and for more people to be able to work on it in parallel it needs to have a software architecture which allows it. Mobile applications usually use presentational software architectures and generally all of the architectural flavours used descend from Model View Controller (MVC) Fowler (2002); Reenskaug (2003). Each platform uses a flavour of MVC as their "default" architecture, alongside those, other architectural patterns have been coined such as Model View Presenter Potel (1996), Model View View Model Garofalo (2011), or View Interactor Presenter Entity Router MutualMobile (2014) in order to provide extra flexibility and testability of those applications.

Many of the codebases do not respect the imposed architecture for various reasons. One of the reasons is the fact that the developers working on the projects do not have the right experience and skill set which result into architectural smells such as Brick Functionality Overloading, Scattered Parasitic Functionality, Logical Coupling or Ambiguous Interfaces Le et al. (2015). Another reason for architectural erosion is the transition from one architectural pattern to another without doing it all over the codebase and all the refactoring needed.

An architectural pattern, which can impose the code to be written in a predefined way or to impose any strict boundaries can not be created without an external system which periodically checks it. Taking into consideration the architectural problems which appear on the mobile platforms, their increasing popularity and the fact that most problems are caused by developers which do not use the pattern correctly and not by requirements or other external factors, we have developed a system which statically checks if a mobile codebase is valid from an architectural point of view, while also highlighting the issues if those exist. Moreover, this system also provides valuable information which can be used by the management to check the architectural health of a codebase and to see tangible results of the refactoring phases.

The novelty of the proposed system is given by the usage of information from the mobile SDKs rather than relying strictly on the information extracted from the codebase as in other approaches Boaye Belle (2016); Corazza et al. (2016); Garcia et al. (2013). In

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

Dobrean, D. and Dioşan, L.
An Analysis System for Mobile Applications MVC Software Architectures.
DOI: 10.5220/0007827801780185
In *Proceedings of the 14th International Conference on Software Technologies (ICSOFT 2019)*, pages 178-185
ISBN: 978-989-758-379-7
Copyright © 2019 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

addition, it uses a simple formalisation of the architectural rules rather than complex specifications Rumbaugh et al. (2004),Hussain (2013), which makes it simply understandable even by less experienced practitioners. Furthermore, by using the proposed system, the architectural pressure points can be easily identified and this can be done early in the development phase by using it in a CI/CD pipeline.

The following section talks about MVC and its particularities. Section 2 analyses the MVC architecture and provides a way of detecting violation of its implementation. In section 3 we present the result of the validation experiments conducted on private and open-source iOS projects with the proposed method. The final section, states our conclusions and some ideas that could be tackled in future work.

2 SOFTWARE ARCHITECTURE CHECKER SYSTEM

The proposed system analyses a mobile application based on the composing components, it clusters them in categories based on the implemented architecture of the codebase and constructs the topological structure of the codebase, a dependency graph. After the graph has been constructed, the relationships between its composing components are analysed and architectural issues are highlighted (Fig 1). By component we mean one different element of the following kind: class, struct, protocol, enumeration.

Figure 1: Software architecture checker system phases.

2.1 Detection

The detection phase of the system takes a mobile codebase and analyses its components. It detects all the classes, protocols, structs and enums together with all their instance properties and methods (both private and public). The result of this detection is an analysis document in which we have stored all the information regarding the codebase, paths of the codebase components, their properties (methods, variables and the names and types of those) together with the size of the files. All of this information is encoded in a structured file (eg. JSON, XML or other type) which will be used by the extracting phase.

2.2 Extraction

There are many approaches for extracting an architecture from a codebase Belle et al. (2013); Corazza et al. (2016); Garcia et al. (2013), or for specifying it Hussain (2013); Rumbaugh et al. (2004). However, none of those approaches are designed specifically for mobile codebases and they are too complex for the purpose of our study.

In order to extract the architecture of the codebase we analyse the file produced by the detection phase. We are constructing the topological structure of the implemented architecture based on the information in that file. The topology is represented by a directed graph in which every node corresponds to a component and has the following informations: name, type, kind (class / struct / protocol / enum name), inherited type, instance and class variables (name and type), instance and class methods (together with parameters names and types), path.

The edges between nodes represent the links in the code between two components. Unlike a classic graph, in the architectural topological structure we can have multiple links between two nodes: a class has multiple references to another class; each of them is translated into an edge, allowing to specifically highlight all the individual codebase issues.

2.3 Categorisation

The most sensitive step in the analysis process is the categorisation one: the previously defined components need to be assigned in a certain category. Previous work has been done in the area of software architecture clusterisation and it was analysed from various perspectives: modules Huang and Liu (2016); Paixao et al. (2018), components Ramírez et al. (2018), layers Belle et al. (2013).

For the purpose of our research we are interested in the layered one. The components of a codebase can be distributed on abstract layers by following two strategies: the responsability-based strategy and the reuse-based strategy. In our MVC-context, the first strategy is implemented. MVC has also been analysed in Chen et al. (2014); Xu and Liang (2014b): an evolutionary algorithm optimises the mapping between the class responsibilities and the pattern roles. In our case, we define several heuristics, specific to mobile applications, that have to be respected by such mapping. Our proposed solution is a deterministic one; the search space is smaller as we are focusing on the codebase alone. In Mariani et al. (2016) the layered architectures are analysed, but the focus is on changing the codebase for avoiding the layered violations.

They do not analyse the architectural style used nor do they specifically focus on mobile platforms. To the best of our knowledge, the architectural style was analysed only in Sarkar et al. (2009) and Maffort et al. (2013), but these approaches are not mobile oriented.

Regarding the architecture conformance, the literature describes two main techniques: reflexion models and domain-specific languages that express the dependency rules. The reflexion models compare the intended architecture expressed as a high-level model created by the architect with the implemented one extracted from the source code and expressed as a concrete model. In general, the high-level model requires a set of refinements in order to identify the architecture violations Koschke (2013). In Maffort et al. (2013) a lightweight specification of the high-level model is proposed in order to mine the structural and historical architectural patterns. Our approach is somehow similar to Maffort's approach, but is focus on mobile applications and integrates information from the mobile SDKs rather than relying strictly on the information extracted from the codebase. The domain-specific languages help the architects to defined the intended architecture by using various constraints expressed in a customised and elaborated syntax Terra and Valente (2009). The constraints we proposed to be used can be easily defined and mined in the checker system.

We propose a novel mobile-inspired approach: the categories represent the layers of the analysed architectural pattern. In the case of the current research those are Model, View and Controller. We can leverage the fact that the mobile applications use certain SDKs for displaying information on the screen. The interaction with the user, both input and output, is manipulated by SDKs provided by the creators of mobile Operating Systems. With this idea in mind, we propose the following heuristics for deciding which component fits in which layer:

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

2.4 Analysis

MVC has been extensively analysed by practitioners DeLong (2017); Kocsis (2018); Orlov (2015) and academia Garofalo (2011); La and Kim (2010); Olsson et al. (2018) on mobile applications and other software systems. The potential issues highlighted such as massive view controllers or violation of single responsibility principle DeLong (2017) constitute the base for our research and represent the architectural issues we want to highlight early in the development phase. Therefore, the last phase of the processing is the analysis of the dependency graph generated by the extraction phase, correlated to the clusterisation on abstract layers. In this study, we are analysing whether or not one of the rules which dictate the MVC pattern are violated. Every rule violation is detected and highlighted. After this step we can say whether or not the codebase respects the MVC architecture and which are the pressure points in the codebase, what should be refactored and which are the components responsible for those violations.

The dependency relation between two layers, defined as two sets A and B, can be formalised as a set $L \subseteq A \times B$ as follows: $L_A^B = \{l = (a,b) | a \in A \wedge b \in B\}$. Note that a link is different to a coupling. The link is unidirectional, while the coupling is bidirectional.

After all the components have been split into the three categories (Model, View, Controller), each component is checked against all the other components from the other two layers for finding dependencies. The dependencies which are forbidden are highlighted and presented to the end user of the system.

In the classic MVC architecture all the dependencies are allowed except for the one between the Model and the Controller. This rule can be expressed formally by defining $L_{Model}^{Controller}$, the relations of Model's components to components of the Controller layer: $L_{Model}^{Controller} = \{l = (m,c) | m \in Model \wedge (c \in Controller)\}$. In order for the application to respect the classic MVC dependencies, $L_{View}^{Other} = \emptyset$ should be true. Based on the MVC flavour used the allowed dependencies rules can be different.

3 EVALUATION

The aim of the analysis is to inspect whether or not the MVC architecture is respected in the codebases of commercially available mobile applications using our proposed system. The rest of the paper focuses on the iOS platform and on Swift codebases; however, the same principles can be applied to other mobile platforms (Android, Window Mobile) and even on some other ranges of presentational applications such as desktop applications. The main differences between iOS and other platforms would be the naming of the components and frameworks used. In this part of the study the focus was on answering the following research questions:

RQ1 - How effective is the proposed categorisation

method compared to manual inspections?
RQ2 - What is the topological structure of mobile codebases using the proposed approach?
RQ3 - Do mobile codebases respect the architectural rules?

3.1 Methodology

Since we inspect iOS codebases, for the validation and analysis of the proposed system we have used the rules of Apple's flavour of MVC Apple (2012b):
Controller: $L_{VC}^{CC} = \{l = (vc, cc) | vc \in ViewControllers \wedge cc \in CoordinatingControllers\} = \emptyset$ meaning that ViewControllers should not depend on other Coordinator controllers
View: $L_{View}^{Other} = \{l = (v, o) | v \in View \wedge (o \in Controller \vee o \in Model)\} = \emptyset$ meaning that all components in the View layer should only depend on components within the same layer
Model: $L_{Model}^{Other} = \{l = (m, o) | m \in Model \wedge (o \in Controller \vee o \in View)\} = \emptyset$ meaning that all components in the Model layer should only depend on components within the same layer

A codebase respects Apple's flavour of MVC Apple (2012b) when all the above rules are respected. Mobile applications do not always use the Coordinator layer as this is a fairly unknown to the vast majority of developers, that is why our analysis has two categorisation approaches:

3.1.1 MVC Approach (SimpleCateg)

Analysing the most common MVC implementation (classic MVC) by identifying the View objects, the Controller (without coordinators - meaning that the code from navigating from a ViewController to another should reside in a child of an SDK defined Controller object) and all the other items were treated as Models.

Layers & Components. The Controller layer does not contain any Coordinating objects. The heuristics involved in this approach are:

H_1 $Controllers \leftarrow CategorisationVCs(Comps.)$

H_2 $Views \leftarrow CategorisationViews(Comps. \setminus Controllers)$

H_3 $Models \leftarrow Comps. \setminus (Controllers \cup Views)$

Dependencies Rules used for Validation.

R_1 $L_{View}^{Others} = \emptyset$ – all View components depend only on other View components

R_2 $L_{Model}^{Others} = \emptyset$ – all Model components depend only on other Model components

3.1.2 MVC with Coordinators Approach (CoordCateg)

Analysing the MVC with coordinating objects in place. Every items which dealt with UIKit defined Controller object was marked as a coordinator object. We have also taken all the objects that deal with those coordinator objects, and put them in the same category as well — Coordinating controllers.

Layers & Components. The Controller layer contains Coordinating objects. The heuristics involved in this approach are:

H_4 $Controllers \leftarrow CategorisationVCsAndCCs(Comps.)$

H_2 $Views \leftarrow CategorisationViews(Comps. \setminus Controllers)$

H_3 $Models \leftarrow Comps. \setminus (Controllers \cup Views)$

Dependencies Rules.

R_1 $L_{View}^{Others} = \emptyset$ – all View components depend only on other View components

R_2 $L_{Model}^{Others} = \emptyset$ – all Model components depend only on other Model components

R_3 $L_{VCs}^{CCs} = \emptyset$ – all ViewController components should not depend on Coordinator components

3.2 Evaluation Metrics

With both approaches we were interested in the same metrics. Our evaluation is two folded: validation of the categorisation process and analysis of how the architectural rules are respected or not.

In the **validation** stage, to measure the effectiveness of the categorisation, we compare the results from manual inspection (that acts as ground truth) to those of our methods. Three metrics are of interest in this validation: accuracy, precision and recall. In the case of multi-class classification problems (in our case we have a problem with 3 classes) the accuracy metric could be misleading since it does not take into account is the analysed data is balanced or not (all the classes have the same number of examples). Precision and recall are better-suited metrics.

In the **analysis** stage, the number of components (*#Comp*) from each abstract layer, the percentage of the Model, View and Controller components from the overall total and how many MVC rules were invalidated in the codebase (based on the analysed flavour) are computed.

We were also interested in the number of dependencies within a layer (*#IntDepends*), as well as in the number of external dependencies (*#ExtDepends*) of a layer. The *#IntDepends* are represented by links in the architectural topology of the codebase which are done between components which reside in the same layer.

#ExtDepends represent the links between the components of a layer and components which reside within the other two layers of MVC. Moreover, the *#CompleteExtDepends* include the external links relative to MVC layers and the links with other SDKs and third party libraries defined types, as well as Swift predefined types (such as String, Int) or codebase defined types (such as closures). These numbers denote the layers which are highly coupled with other layers or libraries and identifying those can ease the refactoring process by highlighting to the developers the items which are too complex and represent an architectural pressure point in the system, and revalidate the layers constructed correctly which are loosely coupled and self contained.

Another important metric is the number of different external links (*#DiffExtDepend*) — the number of different codebase components on which a certain component depends. The components with a large amount of different dependent items which violate the architectural rules are problematic and represent architectural pressure points in the analysed codebase.

Note that architectural change metrics (e.g. architecture-to-architecture, MoJoFM, cluster-to-cluster Le et al. (2015)) can not be used in order to establish which rules are violated, since the conceptual/intended architectures of the analysed systems are unknown.

3.3 Data

In order to test our system we have used some small, medium and large sized private and open-source iOS projects written in Swift. All the external libraries were ignored as well as their codebase are not relevant for the analysed application. In other words, we have analysed the Swift files defined in the project, and none of the external ones or the ones written in another programming language.

We have analysed 5 applications: mobile Web browser - Firefox Mozilla (2018), information - Wikipedia Wikimedia (2018), cryptocurrency wallet - Trust Trust (2018), e-commerce application - private and multiplayer game - private.

Table 1: Short description of investigated applications.

Application	Blank	Comment	Code
Firefox	23392	18648	100111
Wikipedia	6933	1473	35640
Trust	4772	3809	23919
E-Commerce	7861	3169	20525
Game	839	331	2113

As can be seen in Table 1. we have analysed different sized codebase. Blank, comment and code columns refer to the type of the text written in a line.

In order for the categorisation process to be accurate, we have identified all the iOS SDK defined ViewController and View components types defined in the iOS SDK, there are 12 different ViewController items and 40 View ones.

3.4 Results

RQ1 - How Effective is the Proposed Categorisation Method Compared to Manual Inspections? The first evaluation done was for the categorisation phase. We have manually analysed each application and placed each one of its components in one of the Model, View, Controller layers. After creating the ground truth, the system was ran over the studied applications and the results were compared against the baseline. Table 2 presents our findings for each of the applications using the precision, recall and accuracy metrics.

Table 2: The effectiveness of the categorisation process in terms of Accuracy, Precision and Recall.

Appr	Applic	Model		View		Ctrl		Acc
		P	R	P	R	P	R	
Simple	Firefox	96	99	100	98	98	71	95
Coord.	Firefox	96	77	100	96	46	90	82
Simple	Wiki	76	99	100	57	98	89	87
Coord.	Wiki	72	73	100	57	73	95	78
Simple	Trust	78	98	100	67	100	37	82
Coord.	Trust	83	88	100	67	64	70	82
Simple	E-comm	75	100	100	100	100	54	84
Coord.	E-comm	96	78	100	96	78	100	89
Simple	Game	100	100	100	100	100	100	100
Coord.	Game	100	100	100	100	100	100	100

We start the validation by an important metric, the accuracy, which denotes how many components were correctly identified for all the analysed layers. Our experiment shown that by using the SimpleCateg, without the coordinators detection, the proposed system was able to correctly categorise the components in layers with an average accuracy of 89.6% on all of the analysed codebases, while CoordCateg achieved an average accuracy of 86.2%.

An interesting finding is that in the case of the applications where the Coordinator concept was consistently used throughout the code (Trust, E-Commerce) the results were better for the CoordCateg approach, however on the other applications where this concept was not used, the results were worst.

The detection of the Coordinator components is heavily influenced by the way the navigation from one ViewController to another is implemented in the application. If this is scattered all around the codebase and is not extracted in custom components (Coordinators) the detection process will be affected. While building the ground truth, we have prioritised inheritance over the links between components – meaning

that if a component inherits from a View element but has dependencies to other Controller components we have marked it as a View component – as we had no way of knowing what the developers intended.

In the categorisation phase of the CoordCateg approach we are interested in the behaviour while detecting the Coordinators. It is more important to highlight that a component has the behaviour of a Coordinator than to leave it as a View or Model component. Due to how the code was written and the usage of external libraries of View elements, the accuracy for the CoordCateg approach is worst in the case of the Firefox and Wikipedia apps. The difference in the accuracy indicates that in the codebase there is a difference between the behaviour and the intended type of a component, which highlights an architectural misconception.

In the case of the simplest application (from an architectural point of view), both approaches correctly identified the components with an accuracy of 100%. From Precision's point of view, the best classified component is the View one (in both approaches) and no false positive are identified for this layer. From Recall's point of view, the best classified component by the first approach is the Model, while by the second approach is the Controller – such result is somehow correlated with H_4, the improved heuristic focused on Controller layer. In several cases (ECommerce and Game) a perfect recall is obtained indicating that the system produces no false negatives.

If we compare the results of the first approach with those of the second approach we observe how the second approach improves the Controller's recall because it is able to identify more types of components that should belong to this layer (not only that are pure View Controllers).

In the same time the Controller's precision and the View's recall decrease (their are some View components that, due to the filtering order, are labelled as Coordinator objects, breaking the mention metrics). Furthermore, the Model's recall decreases since some Model's components are labelled as Coordinating objects.

The Model's precision increases in some cases (e.g. Trust and ECommerce applications) because there are fewer false positive cases: some Coordinating objects, labelled as Model by the first approach, are labelled as Controller by the second approach. However, the Model's precision decreases when the second approach labelled as Coordinators some Model's components (that manipulate View-Controllers), similar to the View's components wrong classified. The View's precision is constant and reach the maximum, this being a good sign that the proposed system is able to correct recognise all View's components. The View's recall if affected by the usage of external libraries for UI components (e.g. Wiki case).

As a first conclusion, our system is able of categorising the components on the right layers.

RQ2 - What is the Topological Structure of Mobile Codebases using the Proposed Approach? Fig. 2 presents clusters composed from three categories (Model, View, Controller) showing the number of different components (*#Comp*) / *#IntDepends* — the number of internal dependencies for each cluster. The first columns from each application refer to the SimpleCateg, while the second columns denote the CoordCateg approach. As can be seen from the analysis and the sizes of the codebase, a smaller codebase can have more components even if it has fewer number of lines. This is due to the granularity of the architecture. Usually a higher granularity indicates a better architecture.

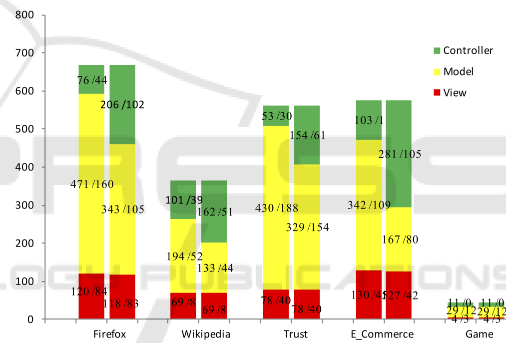

Figure 2: Codebase components, distribution of components on layers.

The *#IntDepends* are increasing in the case of the CoordCateg approach for the Controller layer and decreasing or keeping their status for the other two layers. This decrease of internal dependencies in the View and Model happens due to the migration of various components in the Controller layer as Coordinating items; the remaining items have a higher degree of cohesion among them.

Tables 3 and 4 summarise the analysis of the separation on each one of the codebases, both *#ExtDepends* and *#DiffExtDepend* are presented. The items highlighted in red represent the dependencies which violate the MVC rules. While links Model—Controller and View—Controller should exits, those should be done trough interfaces and the Model and View should have no direct knowledge about the Controller; that is why those items were highlighted as well.

As can be seen from the two approaches, Coord-

Table 3: Codebase analysis - SimpleCateg approach.

#ExtDepends / #DiffExtDepend					
Dependency	Firefox	Wiki	Trust	E-comm	Game
View - Model	21/9	7/3	21/14	72/27	1/1
View - Controller	3/1	-	-	2/1	-
Model - View	203/19	99/11	49/11	122/11	1/1
Model - Controller	161/11	55/8	146/23	560/69	-
Controller - Model	152/41	78/35	74/37	290/62	38/6
Controller - View	403/46	545/35	146/30	637/47	42/13
#CompleteExtDepends					
Model	3001	1393	1745	2018	111
View	598	146	163	274	19
Controller	1615	2212	496	1702	188

Table 4: Codebase analysis - CoordCateg approach.

#ExtDepends / #DiffExtDepend					
Dependency	Firefox	Wiki	Trust	E-comm	Game
View - Model	21/9	7/3	21/14	63/25	1/1
View - Controller	3/1	-	-	2/1	-
Model - View	126/14	91/11	35/11	103/8	1/1
Model - Controller	-	-	-	-	-
Controller - Model	257/41	73/32	264/66	431/62	38/6
Controller - View	448/46	576/35	161/31	674/49	42/13
VC - CC	50/9	8/3	-	49/8	-
#CompleteExtDepends					
Model	2129	957	1228	846	111
View	569	146	163	241	19
Controller	2646	2672	1193	2569	188

Categ produces more accurate results and it does not have any negative side effects on codebases which do not use coordinators at all (Game). The shift from Model and View to more Controller components can be seen in the #CompleteExtDepends section of Tables 3 and 4, as well as in the Fig. 2, the Controller has more items, while the Model and sometimes the View has fewer.

RQ3 - Do Mobile Codebases Respect the Architectural Rules? Our approach intends to highlight the architectural problems and to provide meaningful insights regarding the codebase. To this aim we analyse MVC specific validation rules over different types of dependencies found in the codebase. Note that the purpose of this study was not to compare the extracted architectural topology against a reference one (also known in literature as the conceptual architecture) since it not available for the analysed systems. Hence, in Tables 3 and 4 can be seen that the validation rules R_1 and R_2 are violated in both approaches. One reason for this can be the fact that developers tend to respect the classic MVC dependencies rules and ignore the Apple's flavour of MVC Apple (2012b) dependency rules. However, in the CoordCateg approach (Table 4) the rule violation is more mildly as we have fewer invalidations. This amelioration could be a consequence of the improved heuristic used for identifying Controller layer components (H_4).

In the CoordCateg approach, as can be seen in Table 4 the rule R_3 is violated in 3 of the 5 analysed codebases; the Game codebase does not use Coordinating controllers, while in the case of Trust application the dependency rule is respected. In the case of applications where the Coordinating controllers are well understood, they are usually correctly implemented. However in other cases where this sub-layer appears as a side effect of the complexity in the Controller layer, they are inadequately constructed.

As can be seen in both Tables 3 and 4, the majority of the architectural problems are encountered between the Model and the other layers. This shows that the Model layer is usually wrongly constructed and it is not clear for developers what should reside in there. Moreover the analysis also shows that while mobile projects are split in composing layers, MVC tries to be followed, the dependencies between those layers are not correctly constructed.

3.5 Threats to Validity

Internal Validity. In the case of Coordinating Controllers detection there might be mismatches between the purpose given by the developer to that component and our categorisation process. This shift from Model and View layer might not always be valid as some components were intended to reside in those layers, but they simply are wrongly coupled with ViewControllers. In addition to this this wrong categorisation heavily affects the analysis phase. Moreover the usage of third party libraries for the UI components heavily impacts the categorisation process which drastically impacts the rest of the analysis as those are not covered by our proposed heuristics. Furthermore the analysis was conducted on the iOS platform using the Swift language, for other platforms which use languages which support multiple inheritance some of the heuristics might not apply.
External Validity. The current study focuses on MVC and its flavours on the most popular mobile platforms. By using the current approach, custom architectural patterns can not be analysed, in order to overcome this issue the categorisation process should be enriched with more layers and improving the clustering capabilities.
Conclusion Validity. The study can be ran on more applications to strengthen our findings. Additional metrics can be used for analysing the dependencies in order to give relevant insights regarding the stability, testability and extensibility of the analysed codebase.

4 CONCLUSIONS

Architectural patterns can not enforce their rules on developers. In the domain of software architectures,

the builders — developers, can cross architectural boundaries fairly easy; that is why an external system is needed for enforcing the chosen architectural pattern. Our research provides a distinct technique for analysing the mobile codebases, without the need for using complex ADL languages and validation systems. The proposed method uses the mobile SDKs particularities for constructing an easy to use and understand system which can come in the aid of mobile developers. Discovering the root and sources of the technical debt and highlighting the architectural issues of the codebase benefits both developers as well as management, as they can see the architectural health of the codebase.

The fact that the codebase is split in well defined categories represent just part of the benefits obtained by respecting an architectural pattern. The correct dependencies between layers is what offers all the other architectural benefits (flexibility, testability, maintainability, etc.) and it is more important than just categorising the components. Our results show that those dependencies are not correctly constructed and there is a need for an architectural checker system on mobile platforms for imposing architectural patterns.

As future steps, we plan to extend the system to work with custom defined software architectures, such that it can analyse multiple type of projects. The system can also be integrated into a CI/CD pipeline, highlighting architectural issues early in the development phase before the code goes into the final product.

REFERENCES

Ameller, D. and Franch, X. (2011). Ontology-based architectural knowledge representation: structural elements module. In *CAiSE*, pp. 296–301. Springer.

Apple (2012a). Controller. link.

Apple (2012b). Model-view-controller. link.

Belle, A. B. et al. (2013). The layered architecture revisited: Is it an optimization problem? In *SEKE*, pp. 344–349.

Boaye Belle, A. (2016). *Recovering software layers from object oriented systems: a formalization as an optimization problem*. PhD thesis, École de technologie supérieure.

Chen, X. et al. (2014). A replicated experiment on architecture pattern recommendation based on quality requirements. In *ICSESS*, pp. 32–36. Citeseer.

Corazza, A. et al. (2016). Weighing lexical information for software clustering in the context of architecture recovery. *Empirical Software Engineering*, 21(1):72–103.

DeLong, D. (2017). A better MVC. link.

Fowler, M. (2002). *Patterns of enterprise application architecture*. Addison-Wesley Longman Publishing Co., Inc.

Garcia, J. et al. (2013). A comparative analysis of software architecture recovery techniques. In ICASE, pp. 486–496. IEEE Press.

Garofalo, R. (2011). *Building enterprise applications with Windows Presentation Foundation and the Model View View Model Pattern*. Microsoft Press.

Huang, J. and Liu, J. (2016). A similarity-based modularization quality measure for software module clustering problems. *Information Sciences*, 342:96–110.

Hussain, S. (2013). *Investigating architecture description languages (ADLs) a systematic literature review*. PhD thesis, Linkapings Universitet.

Kocsis, K. (2018). Architectural patterns, MVC, MVVM: What is the hype all about? link.

Koschke, R. (2013). Incremental reflexion analysis. *Journal of Software: Evolution and Process*, 25(6):601–637.

La, H. J. and Kim, S. D. (2010). Balanced mvc architecture for developing service-based mobile applications. In ICEBE, pp. 292–299. IEEE.

Le, D. M. et al. (2015). An empirical study of architectural change in open-source software systems. In IEEE MSR, pp. 235–245.

Maffort, C. et al. (2013). Mining architectural patterns using association rules. In SEKE, 2013, pp. 375–380.

Mariani, T. et al. (2016). Preserving architectural styles in the search based design of software product line architectures. *J. of Systems and Software*, 115:157–173.

Mozilla (2018). Firefox iOS application. link.

MutualMobile (2014). Meet VIPER: Clean architecture for iOS apps. link.

Olsson, T. et al. (2018). Towards improved initial mapping in semi automatic clustering. In ECSA, page 51. ACM.

Orlov, B. (2015). iOS architecture patterns: Demystifying MVC, MVP, MVVM and VIPER. link.

Paixao, M. et al. (2018). An empirical study of cohesion and coupling: Balancing optimization and disruption. *IEEE TEC*, 22(3):394–414.

Potel, M. (1996). MVP: Model-View-Presenter the taligent programming model for C++ and Java. *Taligent Inc*, page 20.

Ramírez, A. et al. (2018). Interactive multi-objective evolutionary optimization of software architectures. *Information Sciences*.

Reenskaug, T. (2003). The model-view-controller (MVC) its past and present. *University of Oslo Draft*.

Rumbaugh, J. et al. (2004). *Unified modeling language reference manual, the*. Pearson Higher Education.

Sarkar, S. et al. (2009). Discovery of architectural layers and measurement of layering violations in source code. *J. of Systems and Software*, 82(11):1891–1905.

Terra, R. and Valente, M. T. (2009). A dependency constraint language to manage object-oriented software architectures. *Software: Practice and Experience*, 39(12):1073–1094.

Trust (2018). Trust wallet iOS application. link.

Wikimedia (2018). Wikipedia ios application. link.

Xu, Y. and Liang, P. (2014b). A cooperative coevolution approach to automate pattern-based software architectural synthesis. *IJSEKE*, 24(10):1387–1411.

Zhang, M. (2018). Photoshop coming to the iPad. link.