MMVMi: A Validation Model for MVC and MVVM Design Patterns in iOS Applications

Mariam Aljamea, and Mohammad Alkandari

***Abstract*—Design patterns have gained popularity as they provide strategies for solving specific problems. This paper explores two common design patterns in iOS development field: Model View Controller (MVC) and Model View ViewModel (MVVM) design patterns. The paper investigates the problems with MVC design pattern. Then, it introduces a validation model that detects MVC problems, and helps programmers make the decision to switch between MVC and MVVM design pattern. The proposed validation model consists of two phases: phase one is to detect MVC problems, and phase two is to validate the relation between MVVM objects. The model was then applied to a Cloudy app as a case study. As a result, the model was able to detect MVC problems. The relation between MVVM objects was also validated. Further, this research provided some recommended solutions to satisfy the relations between MVVM objects in the project.**

***Index Terms*—design pattern, mvc, mvvm, validation, model.**

I. Introduction

OVER the years, softwares have been grown and changed to accommodate users' demands. Users need to interact with softwares through interfaces, where the emerge of interfaces has catalyzed a new demand of design patterns to help keep interfaces user friendly and to improve them [1]. Since then, several design patterns appeared then evolved and matured [2]. Design patterns reuse the same set of ideas to construct a solution to solve certain problems occur commonly [3]. Model View Controller (MVC) design pattern is used frequently to architect interactive software systems [4]. MVC design pattern offers a way to struct application's components into different separate roles [3]. In fact, recent technologies encourage separating the roles of application early in design phase [4]. Thus, roles separating improves the implementation of the application as it composes a robust and flexible application. MVC separates roles as follow: View objects in MVC displays data to users, while user interaction is processing by Controller objects [4]. Application logic and data are the responsibility of Model objects [4]. The Model View Controller (MVC) design pattern is recommended by Apple since its frameworks built based on it [5]. Consequently, any iOS developer will start programming an iOS application based on Model View Controller (MVC) design pattern. For some projects MVC will work perfectly: as desired leading to reusable objects, easily extensible objects and much more numerous benefits. However, for some other projects, MVC design pattern will not work as desired as it will lead to a famous problem in iOS community: the Massive View Controller problem. When Controller objects exceed 150 lines, this is an indicator of Massive View Controller problem. Controller objects will be massive when they handle many responsibilities more than their own. The Model View ViewModel (MVVM) design pattern rescuers gracefully by strictly distributing the roles and responsibilities among objects [6]. By using MVVM design pattern Controller objects will just set the values for UI components while all data logic preparation for UI will be located in ViewModel objects. As a rule of thumb, both of MVC and MVVM have their own advantages and disadvantages, choosing one of them as a design pattern is highly dependents on the project architecture. For instance, JavaScript has different frameworks each of them used either MVC or MVVM according to the framework needs. Namely AngularJS, Angular 2, and Vue.js are based on MVC, while Ember.js is built based on MVVM [7]. In essence, the objectives of this study are:

1) To understand the impact, role and importance of Design Patterns in iOS development.

2) To help developers to make a decision when to use MVC or MVVM.

3) To provide a method to check and validate the relations between MVVM objects.

To address these objectives, we proposed a model that will help developers to indicate the time to switch from MVC design pattern to MVVM design pattern. The proposed model consists of two phases. Phase one will detect if the project has a Massive View Controller problem. The second phase, will check the relations between project's objects if they obey to MVVM correctly or not. Hence, there are three significant contributions in this paper. First, MVC problems is presented. Then, this paper develops a validation model that accomplish these objectives: detecting massive controller objects, and validate the relations between MVVM objects.

The remainder of this paper is organized as follows: section 2 provides a background knowledge of Model View Controller (MVC) and Model View ViewModel (MVVM) design patterns. Section 3 presents the related work. Section 4 introduces MMVMi, our model to validate MVC or MVVM in iOS applications, and explains its two phases. Followed by applying MMVMi to two cases and demonstrating the results. After that, MMVMi evaluation by comparing it with other tools in section 6. The conclusion is described in section 7. The paper concludes with a future work in section 8.

II. Background

Design patterns have been part of application development process as they help and guide developers to develop efficient

Manuscript received October 3, 2017; revised May 9, 2018.

M. Aljamea is a graduate student in the Department of Computer Engineering at College of Engineering and Petroleum - Kuwait University, Kuwait e-mail: (mrm259@gmail.com).

M. Alkandari is an Assistant Professor in the Department of Computer Engineering at College of Engineering and Petroleum - Kuwait University, Kuwait e-mail: (m.kandari@ku.edu.kw).

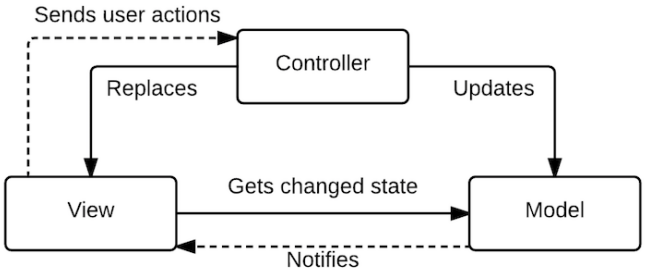


Fig. 1. The traditional MVC design pattern

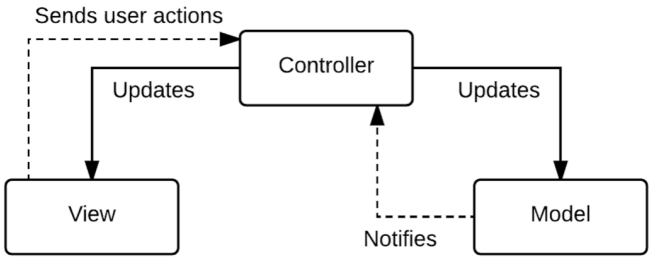


Fig. 2. Apple's MVC design pattern

and robust applications. Design patterns provide a strategy to solve common problems in software development projects [8]. The provided solution is an abstract that is not related to a specific programming language or platform [8]. Design patterns help avoiding or solving a problem but they do not provide an exact solution as they require additional works in order to be adapted into specific projects [8]. Design patterns improve the flexibility of the code which is a good investment as it makes the code adaptable and robust [8]. Developers choose the desired design pattern according to projects' requirements and their experiences [9]. Presentation design patterns are used for GUI-based applications [2]. Separation of Concerns (SoC) is the fundamental concept of presentation design patterns [2]. This section provides a background knowledge about two common presentation design patterns the Model View Controller (MVC) design pattern and the Model View ViewModel (MVVM) design pattern.

A. Model View Controller (MVC) Design Pattern

In 1979 Trygve Reenskaug introduced Model View Controller (MVC) design pattern during his work on Smalltalk at Xerox PARC [2]. The intent of MVC is to separate application's concerns into three objects each of these objects has its own tasks to handle [10]. MVC separates the concerns as follows: object to store and manage data, object to display data, and object to handle user interactions [10]. With this strict separation among application logic and its interface, MVC supports developing extremely maintainable well structured applications [10].

Figure 1 depicts the traditional MVC design pattern. It consists of three entities: Model objects, View objects, and Controller objects. As shown in Figure 1, each of the three entities recognizes the other two entities [6]. Thereby, entity reusability will be reduced and this is considered as a main drawback that makes the traditional MVC design pattern not applicable for iOS applications [6]. In addition, the evolution of iOS applications with their new demands introduced and revealed weak spots in the traditional MVC design pattern [10]. Therefore, to overcome all the limitations in the traditional MVC design pattern, Apple enhanced it as shown in Figure 2 [6]. In Apple's MVC, Controller objects know about View objects and Model objects [11]. Also, Controller objects have the ability to update View objects and Model objects when it needed [11]. While it is not required for Model objects to know anything about View objects or Controller objects [11]. Similarly, it is unnecessary for View objects to know about Model objects or Controller objects [11]. However, View objects respond to user interactions, then they send these interactions to Controller objects by delegation pattern or target-action pattern [11]. Thereby, View objects are totally decoupled from Model objects in Apple's Model View Controller design pattern [12]. As a result, application logic is separated from user interface [13] that leads to more reusable objects and more extensible applications [14]. Apple's MVC divided the roles between objects as follow: (1) Model objects hold and manage application's data, (2) View objects render the interactive visual elements, and send user actions to Controller objects (3) Controller objects bind Model objects and View objects together and contain application logic [15]. In more details Controller objects have the following roles: create views, update views, validate the input, query models, modify models, map user actions to update model, handle segues to other controllers, and hold network logic [5]. Indeed, View objects and Controller objects do their tasks as a pair by allowing user to interact with the rendering objects in the user interface. These rendering objects are View objects. While user interactions are handled by Controller objects [10]. However, MVC decouples View objects from Controller objects and treats each of them as a separate object [10]. With this decoupling, MVC enhances separation of concerns (SoC) concept [10]. Obviously, Apple UIKit frameworks obey to MVC design pattern as their names reveal its philosophy. For example, a UIViewController tells the UIView what to render in the screen, which is according to MVC a UIViewController is a Controller object and a UIView is a View object [11]. To illustrate that, a data source (model) provides a UIPickerView with its data, so UIPickerView is a View object that got its data from a Model object [11]. Apple's MVC design pattern separate application's roles loosely between three main objects: Model object, View object, and Controller object. This loosely separation leads to downside effects by assigning many different roles to Controller objects. Also, there are some parts of the code that developers don't know where to write them as these parts do not belong to View objects or Model objects. For example: table view delegate methods, network requests, loading data from databases, view layout, and much more. Mostly, all these parts will be written in Controller objects. As a result, application based code will end up with massive controllers, which are complex [5]. Consequently, it will be difficult to test these massive controllers as they contain several unrelated roles [5]. In addition, these unrelated roles with their different responsibilities make massive controllers hard to reuse and hard to maintain. Undoubtedly, Controller objects should not contain any logic, they should only act as an intermediary between Model objects and View objects.

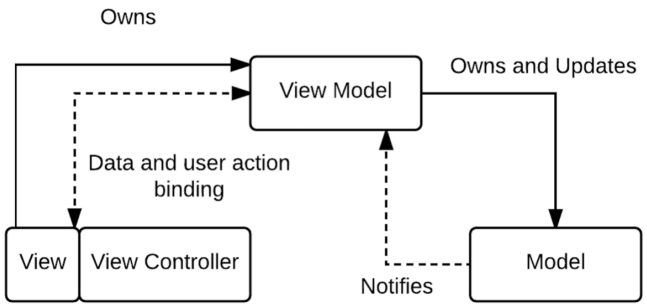


Fig. 3. MVVM design pattern

Also, massive controllers will increase the chance of writing errors in the code without recognizing them as they are long and they are difficult to test [5]. The next subsection will explain a solution to the Massive View Controller problem.

B. Model View ViewModel (MVVM) Design Pattern

In 80s the Model View ViewModel (MVVM) design pattern was introduced in Smalltalk to overcome the limitations of MVC and to gain from some of its strengths [2]. In fact, both MVC and MVVM assert the concept of Separation of Concerns (SoC), which increases code quality [9]. However, MVVM separates concerns more than MVC which reduces complexity and maximizes testability and reusability. MVVM introduced a new object named it ViewModel object. This object has the logic to prepare the data for View object, which was the responsibility of Controller object in MVC [5]. Furthermore, similar to MVC View objects and Controller objects work as pair, but MVVM merged them into one single object and called it View objects [10]. Thereby, MVVM increases data independent and application logic encapsulation [9]. A recent study [16] exploits MVVM design pattern to show that it compacts the based code and makes it flexible. Absolutely, there are various versions of MVVM, iOS community use Microsoft MVVM [10]. Moreover, the study in [17] demonstrates Protocol Oriented Model View View Model (POP MVVM) by using it in developing Tintm3 iOS application. POP MVVM takes advantage of Protocol Oriented Programming paradigm to enhance the original MVVM. The intent of MVVM is to separate application's logic concerns from user interface concerns by distributing distinct responsibilities between three objects: first, View objects to render user interface, Second, ViewModel objects take the responsibility of user interactions and view state, Third, Model objects to handle data [10]. Additionally, to work with application's data, ViewModel objects own Model objects and have a direct access to them [10], as it is demonstrated clearly in Figure 3 [6]. On the other hand, View objects and ViewModel objects cooperate in a new way that is introduced by Microsoft and it is Data Binding [10]. Data Binding improves logic separation between View objects and ViewModel objects [10]. In MVVM, View objects and Model objects have the same roles as in MCV. Indeed, MVVM reduces the roles of Controller objects, by moving some of them to ViewModel objects. As shown in Figure 3, ViewModel objects sit between Model objects and View/ViewController objects [2]. Controller objects in MVVM design pattern have the following roles: create views, update views, map user actions to update model, and handle segues to other controllers [5]. In contrast, the roles of ViewModel objects include: gathering data from Model objects, preparing data for presentation, input validation, and hold network logic [5].

III. Related work

This section discusses the related work and shows how the proposed model related to other existed work. The founded related work are available on GitHub which is a website to host code or software repositories by using Git Source Code Management (SCM) tool [18]. GitHub satisfied the demand of Open Source Software (OSS) projects as it helps developers and programmers to share their code and to collaborate with each other. GitHub enables its users to track other users actions by offering community-visible profiles [18]. This transparent social integration improvs learning from each other by observing and by following other programmers actions such as: the way they code, the recommended ways to solve problems, inspiring the programmers with ideas from observing other works, and much more activities [19]. Luft [20] is a Xcode Plugin that aims to improve the quality of view controllers by helping programmers keep view controllers shorter, and lighter. The main advantage of Luft is: it's availability during coding as it colors Xcode gutter depending on the view controller's status. According to number of lines, Luft has three status for view controller: (1) a light view controller that is less than 150 lines, (2) a bit heavier view controller which is greater than or equal to 150 lines or less than or equal to 300 lines, and (3) a massive view controller that has more than 300 lines. Luft colors Xcode gutter with a green color in the case of a light view controller. However, in a massive view controller case, Luft colors Xcode gutter with a red color. When a view controller gets a bit heavier Luft warns programmers by turning Xcode gutter to yellow color. Currently, the latest stable release of Xcode is version 9.2 which comes with many improvements. Since Xcode version 8, code signing requirements have been changed and the introduction of Xcode extensions. Both of these two factors made adding Xcode Plugin more complicated and sometime impossible [20]. As a result, there is no simple way to use Luft with Xcode current version so this is considered as a huge drawback. It is clear that Luft is related to Massive Controller Detection (MCD) phase in MVC & MVVM Validation Model for iOS (MMVMi). Both of Luft and MCD have the same three status: a light view controller, a bit heavier view controller, and a massive view controller. However, MCD counts view controller lines more accurate than Luft because MCD counts the code without counting comments and empty. Moreover, MCD is a python script that is independent of Xcode. Therefore, any update to Xcode will not affect Massive Controller Detection (MCD) python script. According to Object Relations Validation (ORV) phase, we found the following open sources that are related to our work: objc dependency visualizer [21], and Depcheck [22]. Objc dependency visualizer is a tool that visualizes the dependency in Objective-C project or in Swift project. Objc dependency visualizer produces dependency graph that visualizes classes coupling [21]. This tool used d3js library to give the visualizations a nifty style [21]. On the other hand,

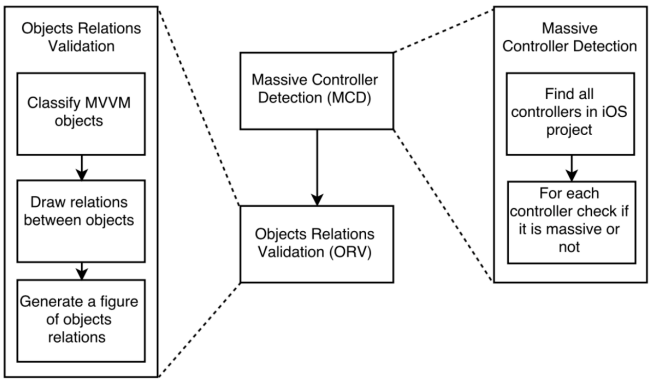


Fig. 4. MVC and MVVM Validation Model for iOS (MMVMi)

Depcheck is a tool that analyzes the dependency in Swift projects. For each class, Depcheck generates a dependency report, that enables programmers to detect classes that are dependent on many other classes. Also, the generated reports help to detect independent classes and the most used classes. Depcheck has the ability to generate a dependency graph for Swift projects. Nevertheless, none of the aforementioned tools visualize the relations in iOS project as the relations between objects in Model View ViewModel (MVVM) design pattern. As far as we know, Object Relations Validation (ORV) is the first tool to generate a dependency graph that visualizes the relations between objects in iOS project and demonstrates project's objects in point view of Model View ViewModel (MVVM) objects.

IV. MVC & MVVM Validation Model for iOS (MMVMi)

The previous section clearly demonstrates the problem with Apple's MVC design pattern. This section, provides a detailed explanations of the proposed Validation Model. Figure 4 depicts MVC & MVVM Validation Model for iOS (MMVMi). The first phase is: Massive Controller Detection (MCD). Then, the second phase is Object Relations Validation (ORV). The next two subsections discuss each of these phases in details.

A. Phase one: Massive Controller Detection (MCD)

The first phase of the proposed Validation Model is Massive Controller Detection (MCD). As shown in Figure 4, MCD consists of two steps: the first step is finding all controllers in the iOS project. Then the second step is for each controller check if it is massive or not. Basically, there are three major controller types in Cocoa frameworks: coordinating controllers, view controllers, and mediating controllers [23]. Coordinating controllers are available on both iOS and OS X, in iOS the role of coordinating controller is involved in view controllers [23]. However, the second type, view controllers are available just on iOS [23]. While the third type which are mediating controllers, are available just on OS X [23]. Our model is developed for iOS. Thus, the model implementation is based on the second type of controller types, which are view controllers. View controller classes in iOS are instances of UIViewController subclasses [24]. These view controller classes are provided by UIKit [23]. UIKit provides programmers with variety of UIViewController subclasses, each subclass has its own special-purpose [23], including UITabBarController, UITableViewController, UICollectionViewController, and more. In addition, programmers can customize UIViewController by subclassing UIViewController or by subclassing any of its subclasses. To illustrate the subclassing: a programmer will subclass UITabBarController to give the app a unique user interface. Therefore, the proposed model have to be intelligent to detect all these cases by finding all the UIViewController and its subclasses that are provided by UIKit, and also the model have to find all the customized classes that are created by programmers to satisfy programmers needs. To accomplish this intelligent detection, MCD used python Regular Expression (RE). The next subsection explains the methodology of Massive Controller Detection (MCD).

Algorithm 1 Massive controllers detection in iOS project

INPUT: iOS project directory F.
OUTPUT: All controllers in the project with detection for the massive ones.

```
for all files f_i in F do
    if f_i is an instance of a subclass of UIViewController then
        Store f_i in C_o
    end if
end for
for all controllers c_i in C_o do
    if line is not a comment or not an empty line then
        increment the total number of lines L
    end if
    if L < 150 then
        not a massive controller
    end if
    if L >= 150 or L <= 300 then
        not a massive controller but warn the programmer to take care
    end if
    if L > 300 then
        a massive controller
    end if
end for
```

Massive Controller Detection (MCD) Methodology: Regular Expression (RE) used to match a set of specifics strings [25]. Patterns in Regular Expression (RE) identifies the format of input values [26]. Thus, MCD used a pattern regular expression to detect controller objects. Algorithm 1 demonstrated MCD methodology. The directory of iOS project contains all the files including model objects, view objects, controller objects, configuration files, etc. MCD's pattern regular expression has the ability to find all controller objects (files) that belong to one of the following controllers: UIViewController, subclass of UIViewController, subclass of UIKit's UIViewController subclasses, custom UIViewController, subclass of custom UIViewController, or even any class that is inherited from UIViewController. In fact, there is no such strict rules to determine number of lines of the massive controller. However, there are conventions

in iOS community that determine the massive controller as the following: when controller has more than 300 lines, it is a massive controller. In contrast, any controller that has less than 150 lines is considered as a thin controller. Another key contribution to mention, MCD aims to help programmers more and to let them get the most benefits of the proposed model, so Massive Controller Detection model warns them when it detects a controller which has 150 lines or more but less than or equal 300 lines. This warning is important because keeping controller objects less than 150 lines not only helps in testing but also helps in maintaining the project, which in terms, helps improving the performance of the project overall. It is obvious that programmers write comments to help them in several ways such as documentations, or posting comments for other programmers. Also, programmers insert new lines to make the file readable and to follow conventions in coding style. With these two reasons in mind, Massive Controller Detection model counts the lines in an accurate way by ignoring all comments and all empty lines.

B. Phase two: Object Relations Validation (ORV)

After phase one, object relations validation (ORV) phase comes, ORV used dot [27] language to draw a directed graph that represents the relations between objects in iOS project. These objects are MVVM objects: Model object, View object, and ViewModel object. The generated graph from ORV shows how MVVM objects are related to each other. For wrong relations cases, ORV will draw these wrong edges with a dotted arrow. It will also detect the exact line number which cases the wrong relation and display it to the user. The methodology of object relations validation (ORV) phase will be explained in the next subsection.

Algorithm 2 MVVM objects relations validation in iOS project

INPUT: iOS project directory F.
OUTPUT: A directed graph with all relations between MVVM objects.

```
1:  for all MVVMObjects f_i in F do
2:      write f_i in its cluster
3:      if f_i has a relation with any other MVVM Objects
        then
4:          draw the relation
5:          if f_i is a wrong relation then
6:              draw it with a dotted arrow
7:              detect the line number that causes the wrong
                relation
8:              display a WARNING message
9:          end if
10:     end if
11: end for
```

Object Relations Validation (ORV) Methodology: ORV follows the same strategy that is used in MCD to find all Controller objects. Additionally ORV will find all View objects. In order to find View objects, ORV modifies MCD pattern regular expression instead of finding all UIViewController. It will find all View objects (files) which are subclass of UIView or any of its subclasses such as: UIButton, UITableViewCell, UILabel, UIImageView, UISwitch and the rest of the subclasses that UIKit provides. All these view classes are provided by UIKit framework for iOS, while OS X view classes are provided by AppKit framework [12]. Furthermore, there are dozens of View objects in Interface Builder Library [12]. On the other hand, there is no direct or simple way to find Model objects, and ViewModel objects because they are not subclasses of known classes. In addition, programmers create Model objects, and ViewModel objects in variety ways. As a result, when users run ORV script, it will ask users to enter the name of Model objects, and ViewModel objects. Then, ORV will generate a dot file that has the attributed graph text. As shown in Algorithm 2, in line 1 to generate the dot file, the script will collect all MVVM objects: Controller objects, View objects, ViewModel objects, and Model objects. Then, in line 2 the script will start writing each object in its cluster: Controller objects in controller cluster, View objects in view cluster, ViewModel objects in view model cluster, and Model objects in model cluster. After that, in line 3 ORV will open each file (object) of MVVM and read it to check if the object has a relation with other objects. If the relation exist, ORV will connect the objects, otherwise ORV will not generate a connection. Also, ORV will check the existing relations are they obey to MVVM rules or not. Then, ORV will draw all illegal relations with a dotted arrow as written in line 6 in Algorithm 2. Moreover, ORV will search carefully to detect the exact line number that is the reason of the wrong relation. After that, ORV will display this line number in a warning message. To accomplish this checking successfully, ORV used python regular expression (RE) pattern to find a match. The used RE pattern is accurate since it checks the file line by line: searching for other objects if they exist in this file. Moreover, sometimes programmers name methods or variables with the same name of other files or part of it. For example, suppose we have a Model object its name: Students, and suppose that in ViewModel object there is a method named: findAllStudents. ORV used word boundaries in the used RE pattern, therefore it will be intelligent enough to ignore findAllStudents and avoid partial matches. The result of running ORV script is a dot file that has all the relations between MVVM objects written as attributed graph text. Finally, the user will run the dot file to generate the directed graph that depicts the relations between MVVM objects in the project. If there is any wrong relation, ORV will depict a dotted arrow. For instance, according to Model View ViewModel design pattern, it is not allowed for Model objects to communicate with View objects. Consequently, if ORV finds Model object that is communicated with View objects, ORV will draw this relation arrow with a dotted arrow to indicate that this relation is a wrong relation. Thereby, object relations validation phase will help programmers or developers to construct a robust application architecture by writing a valid MVVM design pattern with correct relations between all objects.

Figure 5 shows the flow chart of MVC & MVVM Validation Model for iOS (MMVMi). Any iOS project can be written by following Model View Controller design pattern, Model View ViewModel design pattern, or mix of them, or even any other available design patterns. As depicted in Figure

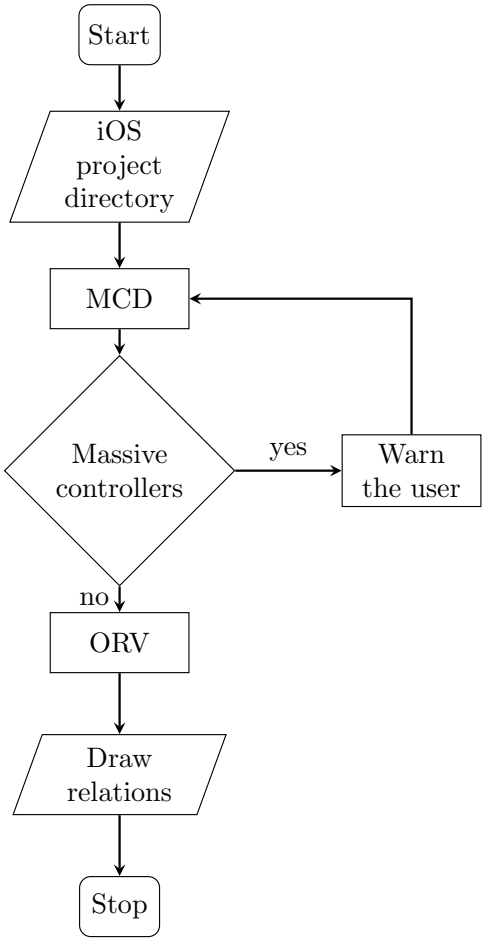


Fig. 5. MMVMi Flow chart

5, the input to MMVMi is iOS project directory. First, MCD detects massive controllers. In case the project has massive controllers, MMVMi advises the user to switch to MVVM design pattern. Next, ORV checks the relations, if all relations are satisfied, MMVMi ends processing. ORV advices the user to fix the wrong relations if it finds relations that are not satisfied by MVVM design pattern.

V. Cloudy App case

In the previous section, MVC & MVVM Validation Model for iOS (MMVMi) was introduced with its two phases. In this section, we apply MMVMi model on iOS application named Cloudy with different two cases. Case one has a valid relations between MVVM objects, while the second case has invalid relations between MVVM objects. The section begins with an introduction to the application. After that, the two phases of MMVMi model are explained and applied on Cloudy app which has valid relations. Then, MMVMi model is applied on Cloudy app with invalid relations.

A. Introduction to Cloudy App

Cloudy is an iOS application that is developed by Bart Jacobs, the founder of Code Foundry. Cloudy is a weather application that provides daily and weekly weather forecasts. Cloudy gets the user current location, and then fetches the weather information from Dark Sky API after "Cloudy" displays the weather information for the user. As depicted

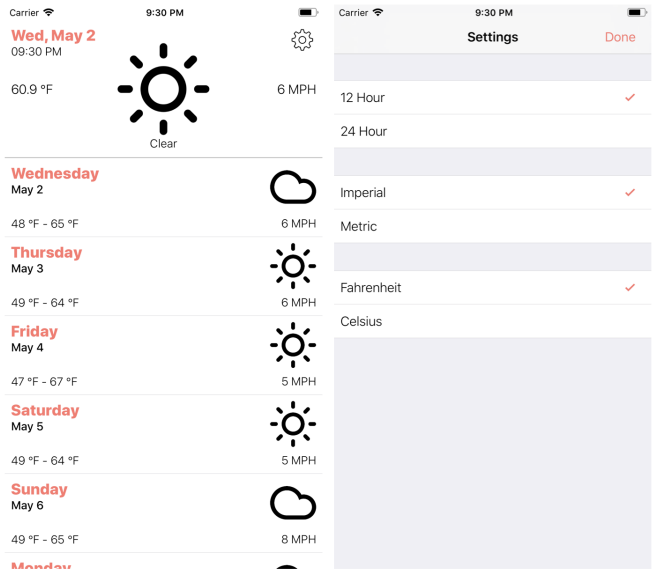


Fig. 6. Cloudy App: (a) Main view, and (b) Settings view

in Figure 6(a), the main view shows the weather information including temperature, windSpeed, summary of the current weather status, and an icon to represent the current weather status. In addition, Cloudy app offers two different formats for time, temperature, and windSpeed. Users can choose their preferences in the Settings view as illustrated in Figure 6(b). Initially, Cloudy app was built with Model View Controller (MVC) design pattern. Then, the author aims to let Cloudy app architecture: robust, and scalable. So, this architecture should have lighter view controllers and should separate concerns in a way that improves testability, and maintainability. To achieve all these goals, the developer refactored Cloudy app from Model View Controller (MVC) design pattern to Model View ViewModel (MVVM) design pattern. Both implementations of Cloudy app MVC-based [28] and MVVM-based [29] are available as open sources on GitHub. The next two subsections provide detailed explanations about applying MMVMi on Cloudy app that has valid relations between MVVM objects. Then, the final section shows the invalid relations between MVVM objects.

B. Case one: Massive Controller Detection phase on Cloudy App

Massive Controller Detection (MCD) is the first phase of MMVMi. We applied it on Cloudy app MVC implementation. Figure 7 shows MCD result, it is obvious that Cloudy application doesn't have any massive controller, even though it is implemented based on Model View Controller (MVC) design pattern. This is common: even if MVC project doesn't have any massive view controller, the developer aims to refactor the code to MVVM design patter. This refactoring has several benefits, one of the main benefits is concerns separation that enhances many non functional requirements such as: testability, maintainability, and many more. So, after refactoring the code, we ran Massive Controller Detection on Cloudy MVVM implementation, the result is shown in Figure 8. This result proves that MVVM improves Cloudy based code by reducing the total number of lines for many of its Controller objects. From the result, we can conclude that

```
CpE-508 — python MMVMi.py — 80×15

This project has 5 controllers.

Massive Controller Detection
```

Controller Name	Total number of lines	Massive controller
RootViewController	138	Thin Controller
SettingsViewController	102	Thin Controller
WeekViewController	94	Thin Controller
DayViewController	68	Thin Controller
WeatherViewController	38	Thin Controller

Fig. 7. MCD result for Cloudy app MVC-based

```
CpE-508 — python MMVMi.py — 80×15

This project has 5 controllers.

Massive Controller Detection
```

Controller Name	Total number of lines	Massive controller
RootViewController	138	Thin Controller
SettingsViewController	91	Thin Controller
WeekViewController	66	Thin Controller
DayViewController	48	Thin Controller
WeatherViewController	38	Thin Controller

Fig. 8. MCD result for Cloudy app MVVM-based

the following Controller objects got benefits from refactoring: SettingsViewController was 102 lines after refactoring it became 91 lines. WeekViewController was 94 lines, while in MVVM implementation WeekViewController is just 66 lines. Thus, with MVVM design pattern WeekViewController got rid of 28 lines. In addition, with Model View Controller design pattern DayViewController was 68 lines. In contrast with Model View ViewModel design pattern, DayViewController has 48 lines. Moreover, it is obvious that both RootViewController and WeatherViewController remain the same. After Cloudy app has been passed MCD phase successfully, it is time to apply ORV phase and validate the relations between objects.

C. Case one: Object Relations Validation phase on Cloudy App

The output of Object Relations Validation phase is two directed graphs where the first graph represents the relations between MVVM objects without depicting the interrelations between objects in the same cluster. This graph is shown in Figure 9. On the other hand, the second graph as illustrated in Figure 10, shows the relations in more details. As demonstrated in Figure 9, the WeatherViewController is not related to any object. However, Figure 10 shows that the WeatherViewController has two relations with two objects in the same cluster: DayViewController and WeekViewController.

Cloudy app has five controller objects: RootViewController, SettingsViewController, WeekViewController, DayViewController, and WeatherViewController. All these controller objects are grouped in Controller cluster as shown in Figure 9. While View Model cluster consists of the following ViewModel objects: SettingsViewTimeViewModel, SettingsViewUnitsViewModel, SettingsViewTemperatureViewModel, DayViewViewModel, WeekViewViewModel, and WeatherDayViewViewModel. In Figure 9, the upper right corner contains Model cluster that has Model objects: WeatherDayData, and WeatherData. On the other hand, as shown in the bottom of Figure 9 View cluster has the following View objects: SettingsTableViewCell, and WeatherDayTableViewCell. According to MVVM objects relations: controller objects act as intermediary between View objects and ViewModel objects. These controller's relations are clearly demonstrated in Figure 9. To illustrate, it let us consider the following example: WeatherDayTableViewCell is a View object, its user actions are interpreted by WeekViewController that is a Controller object. Next, WeekViewController maps user actions to WeekViewViewModel. After that, WeekViewViewModel can update WeatherDayData. WeatherDayData is a Model object so as shown in Figure 3 ViewModel objects own Model objects and update them, that is clearly depicted in Figure 9. Consequently, Cloudy app successfully passed Object Relations Validation phase as all the relations between Cloudy app objects are valid and they satisfied the constraints of Model View ViewModel (MVVM) design pattern.

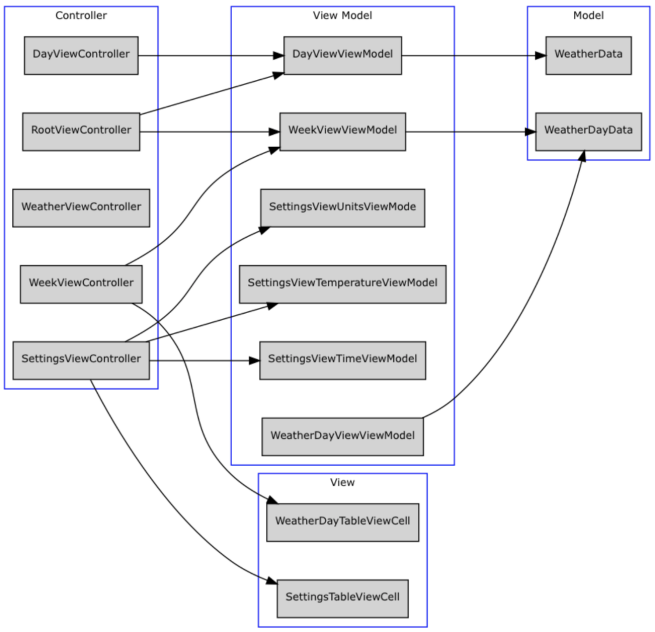


Fig. 9. Object Relations Validation without relations between objects in the same cluster

D. Case two: Cloudy App with invalid relations

In the previous case RootViewController communicates with WeatherDayData via WeekViewViewModel. Because RootViewController is a controller object, and WeatherDayData is a model object. As one of MVVM rules: controller objects are not allowed to communicate with model objects directly, there should be view model objects to manage these communications. In this case, WeekViewViewModel was deleted from Cloudy app. As a result, all its code was moved to RootViewController. Thus, when MMVMi ran on Massive Controller Detection phase as shown in Figure 11, it warned the developer to take care about RootViewController as it has a chance to be a massive controller. Next, Relations Validation phase detects exactly where the wrong relations occur. Then, it shows a warning message with the objects of the wrong relation and the line number that causes this wrong relation. The warning message for our case is illustrated in Figure 12, it shows that in line 16 RootViewController communicates with WeatherDayData which is a wrong relation. Finally, MMVMi generated a directed graph which colored

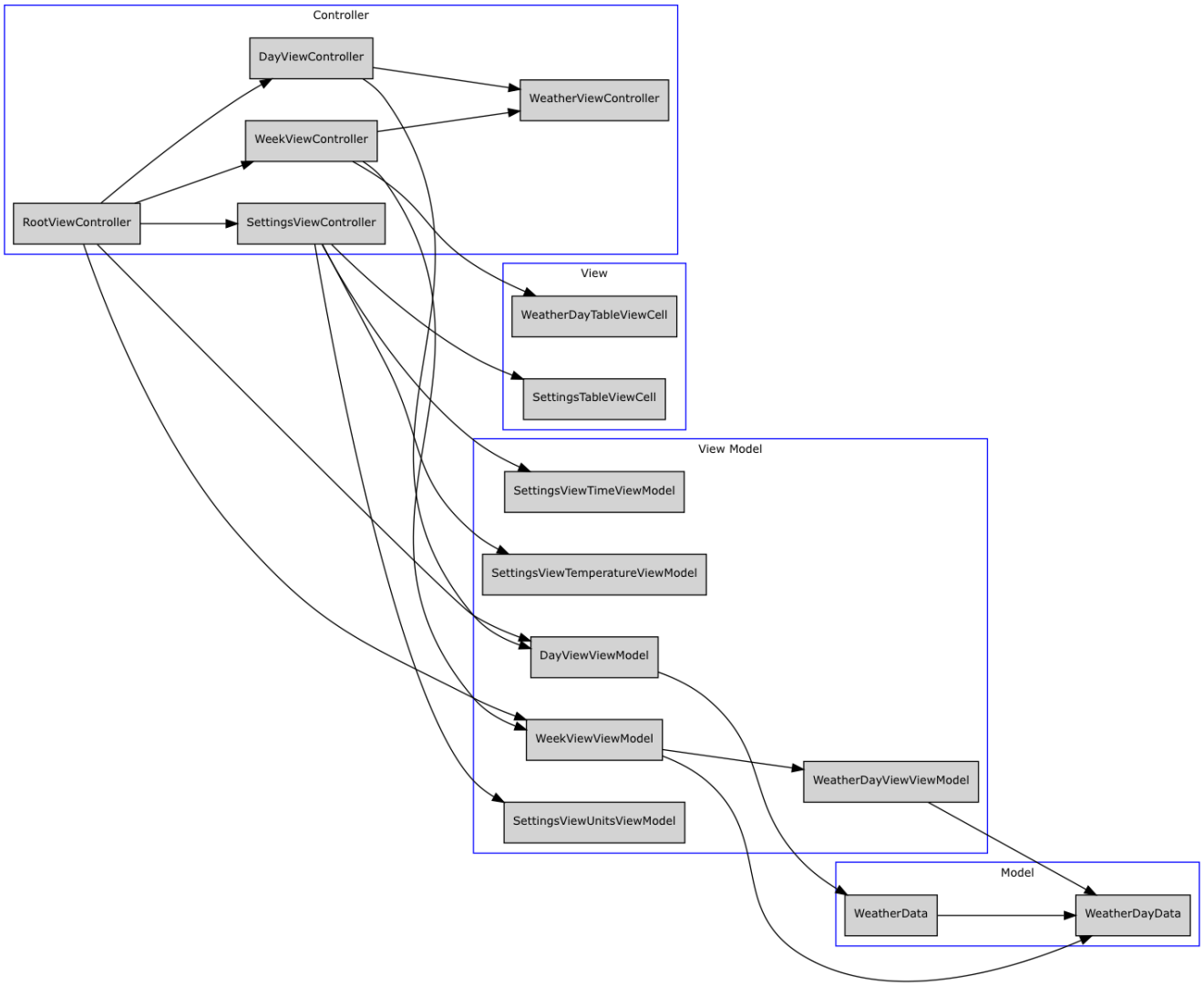
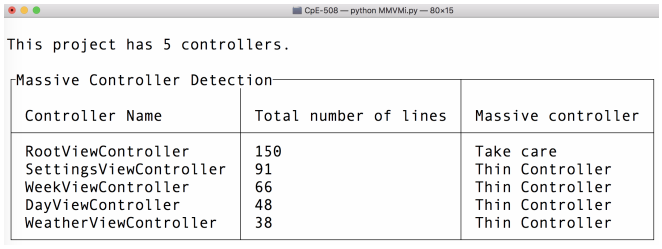


Fig. 10. Object Relations Validation with relations between objects in the same cluster

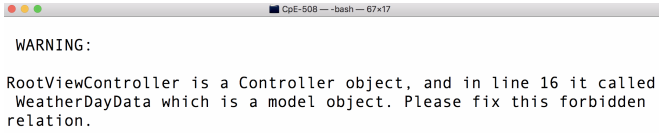


```
This project has 5 controllers.

Massive Controller Detection
```

Controller Name	Total number of lines	Massive controller
RootViewController	150	Take care
SettingsViewController	91	Thin Controller
WeekViewController	66	Thin Controller
DayViewController	48	Thin Controller
WeatherViewController	38	Thin Controller

Fig. 11. MCD result for Cloudy app case 2



```
 WARNING:

RootViewController is a Controller object, and in line 16 it called
 WeatherDayData which is a model object. Please fix this forbidden
relation.
```

Fig. 12. Warning message for wrong relations between MVVM objects

the wrong relation arrow with a dotted line as depicted in Figure 13.

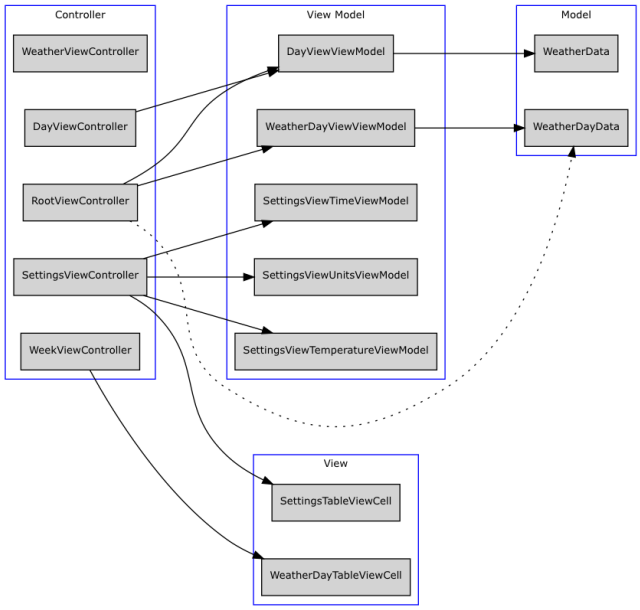


Fig. 13. Invalid relations between MVVM objects

VI. Evaluation and Results

This section evaluates MMVMi model by comparing it with relevant tools such as Depcheck and Objc Dependency

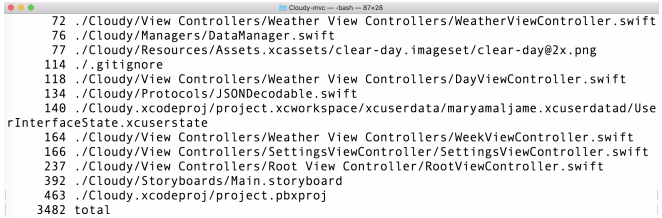


Fig. 14. Detecting massive controllers by using Krzysztof Zablocki script on Cloudy MVC version

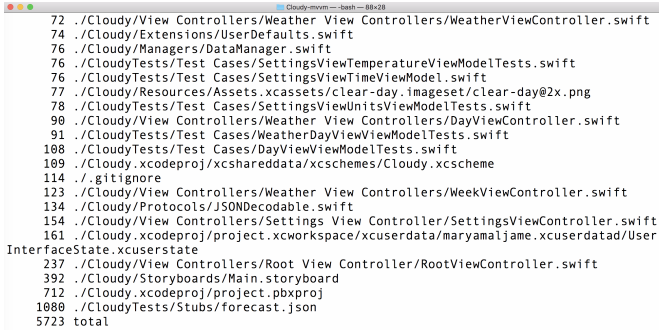


Fig. 15. Detecting massive controllers by using Krzysztof Zablocki script on Cloudy MVVM version

Visualizer tool. The first subsection, evaluates Massive Controller Detection phase. While Object Relations Validation phase is evaluated in the next two subsections. Subsection two, compares Object Relations Validation phase with Depcheck by running both of them on Cloudy app MVVM version. The next subsection, applies Objc Dependency Visualizer on Cloudy app MVVM version. All results and graphs output of the aforementioned tools will be discussed in detail in this section.

A. *Massive Controller Detection phase Comparison*

Massive Controller Detection phase helps developers to detect massive controllers in iOS projects. In a Good iOS Application Architecture: MVVM vs. MVC vs. VIPER talk, Krzysztof Zablocki proposed a script that triggers warnings when the project has massive controllers. Krzysztof Zablocki script outputs all files in the project with their full path and the total number of lines for each file. The results of running Krzysztof Zablocki script on Cloudy app MVC version is shown in Figure 14. Indeed there is no doubt that the output of MCD in Figure 7 is much more abstract than Figure 14. In addition, Krzysztof Zablocki script ran on Cloudy app MVVM version and its output is depicted in Figure 15. On the other hand, the same project was run on MCD and its output is illustrated in Figure 8. By comparing all those different outputs, it is obvious that Krzysztof Zablocki script has a major drawback as it computes the total number of lines for all files, not just the controller files. Accordingly, developers have to scan through all the output files to check the total number of lines for each controller file. In contrast, MCD script searches for controller files implicitly, and then it displays just controller files. Thus, MCD provides a convenient method for developers to detect massive controllers. Furthermore, Krzysztof Zablocki script counts all lines including comments and empty lines, unlike MCD which ignores comments and empty lines and just counts code lines. Thereby, MCD is more accurate. For example, the total number of lines in WeatherViewController is 38 lines as shown in Figure 7 and Figure 8. While Krzysztof Zablocki script shows that WeatherViewController has 72 lines in total as illustrated in Figure 14 and Figure 15.

B. *Comparison with Depcheck*

This subsection compares MMVMi with Depcheck, in terms of visualizing the relations in iOS project. As mentioned in the related work section, Depcheck is an analyzer tool for Swift projects [22]. Basically, Depcheck has three command: analyze, usage, and graph. To report dependencies per classes, run analyze command. While, to count each class

Fig. 16. Depcheck analyze command output

dependencies, run usage command. To generate a dependency graph, run graph command [22]. Figure 16 shows a screenshot of terminal output, when analyze command ran on Cloudy App MVVM version. Obviously, analyze command output is useless as it reports class name with the number of dependencies. Actually, when developers used visualizing tools, they aim to visualize projects details with useful information that help them to inspect the project smoothly. Depcheck's analyze command should be improved by displaying the name of all the dependencies not just the number. Furthermore, even though in case a class is not dependent on any other classes, analyze command will display this class with zero number beside it. For example, in Figure 16, AppDelegate, DataManagerError, SettingsRepresentable, TimeNotation, JSONDecoderError, WeatherViewController and much more all of them have zero dependencies. This is really worthless.

Figure 17 illustrates part of the dependency graph which is generated by running Depcheck's graph command on

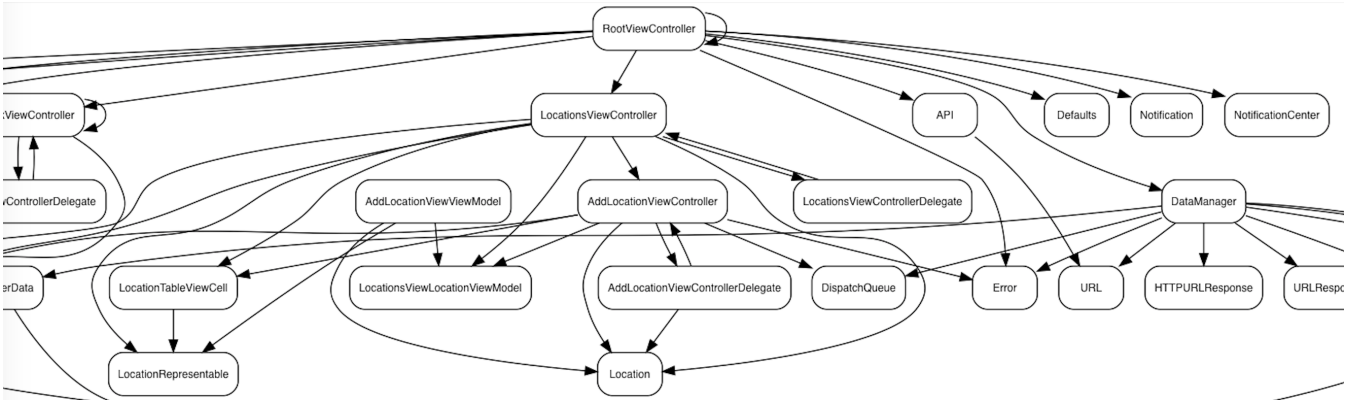


Fig. 17. Depcheck graph command output

Cloudy App MVVM version. Roughly, this graph helps view the relations between objects in the project. However, the graph is huge which makes it unclear and difficult to find a specific relation. On the contrary, Figure 9 depicts MMVMi dependency graph for the same project: Cloudy App MVVM version. Definitely MMVMi graph is more helpful than Depcheck graph for several reasons. Such as MMVMi graph omits unnecessary relations, whereas Depcheck graph draws all relations between objects even the relation between the object and itself as shown in Figure 17: RootViewController has an arrow from its node to itself. Moreover, MMVMi graph assists developers to observe the relations between MVVM objects easily. However, using Depcheck graph might help but in a complex way as it has numerous arrows.

C. Comparison with Objc Dependency Visualizer

Objc dependency visualizer tool illustrates the relations between classes in iOS project in an interactive way. In this subsection, objc dependency visualizer graph will be compared with MMVMi graph. Figure 18 displayed objc dependency visualizer graph for Cloudy MVVM version. In the top right corner, there is a Live editor that gives users the ability to control circle size, charge multiplier, and link strength. Users can control all these options in a simple way by moving and adjusting the sliders according to their needs. Also, there is "show names" option where when it checked all classes name will be appeared on their corresponding nodes. Last but not least, filter option where users can type a regular expression to filter nodes. By comparing Figure 9 with Figure 18, it is clear that Figure 18 shows how each class is linked to other classes which made it crowded and messy. Therefore, in Figure 19 we applied filter to show just Controller objects. After applying Controller objects filter, the arrows relations will be illustrated between only nodes that have controller within their name. It will be better if the Live editor has an option to omit unrelated nodes, and other option to show the relations between the filtered nodes and other nodes.

Figure 20 depicts ViewModel filter, the filter is really useless as it just shows the relation between WeekViewViewModel node and WeatherDayViewViewModel node. However, there are many ViewModel nodes including DayViewViewModel, SettingsViewTemperatureViewModel, SettingsViewTimeViewModel, and SettingsViewUnitsViewModel. Objc dependency visualizer tool dismissed all these ViewModel nodes relations because these relations are not with other ViewModel nodes. Indeed, developers need to check how ViewModel nodes are linked with Model nodes and Controller nodes. This relations check is clearly illustrates in MMVMi graph as shown in Figure 9. Moreover, filter option can be considered as a drawback because in order to use it, users must have a good knowledge in regular expression to write a regular expression that fulfills their need.

VII. Conclusion

Design patterns have a huge impact that affects several aspects of software engineering including development, and architecture. In this work, we focused on iOS application development with two design patterns: Model View Controller (MVC) design pattern, and Model View ViewModel (MVVM) design pattern. MVC suffers from many limitations as discussed earlier. Consequently, developers tend to use Model View ViewModel to overcome MVC limitations. The overall objective of our work is to help developers to stick to the rules of MVC design pattern, and help them to check and detect if they have Massive View Controller problem. In this case, they have to switch to MVVM design pattern. We aim to advice developers to switch to MVVM design pattern as it separates concerns in a better way and it distributes the roles more productively. To achieve these objectives, we have developed a validation model: MVC & MVVM Validation Model for iOS (MMVMi). The presented validation model addressed our goals and aims as it helps developers to validate their projects with MVC rules. The proposed model validates the relations between project objects and check if they satisfy the MVVM objects relations rules. We assert that such a validation model will aid developers and programmers to successfully apply MVC or MVVM on their iOS projects and get the main benefits from applying design pattern correctly. We hope this work will help iOS community and guide them to build robust iOS applications that are well structured with a valid design pattern.

VIII. Future work

Currently, Object Relations Validation phase in MMVMi used dot language to produce the directed graph. We see that this is an aspect that needs to be improved. The output of ORV phase is a simple directed graph that does not allow

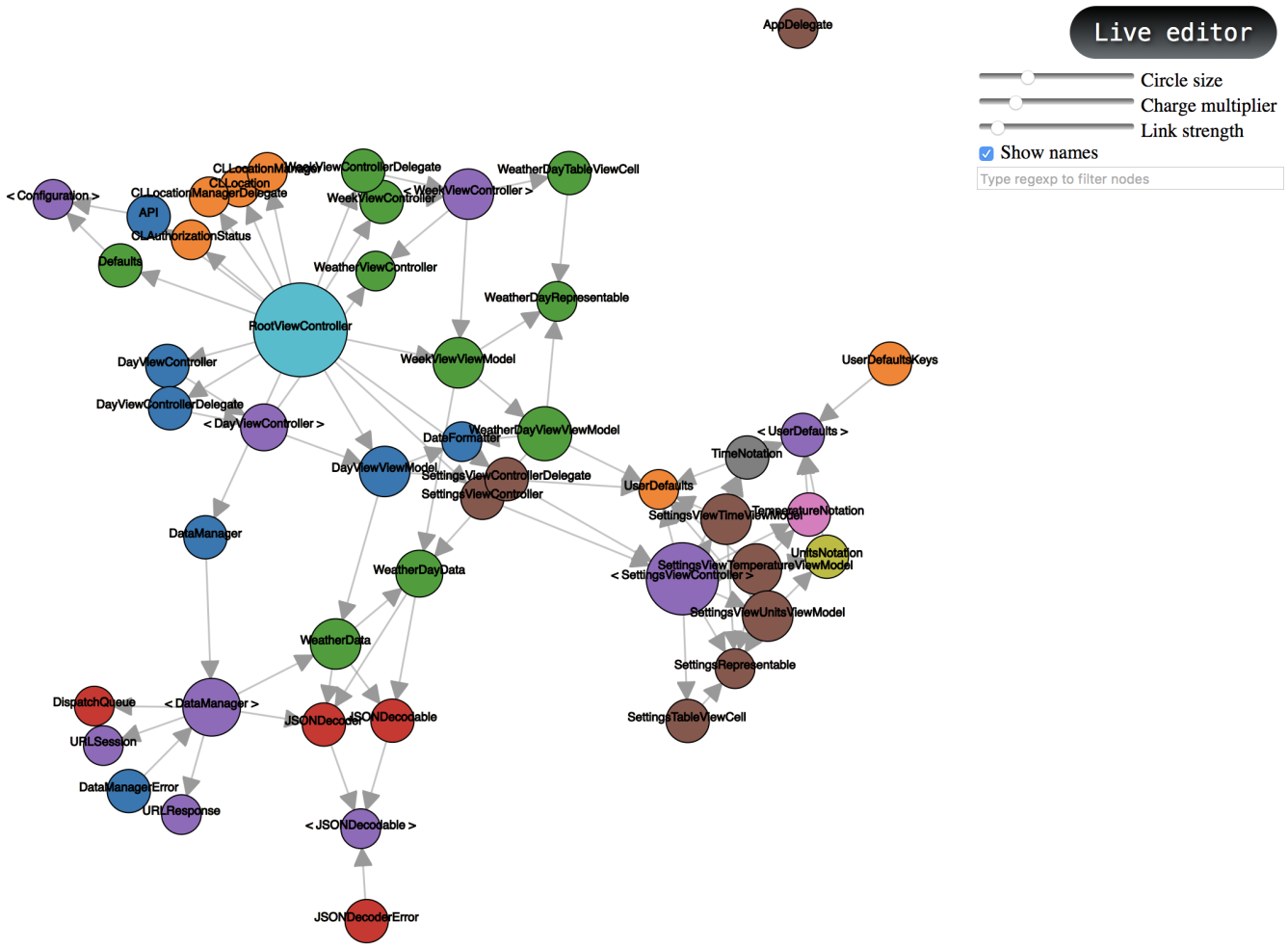


Fig. 18. Objc dependency visualizer output for Cloudy MVVM version

the user to interact with it. User interactions can come in different forms such as giving the user the ability to move a cluster or nodes to anyplace. All these interactions can be done with d3js library and they will be visualized in a powerful interactive way. In fact, in iOS community there are many projects that use both design patterns: Model View Controller, and Model View ViewModel. So, we plan to improve the proposed model in which, the model should have the ability to specify the project type, and it should be able to deal with projects that use both of MVC, and MVVM. Hence, the aforementioned improvements will enhance the MMVMi and will make it more powerful.

REFERENCES

[1] E. Sorensen and M. Mikailesc, "Model-view-viewmodel (mvvm) design pattern using windows presentation foundation (wpf) technology," *MegaByte Journal*, vol. 9, no. 4, pp. 1–19, 2010.

[2] R. Vice and M. S. Siddiqi, *MVVM Survival Guide for Enterprise Architectures in Silverlight and WPF*. Packt Publishing Ltd, 2012.

[3] D. Patterns and C. Pattern, "Model-view-controller," *Microsoft Patterns & Practices, http://msdn. microsoft. com/practices/type/Patterns/Enterprise/DesMVC*, 2003.

[4] A. Leff and J. T. Rayfield, "Web-application development using the model/view/controller design pattern," in *Enterprise Distributed Object Computing Conference, 2001. EDOC'01. Proceedings. Fifth IEEE International*. IEEE, 2001, pp. 118–127.

[5] D. A. Lo, *Sensor Plot Kit: An iOS Framework for Real-time plotting of Wireless Sensors*. University of California, Irvine, 2015.

[6] B. Orlov, "ios architecture patterns," https://medium.com/ios-os-x-development/ios-architecture-patterns-ecba4c38de52, (Accessed on 04/03/2018).

[7] T. Lee and T. Brunner, "A prototype to increase social networking between staff: A web application for companies," 2017.

[8] A. Freeman, *Pro design patterns in swift*. Apress, 2015.

[9] J. Patel, S. Okamoto, S. M. Dascalu, and F. C. Harris Jr, "Web-enabled toolkit for data interoperability support," in *Proceedings of the 21th International Conference on Software Engineering and Data Engineering (SEDE-2012), Los Angeles, CA*, 2012, pp. 161–166.

[10] A. Syromiatnikov and D. Weyns, "A journey through the land of model-view-design patterns," in *Software Architecture (WICSA), 2014 IEEE/IFIP Conference on*. IEEE, 2014, pp. 21–30.

[11] M. Neuburg, *IOS 8 Programming Fundamentals with Swift: Swift, Xcode, and Cocoa Basics*. " O'Reilly Media, Inc.", 2015.

[12] "Model-view-controller," https://developer.apple.com/library/content/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html, (Accessed on 04/03/2018).

[13] A. Allan, *Learning iOS Programming: From Xcode to App Store*. " O'Reilly Media, Inc.", 2013.

[14] R. B'far, *Mobile computing principles: designing and developing mobile applications with UML and XML*. Cambridge University Press, 2004.

[15] K. Topley, F. Olsson, J. Nutting, D. Mark, and J. LaMarche, *Beginning IPhone Development with Swift: Exploring the IOS SDK*. Apress, 2014.

[16] H. Sun, J. Zhang, G. Sun, and Y. Li, "Agricultural traceable and marketing system based on ios-platform and wireless sensor network," *Journal of Computer and Communications*, vol. 5, no. 06, p. 45, 2017.

[17] L. Nguyen and K. Nguyen, "Application of protocol-oriented mvvm architecture in ios development," 2017.

[18] K. Peterson, "The github open source development process," *url: http://kevinp.me/github-process-research/github-processresearch.pdf (visited on 05/11/2017)*, 2013.

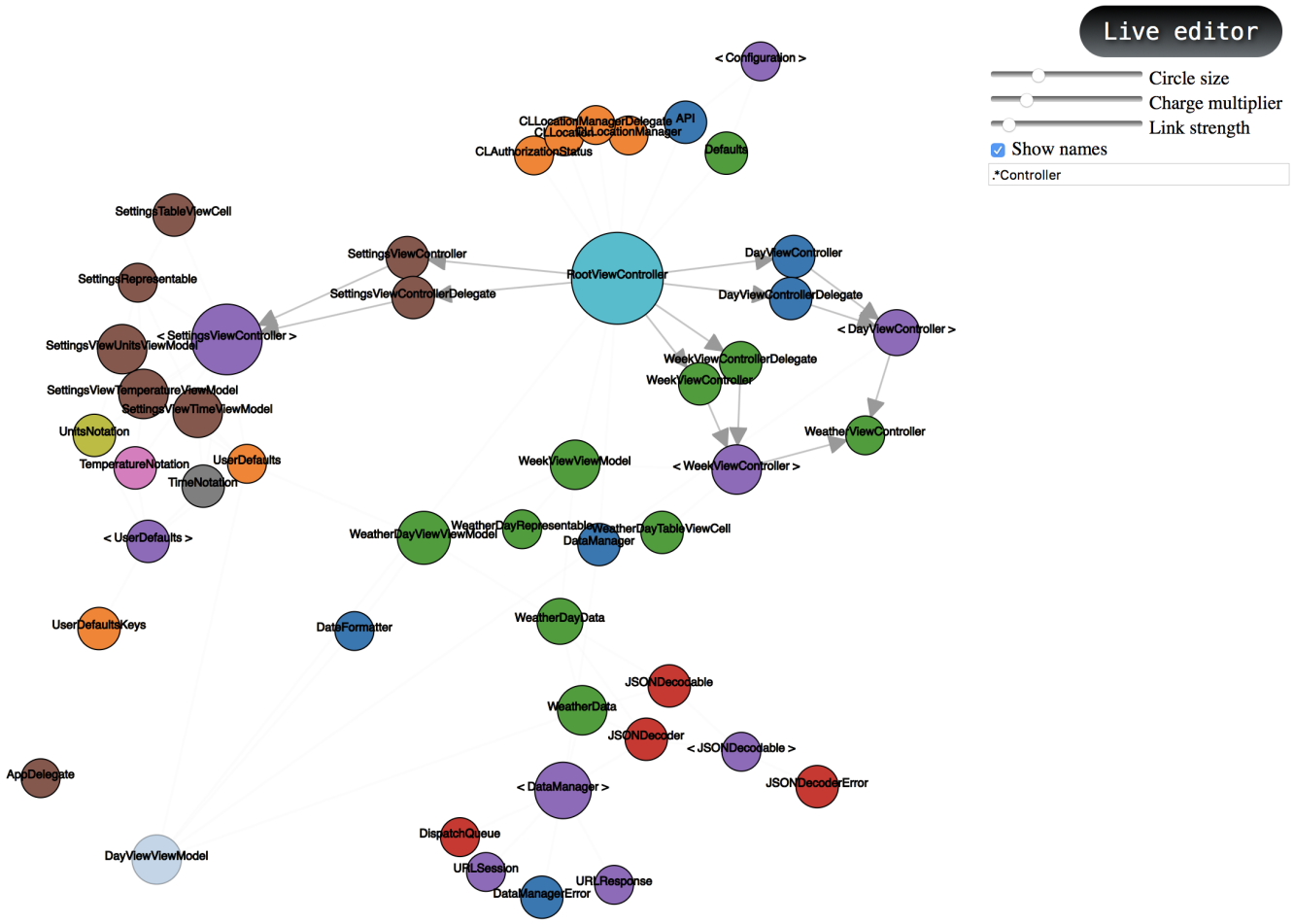


Fig. 19. Applying Controller filter on Cloudy MVVM version

[19] L. Dabbish, C. Stuart, J. Tsay, and J. Herbsleb, "Social coding in github: transparency and collaboration in an open software repository," in *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work*. ACM, 2012, pp. 1277–1286.

[20] "Github - k0nserv/luft: The xcode plugin that helps you write lighter view controllers," https://github.com/k0nserv/luft, (Accessed on 04/03/2018).

[21] "Github-paultaykalo/objc-dependency-visualizer: Objective-c and swift dependency visualizer. it's tool that helps to visualize current state of your project. it's really easy to see how tight your classes are coupled." https://github.com/PaulTaykalo/objc-dependency-visualizer, (Accessed on 04/03/2018).

[22] "Github-wojteklu/depcheck: Dependency analyzer tool for swift projects," https://github.com/wojteklu/depcheck, (Accessed on 04/03/2018).

[23] "Controller object," https://developer.apple.com/library/content/documentation/General/Conceptual/DevPedia-CocoaCore/ControllerObject.html#//apple_ref/doc/uid/TP4000895-CH11-SW1, (Accessed on 04/04/2018).

[24] "Uiviewcontroller-uikit — apple developer documentation," https://developer.apple.com/documentation/uikit/uiviewcontroller, (Accessed on 04/03/2018).

[25] S. Brin, "Extracting patterns and relations from the world wide web," in *International Workshop on The World Wide Web and Databases*. Springer, 1998, pp. 172–183.

[26] H. Hosoya and B. Pierce, "Regular expression pattern matching for xml," in *ACM SIGPLAN Notices*, vol. 36, no. 3. ACM, 2001, pp. 67–80.

[27] E. Koutsofios, S. North *et al.*, "Drawing graphs with dot," Technical Report 910904-59113-08TM, AT&T Bell Laboratories, Murray Hill, NJ, Tech. Rep., 1991.

[28] "bartjacobs/cloudy at mvc," https://github.com/bartjacobs/Cloudy/tree/mvc, (Accessed on 04/03/2018).

[29] "bartjacobs/cloudy: Mastering model-view-viewmodel with swift," https://github.com/bartjacobs/Cloudy, (Accessed on 04/03/2018).

Mariam Aljamea earned bachelor degree in computer engineering from Kuwait University in 2012. Currently she is a master student in computer engineering at Kuwait University and she is working on distribution systems using Apache Spark for her thesis. Also, she is an iOS Developer in Ministry of Education. She is a researcher in apache spark and bioinformatics.

Mohammad Alkandari is an Assistant Professor of computer engineering at Kuwait University, Kuwait, where he has been on the faculty since 2012. He received his Ph.D. degree in computer science at College of Engineering from Virginia Polytechnic Institute and State University (Virginia Tech). He was the director of the Office of Engineering Education Technology at Kuwait University, College of Engineering and Petroleum for 3 years. He is currently the coordinator of Software and Systems Engineering Research Group at Computer Engineering Department. He is a researcher in software engineering, requirements engineering, software project management, software quality assurance, privacy and data protection, and human-computer interaction.

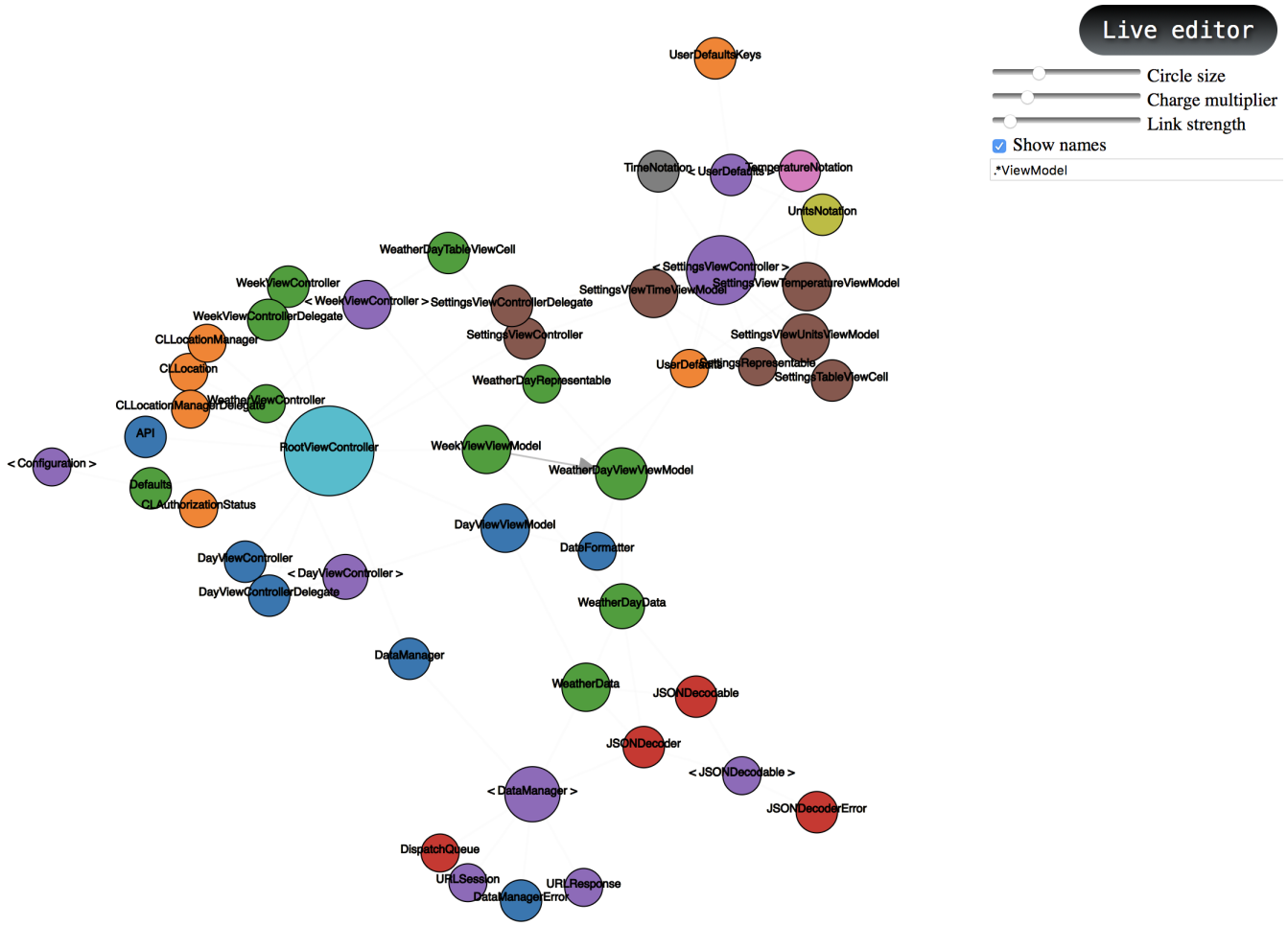


Fig. 20. Applying ViewModel filter on Cloudy MVVM version