electronics

MDPI

Article

Rule-Based Architectural Design Pattern Recognition with GPT Models

Zoltán Richárd Jánki * and Vilmos Bilicki

Department of Software Engineering, Institute of Informatics, University of Szeged, 6720 Szeged, Hungary; bilickiv@inf.u-szeged.hu

* Correspondence: jankiz@inf.u-szeged.hu

Abstract: Architectural design patterns are essential in software development because they offer proven solutions to large-scale structural problems in software systems and enable developers to create software that is more maintainable, scalable, and comprehensible. Model-View-Whatever (MVW) design patterns are prevalent in many areas of software development, but their use in Web development is on the rise. There are numerous subtypes of MVW design patterns applicable to Web systems, but there is no exhaustive listing of them. Additionally, it is unclear how these subtypes can be utilized in contemporary Web development, as their usage is typically unconscious. Here, we discuss and define the most prevalent MVW design patterns used in Web development, as well as provide Angular framework examples and guidance on when to employ a particular design pattern. On the premise of the primary characteristics of design patterns, we created a rule system that large language models (LLMs) can comprehend without doubt. Here, we demonstrate how effectively Generative Pre-trained Transformer (GPT) models can identify various design patterns based on our principles and verify the quality of our recommendations. Together, our solution and GPT models constitute an effective natural language processing (NLP) solution capable of detecting MVW design patterns in Angular projects with an average accuracy of 90%.

Keywords: large language models; GPT; software engineering; design pattern; natural language processing; Web development; Angular

Citation: Jánki, Z.R.; Bilicki, V. Rule-Based Architectural Design Pattern Recognition with GPT Models. *Electronics* **2023**, *12*, 3364. https://doi.org/10.3390/electronics12153364

Academic Editors: Iouliia Skliarova, Jinhyun Kim, Seonah Lee and Suwon Lee

Received: 13 July 2023
Revised: 31 July 2023
Accepted: 4 August 2023
Published: 6 August 2023

1. Introduction

Software development is a multidimensional field that centers on the creation, testing, and maintenance of applications and frameworks. It encompasses a comprehensive process that includes comprehending user requirements, designing software architecture, coding, testing, and maintenance. This process is constantly evolving as new technologies, methodologies, and instruments are continuously introduced. Even though there are a large number of duties in the background of design and development, developers face problems frequently. Later, from recurring problems, well-known solutions will arise. From the solutions, more abstract but still applicable best practices may emerge, and from the best practices, the most abstract and generally applicable design patterns are defined.

There is a vast array of design patterns that are categorized by various aspects and solutions. The Gang of Four (GoF) [1] created a taxonomy of design patterns for object-oriented software development in their book, which is still widely acknowledged today. However, these are software design patterns that provide guidance for implementing a class or function based on the nature of the problem. When considering a piece of software as a system, we can zoom out to its architecture in order to observe its components and their interactions. Obviously, there are taxonomies for architectural design patterns, but their application and interpretation are highly dependent on the underlying technology and framework.

The Model-View-Controller (MVC) pattern was created by Trygve Reenskaug in 1979 [2,3]. It is one of the most common and earliest design patterns. As an objective, it

separates the domain model, the user interface (UI), and their connector component [4]. The first conscious application of MVC was introduced by Krasner and Pope [5] in 1988. They presented how MVC can be implemented in the Smalltalk-80 programming language and environment to enhance the creation of user interface applications. As a result of its popularity, new technologies and frameworks are perpetually adopting MVC or creating new variants of it. The majority of variants utilize the same structure, separating the Model, View, and Controller, but their modes of communication and relationships may vary.

Taligent [6] was the first to implement Model-View-Presenter (MVP), a modified and generalized variant of the original MVC concept that appeared more than a decade after MVC. They maintained the principle of separating the components, but increased the Presenter's responsibility by transforming the domain data for the View and governing its content and behavior. MVP first went live by implementing Dolphin Smalltalk, an open-source Smalltalk programming solution for the Windows platform. Later, additional MVP variants were developed by modifying the communication and responsibilities among the components [7].

Microsoft's John Gossman [8] introduced the Model-View-ViewModel (MVVM) design pattern in 2005. The Windows Presentation Foundation (WPF) was the first framework to implement MVVM principles. Martin Fowler [9] presented MVP and the Presentation Model, which served as its primary motivations. There are currently MVVM variants with minor modifications, such as Application Model and Presentation Model, that have different principles [10]. However, there are numerous MVVM implementation variants as well. There are prominent frameworks, such as WPF, that offer their best practices by employing the MVVM design pattern. However, there are technologies in which MVVM is not predominate but can be readily implemented. MVVM can be used to implement native Android and iOS applications in mobile application development, but it is not the primary approach [11,12]. In addition to structural distinctions, these methods may also have varying effects on performance [13,14].

All of these design patterns are extensively used in software development, and while there are numerous variants of them today, their core concepts remain the same. They are also utilized in Web and mobile development [15,16]. Some modern frameworks are said to be pattern-specific, but there are use cases and practices in which other patterns or variants function more effectively. Since the origin of all Model-View-related design patterns is the same, they are commonly referred to as Model-View-Whatever (MVW) patterns. Developers frequently combine the solutions provided by design patterns to create new best practices. Object orientation is advised to implement these design patterns. React, Vue, and Angular are the front-end frameworks that use the Node.js runtime environment the most frequently today. The history of MVW design patterns reveals that the majority of their origins are related to Microsoft. Since its second edition, Angular has been a framework that supports typed implementation by using TypeScript as its principal development language. We discovered that in Angular application development, it is unclear which MVW design pattern is the most prevalent. AngularJS and Angular 2+ have fundamentally distinct concepts. Throughout the remainder of this article, Angular pertains to the Angular 2+ framework. To examine MVW design patterns in contemporary Web frameworks, we concentrate on Angular projects because only then can we guarantee typed source code.

Numerous researchers are currently concerned with design patterns and their effects on software development, and an increasing number of studies are being conducted to detect them using natural language processing (NLP) techniques and compile global statistics. Continuously and abruptly, OpenAI [17–19] is developing Generative Pre-trained Transformer (GPT) models. Within five years, they released four major model iterations, and with the release of the GPT-3 and GPT-4 models in the guise of chatbots, a plethora of opportunities opened up in numerous fields. Using their Application Programming Interface (API), the properties' creativity and consciousness can be readily controlled. There are large language models (LLMs) that can be fine-tuned, but their capabilities vary considerably.

Our aim is to present how architectural design pattern variants can be applied and combined in modern Web development, and to provide a novel approach for design pattern detection using prompt engineering with GPT models. On the basis of an elaborated taxonomy of MVW design patterns and an examination of open-source projects, we can acquire a clearer understanding of how and why MVW variants are utilized in Angular application development.

The following are the primary contributions of this paper:

- a classification of MVW design patterns for Angular application development with formal definitions and examples;
- a rule-based approach for detecting MVW design patterns in Angular applications using GPT models;
- the results of the accuracy attained in design pattern detection with GPT models;
- the distribution of MVW design patterns in Angular development.

The remaining sections are organized as follows. In Section 2, we provide an overview of the related works. In Section 3, we present the proposed materials and methods. Section 4 presents the experimental results and Section 5 presents a short discussion. In Section 6, we present the limitations of our study. Section 7 concludes the study by presenting the key findings, limitations, and future research.

2. Related Work

Design pattern detection and analysis are active research fields. Koirala [20] provided a thorough comparison of four distinct MVW design patterns in a Microsoft environment by addressing the key issues and proposed solutions. Based on his study, Syromiatnikov and Weyns compiled an inventory of well-documented MVW design patterns utilized in UI architectures [10]. There are similarities between specific design patterns, but new technologies are constantly modifying them and creating new alternatives. They did a decent job of comparing example use cases, but it is difficult to identify design patterns in projects based on a single example and brief description. In addition, the MVW design pattern family has expanded as new technologies have become prevalent and as new best practices have emerged. Software design pattern detection yields promising results, whereas architectural design pattern detection appears to be a more difficult endeavor. N. Nazar et al. attained an accuracy of 80% with their trained Random Forest Classifier [21]. Since that study, NLP has advanced significantly, and LLMs can now perform better. L. Wang et al. [22] trained Visual Geometry Group (VGG) and Support Vector Machine (SVM) classifiers for 12 GoF design patterns with more than 85% stable precision and recall. The sample sets were constructed in the form of Unified Modeling Language (UML) models, and image classification was used to detect design patterns. While GoF design patterns can be discovered by analyzing one or two files within a project, architectural design patterns necessitate the analysis of source codes from multiple files. R. Nord and Z. Kurtz compared machine learning approaches for detecting MVC and assisting with software quality evaluations with a comparatively low degree of precision [23]. With these outcomes, global statistics cannot be compiled. In [24], a novel hybrid approach was presented for detecting MVC architectural layers in Android applications. By exploring the layers, students and beginners can better understand the projects. Similar to our study, Komolov et al. [25] focused on architectural design patterns in their research. They evaluated multiple machine learning strategies and analyzed Android applications with MVVM and MVP as their focal points. The training dataset was founded on source code metrics. Their approach to machine learning has an accuracy of 83%, but it is limited to two main categories and lacks a semantic analysis of the projects. Complex tasks may also require semantic analysis; simple metrics are insufficient. To conduct a reliable semantic analysis, it is necessary to develop a well-defined rule system. This article focuses on Angular projects and provides a detailed classification of MVW design patterns with clearly defined rules. By modifying the classification criteria based on the employed technology,

our method is applicable to other Web frameworks and other technologies. This technique does not require neural network training.

3. Materials and Methods

This section presents a taxonomy and formal definitions of MVW design patterns. In this study, we present a machine learning technique for identifying MVW design patterns through semantic analysis of source code. The semantic analysis is founded on a set of principles that distinguish the design patterns with precision. In addition, we provide Angular framework implementation examples to elucidate what elements and components the language model seeks.

3.1. MVW Taxonomy for Frontend Frameworks

According to the available literature, MVW is a design pattern family consisting of three primary groups: MVC, MVP, and MVVM. It has been discovered, however, that new UI frameworks may modify the original concepts and produce new best practices and, eventually, new design patterns. The same thing occurred with MVW design patterns, as modified alternatives emerged in addition to the traditional MVC, MVVM, and MVP design patterns. Here, we present a taxonomy of MVW design patterns and classify them according to the design pattern group to which they are conceptually closest. The MVW design pattern family that can be found in modern Web development is depicted in Figure 1.

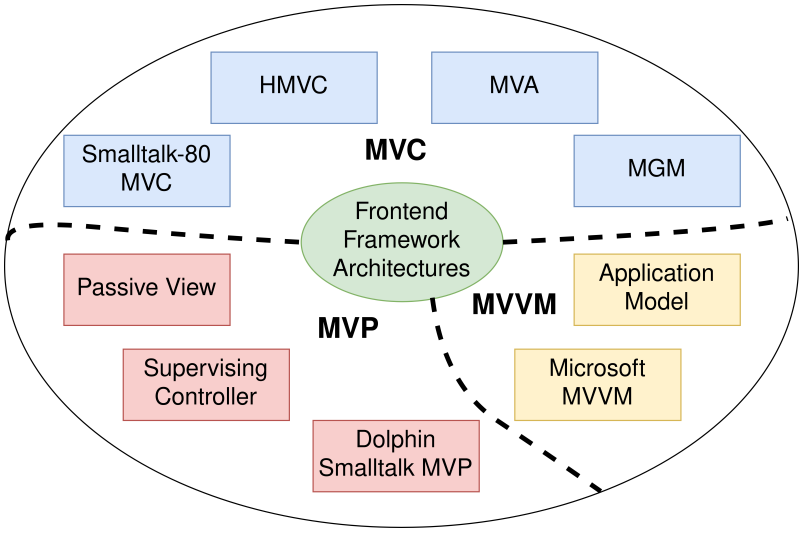

Figure 1. Categories of MVW design patterns and their variants.

MVC has the longest history and the most extensive list of variants. MVC is the foundation for a variety of technologies with architectural modifications. It is additionally frequently employed in the development of desktop, mobile, and Web applications. MVC has significantly influenced iOS development, becoming a standard in this field [26]. Android development follows somewhat distinct principles. MVC has been applicable to both desktop and Web applications since it was first introduced and demonstrated with Smalltalk. In Web development, however, Spring Web MVC was the most influential in the dissemination of the MVC design pattern [27]. In Angular development, we discovered that not only is it possible to implement components using Smalltalk-80 MVC concepts, but it is also a commonly used architectural technique. Since Angular is a component-based framework, hierarchy plays an essential role, and components are frequently nested within one another. Signs of Hierarchical MVC (HMVC) are also common in Angular application development, where hierarchical communication and data transfer between MVC components are essential. The Controller functions as a mediator when the control of UI elements and the manipulation of their content are delegated to the Controller [28]. Typically, this

alternative to MVC is referred to as Model-View-Adapter (MVA). It is also possible to delegate the adapter role and implement adapters for individual elements. This particular design pattern is known as Model-GUI-Mediator (MGM) [29]. These are the four most prevalent MVC design pattern variants that can exist in Angular.

MVVM represents the tiniest subset of the MVW design pattern family. Application Model [30] and Microsoft MVVM are variants. The fundamental idea is to isolate the business model and the View. This induces a layer responsible for the View by providing a model for it, thereby increasing the Model's responsibility. The majority of modern Web frameworks, such as Angular, support MVVM and adhere to its principles. However, it is crucial to note that there are implementation techniques that contradict the use of MVVM, with MVC or MVP characteristics being more common.

MVP is the second-largest group in the MVW family. It consists of three variants: the Dolphin Smalltalk MVP, the Supervising Controller, and the Passive View. In MVP, the Controller has been replaced by the so-called Presenter, which has increased responsibility for the View and its state. In all MVP variants, the Presenter is responsible for validating user input and may manipulate View elements based on the validation result. MVP can be implemented with various strategies in Angular. On the one hand, the Presenter can be a distinct class in the class hierarchy whose sole responsibility is to handle, validate, and process the data. On the other hand, the Presenter can be integrated into a component that manages View directly. The Model can alter based on which domain model properties are included. In some instances, the UI widget properties are also present in the domain model; these are the main implementation strategies of the design pattern. Additionally, the level of responsibility of the Presenter depends on the distinct MVP variants.

The subsequent sections define the variants of the MVW design pattern family using the GoF definition structure.

3.1.1. Smalltalk-80 MVC

Intent: Smalltalk-80 MVC divides a component into three interconnected sections, effectively addressing distinct application concerns. This division facilitates the management of complex applications that involve user interactions and the representation of data.
Also Known As: MVC.
Motivation: A library management system in which the library and its properties are represented by a service and an interface. The property values are depicted in a table format.
Applicability:

- If View does not require additional attributes, the entire logic can be governed by domain data.
- If change detections can be managed using component-implemented functions rather than the Angular engine.

Structure: See Figure 2.
Participants:

- Model: in a form of a service and an interface or class describing the entity's properties and methods.
- View: UI that is visible to the user and display domain data.
- Controller: responsible for receiving the user input and manipulating domain data.

Collaborations:

- User input is transmitted to the Controller, which manipulates the Model based on the input.
- The Domain model modifies and alters View-displayed properties.
- View displays domain data.

Consequences:

- A straightforward implementation consisting of the required properties and methods.
- The source code is easy to understand and the applications can be thoroughly tested.

Implementation: Model can be implemented in multiple files, but a class representing the domain model and its capabilities is required.

Sample Code: See Code S1 in Supplementary Materials.
Known Uses: Spring Web MVC framework, Smalltalk applications, AngularJS applications.
Related Patterns: HMVC, MVA, MGM.

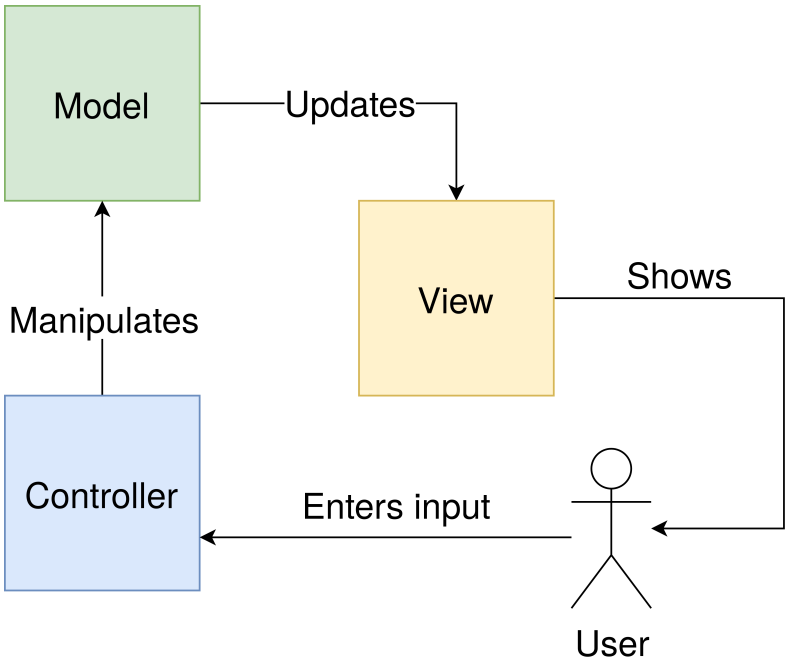

Figure 2. Structure of Smalltalk-80 MVC.

3.1.2. HMVC

Intent: A component is divided into three interconnected sections. This strategic division helps to segregate concerns within a component. HMVC provides a robust framework to manage complex components and modules that involve intricate user interactions and the representation of diverse data.
Also Known As: This pattern has no aliases.
Motivation: A dashboard includes a menu, a timeline, a table, and various charts. The data are already accessible at the dashboard level, but the smaller components (menu, timeline, table) are hierarchically at a lower level, so domain data must be sent to or shared with them. Data are shared via component-based communications, such as input and output directives, between the different subcomponents that use the domain's data.
Applicability: If components start to become difficult to comprehend, it is reasonable to divide them into multiple smaller components by constructing a hierarchy.
Structure: See Figure 3.
Participants:

- Model: a service and interface or class that describes an entity's properties and methods.
- View: UI that is visible to the user and displays domain data.
- Controller: responsible for receiving user input and manipulating domain data. Controllers at a lower level in the hierarchy receive input data and propagate output data. Higher-level controllers that implement entire pages send data to lower-level controllers that implement only a portion of the page and may manipulate a portion of the domain data. The highest-level controller receives data segments and merges them.

Collaborations:

- User input is sent to the Controller, which evaluates it and modifies the Model based on the input.
- Domain model changes and modifies properties that are propagated to sub-controllers and returned to higher-level controllers.
- View displays a portion of the domain data.

Consequences: Implementation is more complicated, whereas source codes and components are simpler to comprehend.
Implementation: It is advised to avoid creating excessively small components because applications can become too fragmented.
Sample Code: See Code S2 in Supplementary Materials.
Known Uses: Angular 2+ applications with @Input() and @Output() directives.
Related Patterns: There are no related patterns.

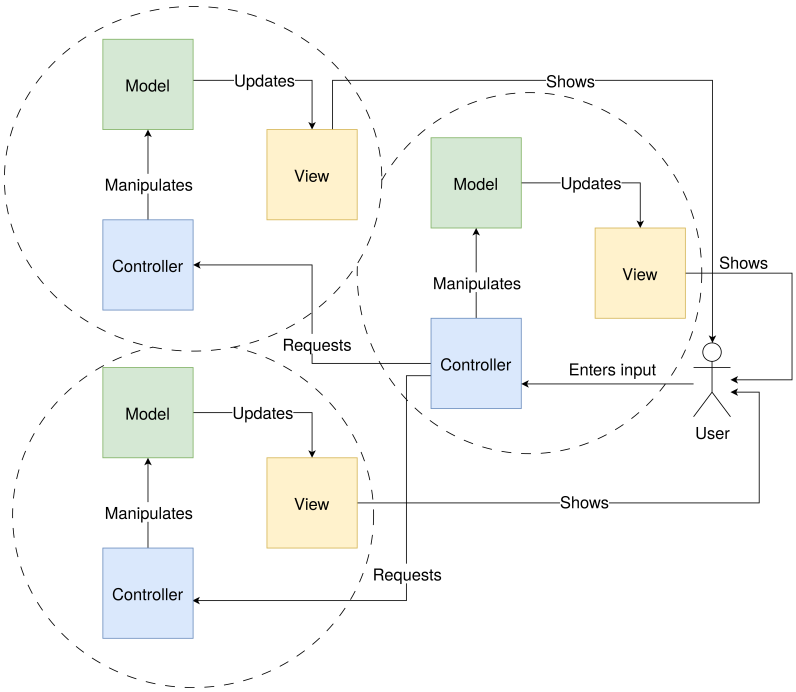

Figure 3. Structure of HMVC.

3.1.3. MVA

Intent: Smalltalk-80 MVC divides a component into three interconnected sections, effectively addressing distinct application concerns. This division facilitates the management of complex applications that involve user interactions and the representation of data.
Also Known As: Model-Mediator-View, Mediated MVC.
Motivation: In an accounting information system, the domain model defines the financial status of the company, whereas the View contains properties that are not explicitly stored but are rather inferred from domain data values. For instance, if the income falls below a certain threshold, the text color changes and a down arrow appears. All page properties are administered by a single controller.
Applicability: Multiple UI elements are served based on the same domain property, but other variables technically refer to the domain property's value.
Structure: See Figure 4.
Participants:

- Model: in the form of a service and interface or class that describes the entity and its properties and methods. It modifies the domain entity properties and inferred properties within the Adapter.
- View: the UI component that displays the domain data and the adapter's properties.
- Adapter: receiving user input and processing domain data.

Collaborations:

- User input is transmitted to the Adapter, which manipulates the Model based on the input.
- The Domain model modifies and alters displayed properties.
- Properties of an adapter are inferred from domain data.

- View displays the domain data and the Adapter's properties.

Consequences:

- Variables and properties are managed by multiple components, resulting in a more complicated implementation.
- Multiple UI module attributes are maintained in a centralized location.

Implementation: It is recommended to alter Adapter properties collectively in the same scope or in separate functions to keep them organized.
Sample Code: See Code S3 in Supplementary Materials.
Known Uses: Applications with multiple UI elements whose appearance is dependent on domain data.
Related Patterns: MGM, Application Model.

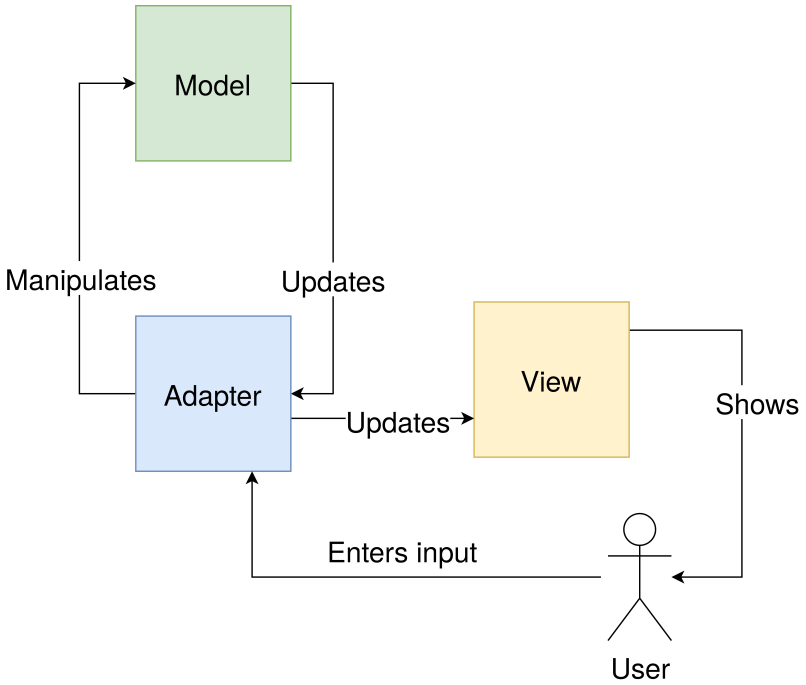

Figure 4. Structure of MVA.

3.1.4. MGM

Intent: In this software development design strategy, the component is divided into four interconnected parts. These sections are linearly connected at endpoints to segregate application concerns efficiently. To enhance the View visualization, domain data are transformed. The management of the distinct properties of UI widgets is assigned to different controllers, ensuring a more specialized and streamlined operation.
Also Known As: This pattern has no aliases.
Motivation: An accounting information system's domain model describes the company's financial status, but the View's properties are inferred from domain data values. If income drops below a certain value, the text color changes and a down arrow appears. Distinct Controllers manage different page properties.
Applicability: In the case of more complex UI widgets (e.g., charts, popups), it may be preferable to manage the Controller as a separate component.
Structure: See Figure 5.
Participants:

- Model: a service and interface or class describing the entity's properties and methods. It modifies the Adapter's domain entity properties and inferred properties.
- GUI: UI that is visible to the user and presents widget components collectively. It is responsible for notifying the Mediator and transmitting user input.

- Mediator: It manages a specific complex GUI and its content via a Model that is only a portion of the entire domain entity and does not include other UI widgets.

Collaborations:

- User input is transmitted to the Mediator, which manipulates Model based on the input.
- The Domain model modifies and alters View-displayed properties.
- Properties of an adapter are inferred from domain data.
- View displays the domain data and Mediator's properties.

Consequences:

- Complex UI widgets are simpler to comprehend and maintain.
- Component is reusable in other sections of the project.

Implementation: Only large and unique components should be implemented with Mediator.
Sample Code: See Code S4 in Supplementary Materials.
Known Uses:

- Ionic framework popup implementations.
- Angular Material mat-table implementation.

Related Patterns: MVA.

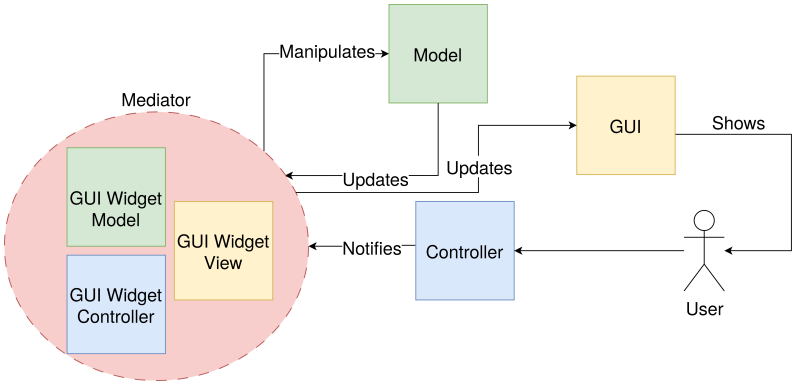

Figure 5. Structure of MGM.

3.1.5. Application Model

Intent: The Application Model design pattern divides a component into four connected sections to implement distinct concerns within an application. The behavioral properties of UI elements are dependent on domain data and are wrapped in an Application Model. For View and Controller and Application Model interaction, data-binding is used.
Also Known As: This pattern has no aliases.
Motivation: In a hotel reservation system, the user can view the rooms' descriptions and ratings. In addition to the rating value, there is a UI widget consisting of five stars. The Application Model manages the star data according to the rating value. The Controller is still capable of updating the View based on domain data. The page's UI elements are all updated based on the same domain model. Through data binding, user input and view content are managed.
Applicability:

- With data binding, data can be sent to the controller and returned to the view without the need for additional function implementations.
- There are distinct View objects created in the business logic that are served by the domain model. These objects are Application Model properties. Additionally, the domain model can be retained and utilized. It has a loose structure overall.

Structure: See Figure 6.
Participants:

- Model: a service and interface or class describing the entity's properties and methods. It updates the Application Model's domain entity properties and objects.
- View: UI that is visible to the user and displays elements containing data.

- Application Model: objects created for the View and supplied data from the domain. It only affects the behavior of UI elements.
- Controller: This component manages user input and receives model updates. It is only applicable to the domain model.

Collaborations:

- User input is transmitted to the Controller, which manipulates the Model based on the input.
- Domain model modifications and updates are returned to the Controller, and the Application Model is also updated.
- Application Model properties are behavioral properties of UI widgets that are inferred from domain data.
- View displays the domain data and Application Model properties.

Consequences:

- Separate logic for the displayed data.
- Difficulty in maintaining domain model and view consistency.

Implementation: Recommended to have a strongly typed Application Model.
Sample Code: See Code S5 in Supplementary Materials.
Known Uses: Angular 2+ and Ionic applications.
Related Patterns: MVA, MGM, Microsoft MVVM.

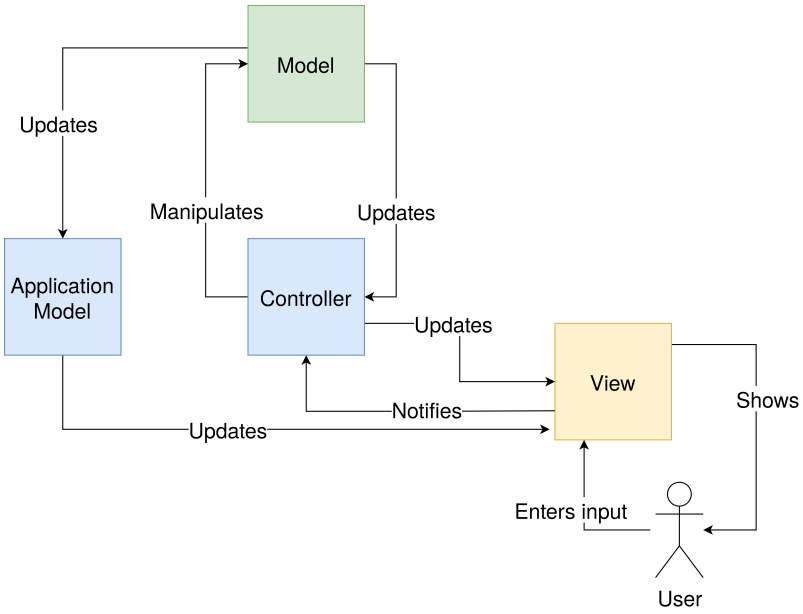

Figure 6. Structure of Application Model.

3.1.6. Microsoft MVVM

Intent: Microsoft MVVM separates a software application into three interdependent sections and maintaining distinct concerns within an application. The behavioral properties of the UI widgets and View contents are dependent on domain data and are encapsulated in a ViewModel. View and ViewModel interaction is managed via data binding.
Also Known As: Presentation Model.
Motivation: In a system for scheduling appointments, domain data are passed to a distinct object that is extended with additional fields, such as ratingColor. The rating data are utilized in both a chart and a table, so a single object can serve multiple UI elements. However, the ViewModel is a copy of the domain data with potential extensions. Through data binding, user input and view content are managed.
Applicability:

- With data binding, data can be sent to the ViewModel and returned to the view without the need for additional function implementations.

- Additional properties required by the view are not maintained in separate objects; rather, the developer is provided with an unstructured and expanded structure.

Structure: See Figure 7.
Participants:

- Model: a service and interface or class defining the entity's properties and methods. It synchronizes the ViewModel.
- View: UI that is visible to the user and displays elements containing data.
- ViewModel: a replica of the domain model with view-specific properties added.

Collaborations:

- Data binding transfers user input to the ViewModel. ViewModel processes it and manipulates Model.
- Domain model modifications are communicated back to the ViewModel.
- ViewModel duplicates the domain entity and extends or transforms the data so that they are suitable for the View. It refreshes the View.

Consequences: Looser ViewModel structure achieved by extending the domain model copy.
Implementation: ViewModel should be a mapping of the domain model to ensure data consistency.
Sample Code: See Code S6 in Supplementary Materials.
Known Uses: Angular 2+ and Ionic applications.
Related Patterns: MVA, MGM, Application Model.

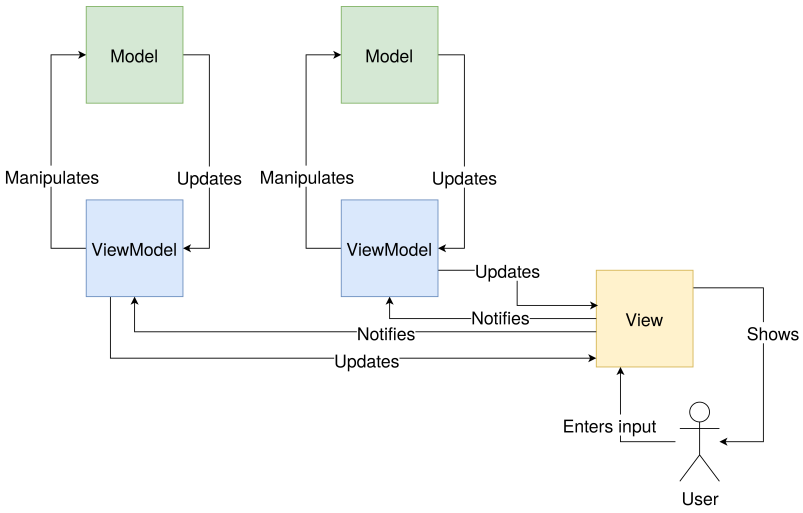

Figure 7. Structure of Microsoft MVVM.

3.1.7. Dolphin Smalltalk MVP

Intent: This design pattern partitions an application into three interconnected sections and maintains distinct concerns within an application. Here, View has less control, and modifications depend on user input and its validation.
Also Known As: MVP.
Motivation: The application form of a hotel scheduling system contains multiple input fields with validators. The status and color of the submit icon are determined by the results of input validations.
Applicability:

- User input must be validated prior to Model modification, and the result must be presented to the user.
- View serves only for presentation purposes; intricate logic is left to the Presenter.

Structure: See Figure 8.
Participants:

- Model: a service and interface or class specifying the entity's properties and methods.

- View: UI that is visible to the user and displays elements containing data.
- Presenter: receives user input, validates it, and alters the Model if the input is legitimate; the View also displays this result.

Collaborations:

- Data binding is used to transmit user input to the Presenter. Presenter processes it and manipulates Model if data are valid.
- Domain model modifications are transmitted back to the Presenter.
- View only displays data and validation results from user input.

Consequences: Presenter has direct access to View and greater control over View.
Implementation: Use FormControls to automatically manage data binding and validation.
Sample Code: See Code S7 in Supplementary Materials.
Known Uses: Angular 2+ and Ionic applications.
Related Patterns: Supervising Controller, Passive View.

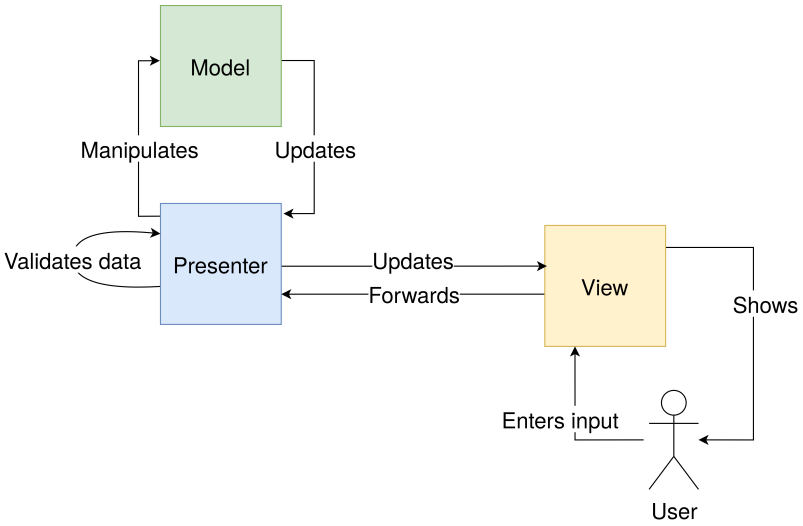

Figure 8. Structure of Dolphin Smalltalk MVP.

3.1.8. Supervising Controller

Intent: Supervising Controller splits an application into three interconnected sections. View is modified based on user input and its validation result. View has less control and Model is not limited to domain data.
Also Known As: Supervising Presenter.
Motivation: The user can customize the displayed UI elements in a chat room application to his or her liking. The colors of elements are saved in the user profile so that it can be reloaded in any environment.
Applicability:

- Domain data may need to be extended with UI behavior that is not dynamically derived from the domain entity's values.
- Application development if review procedures are time-consuming.

Structure: See Figure 9.
Participants:

- Model: a service and interface or class that describes the entity, its properties and methods, and potentially the behavioral properties of the UI elements.
- View: User interface that is visible to the user and displays elements containing data.
- Presenter: receives user input, validates it, and alters the Model if the input is valid; the View also displays this result.

Collaborations:

- User input is forwarded to the Presenter by data binding. Presenter processes it and manipulates Model if data are valid.

- Domain model changes occur and updates are sent back to the Presenter. If there are properties in domain data that refer to other properties, they can be updated immediately.
- View presents data and behavioural properties based on the stored data and the results of the user input validation.

Consequences: Due to the extended domain model, domain data may be overly static.
Implementation: Use objects for each property group that is associated with a UI widget or domain property.
Sample Code: See Code S8 in Supplementary Materials.
Known Uses: Serverless development to ease application updates.
Related Patterns: Dolphin Smalltalk MVP, Passive View.

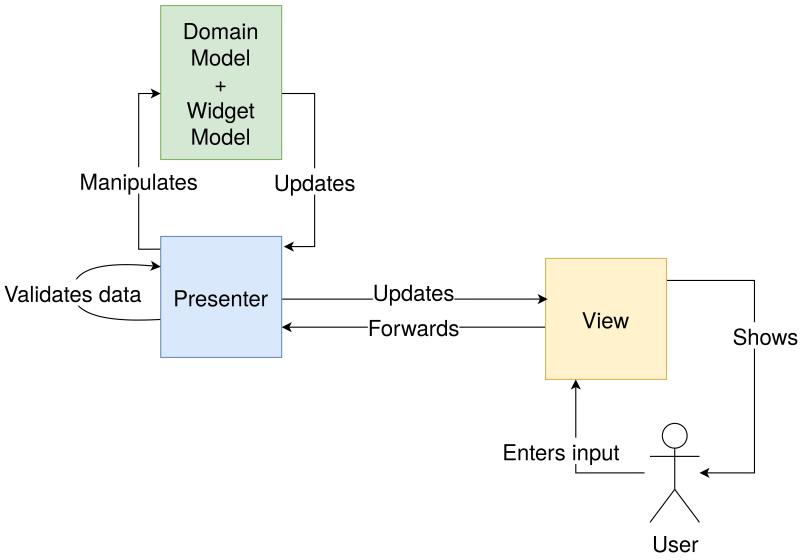

Figure 9. Structure of Supervising Controller.

3.1.9. Passive View

Intent: Passive View divides an application into three interconnected sections and maintains distinct concerns within an application. View has no controller role and UI widgets are display based on the result of user input validation.
Also Known As: Humble View.
Motivation: In an image editor application, the Presenter class implements the mechanism and appearance of the available tools. The user interface is the business logic.
Applicability: The logic and behavior of UI elements are too intricate to implement in the View.
Structure: See Figure 10.
Participants:

- Model: a service and interface or class describing the entity's properties and methods.
- View: UI widget references that must be visible to the user.
- Presenter: receives user input, validates it, and updates Model if input is valid; has complete control over the states and controls of UI widgets.

Collaborations:

- Data binding is used to transmit user input to the Presenter. If data are valid, Presenter processes it and manipulates Model. Moreover, Presenter entirely controls the View elements.
- Domain model modifications are transmitted back to the Presenter.
- View has no control and only a reference to the visible UI element.

Consequences:

- Easy to make spaghetti code.
- Provide increased implementation flexibility for the UI.

Implementation: Use separate classes for complex UI elements to make the source code more maintainable.
Sample Code: See Code S9 in Supplementary Materials.
Known Uses: Implementation of GUI widgets and GUI libraries.
Related Patterns: Dolphin Smalltalk MVP, Supervising Controller.

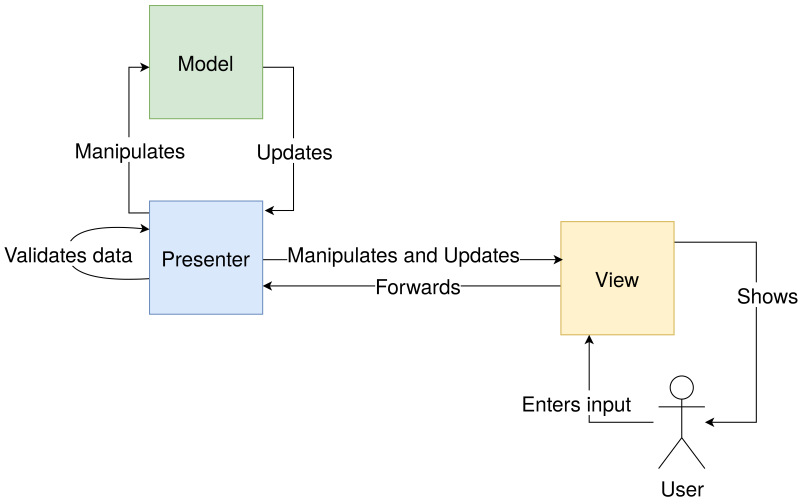

Figure 10. Structure of Passive View.

3.2. Rule-Based Design Pattern Identification in Angular Projects

The manual identification of design patterns is a time-consuming and error-prone process. Some design patterns are readily identifiable, while others are intricate and require complex thought. In addition, human error may result from a superficial examination of the source code, incorrectly used variable and component names, and program codes written in a rarely used spoken language. With the introduction of ChatGPT, LLMs gained tremendous popularity. Due to the fact that GPT models can be programmed using prompts, a new discipline known as prompt engineering arose. It was a major breakthrough to develop a language model that can be used to have topic-independent conversations. In addition to dialogues, we can delegate tasks to it, and if the descriptions are accurate, it returns precise answers within seconds. Training a neural network for the detection of design patterns is not a simple operation; it requires a large number of samples from all classes. Taking into account the MVW design patterns and their subtypes, the task becomes even more challenging. Using the capabilities of GPT models, we developed rule sets for detecting MVW design patterns in Angular projects. These principles define the relationship between the files and the component responsibilities using plain English sentences. Tables A1–A3 in the Appendix A the applicable rules for each design pattern. The R1-R30 rules enable us to refer to a particular rule in the text. In Table A4, we provide an overview of the most essential criteria for determining whether a component applies a particular MVW design pattern. If MVC is used in Angular, data must be passed to components via methods and not data binding. The remaining design patterns send data to the component via data binding. The management of user input can be differentiated between these two approaches. HMVC requires a hierarchical structure, so domain data must also be shared with lower-level components. If a component maintains View-specific attributes, it is either MVA-, MGM-, Application Model-, or Microsoft MVVM-specific. Microsoft MVVM has only a few limitations and offers the finest Angular practices. Validation of user input is a requirement for MVP variants. Moreover, if UI behavioral properties are part of the domain data, the component is deemed to use the Supervising Controller pattern, whereas the Passive View pattern is used if the View has no control.

Due to the fact that an MVW design pattern can only be identified by examining three or four interconnected files, we must locate the connected components. Due to Angular's component-based architecture, the MVW components can be readily separated. Each

Angular component consists of a TypeScript-written component logic, an HTML template, and an interface or class representing the domain entity, typically in the form of a service. To identify the relevant design patterns, we must answer the following questions.

- Q1: What functions do the interconnected files serve?
- Q2: What responsibilities do the interconnected files have?
- Q3: What are the relationships and orientations of file-to-file communication?

If we can determine which file contains the component's implementation, we can readily answer a portion of Q1. The component source must be a TypeScript file with a @Component() annotation and a .ts extension. In the MVW structure, component code denotes Whatever (W). View can be located based on the component's source code by identifying the template or its path in the component's definition. Model can be detected according to the imports and private attributes instantiated in the constructor's argument list. To ascertain what the W in MVW stands for, the model must identify a subset of entirely applied rules. Q2 and Q3 can be resolved by employing the principles' subsets. Rules are completely independent, so there are no dependencies between rules in a subset. A design pattern can only be applied to a component if all of its subset's principles are applicable. It is essential to note that the listed design pattern variations can be combined, and that the application of one design pattern does not preclude the application of another design pattern from the other MVW group.

3.3. MVW Dataset and Evaluation with GPT Models

In our research, we downloaded 18,830 open-source Angular projects from GitHub and chose those with the highest number of forks, which indicates a higher level of project maturity. Obviously, GPT models have limitations for tokens, so we cannot give the model an entire Angular project to identify design patterns. Thus, we must organize the project's content by level of interest. Each project's MVW structure was isolated and analyzed separately. We discovered that some components lack a model, whereas other, more complex components are compatible with multiple models. The dataset analyzed consists of 125 Angular applications. Various GPT models have differing context restrictions. At the time of our research, the gpt-3.5-turbo and gpt-4-0613 models could contain a maximum of 4096 and 8192 tokens, respectively. An Angular component with a template and used service codes contains an average of 5334 tokens, which exceeds the gpt-3.5-turbo token limit. A set of parameters can be used to configure and control the output of GPT models. The models cannot be configured via the ChatGPT interface in the browser, so it is necessary to utilize GPT models via their API. Pricing is determined by the number of tokens composing both the input and output. We sent the source code in a minified form so as to use fewer tokens without sacrificing the efficacy of the models. By adhering to OpenAI's best practices, more than two consecutive whitespaces are removed from the minified code, new lines are substituted with the $\backslash n$ notation, and tabulators are removed so that each source file contains a single line of code. Because the entire file is stored as a single line, single-line comments could be problematic, but the $\backslash n$ notation indicates that the comment has ended. This minification reduces the number of characters by 8% but tokens by an average of 25%. Consequently, the average number of tokens is only 4006; consequently, the majority of components are within the context limit. After minifying the code, the greatest reduction can be observed in Angular templates, averaging 28.33%. Only those MVW components that fit within the context of both GPT models were analyzed in our investigation. Note that the token limit is valid for a request and response pair.

For evaluating the MVW components, we used the Chat Completions API with the following configurations for both models. In accordance with OpenAI's recommendations, we employ two categories of messages: system and user messages, respectively. The system messages can be used to specify the context and provide instructions. In the system message, the source code is provided in minified form while preserving the MVW structure and segregating the various MVW components. The assistant will resolve the requests and remarks contained in the user messages. It is used to provide the provided rule and request

substantiation from the model in order to validate the results. To ensure a straightforward response, we capped the number of tokens in the response at 150. In addition, decisions must be as deterministic as possible, so we set the temperature to 0.0. This variable controls the model's creativity, which is irrelevant because we only care about the facts. Another metric for controlling randomization, Top_p, is used to determine the next token in the response. We used the default value so that we could receive a clear explanation of the decision. In addition, we request that models respond with "no" if the provided source code lacks sufficient information to make a decision. Both the $frequency_penalty$ and $presence_penalty$ parameters are subsequently set to 0.0.

4. Results

During the course of our research, we formulated rule sets for detecting variations in MVW design patterns using GPT models. Here, we strive to respond to the four research questions listed below.

RQ1: How precisely can MVW design pattern variations in Angular applications be detected?

RQ2: Which variation of the MVW design pattern is most prevalent in Angular application development?

RQ3: What design recommendations and best practices can be formulated for Angular applications?

RQ4: What additional opportunities do the results present?

4.1. RQ1: MVW Design Pattern Detection in Angular Projects

After several iterations on our rule sets, we developed a method for instructing LLMs to determine whether a rule can be applied to an Angular component. If all elements of a rule set can be applied to a component, then the component employs the design pattern described by the rule set. First, the gpt-3.5-turbo model was evaluated using the rule sets. The results were quite encouraging, but we discovered that gpt-3.5-turbo cannot identify certain classes. We attempted to fine-tune our rules in order to achieve improved outcomes, but we could have achieved an aggregate average of 78.98%. Numerous false positives were observed for the NO MVW and Microsoft MVVM categories. Figure 11 demonstrates this effect with green bars. Considering the rules listed in Table A1, R16 was the most problematic provision. In many instances, GPT-3.5 did not accurately locate the domain model or confused it with the object defined in the ViewModel. GPT-3.5 identified Microsoft MVVM in approximately 18% of instances. After missing numerous Microsoft MVVM cases and asserting that the component lacks MVW-related indicators, the accuracy of both classes decreased. However, after evaluating the components with GPT-4 and the same configuration parameters, we discovered that the rules adequately characterize the requirements, but that some of them require a model that can comprehend more complex cohesions. This occurred with GPT-3.5 and the Microsoft MVVM rule set. GPT-4 was able to employ the principles, whereas GPT-3.5 committed errors. According to the graph, GPT-4 performs better, or at least at the same level as GPT-3.5. The Dolphin Smalltalk MVP class obtained the lowest level of accuracy, which was 77.8%.

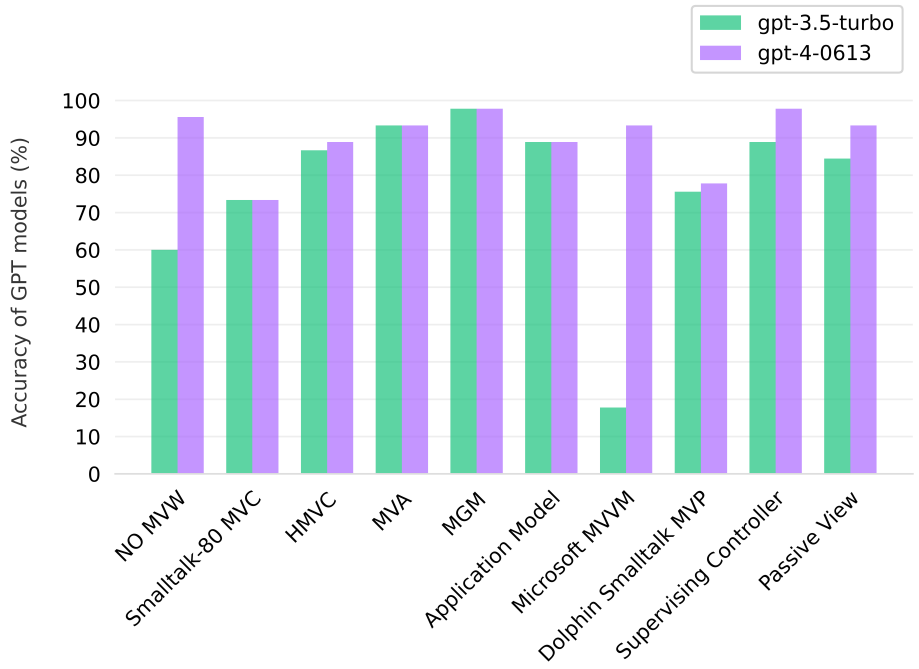

Figure 11. Detection precision of GPT models for MVW design patterns.

R19, R23, and R26, listed in Table A1, generated some confusion because there were components in which the input forms and validators were defined in the Model as functions, but the Presenter invoked the Model function that returned the form and performed the validation in the Presenter. This resulted in a number of false negatives when evaluating MVP variants.

4.2. RQ2: Most Typical MVW Design Pattern in Angular Application Development

In general, Angular is regarded as an MVVM framework, but there are implementations that contradict this. However, MVVM variations are without a doubt the most prevalent, with Microsoft MVVM present in 34.4% of the components. Microsoft MVVM is present in 97.73% of the components that implement any of the aforementioned MVW design patterns. After analyzing 125 mature projects containing 1576 components, we discovered that 64.8% of the components do not conform to any of the enumerated design patterns. In these components, we cannot locate a Model that is related to the other participants, or the component is vacant, so the separation criterion fails at its most fundamental level. Smalltalk-80 MVC is the second most common design pattern, accounting for 11.2% of all components. It represents traditional method-based communication between View and Controller in Angular, and the domain model is not transformed. HMVC and MVP are found in 8% of components and are the third most common design patterns in Angular application development. Application Model is less prevalent in Angular development; only 3.2% of components use it. In Angular, it is extremely uncommon to maintain a distinct model in the component for UI widget behavior. The remaining design patterns only exist in 2% of the components. These results are displayed in Figures 12 and 13. Overall, the Microsoft MVVM design pattern is the most popular among developers. Moreover, we discovered that the same design patterns are typically used in the components of the same project. A particular library typically enforces infrequently used design patterns.

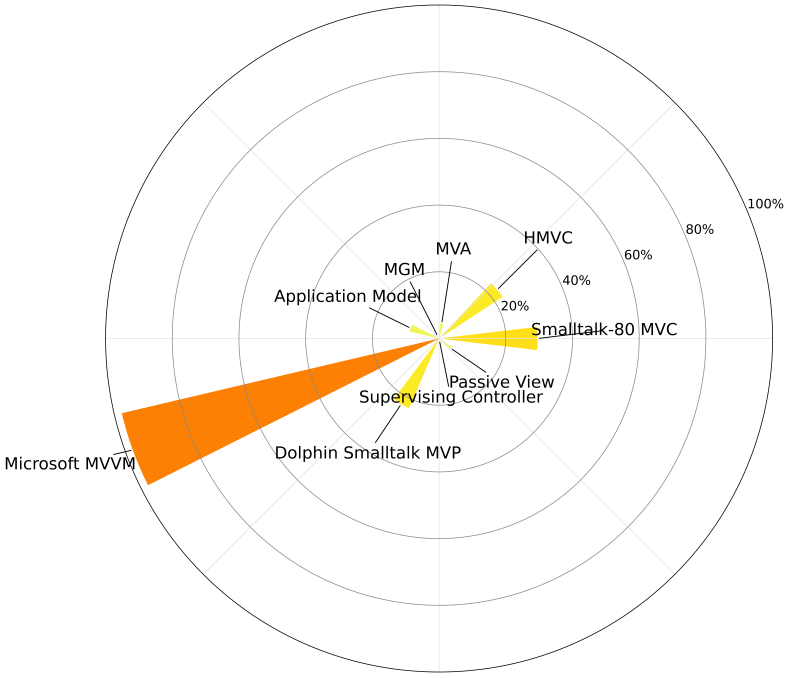

Figure 12. Distribution of MVW design patterns in Angular projects.

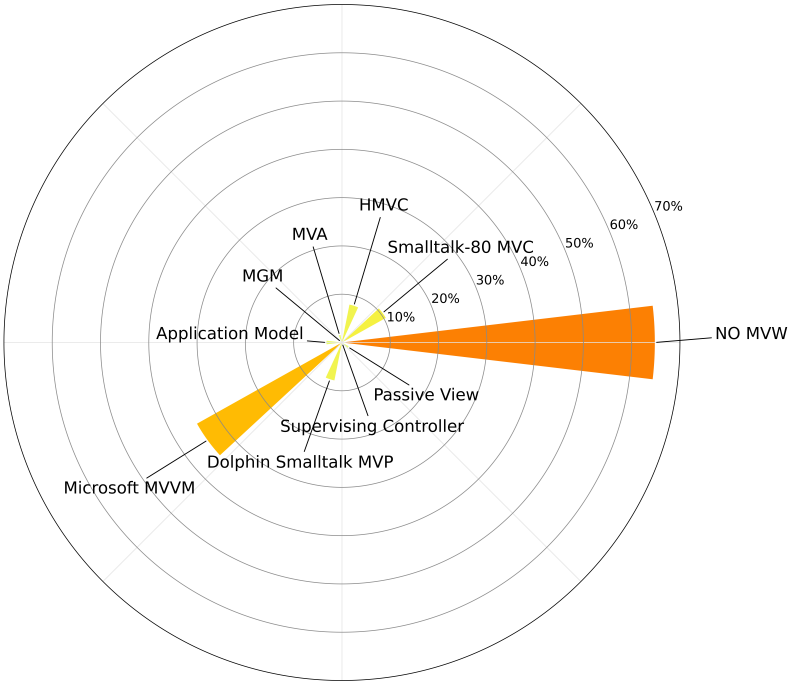

Figure 13. Distribution of MVW design patterns in Angular projects that adhere to the MVW design principles.

4.3. RQ3: Recommendations for Using MVW Design Patterns in Angular Application Development

It is challenging to catalog all implementation variations of how a particular design pattern can be applied to application development. Nevertheless, some design patterns are too specific for a given framework. Here, we provide a list of recommendations for when a

particular design pattern should be used in the development of Angular applications. Due to the fact that our rule sets for detecting MVW design patterns are not example-based but rather descriptive, GPT models identified some code sections that apply a particular design pattern that we overlooked during manual analysis. The sections that follow provide recommendations for using particular MVW design patterns in Angular development. It is essential that specific design patterns do not exclude one another, and in practice, multiple design patterns are typically combined.

4.3.1. Smalltalk-80 MVC

This design pattern is the most fundamental in terms of implementation and component communication. The Angular service should implement the entity's required methods. Only domain-specific attributes and no View-specific properties may be present in the Domain model. The Angular component instantiates the service as a private attribute and transmits the original domain entity to the View. The typical usage scenario involves the use of an async-pipe that presents the domain model's data without component control. When a change event occurs, user input is transmitted to the controller using methods implemented in the component. If additional View logic is required, Smalltalk-80 MVC is not the optimal solution.

4.3.2. HMVC

HMVC is quite similar to Smalltalk-80 MVC, but it is essential to maintain component hierarchy. In Angular, it is advised to divide a page into smaller components if the components are exceedingly large to maintain and may have too much responsibility. Components at a lower level receive the result of a query to the domain model initiated by the component responsible for the entire page. The domain data must be preserved in their original format. Through the use of input and output directives, this hierarchy and communication between two components are managed. User input continues to be managed by methods implemented in lower-level components and delegated back to higher-level components.

4.3.3. MVA

MVA introduces new attributes for managing the properties and behaviors of UI widgets. If the domain model lacks View-specific attributes, MVA can assist in separating the View behavior by adding View-specific attributes to the component. The lifecycle of these attributes is identical to that of their component counterparts. MVA does not require sophisticated objects to maintain the behavior of widgets; only basic component attributes are required. If the behavior of the View, such as color, direction, and size, depends on the value of the domain model, MVA is a viable option. User input must be transmitted directly to the component using change-triggered methods.

4.3.4. MGM

MGM employs a similar strategy, which is beneficial when the UI widget is too intricate to implement in the same component. Complex elements dependent on the domain model, such as charts and calendars, are typically organized into libraries in Angular. In this instance, the UI widget's properties are implemented in the library, and the data must be passed to the library's component. Inferring the behavior of UI widgets occurs within the library. For implementing complex UI libraries in Angular, this design pattern is recommended. User input must be processed via component-defined methods activated by state changes.

4.3.5. Application Model

This design pattern introduces the Application Model as the new fourth component in the MVW structure. It is an object that is exclusive to View. Typically, the data are derived from the domain model, but it is of a distinct class. It is advantageous if the data depicted in the View differ substantially from the data in the domain model. If the database

serves multiple applications, this may occur. In Angular, it is recommended to create a distinct class for display-related data in order to create code that is easily maintained. Here, user input is processed via data binding rather than controller-implemented methods. FormsModule and ReactiveFormsModule are recommended for data binding in Angular.

4.3.6. Microsoft MVVM

This is the most common design pattern in Angular development because the framework's best practices are built using Microsoft MVVM, so developers implicitly employ this design pattern. It is reasonable in a component-based architecture because large pages are broken down into smaller components and View-specific attributes can be managed independently. In Angular, it is recommended that user input management be handled via data binding. If no reasonable alternative design pattern is necessary, it is recommended to stick with this one.

4.3.7. Dolphin Smalltalk MVP

This is a traditional MVP architecture, in which the component serves as the Presenter and connects the Model and View. In the case of form implementation in Angular, it is recommended to use it. The user's input is sent to the component that validates the data and determines whether the input is legitimate. The behavior and visibility of View elements are determined by the validation results, so the Presenter governs the behavior of UI widgets. View is not encumbered with business logic; its sole responsibility is to display data.

4.3.8. Supervising Controller

This design pattern expands the Presenter's control over the View. It implies that the Presenter instructs the View on how to display data and UI elements, as well as what to display. The Model is not limited to the domain entity, as the Presenter may receive information about UI widget behaviors from the domain model. This pattern is advantageous when persisting UI widget settings and appearance. The management of user profiles and preferences is a typical use case. This pattern is also utilized by UI elements incorporated in installable libraries, such as modals. Validation of user input is still required, and data are presented based on the outcome.

4.3.9. Passive View

This pattern gives the Presenter the most control. Although it is uncommon in Angular, it is still possible to implement the entire View or portions of it in component code. This pattern is recommended when the View is extremely complex and contains more logic than View-specific code. A graphical use case, such as using a canvas, necessitates view-specific logic that is simpler and more maintainable to implement within the component. The validation of user input is required, and UI elements are displayed based on the result.

4.4. RQ4: Future Potentials Based on the Results

During the evaluation of our results and answering the research questions, we found that GPT models can not only find MVW design patterns with high accuracy by applying the defined rule sets, but they can also highlight information that humans may miss. This occurred during the validation process for us as well. Currently, using GPT-3.5 and GPT-4 models via API is not free, so we cannot afford to evaluate a larger number of GitHub projects. However, it is evident that our solution does not necessitate training and also functions with a relatively small dataset. Here, we presented a novel methodology of prompt engineering that may give an opportunity to apply this technique in other fields as well. Our approach is a valuable starting point that could be easily improved if the models do not function well on other projects. The models have a firm foundation for perceiving and analyzing diverse coding styles and techniques that are not limited to the Angular framework and any spoken languages, so it is proposed to apply our rule-based

methodology to the evaluation of additional Web applications and framework-specific projects. It is still intended to train a neural network capable of detecting MVW design patterns in Angular projects with high accuracy, but the available dataset is too limited and one-sided, as Microsoft MVVM is the most prevalent design pattern in Angular development. Nonetheless, a Bidirectional Encoder Representations from Transformers (BERT) model was trained. Since we have limited data from numerous classes, we merged them and labeled them based on our taxonomy as NO MVW, MVC, MVVM, and MVP. Our model can detect NO MVW, MVC, MVVM, and MVP with F1-scores of 81%, 40%, 67%, and 80%, respectively, using these four categories. It is evident that there is still a dearth of information in MVC, whereas MVP is readily identifiable. In the future, it is planned to further enhance our rule-based approach, attain a higher level of precision, and expand our dataset using GPT models for automated labeling. It is also intended to generate samples utilizing GPT models and expand the training dataset. In addition, if fine-tuning of GPT models becomes available, we can promptly achieve superior results.

5. Discussion

This investigation effectively demonstrated that Angular is a Microsoft MVVM-dominant framework. However, developers frequently integrate the Microsoft MVVM methodology with other MVW design patterns. According to our evaluations, design pattern usage is not only framework- or library-specific but also heavily influenced by the preferences of developers and development teams. In the components developed by the same team, not only is the use of design patterns but also the organization of source code remarkably similar.

By evaluating our rule sets, we were required to determine the utmost complexity of requests that are still distinct and appropriate for the context. If a rule is too complicated, it must be broken down and simplified. We discovered that it is crucial to employ expressions that cannot affect the model's decision. Due to the model's factual knowledge, it is able to integrate requests with its knowledge and may produce incorrect results. The names of the concrete design patterns have been omitted from the prompts because GPT models are familiar with MVW design pattern variations. The prompt is constructed using the rule that must be applied and the minified source code in order to adhere to the token count restriction. By expanding the context limit of the models, more complicated requests could be sent, and perhaps entire projects could be analyzed.

Considering the relevant literature, it is evident that detecting MVW design patterns is not a simple task. Identifying architectural patterns requires a significant portion of a project, whereas most GoF design patterns can be discovered by analyzing a single class. In the analysis of MVW design patterns in common programming languages such as Java, C#, and C++, we discovered that the detection of design patterns in Angular applications is an unexplored area of research, but all the tools are available to implement and detect different variants of MVW. MVW design patterns are typically managed as three design patterns, MVC, MVVM, and MVP, and the highest-accuracy results were slightly more than 80%. Our attained accuracy of 90% is encouraging, particularly considering that our method does not require a large sample size and can detect not only the MVW categories but also their variants.

6. Study Limitations

There are a few threats to the validity of the results presented in this study. Although the number of involved projects in our analysis may not be representative, the choice of mature projects compensates for this. In addition, we obtained an overall accuracy of 90% on Angular applications by combining the GPT-4 model with our defined rule sets. The 30 presented rules define the MVW design patterns in Angular applications but the rules may not cover all possible implementations. If this is the case, rule sets must be enhanced and expanded. Lastly, it is possible that our recommendations are too Angular-specific.

However, they draw up best practices that could enhance the development process and serve as a gold standard for Angular application development.

7. Conclusions

We developed a taxonomy for MVW design patterns in Angular application development as part of our research. We introduced a novel method for detecting design patterns with LLMs based on a taxonomy. We found that the Microsoft MVVM design pattern is the most prevalent in Angular application development, but the Smalltalk-80 MVC and HMVC design patterns are also quite popular. In the future, it is intended to fine-tune the rule sets and enhance the GPT-4 model's current 90% accuracy. Due to a paucity of samples for specific design patterns, it is more difficult to train a neural network. Nevertheless, by modifying the rule sets, GPT models can analyze a greater number of projects and other types of projects as well.

Supplementary Materials: The following supporting information can be downloaded at: https://www.mdpi.com/article/10.3390/electronics12153364/s1. Each example is contained in a folder bearing the name of the design pattern. Code S1: Smalltalk-80 MVC example in Angular; Code S2: HMVC example in Angular; Code S3: MVA example in Angular; Code S4: MGM example in Angular; Code S5: Application Model example in Angular; Code S6: Microsoft MVVM example in Angular; Code S7: Dolphin Smalltalk MVP example in Angular; Code S8: Supervising Controller example in Angular; Code S9: Passive View example in Angular.

Author Contributions: Conceptualization, Z.R.J. and V.B.; methodology, Z.R.J.; validation, Z.R.J.; writing—original draft preparation, Z.R.J.; writing—review and editing, Z.R.J. and V.B.; visualization, Z.R.J.; supervision, V.B. All authors have read and agreed to the published version of the manuscript.

Funding: The research was supported by the Ministry of Innovation and Technology NRDI Office within the framework of the Artificial Intelligence National Laboratory Program (RRF-2.3.1-21-2022-00004). Project no. TKP2021-NVA-09 has been implemented with the support provided by the Ministry of Innovation and Technology of Hungary from the National Research, Development and Innovation Fund, financed under the TKP2021-NVA funding scheme.

Data Availability Statement: All data were presented and referred in the main text.

Conflicts of Interest: The authors declare no conflicts of interest.

Appendix A

Table A1. Rule sets of MVW design pattern detection in Angular projects.

Design Patterns	Rules
Smalltalk-80 MVC	R1: View (HTML) presents domain data but data can be modified via events (e.g., methods in the component (implemented in TypeScript)) and not via two-way data binding.
	R2: Component file (TypeScript) works with domain data in its original form, there is no transformation of it, and additional attributes cannot be present in the View (HTML).
	R3: View (HTML) properties (e.g., color) are not part of the domain model.
	R4: There are no @Input() and/or @Output() decorators for sending domain data between user-defined components. Also, there are no input and output directives in the View (HTML) as user-defined component selectors.

Table A1. *Cont.*

Design Patterns	Rules
HMVC	R5: Component file (TypeScript) contains @Input() and/or @Output() decorators and domain data are transmitted through component parameters, or in the View (HTML), user-defined component selectors have input and/or output directives.
	R6: Component file (TypeScript) is working with a smaller part of a domain entity; the entire domain entity is not present in the component. Only some properties of the entity are in use.
	R7: Component file (TypeScript) works with domain data in its original form, there is no transformation of it, and additional attributes cannot be present in the View (HTML).
MVA	R8: Domain model data are transformed for a different visualization purpose and the variable containing the transformed value is used in the View (HTML).
	R9: Domain model data transformations are handled in one component that is responsible for an entire page and contains multiple widgets.
MGM	R10: Domain model data are transformed for a different visualization purpose and the variable containing the transformed value is used in the View (HTML).
	R11: Domain model data transformation is handled in a component that is responsible only for a given UI widget.

Table A2. Rule sets of MVW design pattern detection in Angular projects.

Design Patterns	Rules
Application Model	R12: UI widget (HTML) attributes are parts of the Application Model declared in the Component file (TypeScript).
	R13: UI widget (HTML) attributes (e.g., CSS class, color, size, etc.) are set based on the values of the domain model attributes. Only UI widget (HTML) attributes that change the behavior of the content are valid here.
	R14: UI widget (HTML) attributes are not part of the domain model; these properties can be only present in the Component file (TypeScript).
	R15: There are multiple Application Model attributes (declared in the TypeScript code) that refer to the same domain model attribute (e.g., value and color attributes in the Application Model refer to the value in the domain model).
Microsoft MVVM	R16: The Component file (TypeScript) acts as an interface between the domain model and the View (HTML). It should encapsulate any logic needed to convert the domain model data into a form that is suitable for the View (HTML).
	R17: UI widget (HTML) attributes are not part of the domain model; these properties can only be present in the Component file (TypeScript).
	R18: The changes in the ViewModel (Component) attribute values trigger data retrieval from the Model and the View (HTML) is updated through data binding.

Table A3. Rule sets of MVW design pattern detection in Angular projects.

Design Patterns	Rules
Dolphin Smalltalk MVP	R19: Input fields are checked with validators in the component file (TypeScript code), e.g., FormControl value validations.
	R20: UI widget (HTML) appearance (behavior) depends on the result of a validator (e.g., a button is disabled if a validator fails).
	R21: View states (behaviors) are handled in the HTML code, not in the component file, so HTML DOM elements are not accessed explicitly from TypeScript code.
Supervising Controller	R22: Model is maintained only for the domain entity and its attributes; UI widget (HTML) attributes (e.g., colors, validity) are not present in the domain model. R23: Input fields are checked with validators in the component file (TypeScript code), e.g., FormControl value validation.
	R24: UI widget (HTML) appearance (behavior) depends on the result of a validator (e.g., a button is disabled if a validator fails).
	R25: Domain model contains the results of the user input validations, so widget appearance (behavior) explicitly depends on domain model attribute values.
Passive View	R26: Input fields are checked with validators in the component file (TypeScript code), e.g., FormControl value validation.
	R27: UI widget (HTML) appearance (behavior) depends on the result of a validator (e.g., a button is disabled if a validator fails).
	R28: Some UI widgets (HTML DOM elements) are created and/or modified in the component file (TypeScript).
	R29: There are View states (behaviors) that are handled in the component file (TypeScript), so HTML DOM elements may be accessed explicitly from TypeScript code.
	R30: Model is maintained only for the domain entity and its attributes; UI widget (HTML) attributes (e.g., colors, validity) are not present in the domain model.

Table A4. A summary of key considerations for employing MVW design pattern variants in Angular.

Design Patterns	User Data Sent via Methods Triggered by Change Events	User Data Sent via Data-Binding	Domain Data Shared between Compo-nents	View-Specific Attributes Declared within the Component but Not in the Domain Entity	View-Specific Attributes Are Part of the Domain Entity	View-Specific Attributes Refer to the Same Domain Property	View-Specific Attributes Refer to Different Domain Property	Only UI Widget Behavioral Attributes Can Be Declared in the Component	UI Widgets Have Different Controllers	User Input Is Validated and Has an Impact on the Presented Data	View Receives Data from the Component and Is Maintained by It
Smalltalk-80 MVC	X										
HMVC	X		X								
MVA	X			X		X					
MGM	X			X			X		X		
Application Model		X		X		X		X			
Microsoft MVVM		X		X							
Dolphin Smalltalk MVP		X								X	
Supervising Controller		X			X					X	
Passive View		X								X	X

References

1. Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*; Addison-Wesley Professional Computing Series; Pearson Deutschland GmbH: Munchen, Germany, 1998; ISBN 978-0-201-63498-3.
2. Model View Controller History. Available online: https://wiki.c2.com/?ModelViewControllerHistory (accessed on 11 July 2023).
3. Myer, T. *Professional CodeIgniter*, 1st ed.; Wrox: Birmingham, UK, 2008; pp. 7–9.
4. Fowler, M. *Patterns of Enterprise Application Architecture*; Addison-Wesley Professional: Francisco, CA, USA, 2002; pp. 330–332.
5. Krasner, G.E.; Pope, S.T. A Cookbook for Using the Model-View Controller User Interface Paradigm in Smalltalk-80. *JOOP J. Object-Oriented Program.* **1988**, *1*, 26–49.
6. Potel, M. *MVP: Model-View-Presenter the Taligent Programming Model for C++ and Java*; Taligent Inc.: Auburn, CA, USA, 1996.
7. Iordan, A. MVP Architecture and Design Patterns Applied to an Optimal Development of a Soft Used for Shortest Path Problem Study. *Res. Highlights Math. Comput. Sci.* **2023**, *9*, 36–54. [CrossRef]
8. Smith, J. Patterns—WPF Apps with the Model-View-ViewModel Design Pattern. *MSDN Mag.* **2009**, *24*, 135.
9. GUI Architectures. Available online: https://martinfowler.com/eaaDev/uiArchs.html#Model-view-presentermvp (accessed on 12 July 2023).
10. Syromiatnikov, A.; Weyns, D. A Journey through the Land of Model-View-Design Patterns. In Proceedings of the IEEE/IFIP Conference on Software Architecture, Sydney, NSW, Australia, 7–11 April 2014; pp. 21–30. [CrossRef]
11. Indrawan, D.; Kusumo, D.; Puspitasari, S. Analysis of the Implementation of MVVM Architecture Pattern on Performance of iOS Mobile-based Applications. *J. Ilm. Penelit. Dan Pembelajaran Inform. (JIPI)* **2023**, *8*, 59–65. [CrossRef]
12. Sampayo-Rodriguez, C.J.; González-Ambriz, R.; Gonzálczmartinez, B.A.; Aldana-Herrera, J. Processor and memory performance with design patterns in a native Android application. *J. Appl. Comput.* **2022** *6*, 53–61.
13. García, R.F. MVVM: Model–View–ViewModel. In *iOS Architecture Patterns*; Apress: Berkeley, CA, USA, 2023. [CrossRef]
14. Epiloksa, H.A.; Kusumo, D.S.; Adrian, M. Effect Of MVVM Architecture Pattern on Android Based Application Performance. *J. Media Inform. Budidarma* **2022**, *6*, 1949–1955. [CrossRef]
15. Forte, L. Building a Modern Web Application Using an MVC Framework. Bachelor's Thesis, Oulu University of Applied Sciences, Degree Programme in Business Information Technology, Oulu, Finland, 2016.
16. Badurowicz, M. MVC architectural pattern in mobile web applications. *Actual Probl. Econ.* **2011**, *6*, 305–309.
17. GPT-4 Technical Report, OpenAI Blog. 2023. Available online: https://cdn.openai.com/papers/gpt-4.pdf (accessed on 12 July 2023).
18. Radford, A.; Narasimhan, K.; Salimans, T.; Sutskever, I. Improving Language Understanding by Generative Pre-Training, OpenAI Blog. 2018. Available online: https://s3-us-west-2.amazonaws.com/openai-assets/research-covers/language-unsupervised/language_understanding_paper.pdf (accessed on 12 July 2023).
19. Roumeliotis, K.I.; Tselikas, N.D. ChatGPT and Open-AI Models: A Preliminary Review. *Future Internet* **2023**, *15*, 192. [CrossRef]
20. Koirala, S. Comparison of Architecture Patterns MVP(SC), MVP(PV), PM, MVVM and MVC. 2010. Available online: http://www.codeproject.com/Articles/66585/Comparison-of-Architecture-presentation-patterns-M (accessed on 12 July 2023).
21. Nazar, N.; Aleti, A.; Zheng, Y. Feature-Based Software Design Pattern Detection. *J. Syst. Softw.* **2020**, *185*, 111179. [CrossRef]
22. Wang, L.; Song, T.; Song, H.-N.; Zhang, S. Research on Design Pattern Detection Method Based on UML Model with Extended Image Information and Deep Learning. *Appl. Sci.* **2022**, *12*, 8718. [CrossRef]
23. Nord, R.; Kurtz, Z. Using Machine Learning to Detect Design Patterns, Carnegie Mellon University, Software Engineering Institute's Insights (blog). March, 2020. Available online: https://insights.sei.cmu.edu/blog/using-machine-learning-to-detect-design-patterns/ (accessed on 11 July 2023).
24. Dobrean, D.; Diosan, D. A Hybrid Approach to MVC Architectural Layers Analysis. In Proceedings of the 16th International Conference on Evluation of Novel Approaches to Software Engineering (ENASE 2021), Online, 26–27 April 2021; pp. 36–46.
25. Komolov, S.; Dlamini, G.; Megha, S.; Mazzara, M. Towards Predicting Architectural Design Patterns: A Machine Learning Approach. *Computers* **2022**, *11*, 151. [CrossRef]
26. Model-View-Controller. Available online: https://developer.apple.com/library/archive/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html (accessed on 11 July 2023).
27. Li, Y.; Jing, W. Research on Integrated Management System of Physical Education Course Information based on Spring MVC Framework. In Proceedings of the International Conference on Information System, Computing and Educational Technology (ICISCET), Montreal, QC, Canada, 23–25 May 2022; pp. 121–124. [CrossRef]
28. Model-View-Adapter (MVA, Mediated MVC, Model-Mediator-View). Available online: https://stefanoborini.com/book-modelviewcontroller/02-mvc-variations/05-variations-on-the-triad/01-model-view-adapter.html (accessed on 11 July 2023).
29. Bulka, A. Model GUI Mediator. 2001. Available online: https://www.atug.com/andypatterns/AndyBulkaModelGuiMediatorPattern.pdf (accessed on 12 July 2023).
30. Hopkins, T.; Horan, B. *Smalltalk: An Introduction to Application Development Using VisualWorks*; Prentice Hall International (UK) Ltd.: London, UK, 1995.