MVVM Revisited: Exploring Design Variants of the Model-View-ViewModel Pattern

Mario Fuksa, Sandro Speth, and Steffen Becker

Institute of Software Engineering
University of Stuttgart, Universitätsstraße 38, 70569 Stuttgart, Germany
[firstname.lastname]@iste.uni-stuttgart.de

Abstract. Many enterprise software systems provide complex Graphical User Interfaces (GUIs) that need robust architectural patterns for well-structured software design. However, popular GUI architectural patterns like Model-View-ViewModel (MVVM) often lack detailed implementation guidance, leading GUI developers to inappropriately use the pattern without a comprehensive overview of design variants and often-mentioned trade-offs. Therefore, this paper presents an extensive review of MVVM design aspects and trade-offs, extending beyond the standard MVVM definition. We conducted a multivocal literature review (MLR), including white and gray literature, to cover essential knowledge from blogs, published papers, and other unpublished formats like books. Using the standard MVVM definition as a baseline, our study identifies (1) 76 additional design constructs grouped into 29 design aspects and (2) 16 additional benefits and 15 additional drawbacks. These insights can guide enterprise application developers in implementing practical MVVM solutions and enable informed design decisions.

Keywords: Model-View-ViewModel · MVVM · Graphical User Interface (GUI) · GUI Architectural Pattern.

1 Introduction

Graphical User Interface (GUI) architectural patterns like Model-View-Controller (MVC), Model-View-Presenter (MVP), or Model-View-ViewModel (MVVM) play a central role when building robust and complex GUIs for enterprise applications. Many developers use the MVVM pattern, which promises high testability and helps to decouple the GUI from the business logic. While Microsoft originally introduced the pattern for the Windows Presentation Foundation (WPF) application development, in recent years, the pattern has also gained more prominence for mobile development [16]. For example, ViewModels are part of the suggested architecture for Android apps [14], while it is also popular in iOS development [11]. The origin of the MVVM pattern is often defined in Martin Fowlers *PresentationModel*, which describes the idea of separating the presentation state from the View in a dedicated observable data-structure and aims for a Humble View [7, 8].

However, while MVVM is prominently used, it is a set of a few guidelines, and standard MVVM definitions leave many design decisions open. For instance, MVVM does not specify how to structure the GUI at the dialog level [6]. Many developers have their interpretation of the pattern and use specific variants in their implementations. This comes with architectural risks: (1) developers select certain MVVM implementations without having an overview of which design alternatives they could consider. (2) MVVM has implicit trade-offs, which developers often do not know in advance.

While significant research exists on MVC and various GUI architectural patterns, no comprehensive literature study explores design variants and additional trade-offs regarding MVVM. Specifically, gray literature and books often contain essential aspects about the usage of MVVM, which has not been covered by white literature so far. The lack of a systematic review that integrates diverse sources leaves a critical void in the literature. Therefore, many developers and researchers might miss a complete overview of MVVM.

To fill this gap, this paper presents a Multivocal Literature Review (MLR), including a qualitative analysis of a broad amount of white and gray literature. The MLR focuses on the conceptual level of the MVVM pattern and does not analyze specific GUI framework implementation details since, in our perspective, GUI frameworks do not implement or even enforce a specific MVVM design variant. Therefore, we guide the MLR by the two research questions:

RQ1: *Which design variants do developers use when implementing MVVM?*

RQ2: *Which trade-offs do developers mention when applying MVVM?*

As a result, we extracted 76 additional *design constructs*, 16 additional *benefits*, and 15 additional *drawbacks*, which go beyond the MVVM standard definition. We synthesized 29 *design aspects* to categorize those design constructs. Therefore, this paper gives an overview of MVVM design variants and trade-offs to help developers make informed decisions when implementing MVVM.

The paper's remainder is structured as follows: Section 2 describes the MVVM standard definition and trade-offs. Section 3 outlines the MLR process. Section 4 discusses the results. Section 5 details design variants. Section 6 handles threats to validity. Section 7 covers related work. Section 8 concludes the paper.

2 Standard Definition of Model-View-ViewModel

A central element in our MLR is a *standard definition* about MVVM, which we use as the baseline to identify design deviations, extensions, or additional trade-offs. Our standard definition relies mainly on the definition and trade-offs that John Gossman originally introduced in Microsoft blog posts [16,17]. Additionally, we regard an often cited definition of Josh Smith on a Microsoft blog post and two further official documentation sites of Microsoft about MVVM [25, 26, 33].

MVVM is a GUI architectural pattern derived from MVC, where the *ViewModel* replaces the controller and uses a general data-binding mechanism. It specializes Fowler's *PresentationModel* [7]. Figure 1 shows the three components:

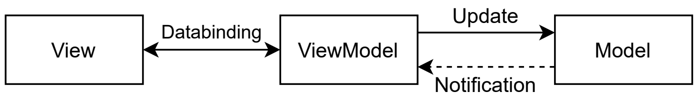


Fig. 1. The Standard Model-View-ViewModel Architectural Pattern.

Model: As in MVC, the *Model* contains the data and business logic completely independent of the GUI. The concrete design of Model classes has almost nothing to do specifically with the MVVM pattern.
View: Also similar to MVC, the *View* consists of visual elements (like buttons, windows, or graphics) and uses one-way[1] or two-way[2] data-binding to ViewModel fields to obtain information to visualize. The View can be data-bound directly to Model elements or by further elements defined by the ViewModels.
ViewModel: The ViewModel handles the *presentation logic* like data transformation, acts as the "Model for the View", and provides information by data-binding. It exposes Commands that the View can use to interact with the Model. ViewModels might contain (sole or extending) validation logic. Further, the *ViewModelLocator* pattern helps to instantiate and locate ViewModel instances.
Relationships: In MVVM, the View knows the ViewModel, and the ViewModel knows the Model. The Model is unaware of the View and the ViewModel, while the ViewModel is unaware of the View.

Standard benefits of MVVM are that ViewModels provide an abstraction of the View and an easier way to unit-test presentation logic. The components (View, ViewModel, Model) are decoupled from each other, supporting developers to swap, create, or maintain more easily. It can reduce boilerplate code in the View while providing good data-binding performance. Further, a developer-designer workflow helps the development team create robust ViewModels, while a design team can focus on user-friendly View designs. Additionally, it cleanly separates the application's business logic and presentation logic.

Standard drawbacks of MVVM are the complexity for simple GUIs, challenges in designing ViewModels up-front in bigger cases, harder debugging of declarative data bindings, and increased memory consumption by binding overhead.

3 Methodology

This section describes the applied MLR process in which we conducted a qualitative analysis of MVVM-focused sources. We used the MLR process of Garousi et al. [13] to follow a structured process to review gray and published literature and extract information to answer our research questions. Figure 2 shows an overview of our specific process involving data entities and activities. We separated into the planning of the MLR, a search process, an attribute/classification design, a data extraction process, and a data synthesis. We provide a replication package[3] which transparently shows the results of each step, like the initial

[1] E.g., updates on a ViewModel's field are automatically reflected to a View's textbox
[2] E.g., in addition to one-way binding, modifications on the View's textbox are automatically reflected to the ViewModel's field
[3] `https://doi.org/10.5281/zenodo.13350488`

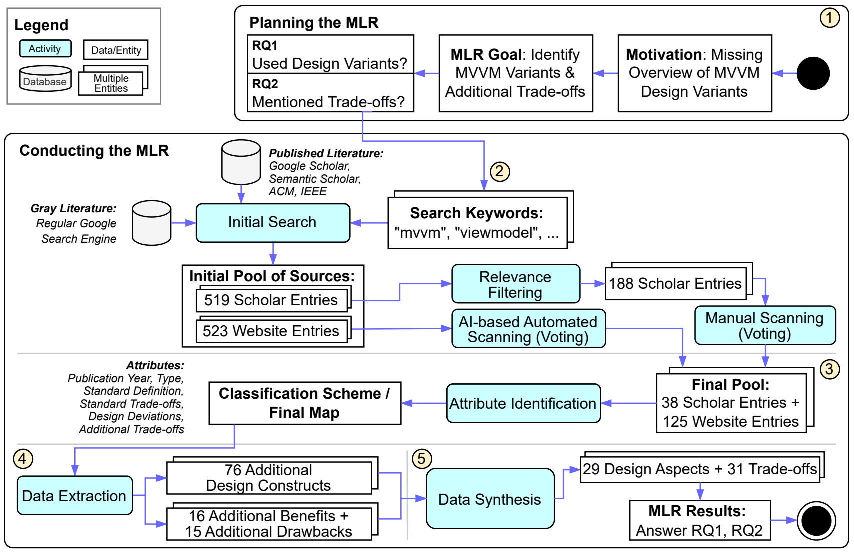

Legend
Activity
Data/Entity
Database
Multiple Entities
Planning the MLR
1
RQ1
Used Design Variants?
RQ2
Mentioned Trade-offs?
MLR Goal: Identify MVVM Variants & Additional Trade-offs
Motivation: Missing Overview of MVVM Design Variants
Conducting the MLR
Published Literature: Google Scholar, Semantic Scholar, ACM, IEEE
2
Gray Literature: Regular Google Search Engine
Initial Search
Search Keywords: "mvvm", "viewmodel", ...
Initial Pool of Sources:
519 Scholar Entries
523 Website Entries
Relevance Filtering
188 Scholar Entries
AI-based Automated Scanning (Voting)
Manual Scanning (Voting)
3
Attributes: Publication Year, Type, Standard Definition, Standard Trade-offs, Design Deviations, Additional Trade-offs
Classification Scheme / Final Map
Attribute Identification
Final Pool: 38 Scholar Entries + 125 Website Entries
4
Data Extraction
76 Additional Design Constructs
16 Additional Benefits + 15 Additional Drawbacks
5
Data Synthesis
29 Design Aspects + 31 Trade-offs
MLR Results: Answer RQ1, RQ2

Fig. 2. Overview of the Applied Multivocal Literature Review (based on Garousi [13]).

search, the attribute scheme with all identified design constructs and trade-offs, or the final data synthesis results. We also included scripts to semi-automate most steps, e.g., to check for duplicate or unrelated search results.

① *Planning the MLR*: First, we planned the MLR and included our motivation for the missing overview of MVVM design variants that people apply in practice. The goal is to identify essential variants of MVVM, which might cover aspects not mentioned in the standard definition. The outcomes of the planning phase are the two research questions that guide our MLR.

② *Search Process*: The search process covers the initial search for gray and published literature, filtering, and voting. First, we defined relevant keywords: "mvvm", "model-view-viewmodel", "viewmodel", and "view model". We varied the combination of keywords depending on the possible search options of the databases. We used the regular Google search engine to find gray literature and multiple databases to find published literature, such as white papers and books. We utilized the tool *Publish or Perish*[4] to search in Google Scholar and Semantic Scholar mainly by title keywords. Further, we did a dedicated search in the ACM and IEEE databases to complement relevant white papers that do not directly contain the keywords' titles. The searches were up-to-date until the beginning of 2024, resulting in 519 scholar entries and 523 website entries.

Next, we filtered and voted on the entries. We first focused on scholar entries and filtered out several entries by exclusion criteria shown in Table 1, which

[4] `https://harzing.com/resources/publish-or-perish`

Table 1. Exclusion Criteria of Scholar Entries.

Criteria	Notes
Not-English	exclude if not written in English
Duplicates	exclude any duplicate entry
Unfocused	exclude if not focusing on MVVM definitions

reduced the number of scholar entries to 188. Since scholar entries allowed us to scan efficiently based on titles, abstracts, and their typical scientific structure, we manually voted them for relevance. Here, we also scanned MVVM definitions if they contain no definition, only standard definition constructs, or if they potentially describe significant design constructs or additional trade-offs. For example, we reject papers that only use MVVM as an implementation detail without a clear MVVM definition. This voting resulted in 38 final scholar entries.

We used another voting approach for websites since we cannot easily filter them. Leveraging the capabilities of ChatGPT (using GPT-4), we used AI-based automated voting. The motivation derives from the lack of standardized website structures, which makes it difficult to scan and filter non-relevant entries without reading each website completely. First, we manually read 20 pivot websites and classified them. We then iteratively improved a prompt for ChatGPT, including our criteria, default definition, and trade-offs, until the pivot websites were classified as expected. We finally used chunks of five URLs and let ChatGPT process the voting. As a result, the AI classified the number of websites into the categories “Standard Definition”, “Extended Definition”, “Extended Trade-offs”, or “No Definition”. The outcome of this voting is 125 website entries.

In our MLR, we focused on the View/ViewModel-specific aspects of MVVM. We largely filtered out design constructs for the Model layer since they are usually not MVVM-specific, i.e., they also apply to MVC and MVP.
③ *Attribute/Classification Design*: We identified relevant attributes as a classification scheme based on the final pool of 38 scholar entries and 125 websites containing potentially significant data. As meta-data, we are interested in the publication year and type (i.e., personal or professional articles, forum discussions, white papers, technical reports, or books). Qualitatively, we are interested in the attribute if a source aligns with the MVVM standard definition or standard benefits/drawbacks. Besides this, we also classified design extensions and additional trade-offs, which extend the standard constructs. To structure results, we then developed a helper language using JetBrains MPS (see our replication package), which prepares the structure of the classification scheme.
④ *Data Extraction*: In the data extraction phase, we analyzed the sources' data to identify relevant design constructs and trade-offs that align with our classification scheme. This qualitative analysis carefully reviewed each voted entry using the prepared classification scheme. Not every entry contained relevant design constructs; e.g., several websites voted by ChatGPT aligned more or less with the baseline standard definition.

Figure 3 overviews the found design constructs and their occurrences across gray and white literature types. We combine similar *design constructs* into eleven

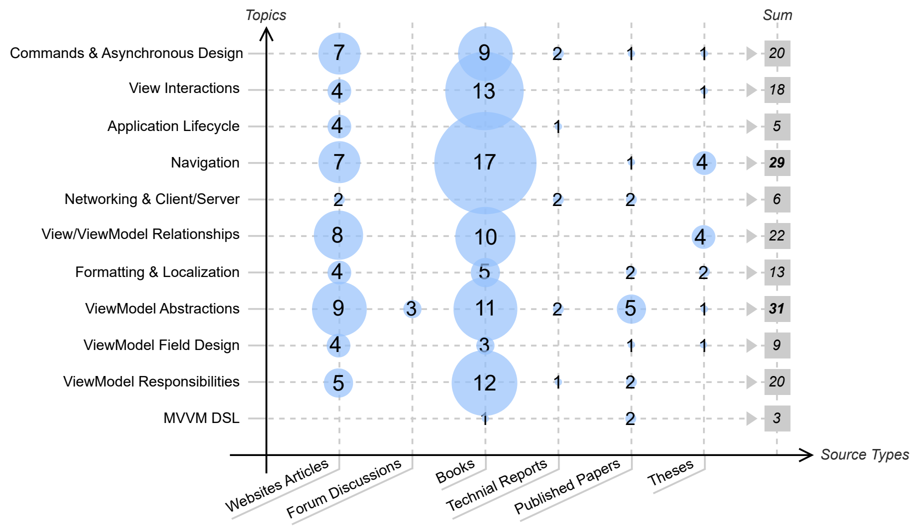


Fig. 3. Distribution of 76 Design Constructs Grouped into Topics.

topics (e.g., formatting and localization) to consume the figure more easily. The overview shows that most constructs are covered by books, followed by website articles. Further, it highlights that the most referenced topics are navigation (e.g., how one ViewModel navigates to another View/ViewModel) and various ViewModel abstraction constructs (e.g., humble View vs. reusable ViewModels). Therefore, we describe those two topics in Subsections 5.1 and 5.2 in more detail and briefly examine the further topics in the joint Subsection 5.3. Finally, we extracted 76 additional design constructs, 16 additional benefits, and 15 additional drawbacks compared to the MVVM standard definition. We discuss a subset of the extracted data in more detail in Section 4. The replication package provides the full overview, including details on their occurrences and explanations.
⑤ *Data Synthesis*: We performed a data synthesis from the extracted data to answer RQ1 and RQ2. For RQ1, we classified the 76 design constructs (e.g., *View has Many ViewModels*) into 29 design aspects (e.g., *View/ViewModel Relationships*) and selected aspects of our particular interest to describe in Section 5. For RQ2, we processed 16 additional benefits and 15 drawbacks. In the next section, we explicitly answer the two research questions as part of the data synthesis, including a discussion of the results.

4 Discussion and Results

This section discusses the results of the MVVM MLR by addressing the two research questions *RQ1* (used MVVM design variants) and *RQ2* (mentioned MVVM trade-offs). We highlight key findings and practical takeaways for enterprise application developers.

MVVM Design Variants (RQ1) The MLR identified 76 additional design constructs grouped into 29 design aspects, representing variants of the standard MVVM definitions. Due to the breadth of design constructs, we further organize the 29 design aspects into eleven topics, as illustrated in Figure 4. We focus on the two most referenced: The *ViewModel abstractions* topic includes twelve constructs in five design aspects (*application structure*, *coupling*, *design*, *humble/reusable*, *model wrapper*), mentioned 31 times (see Subsection 5.1). The *navigation* topic includes eight constructs in three design aspects (*composition*, *responsibility*, *view-based*) mentioned 29 times (see Subsection 5.2). Further topics include command design, view interactions, lifecycle aspects, networking, View/ViewModel relationships, formatting/localization, ViewModel field design and responsibilities, or using an MVVM Domain Specific Language (DSL) in Subsection 5.3. We also synthesized relations between design constructs and standard MVVM:

- *Restricting Constructs*: Nine constructs restrict design rules addressed by the MVVM standard definition. For example, while the standard definition does not limit the cardinalities between View and ViewModel, the construct *View has One ViewModel* does.
- *Extending Constructs*: 43 constructs extend the MVVM standard definition by addressing unmentioned aspects. For instance, *Model-View-Presenter-ViewModel* and *MVVM/Controller* handle the modularization of the ViewModel, addressing the often-mentioned drawback of the ViewModel growth due to many responsibilities without proper modularization.
- *Implementing Constructs*: Twelve constructs provide concrete implementations for standard MVVM aspects. For example, standard definitions mention "asynchronous operations" but lack guidance for handling asynchronous data bindings not firing on the *GUI thread*. Constructs like *Asynchronous Results Handling by Mediator* address this using the mediator pattern.
- *Confirming Constructs*: Seven constructs confirm and clarify standard MVVM "tips" or intentions by providing concrete examples. For instance, the *Coloring in ViewModel* construct confirms the responsibility of data conversion in the ViewModel for color formatting.
- *Divergent Constructs*: Four constructs contradict MVVM standard "tips" or intentions. For example, *Coloring in View* contradicts the intention of placing data conversion logic into the ViewModel by describing how the View can implement the responsibility of formatting colors instead.

These design constructs provide a valuable framework for implementing the MVVM pattern, enabling developers to make well-informed design decisions.

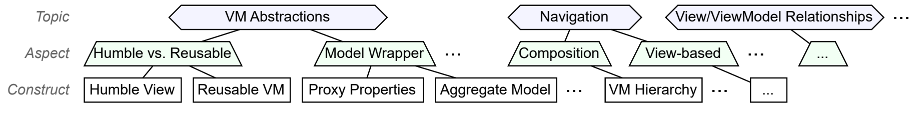


Fig. 4. Design Constructs Hierarchy: Topic ⊃ Aspect ⊃ Construct.

RQ1 Key Takeaways:
The MVVM standard definitions stay vague on crucial design aspects, significantly impacting implementations. We identified design constructs that restrict MVVM rules for specialized design variants, address aspects not covered by standard MVVM, provide concrete implementation guidelines, and confirm or diverge from standard MVVM intentions.

MVVM Trade-offs (RQ2) The MLR identified 16 additional MVVM benefits and 15 additional MVVM drawbacks (Table 2), which the standard MVVM definition does not mention. We highlight the three most cited benefits and drawbacks here.
Benefits: First, eight sources mention the benefit that MVVM supports easier reuse of components like the ViewModel [12]. This is especially beneficial if a ViewModel can be reused for multiple Views. Second, it is stated in five sources (including multiple empirical studies) on mobile applications that MVVM can lead to a better application performance [39]. Third, four sources mention that MVVM achieves a higher decoupling of View and ViewModel since the ViewModel usually does not know the View. In the so-called "Pure MVVM", the decoupling is further increased since the View obtains a ViewModel instance without knowing its concrete type, and data binds to its fields dynamically [37].
Drawbacks: Twelve sources state that the ViewModel usually has too many responsibilities. Inexperienced developers, in particular, might place too many responsibilities into the ViewModel without considering modularization. The unclear definition of MVVM could be a possible reason [12]. Second, seven sources discuss the high learning curve, which can hinder developers from applying the MVVM pattern efficiently [1, 11, 37]. Third, seven sources mention that MVVM involves substantial boilerplate code, mainly if weak tooling support is used and glue code for data-binding has to be written manually [37].

RQ2 Key Takeaways:
Applying the MVVM pattern yields many benefits and drawbacks that developers might not know explicitly. Understanding additional trade-offs can help evaluate the pattern or its design variants more effectively and support making informed decisions when implementing MVVM.

These results highlight the importance of understanding the various design constructs and trade-offs associated with the MVVM pattern. By leveraging the additional insights from our MLR, enterprise application developers can make informed decisions and tailor their implementations to suit specific project needs better while avoiding common pitfalls.

5 Design Aspects

This section discusses the resulting design aspects of the MLR. We selected design constructs of particular interest, which we discuss in a bit more detail.

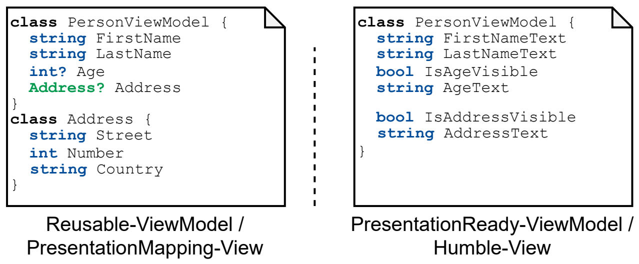
Table 2. Additional Benefits and Drawbacks of MVVM (with Occurrences Number).

Benefit	No.	Drawback	No.
Easier Reuse of Components	8	Many Responsibilities in ViewModel	12
Better Performance vs. MVC/MVP	5	Lot of Boilerplate	7
Increased Decoupling	4	High Learning Curve	7
Less Boilerplate by Library	3	Difficult Testability	3
UI Requirements Quickly Adapted	3	Developer-Designer Workflow Issues	2
View Easily Replaced/Extended	3	Lack of Pattern Guidance	2
N-Tier: Incr. Security/Performance	2	Async Fetching/Threading	2
Different UI Technologies	2	Poor Reusability	1
Development Speed Increased	2	More Classes/Components	1
Easier to Cache View-state	2	Repeated Code in ViewModels	1
Easier Debugging	1	UI-Framework Features Testability	1
Less Imperative Code	1	Complex User Interactions Impl.	1
Well-organized Design	1	3rd-Party Library Issues	1
Reduced Energy Consumption	1	Command Impl. Overhead	1
Reduced CPU Usage	1	Higher CPU Consumption	1
Easier to Maintain Lifecycle	1		

5.1 ViewModel Abstractions

This subsection discusses design constructs that focus on designing ViewModel abstractions. The varying ViewModel abstractions significantly impact the implementation of the MVVM. Therefore, we discuss them in more detail.

Reusable ViewModel vs. Humble View There are two alternatives on how strictly a ViewModel is oriented to a specific View. The first alternative focuses on flexibility and reusability across different Views (i.e., different information formats). The second alternative defines a ViewModel supporting a Humble View to maximize testability and GUI framework exchangeability. Figure 5 illustrates the distinction with a simple example of a `PersonViewModel`: a reusable ViewModel vs. a Humble View design. Both alternatives have different impacts on reusability and testability.



```
class PersonViewModel {
  string FirstName
  string LastName
  int? Age
  Address? Address
}
class Address {
  string Street
  int Number
  string Country
}
```

```
class PersonViewModel {
  string FirstNameText
  string LastNameText
  bool IsAgeVisible
  string AgeText

  bool IsAddressVisible
  string AddressText
}
```

Reusable-ViewModel / PresentationMapping-View
PresentationReady-ViewModel / Humble-View

Fig. 5. Reusable ViewModel vs. Humble View.

Reusable-ViewModel/PresentationMapping-View: The first alternative defines reusable ViewModels, which can be used in multiple Views in a many-to-one relationship (*ViewModel 1:n View*). The ViewModel knows as little as possible about the specific View details and provides abstract data the View can consume. This implies that the presentation mapping of the ViewModel data to certain GUI widget features (e.g., textbox visibility) is the responsibility of each View. Therefore, unit testing reusable ViewModels does not cover presentation mapping logic placed in the View [19]. To fully cover the full presentation logic, the View also needs to be tested. For example, a `PersonViewModel` provides more generic information like age information or an optional Address object, which the View needs to map to boolean or string representations.

PresentationReady-ViewModel/Humble-View: The second alternative defines View-specific ViewModels, designed in a one-to-one relationship with a View (*ViewModel 1:1 View*). Unlike the first alternative, the ViewModel contains the presentation mapping logic, making it *presentation-ready* with a concrete intent on how information maps to GUI widget features. Consequently, the ViewModel provides information primarily as formatted strings or booleans, transforming the View into a *Humble Object* with minimal presentation logic. This supports unit testing of ViewModels covering most of the presentation logic, including mapping logic [19,37]. For example, the `PersonViewModel` provides presentation-ready fields like a boolean to control the address information visibility. Further, ViewModel fields like `FirstNameText` or `AgeText` have a concrete intent on how they should be mapped to a GUI widget. However, a Humble View limits the ViewModel reusability across Views with different information formats. At the same time, some sources explicitly state that reusing ViewModels should not be a premature goal [3] and almost never happens in practice [34].

Coupling and Model Wrappers Another particularly interesting aspect from our perspective is the coupling of the ViewModel to GUI frameworks. Suppose developers use GUI framework-specific helper classes like observables, command base classes, or visibility enumerations. In that case, the ViewModels are coupled to the framework and cannot be easily reused for other GUI frameworks in the future. Alternatively, if developers strictly avoid using such utility classes, they might develop them themselves [10,27]. This makes the ViewModels truly independent of GUI frameworks, and the GUI framework can be migrated without touching them. A further essential aspect of ViewModels is whether it exposes Model objects (e.g., business entities). The reviewed sources state two options:

Aggregate Model: The ViewModel directly exposes Model objects, which implies that the Model objects support observability for data-binding [37].

Model Wrappers: Instead of exposing Model objects, the ViewModel acts as a Model wrapper and provides proxy properties of any Model property [1]. For example, a Model `Person` class with a name is wrapped into a `PersonViewModel` with a dedicated observable name proxy property.

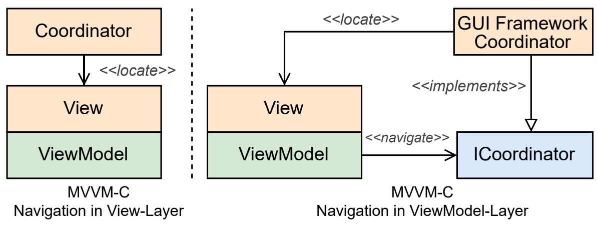


Fig. 6. MVVM-C with Different Navigation Placement.

5.2 MVVM with Navigation

Navigation and routing of Views are important responsibilities in many enterprise applications. This subsection discusses three design constructs: MVVM-C, hierarchical ViewModels, and ViewModel navigation events.

MVVM-C In MVVM-Coordinator (MVVM-C), navigation is explicitly included, which extends MVVM by a *Coordinator* component responsible for navigation. We see two options, as illustrated in Figure 6: The first option places the coordinator into the *View-layer*, solving the navigation using GUI framework-specific tooling. The second option introduces an abstract coordinator or navigation system in the *ViewModel-layer*, providing a GUI framework-independent API for navigating from one ViewModel to another. The abstract coordinator accepts either a type-information about the target ViewModel or it takes a context path (e.g., a URI) to locate the target ViewModel plus context [11].

Hierarchical ViewModels In projects with hierarchical Views, developers can create dedicated ViewModels for each View. This approach mirrors the View hierarchy in the ViewModel layer. For instance, in a master-detail scenario, a MasterViewModel might contain a DetailViewModel object, supporting direct context navigation to the details [37].

ViewModel Navigation Events When ViewModels are decoupled and require navigation capabilities, observer or event mechanisms offer an effective solution. The ViewModel triggers navigation events, which the View or an external component listens for handling navigation logic [3].

5.3 Further Design Aspects

This section briefly outlines nine further topics from the MLR results, highlighting a subset of the most relevant design constructs. Our replication package[5] discusses all design constructs in more detail and examples.

Command Design and Handling of Asynchronous Results WPF introduces first-class framework support for MVVM commands. However, some GUI frameworks do not have such support, and developers must design commands

[5] `https://doi.org/10.5281/zenodo.13350488`

more explicitly. One idea is that the ViewModel provides usual methods, which are called by event handlers (e.g., `OnButtonClicked()`) in the View [1].

Several sources mention asynchronous processing in the ViewModel (e.g., network calls on another thread). The result handling code then has to update the ViewModel, which is usually data-bound to properties of GUI widgets and hence can only be updated from the GUI thread. One idea is to introduce an abstracted dispatcher as a *Mediator*, which provides a GUI framework-independent API to run code on the GUI thread. Using Dependency Injection (DI), the actual GUI framework-dependent implementation is passed to the ViewModel objects, such that it can be used in result handles of asynchronous calls [18].

ViewModels also might prevent further actions while an asynchronous call is still running. Developers can use a *Busy Flag* to visualize information, which is set until the result handler is processed [1].

View Interactions ViewModels often have to fulfill the requirement to interact with the View, e.g., to let the View show a message box to the user. While MVVM, by default, only defines that the View knows the ViewModel instance, the ViewModel cannot directly call the View. We reviewed several design constructs to solve this problem: (1) Introduce a View interface, similar to MVP, which the ViewModel uses for View interaction [27]. (2) Provide events[6] in the ViewModel which the View can subscribe to [37]. (3) Using an interaction service that the ViewModel uses through an interface [37]. (4) Using *Pub/Sub* messaging to publish/subscribe messages. Depending on the programming languages, those options provide a way to solve the interaction problem [37].

Application Lifecycle Aware ViewModels In mobile apps (e.g., on Android), developers must manage the application lifecycle. For example, if a user pauses an app and resumes it later. Whenever a state is stored in ViewModels, developers should ensure that the state is valid on a resume.

In some MVVM frameworks like Android Jetpack or MvvmCross, explicit support is provided to make ViewModels lifecycle-aware. The idea is that ViewModels know about the application lifecycle's creation, pausing, or resuming events to control a consistent state. A bundle object can store and restore the state, which encapsulates the persistence of data [15, 29].

Networking and Client Server In client/server architectures, MVVM can be essential in structuring the data sent over the network. One design construct defines *Remote ViewModels*, which Singh introduces in a paper as the Remote-Model View Remote-View-Model (RMVRVM) pattern [32]. In RMVRVM, the ViewModel is sent over the network while the server stores the View state to optimize further updates (i.e., send only deltas of an updated ViewModel). Singh also discusses RMVRVM in the context of energy efficiency [32].

View/ViewModel Relationships In this aspect, we consider any statements about the View/ViewModel cardinalities that are not stated in the standard definition. Reviewed sources mention different combinations, namely that the

[6] For example, the C# language `event` keyword

View has one or many ViewModels [19,20,30] or that the ViewModel has one or many Views (e.g., when developing a wizard) [1, 19, 20]. Further, some sources explicitly mention a one-to-one relationship, which implies a more strict MVVM version. It implies that the View and ViewModel are a *tandem* developed together [2, 3, 34].

Formatting and Localization Some sources mention design constructs on how the ViewModel or View formats data. For example, coloring can be solved in two ways: (1) The ViewModel provides the color of a text box (as a string color code or logical name). (2) The ViewModel provides a logical enumeration state, and the View is responsible for mapping it to a concrete color [2, 20].

Another design construct is about how the ViewModel exposes numeric information. The ViewModel can either provide the integer or format it to the presentation-ready string, which the View directly displays to the user [20].

Multiple sources deal with the responsibility of localization. If done in the View, the ViewModel has to provide some logical strings, which the View-layer then localizes using dictionaries. If the ViewModel orchestrates the localization, the View is free of this responsibility, and the ViewModel uses a dictionary component that it can use to translate strings [38].

ViewModel Field Design This aspect deals with how developers can design ViewModel fields concretely. One design construct avoids Model types in ViewModels (allowed by the default definition). It allows only using standard types like integer or string, which decouples the View/ViewModel from the Model [24].

Another design construct focuses on visibility information in ViewModel fields. Instead of using GUI framework-specific visibility types, ViewModels use simple boolean types [1].

Further design constructs discuss different orientations when developers design ViewModel fields: (1) View orientation [28]. (2) Explicitly independent of the View [1]. (3) Model orientation by reusing Model types [18].

ViewModel Responsibilities When designing more complex ViewModels, developers should care about the responsibilities placed in the ViewModel. Sources discuss different ideas, e.g., how bindings are refreshed, how dirty flags indicate state changes, or where validation occurs.

We highlight two further ideas explicitly. First, for list items, filtering, sorting, etc., can be done in View or the ViewModel [1]. Second, modularisation plays a key role in complex scenarios, where input logic could be placed into a separate *Controller* to take this responsibility out of the ViewModel [41].

MVVM Domain-specific Languages A team can leverage a DSL to specify ViewModels and to ensure a consistent MVVM implementation. First, developers could use internal DSLs by using fluent API builders, which assist in implementing ViewModel commands or data [12]. Alternatively, external DSLs can help design a ViewModel's API programming language-independent [10].

To test ViewModels, test engineers might also utilize external DSLs, as demonstrated by the *ViMoTest* approach. Especially when using projectional editors, GUI widgets could be pre-rendered in a test case [10].

6 Threats to Validity

In this section, we discuss threats to the validity of our MLR study.
Construct Validity: We used ChatGPT to vote and filter websites automatically. Since ChatGPT's nature is non-deterministic and sometimes unreliable, we might have included false positives and false negatives. In particular, false negatives could negatively affect our results since we might not cover relevant aspects. To mitigate this threat, we confirmed the correctness by checking a random selection of the voting results.

A further threat is about subjective interpretation. The design and application of the classification scheme might involve biases or inconsistencies in categorizing and analyzing data. Further, the manual voting process for scholar entries might introduce selection bias, affecting the relevant sources.
Internal Validity: While we assume that our search did not scan every online resource, our initial Google search yielded over 500 websites, providing a substantial foundation. We have not applied further methods like systematic snowballing since checking every website for references is a considerable effort. Since we reviewed a substantial number of websites, it is unlikely that we missed crucial concepts not covered by the reviewed literature entries.
External Validity: As professionals with specific backgrounds wrote many reviewed sources, our results might be more applicable to specific applications (e.g., enterprise applications) and less to others (e.g., mobile apps or games).

Further, many reviewed sources focus on MVVM inherently integrated into specific technologies like WPF. Therefore, our findings might be limited to the ecosystems where MVVM is commonly used.
Reliability: The reproducibility of our search and selection process based on AI tooling like ChatGPT introduces challenges to reproduction by other researchers, impacting the reliability of the MLR process.

Further, our data synthesizing from extracted design constructs, benefits, and drawbacks into classifications to answer research questions involves subjective judgment, which might vary among researchers.

7 Related Work

This section discusses related work about MVVM or MV* overview studies.

Wongtanuwat et al. created a systematic guideline on detecting the correctness when applying MVVM in Objective-C programs [40]. Weissenberg discusses best practices and lessons learned using the MVVM pattern in an industrial WPF application [38]. While these papers specifically discuss the MVVM pattern, they are context-specific and do not analyze MVVM in a literature review.

Lou compared the MVC, MVP, and MVVM patterns for native Android app architectures regarding testability, modifiability, and performance [22]. Similarly, Sholichin et al. reviewed MVC, MVP, MVVM, and VIPER in the context of iOS architectural patterns [31]. Further, Magics-Verkman et al. compared MVC, MVVM, and MVI for testability and performance in iOS mobile application development [23]. These studies used concrete implementations of MVVM. They quantitatively compared them to other GUI architectural patterns for quality attributes, while our study qualitatively analyses MVVM by a literature review.

Lappalainen and Kobayashi qualitatively compared MVC, MVP, and MVVM by reviewing literature [21]. Syromiatnikov and Weyns selected several GUI architectural patterns like MVVM, reviewed sources describing those patterns, and qualitatively classified them as a landscape of GUI design patterns [35]. While these studies qualitatively review MVVM, they focus on a more extensive landscape of GUI architectural patterns and use no systematic literature survey.

Daoudi et al. empirically studied the occurrence of MVC, MVP, or MVVM in Android apps [5]. Chekhaba et al. introduced the machine learning tool Coach to identify MVC, MVP, or MVVM in Android apps [4]. Unlike our study, they do not qualitatively analyze design variants of MVVM using literature reviews.

Verdecchia et al. performed a systematic mixed-method empirical study on Android app architectures, including semi-structured interviews, gray literature, and white literature [36]. While they review the literature, our paper focuses specifically on MVVM and analyses design aspects and trade-offs more deeply.

8 Conclusion

This paper presented an MLR with a qualitative analysis of white and gray literature about the MVVM pattern. We used the standard definition and standard trade-offs of MVVM from familiar standard sources like Gossman's original blog post, which introduced MVVM. Based on a selection of 519 scholar entries and 523 websites, we filtered out 38 scholar entries and 125 websites, which extend design aspects or trade-offs compared to the standard definition. We then extracted 76 additional design constructs, 16 additional benefits, and 15 additional drawbacks. Finally, we categorized the design constructs into 29 design aspects and briefly described a subset in this paper. We published a detailed replication package with the results of the selection process, data extraction, and synthesis.

The synthesized design aspects and trade-offs provide an overview of the usage variants of the MVVM pattern. Practitioners, like enterprise application developers, can use them to have an explicit catalog of aspects that might be interesting in concrete MVVM implementations. Researchers can also benefit from this overview using the selected sources or the results of the replication package when further studying MVVM.

Future work could study the extracted design aspects and trade-offs, systematically analyzing conflicts between design constructs and their associated trade-offs. Such an analysis could lead to an assessment of the design constructs and recommendations on which constructs to adopt and which to avoid. Additionally, there is the potential to create formal pattern descriptions of both the standard MVVM definition and a subset of relevant design variants. Further, we will elaborate on the Humble View idea by applying the pattern in our doctoral ViMoTest project, where the ViewModel provides a presentation-ready abstraction by orienting directly on GUI widgets [9, 10].

References

1. Anderson, C.: The Model-View-ViewModel (MVVM) Design Pattern, pp. 461–499. Apress, Berkeley, CA (2012). https://doi.org/10.1007/978-1-4302-3501-9_13
2. Brumfield, B., Cox, G., Hill, D., Noyes, B., Puleio, M., Shifflett, K.: Developer's Guide to Microsoft Prism 4: Building Modular MVVM Applications with Windows Presentation Foundation and Microsoft Silverlight. Microsoft Press (2011), ISBN: 978-0-73565-610-9
3. Burns, K.: Introducing MVVM, pp. 127–140. Apress, Berkeley, CA (2012). https://doi.org/10.1007/978-1-4302-4567-4_9
4. Chekhaba, C., Rebatchi, H., ElBoussaidi, G., Moha, N., Kpodjedo, S.: Coach: classification-based architectural patterns detection in Android apps. In: Proceedings of the 36th Annual ACM Symposium on Applied Computing. p. 1429–1438. SAC '21, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3412841.3442018
5. Daoudi, A., ElBoussaidi, G., Moha, N., Kpodjedo, S.: An exploratory study of MVC-based architectural patterns in Android apps. In: Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing. p. 1711–1720. SAC '19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3297280.3297447
6. Engelschall, R.S.: Hierarchical user interface component architecture. BoD–Books on Demand (2018)
7. Fowler, M.: Presentation Model (Jul 2004), `https://martinfowler.com/eaaDev/PresentationModel.html`, accessed: 2024-06-18
8. Fowler, M.: HumbleObject (Apr 2020), `https://martinfowler.com/bliki/HumbleObject.html`, accessed: 2024-06-18
9. Fuksa, M.: ViMoTest: A Low Code Approach to Specify ViewModel-Based Tests with a Projectional DSL Using JetBrains MPS. In: Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings. p. 189–194. MODELS '22, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3550356.3558513
10. Fuksa, M., Speth, S., Becker, S.: Applicability of the ViMoTest Approach for Automated GUI Testing: A Field Study. In: 2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C). pp. 821–830 (2023). https://doi.org/10.1109/MODELS-C59198.2023.00131
11. García, R.F.: MVVM: Model–View–ViewModel, pp. 145–224. Apress, Berkeley, CA (2023). https://doi.org/10.1007/978-1-4842-9069-9_4
12. Garofalo, R.: Building enterprise applications with Windows Presentation Foundation and the model view ViewModel Pattern. Microsoft Press (2011), ISBN: 978-0-73565-092-3
13. Garousi, V., Felderer, M., Mäntylä, M.V.: Guidelines for including grey literature and conducting multivocal literature reviews in software engineering. Information and Software Technology **106**, 101–121 (2019). https://doi.org/10.1016/j.infsof.2018.09.006
14. Google, n.d., O.H.A.: Guide to app architecture (Feb 2021), `https://developer.android.com/jetpack/guide`, accessed: 2024-06-18
15. Google, n.d., O.H.A.: LiveData overview (Feb 2024), `https://developer.android.com/topic/libraries/architecture/livedata`, accessed: 2024-06-18
16. Gossman, J.: Introduction to model/view/viewmodel pattern for building wpf apps (Oct 2005), `https://docs.microsoft.com/de-de/archive/blogs/johngossman/`

introduction-to-modelviewviewmodel-pattern-for-building-wpf-apps, accessed: 2024-06-18
17. Gossman, J.: Advantages and disadvantages of M-V-VM (Apr 2006), https://docs.microsoft.com/en-us/archive/blogs/johngossman/advantages-and-disadvantages-of-m-v-vm, accessed: 2024-06-18
18. Hall, G.M.: The ViewModel, pp. 81–110. Apress, Berkeley, CA (2010). https://doi.org/10.1007/978-1-4302-3163-9_4
19. Kay, R.M.: How to Use Model-View-ViewModel on Android Like a Pro. https://www.freecodecamp.org/news/model-view-viewmodel-android-tutorial (Dec 2020), accessed: 2024-06-18
20. Kouraklis, J.: MVVM as Design Pattern, pp. 1–12. Apress, Berkeley, CA (2016). https://doi.org/10.1007/978-1-4842-2214-0_1
21. Lappalainen, S., Kobayashi, T.: A Pattern Language for MVC Derivatives. In: Proc. 6th Asian Conference on Pattern Languages of Programs (2017), http://www.washi.cs.waseda.ac.jp/wp-content/uploads/2017/03/Sami-Lappalainen.pdf, accessed: 2024-06-18
22. Lou, T.: A Comparison of Android Native App Architecture – MVC, MVP and MVVM. Master's thesis, Aalto University. School of Science (2016), http://urn.fi/URN:NBN:fi:aalto-201610124940
23. Magics-Verkman, H., Zmaranda, D.R., Győrödi, C.A., Győrödi, R.c.: A Comparison of Architectural Patterns for Testability and Performance Quality for iOS Mobile Applications Development. In: 2023 17th International Conference on Engineering of Modern Electric Systems (EMES). pp. 1–4 (2023). https://doi.org/10.1109/EMES58375.2023.10171619
24. Manferdini, M.: MVVM in SwiftUI for a Better Architecture. https://matteomanferdini.com/mvvm-swiftui (Dec 2023), accessed: 2024-06-18
25. Microsoft: The MVVM Pattern (2012), https://learn.microsoft.com/en-us/previous-versions/msp-n-p/hh848246(v=pandp.10), accessed: 2024-06-18
26. Microsoft: Model-View-ViewModel (MVVM). https://learn.microsoft.com/en-us/dotnet/architecture/maui/mvvm (2022), accessed: 2024-06-18
27. Mishra, A.: The MVVM Architectural Pattern, pp. 43–60. Apress, Berkeley, CA (2017). https://doi.org/10.1007/978-1-4842-2689-6_3
28. Moliński, D.: Flutter architecture: implementing the MVVM pattern (Feb 2022), https://fivedottwelve.com/blog/flutter-architecture-implementing-the-mvvm-pattern, accessed: 2024-06-18
29. MvvmCross: Introduction to Model/View/ViewModel pattern for building WPF apps. https://www.mvvmcross.com/documentation/fundamentals/viewmodel-lifecycle (Aug 2023), accessed: 2024-06-18
30. Rock, V.: Using MVVM for enhanced cross platform development of mobile and desktop application. Master's thesis, Master's Thesis (2015), https://diglib.tugraz.at/using-mvvm-for-enhanced-cross-platform-development-of-mobile-and%2Ddesktop-applications-2015, accessed: 2024-06-18
31. Sholichin, F., Isa, M.A.B., Halim, S.A., Harun, M.F.B.: Review Of IOs Architectural Pattern For Testability, Modifiability, And Performance Quality. Journal Of Theoretical And Applied Information Technology **97**(15) (2019), https://www.jatit.org/volumes/Vol97No15/3Vol97No15.pdf, accessed: 2024-06-18
32. Singh, L.: RMVRVM – A Paradigm for Creating Energy Efficient User Applications Connected to Cloud through REST API. In: 15th Innovations in Software Engineering Conference. ISEC 2022, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3511430.3511434

33. Smith, J.: Patterns - WPF Apps With The Model-View-ViewModel Design Pattern (2009), `https://learn.microsoft.com/en-us/archive/msdn-magazine/2009/february/patterns-wpf-apps-with-the-model-view-viewmodel-design-pattern`, accessed: 2024-06-18
34. Stein, G.: Introduction to Model/View/ViewModel pattern for building WPF apps. `https://www.linkedin.com/pulse/mvvm-fashion-trend-gregory-stein` (Mar 2021), accessed: 2024-06-18
35. Syromiatnikov, A., Weyns, D.: A Journey through the Land of Model-View-Design Patterns. In: 2014 IEEE/IFIP Conference on Software Architecture. pp. 21–30 (2014). https://doi.org/10.1109/WICSA.2014.13
36. Verdecchia, R., Malavolta, I., Lago, P.: Guidelines for architecting android apps: A mixed-method empirical study. In: 2019 IEEE International Conference on Software Architecture (ICSA). pp. 141–150 (2019). https://doi.org/10.1109/ICSA.2019.00023
37. Vice, R., Siddiqi, M.S.: MVVM Survival Guide for Enterprise Architectures in Silverlight and WPF. Packt Publishing Ltd (2012), ISBN: 978-1-84968-342-5
38. Weissenberg, C.: Model-View Design Patterns. Tagungsband p. 102 (2019), ISBN: 978-3-00-064236-4
39. Wisnuadhi, B., Munawar, G., Wahyu, U.: Performance Comparison of Native Android Application on MVP and MVVM. In: Proceedings of the International Seminar of Science and Applied Technology (ISSAT 2020). pp. 276–282. Atlantis Press (2020). https://doi.org/10.2991/aer.k.201221.047, `https://doi.org/10.2991/aer.k.201221.047`
40. Wongtanuwat, W., Senivongse, T.: Detection of Violation of MVVM Design Pattern in Objective-C Programs. In: Proceedings of the 8th International Conference on Computer and Communications Management. p. 54–58. ICCCM '20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3411174.3411193
41. Zarifis, K., Papakonstantinou, Y.: In-depth Survey of MVVM Web Application Frameworks. Tech. rep., Technical report of UCSDSE, University of California (2016), `https://dbucsd.github.io/paperpdfs/2016_4.pdf`, accessed: 2024-06-18