Using Hexagonal Architecture for Mobile Applications

Robin Nunkesser
Hamm-Lippstadt University of Applied Sciences, Marker Allee 76–78, 59063 Hamm, Germany

Keywords: Hexagonal Architecture, Software Architecture, Mobile, iOS, Android.

Abstract: Complex mobile applications require an appropriate global architecture. If used correctly, the high-level design patterns officially recommended for iOS and Android such as MVC, MVVM, and MVI/MVU may make an important contribution to the architecture, but they often require supplementary architectural concepts. General architectures such as Clean Architecture may come to the rescue but leave room for interpretation as to how they work best on iOS and Android. This paper discusses using Hexagonal Architecture as the fundamental global architecture for mobile architectures, providing an extendable approach suitable for small and large projects and helping to achieve more independence from frameworks and external agencies and better testability.

1 INTRODUCTION

When Apple published the iPhone SDK in 2008, they recommended adhering to the guidelines of the Cocoa Fundamentals Guide. A central role is played by their flavor of the Model View Controller (MVC) pattern originally introduced by Smalltalk (see e.g. Reenskaug, 1979), which is described by Apple Computer, Inc. (2006) as a "high-level pattern in that it concerns itself with the global architecture of an application". Technically, it is mainly a combination of GoF patterns from Gamma et al. (1995) and a grouping of objects into the three categories of models, views, and controllers (see Figure 1).

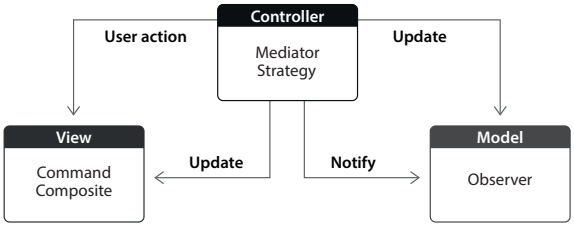

Figure 1: Reproduction of Apple MVC from Apple Computer, Inc. (2006).

Google also recommended MVC and variations such as Model View Presenter (MVP) and Model/View/ViewModel (MVVM) (e.g. in Android Jetpack ViewModel) for implementing Android apps. Since the introduction of SwiftUI and Jetpack Compose, Apple and Google have recommended similar MVC variations of Model View Controller inspired by Model-View-Intent (MVI; introduced by Staltz, 2015) and Model View Update (MVU; introduced by Czaplicki, 2016). SwiftUI and Jetpack Compose introduce state management concepts that facilitate these newer variations.

As stated above, MVC, MVP, MVVM, MVI, and MVU (abbreviated to MVX in the following) are high-level patterns contributing to the global architecture, but they are clearly not global architectures which suffice for complex mobile applications.

There are basically three choices for the global architecture: use none in addition to MVX, use an architecture specifically proposed for mobile applications, or use a known general software architecture.

In a first step, this paper shows advantages and disadvantages of these three choices resulting in a recommendation of using a general software architecture for mobile applications with nontrivial business logic. As a second step, possible disadvantages of choosing the currently popular Clean Architecture (Martin, 2017) are shown in combination with arguments in favor of Hexagonal Architecture (introduced by Cockburn, 2005). In a final step, challenges and solutions when using Hexagonal Architecture for mobile applications are presented.

2 RELATED WORK

There are many articles on architectural topics for mobile applications in blogs and non-reviewed digital

Nunkesser, R.
Using Hexagonal Architecture for Mobile Applications.
DOI: 10.5220/0011075100003266
In *Proceedings of the 17th International Conference on Software Technologies (ICSOFT 2022)*, pages 113-120
ISBN: 978-989-758-588-3; ISSN: 2184-2833

publications, but the focus here is on scientific publications.

To the best of our knowledge, there are scant papers on global architectures for mobile applications. Sommerville (2020) mostly treats mobile applications and web applications equally and only seldom expands on the special requirements of mobile apps. Salazar and Brambilla (2015) concentrate on a high-level process to guide application developers in the task of designing a suitable software architecture.

The high-level MVX design patterns are considered by more papers. La and Kim (2010) propose an adaption of MVC for large-scaled mobile applications. Shahbudin and Chua (2013) also propose an adaption called Extended MVC. Sokolova and Lemercier (2014) propose another adaption called Android passive MVC. Aljamea and Alkandari (2018) focus on MVC and MVVM on iOS, Plakalovic and Simic (2010) focus on MVC and PAC (Presentation-Abstraction-Control; another high-level design pattern). Dobrean and Dioșan (2019) present a comparative study of MVC, MVP, MVVM, and VIPER (an architecture proposed for iOS in Gilbert and Stoll, 2014). This is the only of the mentioned papers, where a global architecture (VIPER) is considered. However, VIPER is iOS only.

Apart from the work on MVX, some papers are considering special cases. Wichmann et al. (2009) focus on the specific task of reusability for mobile applications with location-based services.

None of the mentioned papers focuses on the use of general global software architectures for mobile applications.

3 ARCHITECTURAL CHOICES FOR MOBILE APPLICATIONS

Before we delve into architectural choices for mobile applications, it is necessary to look at the special requirements of mobile app engineering.

3.1 Requirements of Mobile App Engineering

Mobile app engineering has some specific aspects that differ from traditional software engineering. Most of these aspects are based on the mobile hardware the software runs on. Only some aspects derive from the common operating systems for mobile devices.

- A gap between development and runtime hardware
- A large amount and variety of sensors and actors
- Need for efficient code (memory, power, ...)
- Short release cycles of hardware and software
- Customization for individual device types
- Data storage, synchronization, and offline availability
- A high amount of UI code
- Event and lifecycle-based programming
- Concurrency with a special role for the UI thread

This list is not meant to be exhaustive. It focusses on aspects that are relevant for architectural choices. The works of Wasserman (2010), König-Ries (2009), Vollmer (2017), Knott (2015), and Sommerville (2016, 2020) may be of interest for details and further aspects.

3.2 Architectural Goals for Mobile Applications

Two architectural goals are especially helpful in coping with the mobile specific aspects of software development: independence of frameworks and external agencies and testability.

3.2.1 Independence

Independence from frameworks and external agencies is essential for mobile applications. It can be helpful in coping with the gap between development and runtime hardware by enabling code components to run on both hardwares, helping with unit tests and reuse beyond iOS and Android. By offering better flexibility, it also helps with the short release cycles, large variety of hardware and software, and customization needs.

3.2.2 Automatic Testing

Many of the stated aspects of mobile applications contribute to the complexity of testing for mobile applications. While the benefits of automatic testing are well known (see e.g. Rafi et al., 2012), we also know (e.g. from Beller et al., 2015) that many developers do not test. This is especially true for the mobile world, where we should divide automatic tests into at least three classes: local unit tests (on the development hardware), unit tests (on the runtime hardware), and UI tests. One of the main benefits of unit tests given by Beck (2002) is rapid feedback, which is only achieved by local unit tests. Therefore, good global architectures support a high ratio of local unit tests.

3.2.3 Modularity

Modularity is another architectural goal that is helpful for software in general. The ability to divide a software project into separate modules helps with reuse and may also be useful for the before mentioned goals independence and testability.

Table 1 shows an overview of these goals.

Table 1: Overview of important architectural goals.

Independence	Independence of external agencies
Testability	Amenability to automatic tests, and especially local unit tests
Modularity	Capability to be divided into modules

3.3 Assessment of the Architectural Choices

Currently, developers and architects typically either use the endemic (see e.g. Nunkesser, 2018) high-level MVX patterns, mobile specific architectures, or a mobile adaption of a global architecture.

3.3.1 Only MVX

Modern smartphones have a lot of computing power and developers have the freedom to execute complex tasks on the device or in the cloud. For apps with a lot of business logic, view-focused MVX patterns are not enough. Even apps that make heavy use of cloud services may be in need of an architecture that allows flexibility in changing these services and testability for the external dependencies.

3.3.2 Mobile Specific Architectures

Mobile specific architectures typically use a tight integration of third party code. Functional Reactive Programming is a popular example that tightly integrates external dependencies, which in itself prohibits achievement of the desired independence from frameworks and external agencies.

In summary, using mobile adaptions of global software architectures is the most promising approach for achievement of the stated architectural goals.

3.3.3 Mobile Adaptions of a Global Architecture

VIPER (Gilbert and Stoll, 2014), VIP (Law, 2019), and CleanArchitectureRxSwift[1] attracted a lot of attention on iOS. Android-CleanArchitecture[2] (see also Cejas, 2019), Wojda's Android Showcase[3], and Kušt's Clean Architecture Tutorial (Kušt, 2019) are popular examples on Android. All of them apply Martin's Clean Architecture to mobile apps.

Clean Architecture itself integrates ideas from preceding architectural concepts such as EIC/EBI by Jacobson et al. (1992), Hexagonal Architecture (a.k.a. Ports and Adapters) by Cockburn (2005), Onion Architecture by Palermo (2008), and DCI from Coplien and Bjørnvig (2011) and adds principles and concepts. Clean Architecture and derived architectures are typically well-suited to the architectural requirements of more complex mobile applications.

One major problem posed by Clean Architecture, however, is the lack of official reference implementations for different platforms (in Martin's books, his blog, training videos, and GitHub account, for example). The official examples are FitNesse[4] (a real-world project not tailored to learning Clean Architecture), Payroll (e.g. in Martin and Martin, 2006), made with similar principles but not the same naming conventions as Clean Architecture), Video Sales (in Martin, 2017, which is surprisingly brief), and the CleanCodeCaseStudy[5], which is well suited to learning Clean Architecture but not promoted as a Clean Architecture example. There is even an example for iOS[6], which is very different from VIP and VIPER, but also lacks some useful ideas presented in Martin (2017).

The suboptimal situation with regard to official reference implementations leads to problems: a large variety of unofficial examples (as of January 2022, a search for "Clean Architecture" on GitHub returns 826 repositories with Swift examples and even 3.303 repositories with Kotlin examples) with little quality assurance and often conflicting concepts. To elicitate this problems, we consider the six popular approaches mentioned. Table 2 provides an evaluation of these approaches with regard to the criteria shown in Table 1. In addition the faithfulness to Clean Architecture is evaluated.

The assessment of these criteria uses the following scheme:

High	The approach fulfils the criterion or shows only a minor deviation.
Medium	The approach fulfils the criterion in some but not all aspects.
Low	The approach does not fulfil the criterion.

[1] https://github.com/sergdort/CleanArchitectureRxSwift

[2] https://github.com/android10/Android-CleanArchitecture-Kotlin

[3] https://github.com/igorwojda/android-showcase

[4] https://github.com/unclebob/fitnesse

[5] https://github.com/cleancoders/CleanCodeCaseStudy

[6] https://github.com/unclebob/MACS_GOMOKU

Table 2: Comparison of existing Clean Architecture implementations.

	Independence	Testability	Modularity	Faithfulness
VIPER	High	Medium	Low	Medium
VIP	High	High	Low	High
CA-RxSwift	Low	Medium	High	Medium
Android-CA	Medium	Medium	Medium	High
Wojda	Medium	High	High	High
Kušt	High	High	High	Medium

Of the examples considered, none matches all expectations fully. As a representational example let us delve into the assessment of CleanArchitecture-RxSwift's independence as low. The dependance on RxSwift clearly justifies this assessment.

A problem in the application of Clean Architecture is that a suboptimal starting point has led to a variety of embodiments with sometimes conflicting and sometimes even incorrect interpretations. As a consequence, Clean Architecture examples for iOS and Android tend to give a lot of platform-specific guidance, often sacrificing independence and sustainability.

It is useful to reiterate that there are predecessors to Clean Architecture like Hexagonal Architecture, that were also designed with independence, testability, and modularity in mind, but may be easier to understand, implement, and extend.

4 HEXAGONAL ARCHITECTURE

Hexagonal Architecture was among the first architectures to break up traditional layering in favor of what would later become "onion" layering.

Cockburn (2005) introduces Hexagonal Architecture (also known as Ports and Adapters) and explicitly states the aim to:

> "Allow an application to equally be driven by users, programs, automated test or batch scripts, and to be developed and tested in isolation from its eventual run-time devices and databases."

The idea incorporates an application core with ports (protocols or interfaces) that define how the application may be used (by "driving adapters"; port implementations are in the core) and what data the application needs (provided by "driven adapters" implemented externally) (see Figure 2 for an illustration). The application core is technology-agnostic and contains the business logic of the application. Typical driving adapters are the UI, a testing framework, and external applications. Typical driven adapters are *repositories* which allow read operations and often also create, retrieve, and update operations and *recipients* in receipt of data. Driving adapters and driven adapters are *configurable dependencies*. Two details are missing from the description: where is the starting point of the application and where are the dependencies configured? These decisions are problem and platform dependent and are therefore not part of the architecture description.

Hexagonal Architecture has a very easy concept for achieving the main architectural goals discussed here. The biggest advantage, however, is that it may be easily combined with known concepts. Vernon (2013) states: "Because the Hexagonal Architecture is versatile, it could well be the foundation that supports other architectures required by the system. [...] The Hexagonal style forms the strong foundation for supporting any and all of those additional architectural options."

Hexagonal Architecture with MVVM for the GUI and EBI for the application core, for example, is a valid implementation of Clean Architecture. Using DDD (Evans, 2003) for the application core provides the basis for Palermo's Onion Architecture (Palermo, 2008) and Graça's Explicit Architecture (Graça, 2017). Building your architecture upon this idea also facilitates using MVI/MVU.

4.1 Reference Example

Several reference examples for Hexagonal Architecture exist. This paper focusses on hexagonalThis[7] because it is relatively new (2017) and approved by Alistair Cockburn. hexagonalThis is a .NET Console application written in C#. The example gives the architectural frame and a basic implementation of an application that provides poetry.

Figure 3 shows the class diagram of the application without the modules used for automatic testing.

The application core consisting of `PoetryReader` has access to a poem repository through the port `IObtainPoems`. The poem repository port is defined technology-agnostic, in the example a `PoemFileAdapter` that reads from an embedded JSON file is used.

[7]https://github.com/tpierrain/hexagonalThis

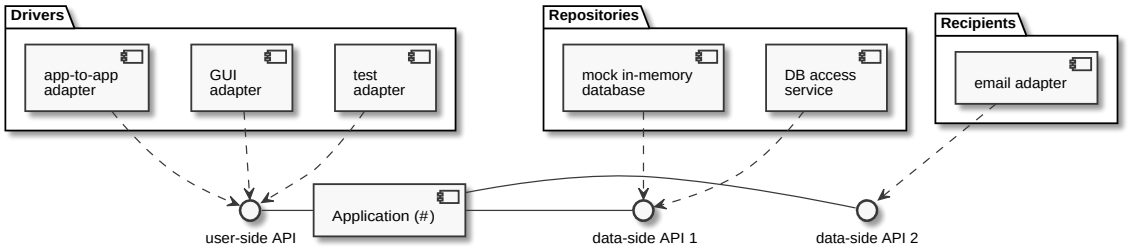

Figure 2: Hexagonal Architecture (Reproduction from Cockburn, 2005).

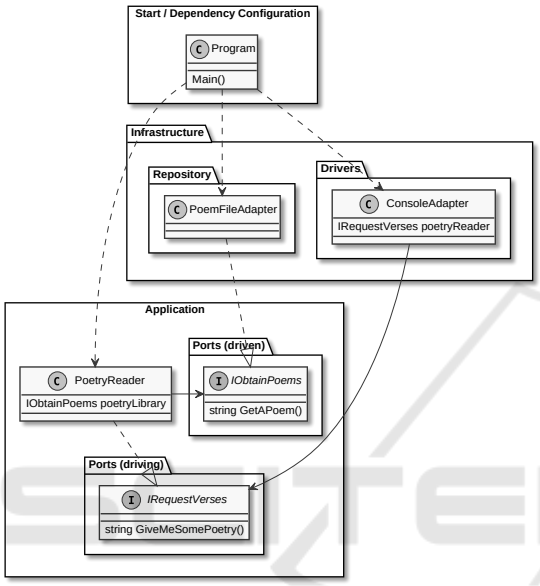

Figure 3: Minimal Example.

The application core offers the method `GiveMeSomePoetry` through the `IRequestVerses` port to driving adapters. In this example, the `ConsoleAdapter` is used to provide a console-based UI.

The concrete dependencies are configured in the separate `Program` class, which is also the application's entry point.

4.1.1 Independence and Testability

The `PoemFileAdapter` and the `ConsoleAdapter` are configurable dependencies. The application itself is independent of them through the use of ports. Keeping dependencies configurable does not necessarily mean that a dependency injection framework is needed, however. It may be enough to consider which dependencies have to be flexible and organize them in a transparent way that is easy to change. In hexagonalThis, the dependencies are configured directly in the application's startup code.

The `PoemFileAdapter` and the `ConsoleAdapter` are also amenable to testing. They are separated according to concern and independent of each other and the application core. However, the application itself is also amenable to testing, as a test adapter may use the `IRequestVerses` port and a mock implementation may implement the `IObtainPoems` port.

A typical development sequence with automated tests is shown in Table 3.

4.1.2 Modularity

The criteria from Table 1 used in Section 3.3 to compare Clean Architecture implementations also encompass modularity. Hexagonal Architecture clearly supports modularity, as every adapter that implements a port may be extracted to a separate module.

4.1.3 Comparison

There are two major beneficial differences in comparison with the considered implementations in Section 3.3. The reference example is basic and therefore easy to understand but also expandable and so enough to deliver the idea of Hexagonal Architecture. In addition, no external and no platform-dependent technologies are used and additional high-level MVX patterns or concepts like EBI and DDD may be integrated.

4.2 Considerations for Mobile Applications

Section 3.1 contains specific aspects of mobile application development. Some of these aspects have to be addressed so that Hexagonal Architecture can be implemented in mobile applications. Among the specific aspects, the gap between development and runtime hardware when developing mobile applications has the highest impact for the application of Hexagonal Architecture. The aspects considered in the following are:

Table 3: Exemplary Development Stages for Hexagonal Architecture.

Stage	User-side	Application	Data-side
1	Test cases	Hardcoded	-
2	Test cases	Real	Test doubles
3	Real	Real	Test doubles
4	Test cases	Real	Real
5	Real	Real	Real

Modularity	How is modularity achieved?
Dependencies	How are dependencies configured?
Concurrency	How are asynchronous calls handled?
Platform dependency	How do we handle code that depends on a mobile device?

4.2.1 Modularity

At the same time, modularization is the technique that provides the greatest benefits and poses the greatest challenges when implementing Hexagonal Architecture. By definition, Hexagonal Architecture uses a modular approach and therefore the high-level packaging of code is straightforward. A lot of different choices nevertheless exist with regard to the concrete packaging.

Brown (2017) discusses different methods of code packaging and the question of enforcement and creating separate assemblies for the components. While iOS and Android do not include modularization in the default templates, it is fairly easy to modularize the code. This also holds true for many cross-platform frameworks. Separating the code into real modules in mobile applications has a lot of advantages: testability, possible division of platform-dependent and platform-independent code, flexibility and, of course, easier enforcement of architectural principles. Therefore, the best approach is to create separate assemblies.

This will ultimately result in projects divided into platform-dependent modules and platform-independent modules. While this is good for testability and reusability, some challenges arise when considering the specific separation and when crossing boundaries. They will be addressed in the following.

4.2.2 Configurable Dependencies

In implementations like hexagonalThis, the dependencies are configured from a separate module. While this is the cleanest solution, mobile applications have the distinctive feature of incorporating platform-dependent and platform-independent modules. A module that configures the dependencies needs to be platform-dependent. In addition, the starting point of a mobile application from a typical developer's point of view differs from projects like hexagonalThis due to the event and lifecycle-based programming model.

The pragmatic solution for configuring dependencies is therefore to keep the application entry point, the UI, and the dependency configuration in the same module.

Some larger projects, multi-platform projects, and projects with configurable UI on the other hand benefit from having the dependency configuration done in a separate module or from using a dependency injection framework.

4.2.3 Concurrency

Asynchronous calls are very important in mobile applications and are typically implemented indirectly with concepts such as callbacks and completion handlers or more directly with asynchronous functions and operations. Although iOS and Android both prefer asynchronous functions and operations at the moment, there is a good chance that different concepts will be used heterogeneously in complex projects. It is therefore important to integrate a homogeneous asynchronous programming concept into the ports.

4.2.4 Platform Dependency

The mobile UI typically has platform-dependent code, at least. It is also possible, however, that the application core or an infrastructure implementation may need code that depends on iOS or Android. There are cases (like reading sensor values) where the whole module needs to be platform-dependent as a result. It is good practice to keep the platform-dependent modules small or to split a module into a platform-dependent and a platform-independent part, if necessary.

In these situations, it is necessary to trade the use of platform-dependent modules off against the implementation independence of the ports. To continue the example of sensor values, there are at least two possible solutions:

- Use a platform-dependent infrastructure module that directly accesses the required sensor

- Build an implementation-dependent port that defines the required sensor values as input values from another platform-dependent module

As mentioned above, the solutions offer different trade-offs and deciding the best solution depends on the specific app.

4.3 Porting the Reference Example

To fully understand the applicability of Hexagonal Architecture for iOS and Android it is helpful to port the reference example to iOS and Android. Although the example is simple, it gives the opportunity to integrate the mentioned topics modularity, configurable dependencies, asynchronous calls, and platform dependency.

In contrast to the original hexagonalThis, `GetAPoem` of the `IObtainPoems` port should be called asynchronously as we do not know the data source of the poems and it might take some time to retrieve them.

The access to a bundled file, on the other hand is an example where we may need platform-dependent code.

4.3.1 Considerations for Mobile Platforms

The configuration of dependencies is done pragmatically at application startup and with constructor-based dependency configuration.

4.3.2 Considerations for iOS

Modularization can easily be achieved by using XCode Workspaces and swift packages.

As stated above, asynchrony should be integrated into the ports, which results in

```
func GetAPoem(completion: @escaping
    (Result<String,Error>) -> Void)
```

or

```
@available(iOS 15.0.0, *)
func GetAPoem() async -> String
```

For the file access, platform-dependent code was required prior to Swift 5.3, but is no longer necessary. Since Swift 5.3, it has been possible to bundle resources with swift packages.

4.3.3 Considerations for Android

Modularization can be easily achieved by using Modules.

Again, asynchrony should be integrated into the ports, which results in

```
suspend fun getAPoem() : String
```

Access to embedded files is typically achieved with assets which may be accessed through the app context. This is clearly a platform-dependent solution. In platform-independent modules, however, resources may be easily accessed with the help of the class loader.

5 CONCLUSION

Popular modern architectural approaches such as Clean, Onion, and Explicit Architecture are essential for complex mobile applications because they assure independence from external frameworks and testability. However, the lack of quality-assured reference implementations for iOS and Android means a range of implementations and examples of varying quality. One reason for this is that Clean Architecture is often seen as a lock, stock, and barrel approach and is also frequently integrated too tightly with platform-specific or even external techniques.

Hexagonal Architecture is a predecessor of Clean Architecture that helps to achieve similar architectural goals important for mobile applications. In contrast to Clean Architecture, it is easier to implement and understand, but also offers the flexibility to be extended or used in a mix-and-match approach.

There are challenges involved in adapting Hexagonal Architecture, but these may be overcome without losing the advantages of the architecture or adding too much complexity.

REFERENCES

Aljamea, M. and Alkandari, M. (2018). Mmvmi: A validation model for mvc and mvvm design patterns in ios applications. *IAENG International Journal of Computer Science*, 45(3):377–389.

Apple Computer, Inc. (2006). Cocoa fundamentals guide.

Beck, K. (2002). *Test Driven Development: By Example*. Addison-Wesley Professional, Boston, MA.

Beller, M., Gousios, G., Panichella, A., and Zaidman, A. (2015). When, how, and why developers (do not) test in their ides. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, page 179–190, New York, NY, USA. Association for Computing Machinery.

Brown, S. (2017). The missing chapter. In Martin, R. C., editor, *Clean Architecture - A Craftsman's Guide to Software Structure and Design*, pages 303–321. Prentice Hall, Englewood Cliffs, NJ.

Cejas, F. (2019). Architecting android...reloaded. https://fernandocejas.com/blog/engineering/2019-05-08-architecting-android-reloaded. Last accessed: 04/29/22.

Cockburn, A. (2005). Hexagonal architecture. http://alistair.cockburn.us/Hexagonal+architecture. Last accessed: 08/22/18.

Coplien, J. and Bjørnvig, G. (2011). *Lean Architecture: for Agile Software Development*. Wiley.

Czaplicki, E. (2016). The elm architecture. https://guide.elm-lang.org/architecture/. Last accessed: 04/29/22.

Dobrean, D. and Dioșan, L. (2019). A comparative study of software architectures in mobile applications. *Studia Universitatis Babeș-Bolyai Informatica*, 64(2):49–64.

Evans, E. (2003). *Domain-Driven Design*. Addison-Wesley Professional, Boston, MA.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, Boston, MA.

Gilbert, J. and Stoll, C. (2014). Architecting ios apps with viper. https://www.objc.io/issues/13-architecture/viper/. Last accessed: 04/29/22.

Graça, H. (2017). Ddd, hexagonal, onion, clean, cqrs, ... how i put it all together. https://herbertograca.com/2017/11/16/explicit-architecture-01-ddd-hexagonal-onion-clean-cqrs-how-i-put-it-all-together/. Last accessed: 04/29/22.

Jacobson, I., Christerson, M., Jonsson, P., and Övergaard, G. (1992). *Object-Oriented Software Engineering*. ACM, New York, NY, USA.

Knott, D. (2015). *Hands-On Mobile App Testing*. Addison-Wesley Professional, Boston, MA.

König-Ries, B. (2009). Challenges in mobile application development. *it Inf. Technol.*, 51(2):69–71.

Kušt, I. (2019). Clean architecture tutorial for android: Getting started. https://www.raywenderlich.com/3595916-clean-architecture-tutorial-for-android-getting-started. Last accessed: 04/29/22.

La, H. J. and Kim, S. D. (2010). Balanced mvc architecture for developing service-based mobile applications. In *2010 IEEE 7th International Conference on E-Business Engineering*, pages 292–299.

Law, R. (2019). *The Clean Swift Handbook*.

Martin, R. and Martin, M. (2006). *Agile Principles, Patterns, and Practices in C#*. Robert C. Martin Series. Pearson Education.

Martin, R. C. (2017). *Clean Architecture - A Craftsman's Guide to Software Structure and Design*. Prentice Hall, Englewood Cliffs, NJ.

Nunkesser, R. (2018). Beyond web/native/hybrid: A new taxonomy for mobile app development. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*, MOBILESoft '18, Piscataway, NJ. IEEE Press.

Palermo, J. (2008). Onion architecture. https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/. Last accessed: 04/29/22.

Plakalovic, D. and Simic, D. (2010). Applying mvc and pac patterns in mobile applications. *ArXiv*, abs/1001.3489.

Rafi, D. M., Moses, K. R. K., Petersen, K., and Mäntylä, M. V. (2012). Benefits and limitations of automated software testing: Systematic literature review and practitioner survey. In *2012 7th International Workshop on Automation of Software Test (AST)*, pages 36–42.

Reenskaug, T. (1979). Models - views - controllers. Technical report, Xerox PARC.

Salazar, F. J. A. and Brambilla, M. (2015). Tailoring software architecture concepts and process for mobile application development. In *Proceedings of the 3rd International Workshop on Software Development Lifecycle for Mobile*, DeMobile 2015, page 21–24, New York, NY, USA. Association for Computing Machinery.

Shahbudin, F. E. and Chua, F. (2013). Design patterns for developing high efficiency mobile application. *Journal of Information Technology & Software Engineering*, 3:1–9.

Sokolova, K. and Lemercier, M. (2014). Towards high quality mobile applications: Android passive mvc architecture. *International Journal On Advances in Software 1942-2628*, 7:123 – 138.

Sommerville, I. (2016). *Software Engineering*. Pearson, London, 10th edition.

Sommerville, I. (2020). *Engineering Software Products*. Pearson, London.

Staltz, A. (2015). Model-view-intent. https://cycle.js.org/model-view-intent.html. Last accessed: 04/29/22.

Vernon, V. (2013). *Implementing Domain-Driven Design*. Addison-Wesley Professional.

Vollmer, G. (2017). *Mobile App Engineering: Von den Requirements zum Go Live*. dpunkt.verlag, Heidelberg.

Wasserman, A. (2010). Software engineering issues for mobile application development. pages 397–400.

Wichmann, D., Pielot, M., and Boll, S. (2009). Companion platform - modulare softwareplattform zur schnellen entwicklung von mobilen anwendungen. *it - Information Technology*, 51(2):72 – 78.