b
UNIVERSITÄT
BERN

Assessing and Improving the Software Quality of an iOS App Framework

Bachelor Thesis

Alain Stulz
from
Bern BE, Switzerland

Faculty of Science
University of Bern

07 February 2020

Prof. Dr. Oscar Nierstrasz

Software Composition Group
Institut für Informatik
University of Bern, Switzerland

Abstract

Creating and maintaining high-quality software is an essential topic in Software Engineering. While mobile application development is a relatively young discipline, it has evolved particularly rapidly. The quick pace requires complex mobile projects to be highly flexible and easily maintainable to stay relevant over time. In this thesis, we examine a framework designed to build iOS applications, which was created in the early 2010s and seems to have fallen behind in some areas. We answer *"How can we assess the quality of our system?"* by defining our understanding of software quality and subsequently collecting and analyzing data from several sources. In a second step, we address *"How to improve the existing system's quality?"* through setting conventions for developers, performing maintenance, and refactoring specific areas in the code. In this context, we also explore different techniques to increase unit test coverage. Furthermore, we analyze the question *"What would constitute a better software design?"* by selectively rewriting parts of the system's functionality. Finally, we take a look at the project's future and recommend that the company should consider a rewrite over refactoring to better cope with changed software requirements and technology.

Contents

1 Introduction

Apps with love AG is a digital service agency based in Bern, Switzerland [Ber20]. It specializes in designing and developing mobile and web applications for a wide variety of business customers, from small startups to big enterprises. The company was registered in 2010 by four founders and has since grown to employ over 35 people in locations in Bern and Basel.

Initially, the founders wanted to focus on developing their own app ideas and market them directly to end-users. However, they realized that there was a more viable target market in catering to other businesses that needed expertise in the rather new mobile app development industry.[1]

[1]Oswald, Olivier. Co-Founder, CTO (2019)

1.1 Festival App Projects

One of the earliest clients of the newly-founded company was the telecommunications provider Swisscom AG. As a sponsor for various music festivals throughout Switzerland, they envisioned providing visitors with a customized mobile app to simplify their festival experience.

Apps with love, and its partner company Moqod,[2] subsequently developed their first version of a "festival buddy app" for Swiss music festivals such as the Bern-based Gurtenfestival. The app consisted of a program of festival acts, a map of the venue, sections with news and relevant information, as well as a picture wall [Ber11]. (See Figure 1.1)

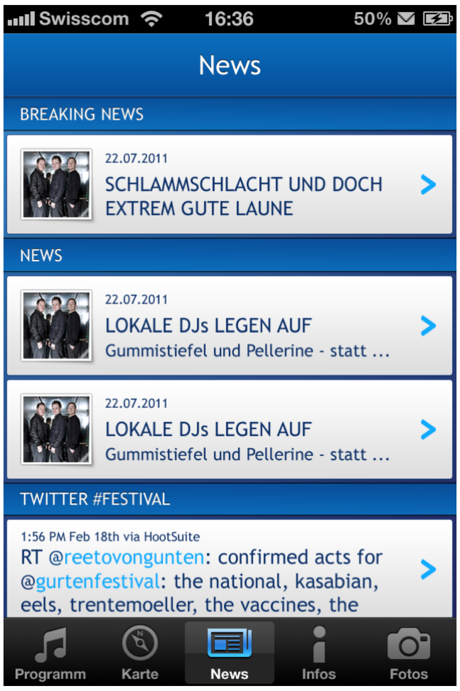
Two significant challenges in this project became apparent: Firstly, the content needed to be dynamically adjustable after the release of the app, as updating an app bundle would otherwise take several weeks. Secondly, the app was required to be fully offline-capable, as access to the internet was at the time often limited, partly due to carrier tariffs, partly due to network congestion at popular events, and partly due to high energy consumption of network requests.

The team developed a system consisting of native mobile frontend clients and a backend with a content management system to overcome these challenges. They would ship an initial version of the app content along with the app binary as part of a local cache. The app would then perform periodic network requests to the backend, if possible, and receive a delta update, bringing the app to the latest content version while minimizing the amount of network traffic needed.

1.2 Framework

The team applied this solution architecture by releasing a separate app for each festival. Initially, these were stored in separate folders inside a mono-repo hosted with GitHub. From there, the shared code diverged into different branches, as the projects needed customized features. Around 2015, they split this single repository into one repository for shared code and separate project-specific repositories, where shared code was imported through Git submodules, and additional customization could be made. Thus, the shared code was now available as a framework on which developers could build a multitude of apps rather efficiently.

[2]Founded in 2008, Moqod is a software development company based in Russia and the Netherlands.


Swisscom 16:36 50%

News

BREAKING NEWS

22.07.2011
SCHLAMMSCHLACHT UND DOCH EXTREM GUTE LAUNE

NEWS

22.07.2011
LOKALE DJs LEGEN AUF
Gummistiefel und Pellerine - statt ...

22.07.2011
LOKALE DJs LEGEN AUF
Gummistiefel und Pellerine - statt ...

TWITTER #FESTIVAL

1:56 PM Feb 18th via HootSuite
RT @reetovongunten: confirmed acts for @gurtenfestival: the national, kasabian, eels, trentemoeller, the vaccines, the

Figure 1.1: Screenshot of the first Gurtenfestival Buddy App Version

The framework was continuously expanded and used in a variety of apps over the following years. It gained a multitude of powerful features such as remote notifications, geofencing, indoor location using Bluetooth beacons, and live surveys.

1.3 Problem

With the continued growth of the codebase and its number of usages, maintaining the framework became increasingly difficult. Hard deadlines, as are typical for event projects, made developers build in “quick fixes,” which reduced the code quality. Also, new developers joined the company, and know-how exchange between the original framework creators and the new developers was challenging the organizational structure.

Ultimately, these issues led to fewer usages of the framework in newer projects, with more code built custom for individual projects. These projects often do not share a common software architecture. The lack of commonality can lead to a “silo organization,”[3] hindering the sharing of knowledge and improvements between projects and developers.

[3]Organizational structure in which members tend not to share know-how, goals, tools, etc.

1.4 Overview

In this project, we focus solely on the iOS implementation of this system, ignoring the Android and Backend components. We aim to address three research questions:

How can we assess the quality of our system? (Chapter 3)
We first analyze where to focus our attention to get the best view on the existing system's quality, finding that we should focus primarily on non-functional quality attributes, such as maintainability. We then discuss our methodology for gathering quality data from various sources, including the setup of static code analysis and visualizing the project's dependency architecture. Our finding is that the framework exhibits issues with the evolvability and understandability of the code. We notice code areas with high complexity and uncontrolled coupling, as well as the absence of unit tests and integration test infrastructure.

How to improve the existing system's quality? (Chapter 4)
Using our previous results, we aim to improve on the existing system. We set new developer processes into place by addressing issues with the current workflow, such as unclear branching. We also improve the test infrastructure by setting up continuous integration between the framework and different client projects, providing developers with instant build feedback. Next, we explore different strategies for increasing test coverage and performing maintenance on the code-level, followed by exploring opportunities for code refactoring.

What would constitute a better software design? (Chapter 5)
To answer this question, we reimplement select features of the current system. We specify an architecture that could better support the software requirements. We then discuss the benefits and drawbacks of our solution in comparison to the existing implementation and end on the conclusion that, due to the changed underlying requirements, rewriting the features that continue to be used would be more efficient.

2 Related Work

Creating high-quality software is universally accepted to be one of the central goals and challenges of the software engineering field. As such, the concept of quality and the techniques to develop software with high quality have been the focus of intense discussion. In the following sections, we want to take a closer look at several topics related to this project.

2.1 Software Quality

Quality is a product aspect that remains elusive and hard to define. Garvin argued that it could be described in five different approaches: The *transcendent view* as something that can be recognized but not explicitly defined. The *product-based view* as a measurable combination of the product's attributes. The *user-based view*, using the metric of user satisfaction, that is, how well the product fulfills its users' needs. The *value-based view*, regarding the relation between value brought to the user and product cost. And finally, the *manufacturer-based view*, identifying quality as "conformance to requirements" [Gar84]. For software, this last view explicitly includes non-functional requirements such as maintainability, which may comprise a significant part of a manufacturer's understanding of quality, but have only an indirect impact on a user's perception thereof.

While Pressman and Maxim rightfully state that "in reality, quality encompasses all of these views and more" [PM15, p.413f.], we will focus mostly on a manufacturer-based view from here on, as it is most approachable to us as developers of the subject system. The consumer's view of product quality is thereby not neglected, as providing value to users is ultimately the goal of developing software. Instead, we use an introspective approach as a means to improve our development of products in the future.

It is noteworthy that while we can apply Garvin's concept in this context, he did not develop it with software engineering in mind, but rather as a generalized view of product quality. Neither can it be neglected that the models created in regards to quality are countless and multifaceted. In Chapter 3, we attempt to find an interpretation of quality that can be applied to our specific problem, helping us to understand underlying issues and to take steps to mitigate these.

2.2 Legacy Software

Legacy systems are often associated with decades-old software that businesses deeply integrated into their core processes. There is a connotation of concepts, languages, and even hardware that is no longer at the current state of technology [Som16, p.261f.]. Such a definition would have partial applicability to our project, as the mobile application field is young compared to the history of software development. However, despite its relatively new appearance, the field has seen a particularly rapid evolution in hardware and software capability and industry adoption, disrupting entire economies with new possibilities. The problem of legacy software may prove even more important when faced with the speed of growth and evolvement in the mobile application business.

A connection becomes even more apparent as we take other, more open definitions into account. Feathers defined legacy software as follows: "To me, legacy code is simply code without tests. [...] With tests, we can change the behavior of our code quickly and verifiably. Without them, we really don't know if our code is getting better or worse." [Fea04]

Taking these definitions into account, we can conclude that the system we are dealing with is indeed legacy software. There are various recommendations on how to improve such systems, including refactoring, wrapping, or rewriting system components. Each strategy entails different opportunities and risks. The book *Object-Oriented Reengineering Patterns* goes into more detail about selecting and applying these strategies [DDN03, p.19ff.].

2.3 Software Refactoring

"Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure [Fow99, p.9]." It can be used both as a way to improve legacy code and as a tool that any developer should continuously apply when writing code, to prevent said code from becoming *legacy code* in the first place.

Multiple strategies for rewriting have been specified in the past, most notably by Fowler [Fow99] and Feathers [Fea04]. They rely on the same basic concept that code first needs to be placed under tests before developers can edit it with the confidence that the system as a whole continues to work as expected. These works offer guidelines for relatively specific scenarios, as we will see later in our project.

2.4 Software Rewriting

In contrast to refactoring an existing system, rewriting a system describes the act of implementing a (part of a) system without reusing its source code, aiming to achieve the original functionality with a new solution instead. Developers tend to choose this approach if they feel that it would be more efficient than refactoring, or if the existing system prevents improvement, for example, due to past technology decisions. Both approaches come with their benefits and drawbacks, and we will compare them for our specific case in Section 5.4.

3 Quality Assessment

Developers who join the Apps with love team and first come into contact with the iOS implementation of our framework, often are under the impression that the code is not very well structured and is relatively hard to understand. As we learned in the previous Chapter, this intuitive feeling of quality, or lack thereof, is only one aspect of what defines quality. Now, we want to obtain a more objective understanding of the system's current state.

As developers analyzing our own system, we naturally tend to take a manufacturer's perspective, as described in Section 2.1. This point of view allows us to define our understanding of quality more formally by focussing on *software requirements*.

We can roughly classify requirements as functional and non-functional: [Som16, p.105-111]

- Functional requirements describe what a system should (be able to) do
- Non-functional requirements do not directly impact the features of a system but specify aspects that are significant for the system's architecture

Non-functional requirements are often described as "ilities" due to the common suffix present in many of these attributes. Notable examples for this are testability, maintainability, usability. Attributes not following this syntax include performance and security requirements [CBN13].

This same classification can also be applied to our understanding of quality, allowing us to differentiate between [Cha13]:

- Functional quality, the ability of given software to match the users' needs, i.e., functional requirements
- Structural quality, the system's ability to meet non-functional requirements
- Process quality, the methods used to develop the software

At this point, our system has been in use and successful for several years. It follows that the original functional requirements have been and continue to be fulfilled by the existing system. Any potential quality problems would have to either stem from unfulfilled non-functional requirements, or changes in functional requirements.

The latter would only be an issue if the system showed a lack of maintainability and evolvability, which in turn are aspects of structural quality. By extension, we should also find corresponding factors in process quality that could be used to reduce or mitigate issues with structural or functional quality.

We can conclude that we should focus first and foremost on assessing structural quality and non-functional requirements in order to gain a better picture of our system's status and any potential problems. In a later step, we could then try to ascribe those structural quality issues to their origin in the development process.

3.1 Methodology

To improve our understanding of the system, we considered multiple data sources: Through the usage of static analysis tools, we collected objective data on code-level quality. Through interviews with developers, we gained insights into hinderances to productivity, both on the code level and in the development organization. By combining objective metrics with the subjective hindrances to development, we can establish a cause-and-effect relationship between issues in the development process and the resulting code quality. We will discuss the results of the collected data points in Section 3.2.

3.1.1 Developer Interviews

To gain insights into developer productivity and needs, we conducted brief interviews with developers and product managers. We asked them about the benefits and drawbacks they saw when working with the framework. We then clustered the results and categorized them into code-level, module-level, and organization-level topics. Next, we prioritized the topics based on their difficulty to resolve and the timeframe in which we could achieve an improvement. Finally, we derived possible actions to improve these specific areas.

We also listed the most critical non-functional quality attributes (“ilities”, as described earlier), and short subjective assessments of the current state of each area. Then we matched the previously received feedback with the listed quality attributes and prioritized the areas based on the apparent potential for quality improvements. The outcome was a prioritized list of actions for quality improvements and possible objective metrics for the most critical quality attributes (See Table C.2).

3.1.2 Static Analysis

Static analysis of program code uses software tools to scan the source code of a project and detect possible issues [Som16, p.359f.]. Often, these tools can also derive metrics that provide objective data about a program.

We used SonarQube[1] as a static analysis tool, which is compatible with the mixed Objective-C and Swift source code in our framework. During the build process, a

[1]Available at `https://sonarqube.org`. More information in Section A.6.

code coverage file is generated. SonarQube monitors branches and pull requests in the framework repository, and generates reports automatically. We used the default quality profiles, but the rules could be customized if necessary. Code-level analyzers are somewhat prone to report false-positives, so their results should undergo careful examination.

In addition to static analysis, we gathered data about the module-level structure in the framework and the organization-wide usage of framework components.

3.1.3 Dependencies and Usages

Module-Level Currently, we use CocoaPods[2] to manage code dependencies. This tool allows for resolving and importing dependencies to third-party code as well as our framework code in our various projects. Additionally, the framework uses CocoaPods internally to integrate different modular components.

Any CocoaPods module can import other modules by specifying them in their `Specfile`.[3] We analyzed these imports and created a visualization of module dependencies, allowing us to get a high-level picture of the architectural framework structure and its possible issues. The process and results are shown in Section B.1.

Organization-Level On the organizational level, we were particularly interested in the total number of projects using the framework, and additional data on which parts of the framework were most and least often used. To achieve this, we exported a list of all project repositories in our organization and filtered out projects concerning platforms other than iOS.

We then prioritized the list of projects using the following factors:

- Past financial value of project to company
- Future expected financial value
- Likelihood and expected impact of future changes

[2]A dependency manager for Swift and Objective-C projects. Available at `https://cocoapods.org`.

[3]The `Specfile` contains metadata CocoaPods uses to make a module available to other modules and projects.

with values from 0 *(no future value/changes unlikely)* to 2 *(high value/major changes expected)*. We calculated the total priority by multiplying the individual factors, resulting in a priority scale from 0 to 8. We selected projects with high priority (≥ 4) for further analysis and tracking throughout our reengineering process.

We then performed a detailed usage analysis on the selected projects by individually cloning the repositories, extracting the imports from the `Podfile`[4] references. Additionally, we gathered data about the actual usage of the imported classes. We then mapped this to the modules containing the used classes.

This analysis resulted in a total count of module-level imports and usages of the tracked projects. We then used this data to derive a strategy for further reengineering work, as outlined in Chapter 4.

3.2 Results

3.2.1 Developer Interviews

The positive feedback about the framework mostly focussed on the business value that the framework provides in projects: Developers find it easier to start new projects if they can work with an existing toolkit instead of starting from scratch. It also ensures consistency, allowing developers to start working on existing projects quickly, should they need to. Additionally, bugfixes and improvements to the framework are available to all projects using the shared source code.

While projects benefit from the existence of shared logic, maintaining the framework itself was connected to some negative sentiment. There were multiple factors mentioned here, primarily:

- Code is hard to understand, navigate and change
- (Side-) Effects of changes are not visible
- Updating projects to newer framework versions takes much work

From the responses, we can also identify some concrete problems with structural quality, as well as factors in process quality that we might consider causes of those structural issues. We again attributed these to the level where the problem stems from, respectively, where a solution would most likely need to be applied. See Table C.1 in the Appendix for a full list of the collected results.

[4]The `Podfile` specifies the dependencies CocoaPods should import into a project.

3.2.2 Static Analysis

With SonarQube we performed an initial analysis of the repository state at the project start. There were 65'000 source lines of code analyzed.

Bugs In total, there were 66 bugs reported, consisting of 8 major and 58 minor bugs. Upon closer inspection of the bugs reported as *major*, we found that the issues were not as severe, and could also be classified as *minor*. Note that this is a purely rule-based bug recognition, as opposed to bugs reported by developers or users.

Code Smells More interestingly, there were almost 3'400 reported code smells, with one blocker, 142 critical, 2'400 major, 771 minor, and 213 info level severity reports. The effort to resolve the code smells was estimated at 77 days by SonarQube, based on the triggered rules and an expected cost of development per source line of code.

The tool reported over 900 code smells due to commented out blocks of source code with severity level *major*. From code inspections, it becomes apparent that this was most often code that was at one point no longer needed, but developers did not feel confident enough to remove the affected code entirely. From a maintainability perspective, it would be beneficial to remove this code to reduce cognitive load when reading through classes. Also, multiple files were commented out entirely, indicating that the affected classes were no longer needed. These should also be removed to make it clear that the classes are no longer available.

Around 600 reported major code smells were found in functions that requested unused parameters. In iOS development, the delegation design pattern is used quite extensively, both in system APIs and in custom code. This pattern can explain many of these indicated code smells. The function caller is often passed as a reference, regardless of whether the callee uses it or not. Another big factor is error handling in asynchronous messages, where the system may create an error object to pass information back to the caller, which is in itself not a problem. We conclude that these smells need individual examination, and we can make no general statement about their validity at this point.

400 code smells with major severity were reported because include-statements were not placed at the very top of the source file. We can easily explain this because Objective-C differentiates between file (preprocessor) `#import` and module `@import`. If a module import was located above a preprocessor import, the tool marked this as a rule violation. These violations can, therefore, be dismissed as false-positives, because they do not affect the validity or understandability of the code.

Other notable results include 130 `TODO` and 18 `FIXME` statements (*info* and *major* severity), 120 usages of code marked by developers as deprecated (minor severity), and 84 pieces of code marked as deprecated (info). These reports could indicate that the maintainability of the project is impaired, as `TODO`s are not getting resolved, and code is deprecated but rarely removed due to dependencies.

Furthermore, there were indications of code and architecture complexity issues, shown by 41 functions with excessive cognitive complexity ($\text{max} = 201, \text{avg} = 44, \text{allowed} = 25$) and 45 classes with more than five ancestors. ($\text{max} = 11$) This report confirms the developer sentiment that parts of the code are hard to understand. The class ancestry issue seems to stem from the use of subclassing to extend view controllers with functionality such as logging. Ideally, this could be solved by using the Swift programming language to create class extensions.

Test Coverage The amount of code covered with unit tests is meager ($\leq 1\%$), which makes maintaining, changing, or expanding the business logic difficult due to possible unintended side effects. Developers cannot tell if any changes they made have effects on other parts of the framework. Additionally, the code was not originally written with testability in mind, meaning that adding tests can take much effort, as we also experienced later in our refactoring work.

Duplications With a reported 2.1% duplicated lines (≈ 2000 lines total), code duplications – although developers generally should avoid them – are not a quality concern in this case.

3.2.3 Dependencies and Usages

Module-Level During static analysis on the module-level, we were most interested in the dependency relations between different modules. We first gathered a broad overview by visualizing the module dependencies. (See description and figures in Section B.1)

In the dependency visualization, we can see the layered architecture of the system, with network and persistence (data access) layers at the bottom. Building on that layer is `ABFCoreUI`, which is referenced by most of the required modules,[5] for example, `ABFInfoModule`.

[5]Modules that are depended on by core parts of the framework and must be included to be able to build the project

We can also see some structural problems in this graph. One is the heavy dependency on the `ABFDataAccessLayer` module, which works as a sort of *god object*, a common architectural anti-pattern. Despite its name, much of the functionality of this module can be ascribed to the single `ABDataAccessLayer` class, which combines many different responsibilities, including directly handling a database from within the class code. This class would need to be split and separated adequately through interfaces, increasing flexibility and maintainability of the system.

Also apparent is the odd placement of `ABFCoreApp`. We consider other modules to be higher-level concrete implementations built upon the core components, so the core components should not depend on those concrete implementations. Should we, for example, choose to use a different implementation of `ABFPreCacheManager`, we would also have to change code in `ABFCoreApp`, which would then have an unwanted impact on other projects. This structure should be changed, so that core components and modules both only depend on abstractions, something Sommerville described as *inversion of control in frameworks* [Som16, p.445].

The same also applies to dependencies between different modules, which should be separated by abstractions so that the system is less rigid, and developers could easily inject their custom implementations if required by a project.

Organization-Level We analyzed 44 repositories from iOS projects in our organization and found that 24 of them used some version of the framework. During prioritization, we found that of those 24 projects, six were discontinued, and an additional two use deprecated frameworks that would require migration to a newer version.

Of the remaining 16 projects, we classified eight as low-priority, three as medium, and six as high. Projects with medium and high priority were further analyzed, and we tracked seven projects throughout the changes made in the framework as part of this thesis. One project with an initially high priority was discontinued during our work, so we stopped tracking it.

In the projects that were further analyzed, we tracked the imports and actual usages of framework modules. We then applied rules based on the relation of imports and usages to decide on strategies for further work with each module.

From a total of 49 modules, we found that 14 had no usages within the medium- to high-priority projects tracked. We recommended the removal of these modules to facilitate further maintenance of the framework. We recommended considering six modules with a high amount of usages (in more than five of the nine tracked projects) for further quality analysis and improvement.

Most projects seemed to import a vastly higher number of modules than they used. In the most extreme case, a project imported 24 modules but referenced only four of these within the project code. This pattern can be attributed to framework-internal dependencies on other modules, and in turn, to excessive coupling within the framework.

For 17 of the modules, we found significant differences between imports and actual usages, indicating that there were issues with dependencies on these modules. We recommended that the usages of these modules from within the framework core should be revised and better separated through abstractions. This improved separation would reduce the need for projects to import these modules without actually using them.

We recognized one more organizational issue, in that no feature or issue backlog exists. Developers generally track bugs and enhancements in customer projects, which hinders the exchange between framework developers and reduces the amount of data we could analyze. For example, we would have been interested in the average time-to-fix for bug reports, which would represent a good metric for maintainability.

3.3 Conclusion

We found that our framework – while it provides value to our business and our developers – exhibits issues with code-level quality and software architecture. We also conclude that these issues mostly affect maintainability, more specifically in the areas of evolvability and understandability. Factors involved are missing tests and test infrastructure, high program complexity, and uncontrolled coupling and dependencies between classes and modules.

A self-enhancing aspect to our quality issues is apparent. A lack of maintainability, if left uncontrolled over time, can lead to a lack of maintenance, which in turn further decreases maintainability. This finding is in line with Lehman's *law of increasing complexity*, which states that systems are bound to increase in complexity unless developers work to maintain or reduce it [Leh80].

Possible solutions to these quality issues include the removal of deprecated code and refactoring of existing code to increase testability and reduce complexity. We recommend the setup of testing and quality control infrastructure, and the consideration of architectural changes to reduce coupling between core components and modules, especially those with less frequent usage.

We also recognize the need for adjustments to the organization's development process, for example, in the way developers share know-how, track issues, and perform code reviews. The implementation of these adjustments exceeds the scope of this thesis.

In our further work, we aimed to address some of the code-level and architectural challenges.

4 Software Refactoring

Following the analysis and categorization of quality issues and their causes, we derived specific actions that could help improve quality over time.

We prioritized those actions by the estimated effort required and the time frame in which we could likely implement them. We began with relatively easy fixes that mostly required one-time actions, such as implementing a repository branching strategy and setting up a continuous integration system. Later, we focussed on adding unit tests and refactoring specific parts of the framework.

4.1 Repository Branching

When we initially started work on this thesis, the most recent release version of the framework was located on a feature branch, while the `master` branch tracked an older framework version. This older version is used by some legacy projects that did not receive updates in the last year. Developers would push their changes directly to these branches without further review. This system could confuse developers and hindered the setup of a reliable strategy for continuous integration.

As part of our work, we created a fork of the framework's repository – so as not to influence ongoing development activity with our changes – and put a branching system into place. We removed the legacy framework version and implemented the *GitLab flow* branching strategy, which our team had previously evaluated [Git20].

Our new system relies on using three main branches: `master`, `testing`, and `production`. While developers could push directly to the `master` branch, the other two are set up as protected branches, preventing direct pushes. This restriction forces developers to use merge requests, which in turn serves to improve communication and code quality. It also allows us to set up continuous integration within the framework and with different client projects.

4.2 Versioning

In the existing framework implementation, projects would import the framework by cloning the entire framework repository using Git submodules. This solution meant that any given project pointed to one single commit in the framework's versioning history, causing all projects to use slightly different versions.

After applying a branching strategy, we were able to move the dependency specifications (`Specfiles`, see Section 3.1.3) into a separate repository. Using our dependency management tool, CocoaPods, we can now release and import individual components using semantic versioning. This new approach allows for more fine-grained control of project migrations, and, in combination with a release roadmap, could help keep projects up-to-date with framework changes.

4.3 Continuous Integration

Continuous Integration describes the process of perpetually rebuilding the source code, after changes in the codebase have been made [Som16, p.742f.].

In our situation, we use it to ensure that changes in the framework are compatible with the projects that use the framework.

We chose to set up GitLab CI to build the framework with every merge request to the `testing` and `production` branches and following every completed merge to these branches. In the CI build process, we leverage *fastlane*, a popular iOS build pipeline tool, to build the test project placed within the framework repository. We then run any provided unit tests and scan the code coverage for further analysis by SonarQube.

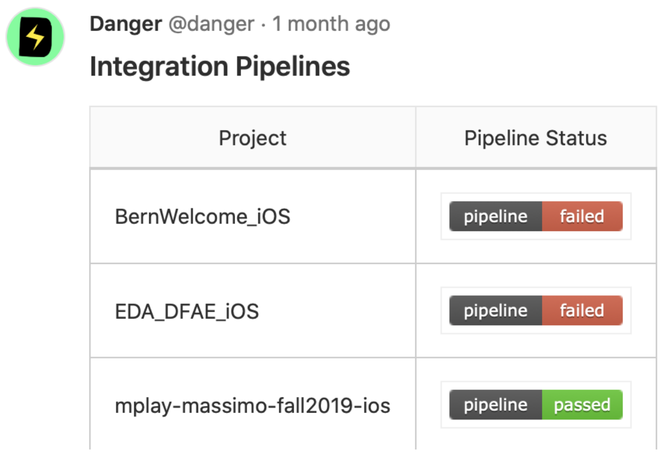
If these stages pass successfully, we run integration builds with selected high-priority client projects. Builds in client projects are triggered using the GitLab API, and always use the latest version of the framework's `master` branch. Continuously building our projects helps us ensure that changes made in any given pull requests do not impact compatibility with existing projects. If a developer were to make an incompatible change in the framework, the build attempt would fail, and they would receive notice quickly. They can then change their implementation in the framework or perform necessary migration steps in the affected project. This improved availability of feedback helps developers make changes within the framework more confidently.

Using this approach, we only get a rough picture of change compatibility. We recommend the setup of additional project-level sanity checks, such as automated UI tests, which run as a part of the CI build. This kind of setup could help detect issues with functional changes that do not impact the binary interface to the framework, but which break project-level logic. A simple approach would be to leverage UI tests to take screenshots of crucial app screens. Framework maintainers could then manually review the screenshots before completing a merge request. More elaborate setups could use project-level UI test logic to perform certain checks automatically.

4.4 Build Status Feedback

With the approach mentioned above, we cannot get feedback on the status of triggered builds, meaning that developers would have to check the build results manually. A paid feature in GitLab would allow developers to see the results of downstream builds in the framework build pipeline.

Since we are currently not willing to incur the associated expenses, we used another approach that relies on the open-source code review tool *Danger*. It is integrated into the *fastlane* build pipeline and adds a comment to the merge request that triggered the build. Developers can see the build status in this comment and directly access more detailed build information. (See Figure 4.1)


Danger @danger · 1 month ago

Integration Pipelines

Project	Pipeline Status
BernWelcome_iOS	pipeline failed
EDA_DFAE_iOS	pipeline failed
mplay-massimo-fall2019-ios	pipeline passed

Figure 4.1: Build pipeline feedback in GitLab provided by Danger

4.5 Test Coverage

During our review of strategies for improving existing software systems, we noticed that a typical recommendation is adding more automated unit tests. Feathers describes that developers can make changes to legacy code in two ways: *Edit and Pray* or *Cover and Modify* [Fea04]. The first approach relies on manual testing after making changes, which is unreliable and time-consuming. The second approach refers to the convention of first writing unit tests for code that developers need to change, and only then modifying the source code.

Using this approach, maintainers can make sure that changes they make do not affect the functionality of the modified code in unforeseen ways. Unfortunately, covering existing code with tests is usually not trivial, because the authors of the code did not write the code to be testable. Such code often includes network or database requests and calls to other code that cannot be controlled by the test environment. Additionally, methods in such code are often long and relatively complex, meaning that is is harder to write tests for them.

Because we wanted to change the structure of some code to improve our framework, we reasoned that we should add more unit tests throughout the project. We found that impediments to testability, such as excessively long methods involving multiple dependencies and network calls, are generally prevalent in the code in our framework. To achieve the best results, we selected specific parts of the code where we identified opportunities for coverage improvements during our process.

4.5.1 Focus

We focussed on the three framework core modules: `ABFCoreApp`, `ABFCoreUI`, and `ABFDataAccessLayer`. The data access layer is responsible for storing, retrieving, and updating app data. In order to do this, it accesses network and database systems. It also contains base implementations for customizable framework modules.

The core app module controls the rest of the application throughout its lifecycle. It contains higher-level logic for accessing data through the data access layer. Also, the core app module is responsible for routing between different app screens and various other functions, such as switching between content languages, which need app-wide coordination.

Finally, the core UI module contains base implementations for UI components, which can be used and customized by other modules in the framework. It also stores design configurations.

These three modules captured our interest because they entail logic that is accessed and used from almost every other framework module. As an additional factor, we considered the frequency of changes made per file, with developers changing files in these core modules relatively often. (See Section B.2 in the Appendix)

To prepare for working on these code parts, we first analyzed the existing implementation to gain a better understanding of its inner workings.

As a preparation step for further work, we ensured that the code was formatted and commented more consistently. There are useful tools to improve code formatting automatically. For Objective-C code, we used the `clang-format` utility. For Swift code, we made use of the autocorrect function in the open-source library *Swiftlint*.

Also, we added small comments in parts where we saw a need or an opportunity for refactoring. We changed the order of properties and methods to adhere more closely to the program flow, because some of the classes we handled had a high complexity, and their functionality was not immediately apparent.

4.5.2 Strategies

To increase the test coverage as intended, we tried multiple different strategies to overcome hindrances to testability. We will describe our approaches as case studies of classes we encountered.

Mock Classes The `ABFModuleFactory` class enables part of the framework's modularized architecture. Modules always subclass `ABFBaseModule` and can contain a separate database and other logic. `ABFModuleFactory` serves to instantiate and manage the lifecycle of these custom modules. It works by finding all subclasses of `ABFBaseModule` and storing a reference to them. Any module will receive a call to perform custom logic when the app starts. Also, the module factory notifies modules of significant events, such as when app content is reset, e.g., due to a change in the user's language preference.

This class does not require any other classes to be initialized and performs logic on a common base class. This structure allowed us to place the class under unit tests relatively quickly. We accomplished this by creating a mock subclass of `ABFBaseModule`, which can be inspected by test code to check that operations are performed as expected.

Preprocessor Macros Contrary to `ABFModuleFactory`, the `ABBusinessEngine` class initializes and uses different objects within and outside of its module. For example, it also calls a method on `ABFModuleFactory` during its initialization process. Since this would typically mean that we could not reliably test the module factory class, we found a way to change the behavior of `ABBusinessEngine` when running tests without affecting the class otherwise.

With the iOS build system, we can leverage C compiler features – in this case, the C preprocessor macros – to change code behavior during builds. This way we can change the code that is compiled into the build output for different environments.

In our test target, we can define a preprocessor flag by putting a command `#define UNIT_TEST` in any of our test source files. In the class where we want to change the behavior, we can then use `#ifdef UNIT_TEST` to use an alternative implementation. Consider the initializer we want to change in `ABBusinessEngine`:

```
- (instancetype)init
{
    self = [super init];
    if (self)
    {

#ifdef UNIT_TEST
        // altered behavior for tests
        return self;
#endif

        // normal behavior,
        // continue with initialization
    }
}
```

In the unit test environment, the initializer will return before any side effects, such as calls to our test subjects, could occur. Using this approach, we can create an instance of `ABBusinessEngine` and pass it to any objects that would require it. We must notice, however, that performing any operations within the class could result in undefined behavior because not all of its properties have been initialized.

Thankfully, if we build our code for production – or any other environment than where our `#define` command is placed – we can be sure that the class behavior remains unchanged because the compiler strips out all of the code we have inserted if our flag is not present. This is an advantage over directly modifying code to support tests because we have a clear separation between the test and production environments. Essentially, the production environment remains unchanged.

Spy Delegates The iOS system classes make extensive use of *delegation*, a design pattern in which an object coordinates its logic with another helper object called delegate. Apple's Cocoa documentation [App18] describes this as follows:

> *The delegating object keeps a reference to the other object—the delegate—and at the appropriate time sends a message to it. The message informs the delegate of an event that the delegating object is about to handle or has just handled. [...] The main value of delegation is that it allows you to easily customize the behavior of several objects in one central object.*

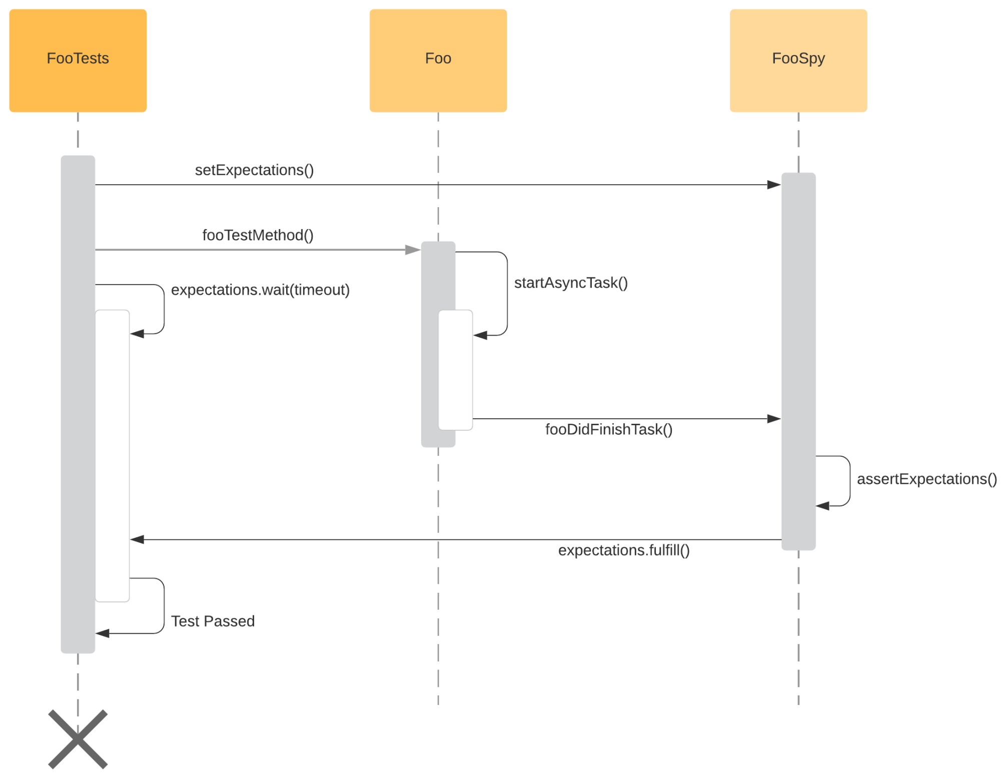


Figure 4.2: Sequence of Actions in the Test Spy Pattern

Since iOS developers also tend to use this pattern in their custom code, we wanted to leverage its customizability to facilitate the creation of unit tests. We achieved this by combining delegation with the *Test Spy* pattern. We found these two patterns to generally be neatly compatible with each other.

The test spy construct works by using a mock/double class to capture indirect outputs made from the tested object to another component [Mes07, p.538]. (See Figure 4.2)

In this instance, `FooSpy` replaces a `FooDelegate` object that would normally be used by `Foo` as its delegate. The `FooSpy` object implements the same underlying delegate interface, meaning that the `Foo` class can continue to use the methods normally abstracted through the delegation pattern, visible in the typical `fooDidFinishTask(_:)` method call.

The test class, `FooTests`, injects its expectations into the spy class. For example, we could expect `fooDidFinishTask(_:)` to be called after some time to indicate that the asynchronous task has successfully completed. The fulfillment or timeout of expectations in the `FooTests` class is handled by the `XCTest` framework iOS developers typically use for testing.

In addition to this test functionality, we can benefit from optional protocol (interface) methods to gain deeper insights into the state of `Foo`, which would otherwise not be accessible from delegates. Consider the following extension of our delegate protocol:

```
 -(void)fooDidFinishTask:(Foo *)foo;

@optional
 -(BOOL)shouldStartTask;
 -(void)fooWillStartTask:(Foo *)foo;
```

Using this construct, our `FooSpy` could implement the additional, optional methods to control the `Foo` class. For example, it might be desirable to skip the asynchronous task for some tests, in which case the `FooSpy` would simply answer the `shouldStartTask` call with `NO`.

Furthermore, we could measure the time needed to complete the asynchronous task by starting a timer as soon as `Foo` calls the `fooWillStartTask(_:)` method. Meanwhile, the original implementation of `FooDelegate` can remain unchanged.

One more benefit of this approach is that we do not have to significantly change the tested class to implement this. Delegate methods are simple to call at any point in the code. However, if we want to use the mechanism of optional delegate methods, we do have to perform an additional safety check before calling delegate methods:

```
if ([self.delegate respondsToSelector:
                @selector(fooWillStartTask:)])
{
    [self.delegate fooWillStartTask:self];
}
```

Unfortunately, in the Swift programming language, optional protocol methods are not directly available as they are in Objective-C. However, we can make this work by either using bridging to Objective-C with the `@objc` keyword or protocol extensions to provide default implementations for optional methods. For example, a default implementation for the `shouldStartTask()` method would simply return `true`.

In summary, we found that spy delegates can be a useful tool when implementing tests for classes that use the delegation pattern, as is often the case in iOS development. They require only minimal modifications on the delegating object, and no modifications in the delegate object, while providing both insights into the tested object and a mechanism to change the behavior of the test subject.

4.6 Compiler Warnings

When we started work on this thesis, performing a complete build of the framework would generate over 12,000 warnings. This high number is a problem because it obscures warnings that are generated as a result of ongoing development. Additionally, a high number of warnings may discourage developers from fixing any warnings at all because the task can feel daunting.

Missing *nullability annotations* caused most of these warnings. This feature was introduced in 2015 to improve compatibility between the Objective-C and Swift programming languages. In Swift, the nullability of properties – that is, whether the property is ever expected to have a null value – is specified through optionals. This construct makes the code safer because it prevents crashes due to unexpected null values. It is worth noting that the compiler generated one warning per usage of such an unmarked variable, not per definition. This means that one single line could trigger dozens of warnings, inflating the total number of warnings.

Apple extended Objective-C to support such behavior when headers are bridged to Swift. For every property specified in a header file, developers should specify whether the code is designed to accept a nil value passed as a parameter.

We worked on adding such nullability annotations to the core modules in the framework, in order to reduce the number of warnings generated. At the time of writing, we were able to reduce it to around 500. While there are still some warnings of this kind present in different modules, our change has served to reveal warnings about relevant issues like usages of deprecated system API usages, incompatible types, and unused variables.

If Apps with love choose to continue to develop and support the framework, they should take further action to reduce the number of warnings. Fewer warnings help developers keep the code up to date with changes to system behavior, programming languages, and internal code structures.

4.7 Deprecated Code

Framework maintainers typically mark code as *deprecated* to discourage its usage, perhaps because a more efficient or safer implementation replaced it, it contains known flaws or is otherwise deemed no longer worth supporting. As opposed to directly removing the affected code, marking it as deprecated serves to keep compatibility with existing usages of the affected features. Warnings may be generated by the compiler to let developers know that they should remove usages of affected interfaces.

The act of marking code as deprecated shows the intention of a developer that code should ideally be removed after a certain time has passed. This forces other developers to migrate their usages to replacements provided by the framework, or otherwise adjust their implementation. Removing code that has previously been marked as deprecated reduces the overall size of the codebase and makes the rest of the code easier to maintain because developers can focus their attention on more relevant parts.

During our work, we noticed that multiple classes and methods had been marked as deprecated several years prior. We removed those code parts which were no longer used in our tracked high-priority projects. We also removed code parts that were commented out but not yet deleted. Furthermore, we identified modules which our tracked projects did not use, and removed them as well.

Our changes helped to reduce the overall system size from over 65’000 source lines of code to around 47’000. We noticed that this initial removal also made even more parts of the code redundant because they had only been used in those previously deprecated parts, which can create a sort of avalanche effect throughout the project.

We noticed that some deprecated code had too many usages from different locations to be deleted in the timeframe of this thesis. One example of this situation was color handling. The framework code contains specific hard-coded color values. We would remove these values need because they are no longer state of the art in color management. The iOS operating system relies on different color representations to support both light and dark appearances, as well as different color spaces.

However, because these seemingly trivial color values are used not only within the framework but also within different dependent projects, their removal was not possible at this time. Instead, an alternate mechanism for color configuration would need to be developed and implemented throughout all dependent client projects before deletion.

For further work, we recommend prioritizing the removal of features that are no longer relevant to the business to reduce the overall codebase size and increase maintainability.

4.8 Code Refactoring

The work outlined in the previous sections was performed partly as preparation for, and partly in unison with software refactoring. We can broadly ascribe our refactoring work to two categories: restructuring of overly complex classes with usages throughout the framework, and smaller, topical improvements.

The first category mostly revolved around the classes `ABDataAccessLayer` and `ABBusinessEngine`. We described the role, and reason for selection, of these classes in Section 4.5.1. To reiterate, the classes are exceedingly complex and tie together significant parts of the app's functionality.

An important step that we consider part of refactoring was reordering the class headers and method implementations to represent the program flow more accurately. This improved structure helped us understand the logic of these classes better. We added code markings to differentiate between major sections in the classes. We also placed comments where we identified specific actions that should be taken.

Focussing on `ABFDataAccessLayer`, we initially struggled with testability. The problem in this class was the multitude of database calls embedded within the rest of the class logic. By splitting these calls into a separate class, `FestivalStore`, we could then test the data access layer class independently of any database by using a mock `FestivalStore`. Future work in this area would consist of finding such opportunities for refactoring, applying the outlined strategies, and gradually improving test coverage and architecture.

The second category of smaller improvements involved usages of deprecated system interfaces. For example, the framework uses a utility class to present an alert popup view to the user. Since iOS 8, alerts should use `UIAlertController`, whereas previous versions used `UIAlertView`. The utility class used an external library named `RMUniversalAlert` to address this. It featured an implementation compatible with both versions. Since our current projects only support iOS 10 and above, we no longer need this external dependency. (At the time of writing, the most recent major version is iOS 13.) We removed the library, which had been last updated in 2016, and used a small Swift class to implement this behavior. We identified many locations where such smaller refactorings could benefit the overall code quality.

4.9 Summary

In this chapter, we took action to improve the quality of our existing software system. We started by setting branching conventions for developers and changing our mechanism for versioning and importing dependencies. These changes enabled us to set up a continuous integration process, as well as a system to automatically provide build feedback, giving developers the ability to gauge their changes' impact on client projects quickly.

We then explored different strategies for increasing the test coverage. We learned about the utility of preprocessor macros to quickly alter the behavior of classes in order to prevent unintended side effects when testing. Furthermore, we introduced the concept of Spy Delegates and explored its application in the context of iOS development. We discussed our steps in maintenance, reducing the number of warnings and the amount of deprecated code. Finally, we briefly touched the topic of code refactoring, learning that refactoring in our context typically falls into two classes: small improvements where code was out of date, and larger sections with centralized logic that should be better separated.

In the process of applying our changes, we learned that while our improvements to the development and testing setup can speed up development time, refactoring the framework to fix most of the apparent quality issues would be an endeavor taking a significant amount of effort. We will now compare the approach of refactoring the existing framework to rewriting parts of the codebase, which could be another viable option to provide a better system in the future.

5 Software Rewriting

We wanted to compare our refactoring strategy to a second approach, which is rewriting parts of the existing system. Our company's requirements have changed over time, and the original framework no longer suits our business needs. Event apps used to be a central part of our revenue streams, but have become much less important over time.

Consequentially, parts of the framework have become mostly irrelevant, while numerous desirable new features are currently missing. By cherry-picking existing functionality and re-implementing it under consideration of these changed requirements, we believe we might be able to achieve a more flexible solution that we can use in various projects.

To compare both approaches, we implemented a small part of the existing functionality using a more modern toolchain. If we find evidence that our new solution can achieve similar or better results than refactoring in a comparable time frame, we might consider this an alternative to adapting the existing framework. In Section 6.1, we will compare the outcome of both approaches as a basis for a business decision.

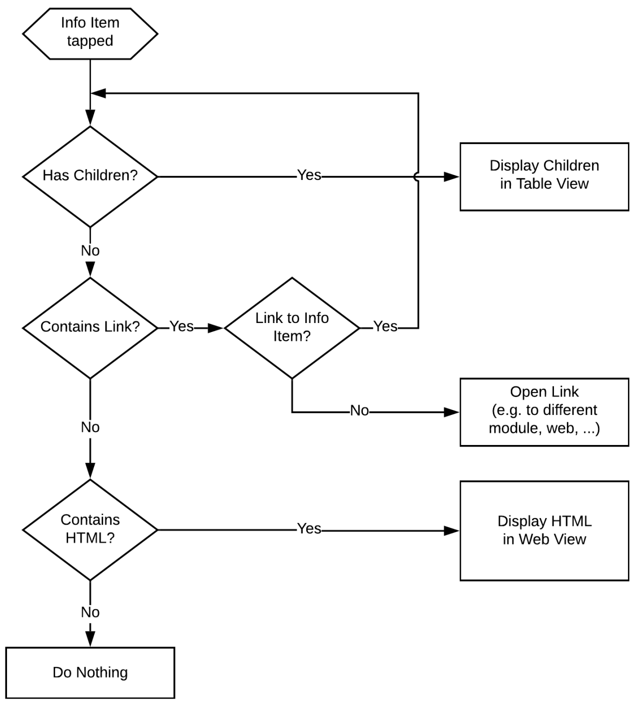


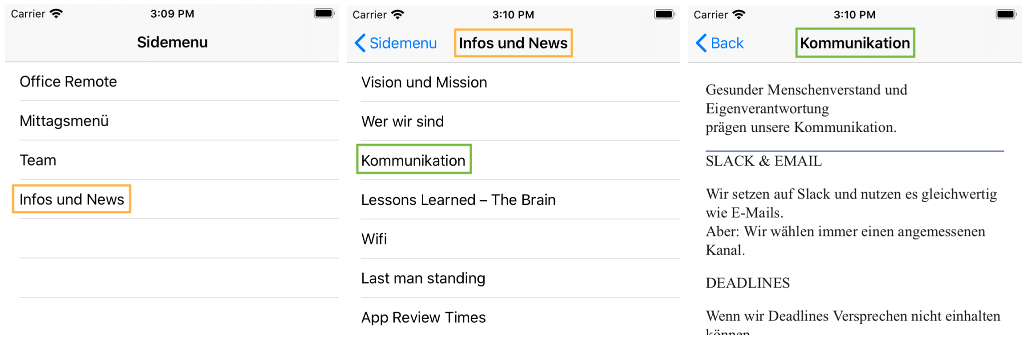
Figure 5.1: Linking Logic of Info Items

5.1 Functionality

Let us describe the features we selected for rewriting. We aimed to select a well-defined, definite subset of the existing system's functionality.

One central functional element many of our apps share are so-called Info Items, which are a versatile feature for displaying nested content as a tree structure. The children of a shared root node commonly represent items in a menu, typically a side menu or tab bar. When the user taps on a node, the system determines which screen to show next, most commonly a list of child items or a detail screen presenting HTML content. (See Figure 5.1 and Figure 5.2)

The info item feature depends on shared functionality from the framework core. This module handles a one-way synchronization process with a server-hosted content management system. It communicates with an API providing delta-updates for the entire app content. The core module stores and manages the app's configuration data as received from the backend, and passes data to all the app modules. The modules are free to decide which data they need and how they want to store it.



Carrier 3:09 PM

Sidemenu

Office Remote

Mittagsmenü

Team

Infos und News

Carrier 3:10 PM

Sidemenu Infos und News

Vision und Mission

Wer wir sind

Kommunikation

Lessons Learned – The Brain

Wifi

Last man standing

App Review Times

Carrier 3:10 PM

Back Kommunikation

Gesunder Menschenverstand und
Eigenverantwortung
prägen unsere Kommunikation.

SLACK & EMAIL

Wir setzen auf Slack und nutzen es gleichwertig
wie E-Mails.
Aber: Wir wählen immer einen angemessenen
Kanal.

DEADLINES

Wenn wir Deadlines Versprechen nicht einhalten

Figure 5.2: Info Items: Root, List and Detail Screen

5.2 Technology

Our existing framework is mostly written in Objective-C. For the new implementation, we used the Swift programming language that Apple introduced in 2014 and has become the default language for native iOS projects.

While the existing project used CocoaPods for dependency management, we chose to use Swift Package Manager, which has been integrated into the Xcode IDE since the last major version. It simplifies the process of specifying new modules, as well as using code from local and remote sources. We found it integrates neatly into our workflow, as it provides an easy way of specifying and importing packages (see Section B.3). One drawback is that it does not provide the ability to ship assets or other binary files in packages as of yet. However, the Swift contributors may well implement this in the future.

5.3 Architecture

Based on the existing implementation, we decided to use a similar approach keeping our code separated into different modules. For our purposes, we created two packages, *Core* and *Info*.

The Core package does not use any iOS-proprietary libraries and is therefore fully cross-platform compatible with all Apple-owned platforms and Linux. It contains code to ensure the compatibility of modules, such as shared protocol definitions and models. Furthermore, it provides shared services such as linking between modules and retrieving updates through a shared API.

The Info package depends on the Core package and serves as a reference implementation of a module. We will describe our solution in detail by building up a picture of the Info module implementation step-by-step.

Our design for a module bases itself on the *Clean Swift* architecture [Law15], which is, in turn, an adaptation of Robert C. Martin's *Clean Architecture* [Mar12].

This architecture seemed ideal because it complements a modular structure due to its strict separation of concerns, allowing project developers to use their custom logic if the need arises. We also learned that our Android developers successfully used a similar design in their implementation.

Clean Swift splits the basic structure of an app into scenes. For example, the *Info List* and *Info Detail* screens each represent one scene. At the core of a scene are three different classes with distinct responsibilities:

- The View Controller handles displaying information to – and receiving inputs from – the user. It requests data from the Interactor.
- The Interactor receives requests from the View Controller and stores or loads data.
- The Presenter receives data from the Interactor and tells the View Controller how to display this data.

Together, they form the View-Interactor-Presenter cycle through which the app handles its business logic. Protocols separate the logic between classes. The View Controller sets up the scene and is the only class that contains direct references to other classes – the rest of the classes only know their protocol interfaces. (See Figure 5.3)

The architecture describes two additional concepts: Routers and Workers. View Controllers use Routers to coordinate navigation to another scene in the app. Interactors offload work such as fetching data to Workers. (See Figure C.1, Page 57)

We combine multiple scenes with shared data structures to form modules. For example, the Info Module combines the Info List and Info Detail scenes. Classes within a module can share underlying model classes and workers.

A module also contains a mechanism for storing and updating its underlying data. It achieves this using the Repository, which acts as a single source of truth for the module. A Repository is free to decide how to store data. Most typically, it would use a database, but we have also successfully implemented an in-memory store for debugging purposes.

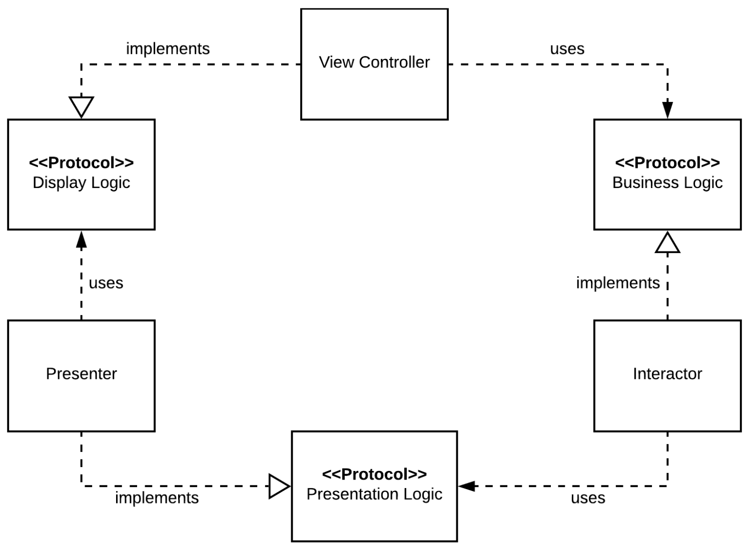

View Controller
implements
uses
<<Protocol>> Display Logic
<<Protocol>> Business Logic
uses
implements
Presenter
Interactor
implements
<<Protocol>> Presentation Logic
uses

Figure 5.3: Relation between View Controller, Interactor and Presenter

5.3.1 Data Updates

Repositories use an UpdateMapper to subscribe to an Updater located in the framework core, which allows them to get notified when the core receives new data. (See Figure C.2, Page 58)

Since we use one central API for all modules and entities, the core needs to be agnostic to the content entities it receives. The only pieces of information processed in the core are related to app configuration and content versioning. The core passes the complete received data to each subscribed UpdateMapper.

The mapper filters out only the entities relevant to its module and passes it to the Repository for storage. The Repository also uses the VIP cycle to update any scenes currently displayed to the user.

5.3.2 Linking

Another addition we needed in order to implement our Info Item feature was centralized linking. An Info Item can contain a link of the form `module/key?query=param`.

When a View Controller encounters such a link, it uses its Router to determine how to link to the next scene. If the link points to a screen within the same module, this works without any issue. The Router can get access to all the data it needs and performs the navigation.

However, because we do not want modules to have any knowledge of each other, the Router does not know how to handle a link to a different module. For this case, we have added a Linker service to the framework core. (See Figure C.3)

During the app start, the Linker is initialized and configured with a link map, giving it a reference to all modules and their link strings. If a Router wants to navigate to a different module, it asks the Linker for an abstract representation of the link target. Using this construct, the Router can pass the link query data to the target without needing to know anything about the target screen's implementation.

5.3.3 View Configuration

A big difference from the existing implementation lies in the way we configure the appearance of views.

The current framework creates its views programmatically and relies on two mechanisms to adapt appearance: Firstly, View Settings objects, which client applications can override to adjust parameters like color schemes. Secondly, the subclassing of views allows for more in-depth adjustments, for example, creating a custom appearance of a table view cell.

In our new implementation, the entire logic about view appearance is solely in the client project, not in the framework itself. Using this approach, we can leverage Storyboards, which allow us to create our views in a graphical editor within Xcode. One advantage of this is that client projects can customize their appearance more easily. (See Figure 5.4) Furthermore, we can benefit from the simplified ability to support auto layout, which handles sizing for different devices, and easier debugging of light and dark system appearance styles.

The only knowledge the framework needs to have about its views is a unique string to identify the view, as well as a reference to any properties of the view (typically subviews) that the view controller needs to access.

Framework modules run a check during app startup to ensure that no views are missing. In debug mode, the app terminates to let the developer know instantly that a screen is missing. Views can only check the presence of required view properties once the respective screen is opened. We achieve this through assertions placed in the view setup code.

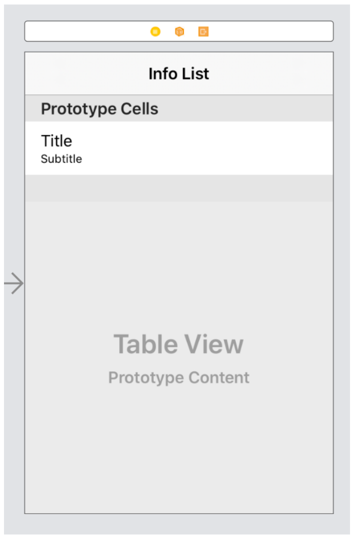


Figure 5.4: Customization of the Info List Scene in Xcode Storyboard

5.4 Comparison

We see multiple advantages but also drawbacks to our solution. One notable improvement is the exclusive use of Swift instead of Objective-C. The new programming language is available since 2014 and introduces many new features, like optionals and improved type-safety. Another new concept is Swift Package Manager, which simplifies the declaration and import of dependencies.

A clearly defined and documented architecture helps developers structure their code. It allows exchanging specific implementations quickly, which enables unit testing through mocking, as well as easy customization of framework code for specific projects.

The Core module supports cross-platform capability, which was not possible in the previous framework and can help us develop for different platforms such as macOS or tvOS. The use of Storyboards enables powerful customization of views while keeping the framework implementation simple.

Inherently, there are also drawbacks to creating a new version of the framework from scratch. Perhaps the most significant one is the big upfront time and cost expense required to create a minimum viable version of the system. Another factor is the risk of repeating mistakes that had been fixed during the maintenance of the existing framework over several years. Also, we can expect a significant period where we would have to maintain both the new and old framework versions – to provide support for projects, and until the new framework is on feature-parity with the current version.

5.5 Summary

In this chapter, we first selected and described certain existing framework features. Then, we compared the language and dependency management of the existing implementation to an approach using newer technology. We outlined the *Clean Swift* architecture, which serves to strongly separate classes and components using interfaces. We described our generic mechanism for updating data and linking between different modules, as well as for allowing projects to more easily customize their design. Finally, we drew a comparison between the existing system and our new solution, finding that a new implementation could potentially benefit the organization, particularly with a more thoroughly structured architecture, technology more fit for customization, and a smaller overall system size.

6 Conclusion and Future Work

During this project, we first looked to define quality in the context of our project and found that we should focus on non-functional requirements such as maintainability and flexibility.

We then assessed the system's structural quality by investigating the developer's needs, using static analysis, and collecting data on module-level and organization-level framework usage. We found issues with the project's code-level quality, software architecture, and the collaboration structure.

Next, we sought to improve the quality of our existing system. We defined a strategy for branching and versioning in our Git repository. We also set up automatic integration tests between the framework and the most vital client projects to ensure the compatibility of changes in either location.

We worked to increase the test coverage in fundamental framework classes, exploring different strategies for covering classes that were previously not testable. We reduced the number of compiler warnings through adding nullability annotations and removing or replacing deprecated code. We then performed refactoring in significant locations and in smaller methods that contained deprecated code.

In the last part, we started a new implementation of select features. We analyzed the existing implementation and specified an architecture that could support our requirements. We also discussed the benefits and drawbacks of our solution in comparison with the existing implementation.

6.1 Recommendation

We want to provide a decision basis to select a strategy for further work, especially for deciding between refactoring the existing system and starting a new implementation. In our work on this thesis, we have seen that both refactoring and rewriting the framework entail costs in time and money.

There are good reasons for either approach, and there exist countless stories of developers struggling with failed rewrites, but also with upgrading legacy systems.

The main benefits of keeping and refactoring the system lie in the years of experience built on – and improvements applied to – the existing codebase. It seems less precarious to keep the current implementation because the company knows it works just well enough for the existing projects. If the developers perform a rewrite, they run into the risk of repeating mistakes they previously spent time ironing out in the current system.

However, we need to acknowledge that many projects that the company creates nowadays do not use the framework at all. We think this is mainly the result of a change in the underlying requirements this system needs to fulfill. Event apps, which were the original purpose of the framework, are now considered small niche projects. While the company extended the code to support an ever-increasing number of features, the system lost its flexibility to the point where it caters well to the event app market, but not to the majority of projects on which the company is working.

We recognize the continued need for a framework that increases commonality between different projects. Using a framework simplifies development, both during the start and maintenance phases of a project. Developers like the ability to expand on an existing system, and having a shared architecture makes collaboration across projects more efficient.

Meanwhile, a rewrite could serve to improve the overall quality of resulting applications through the support of newer features where the current framework is falling behind. For example, we see opportunities for improvement in accessibility, dynamic font sizing, cross-platform support, and design customization. We would also expect the overall system size to be significantly smaller than the current implementation, which could improve performance and maintainability.

A rewrite could also benefit from some of the improvements to the setup we achieved in our refactoring process, for example, the continuous integration workflow. Also, the developers could refer to the existing Android version of the framework for architectural guidance. This implementation seems to be more universally applicable to projects.

We expect that a new implementation could benefit the company in an overall shorter amount of time than refactoring the existing system. The company is currently starting most projects from scratch – a small toolkit could start to be used in new projects relatively quickly, and expanded over time.

Taking all these reasons into account, if the company wants to build a toolkit on which it can base most of its projects, we would, therefore, recommend rewriting the parts of the system which the company still considers useful.

However the project team may decide, a considerable reengineering challenge lies ahead. Seeing as though both approaches come with their risks and opportunities, in the context of a rapidly developing field of technology, the only wrong option may be to keep standing still.

6.2 Future Work

For the company maintaining the system, the next steps could be setting the strategic direction by establishing shared priorities and goals, as well as determining responsibilities for maintaining the existing, and possibly designing a new system [DDN03, p.19ff.].

Further possible actions that we found during our work should also be considered, for example, setting up a bug tracker for the framework instead of handling bugs on a project level could provide metrics to improve the analysis of the software quality.

Developing a new implementation would open up the possibility of further comparison between the new and old implementation, possibly revealing learnings about either system's traits, benefits, and drawbacks.

Lastly, more research should be made in the area of techniques to improve testability, such as expanding on delegate testing and finding similar opportunities in other patterns to facilitate the maintenance of similar systems.

Acknowledgement

I want to thank Prof. Dr. Oscar Nierstrasz for the supervision and guidance throughout this thesis, and to everyone else who provided their support, insight, and valuable feedback.

Bibliography

[App18] APPLE INC.: *Cocoa Core Competencies.* https://developer.apple.com/library/archive/documentation/General/Conceptual/DevPedia-CocoaCore/Delegation.html, April 2018

[Ber11] BERNERZEITUNG.CH/NEWSNETZ: *Gurtenfestival Buddy-App: ein Muss für Openair-Fans.* https://www.bernerzeitung.ch/region/bern/gurtenfestival-buddyapp-ein-muss-fuer-openairfans/story/23865190, July 2011

[Ber20] BERN, Canton of: *Commercial Register.* https://be.chregister.ch/cr-portal/auszug/auszug.xhtml?uid=CHE-116.029.116, January 2020

[CBN13] CHEN, L. ; BABAR, M. A. ; NUSEIBEH, B.: Characterizing Architecturally Significant Requirements. In: *IEEE Software* 30 (2013), March, Nr. 2, S. 38–45. http://dx.doi.org/10.1109/MS.2012.174. – DOI 10.1109/MS.2012.174. – ISSN 1937–4194

[Cha13] CHAPPELL, David: *The Three Aspects of Software Quality.* http://davidchappell.com/writing/white_papers/The_Three_Aspects_of_Software_Quality_v1.0-Chappell.pdf, 2013

[DDN03] DEMEYER, S. ; DUCASSE, S. ; NIERSTRASZ, O.: *Object-Oriented Reengineering Patterns.* 2003. – available at: http://scg.unibe.ch/download/oorp/ (Jan. 2020)

[Fea04] FEATHERS, Michael C.: *Working Effectively with Legacy Code*. Prentice Hall, 2004

[Fow99] FOWLER, Martin: *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999

[Gar84] GARVIN, David A.: What Does ”Product Quality” Really Mean? In: *Sloan Management Review* (1984)

[Git20] GITLAB INC.: *Introduction to GitLab Flow*. `https://docs.gitlab.com/ee/topics/gitlab_flow.html`, 2020

[Law15] LAW, Raymond: *Clean Swift iOS Architecture for Fixing Massive View Controller*. `https://clean-swift.com/clean-swift-ios-architecture/`, 2015

[Leh80] LEHMAN, Manny M.: On understanding laws, evolution and conservation in the large-program life cycle. In: *Journal of Systems and Software* 1 (1980), S. 213–221. `http://dx.doi.org/10.1016/0164-1212(79)90022-0`. – DOI 10.1016/0164–1212(79)90022–0

[Mar12] MARTIN, Robert C.: *The Clean Architecture*. `https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html`, August 2012

[Mes07] MESZAROS, Gerard: *xUnit Test Patterns*. First Edition. Addison-Wesley, 2007. – see also `http://xunitpatterns.com/Test%20Spy.html`.

[PM15] PRESSMAN, Roger S. ; MAXIM, Bruce R.: *Software Engineering – A Practitioner’s Approach*. 8th Edition. McGraw-Hill, 2015

[Som16] SOMMERVILLE, Ian: *Software Engineering*. 10th Edition. Pearson, 2016

A

Anleitung zu wissenschaftlichen Arbeiten

In this chapter, we want to show how we built a productive toolchain for iOS application development and deployment. We use various tools to perform tasks such as linting, dependency management, continuous integration, and more. We will give an overview of tools that developers can install on their local machines and then go on to server applications, integrating everything to build a streamlined workflow. A visualization of the complete workflow can be seen in Figure A.1 on Page 52.

A.1 Linting

Linters analyze source code to provide developers with feedback about code styling issues and other potential problems that can be diagnosed with rule-based recognition. The original lint utility was written for the C programming language, but today a multitude of similar tools exists for different platforms and languages.

For iOS development, such tools exist both for the classic Objective-C as well as the newer Swift programming language. A well-known tool for Objective-C is *OCLint*, which runs on macOS and Linux.[1] In Swift development, a commonly used tool is *SwiftLint*, which can also be integrated into the Xcode IDE.[2]

Typically, we customize the default rules for SwiftLint to reduce the number of false positives and better reflect our workflow. To achieve this, we place a `.swiftlint.yml` file in the project root folder. Our most commonly modified rules are `line_length`, `force_cast` and `trailing_whitespace`. We feel that those rules can be overly limiting. For example, restricting our line length to 80 characters does not provide a sufficient benefit to us to justify the added amount of work.

A.1.1 Autocorrection

A significant advantage of using linting tools is the ability to autocorrect both new and existing files. During our refactoring work, we used the `clang-format` command to clean up files where we found the code formatting to be inconsistent. This tool ships along with the clang LLVM compiler on macOS. We used the command `clang-format -i <file_name>` to achieve this automatic formatting.

Similarly, we can use the `swiftlint autocorrect` command to format Swift files to comply with our styling conventions.

A.2 Dependency Management

For iOS development, the most commonly used tools for dependency management are *CocoaPods*, *Carthage*, and the *Swift Package Manager*.

While Carthage primarily focusses on resolving and downloading dependencies, with developers having to add the files to their project manually, CocoaPods uses a more elaborate system to import dependencies into an Xcode workspace automatically. Similarly,

[1]See also `http://docs.oclint.org`.
[2]See also `https://github.com/realm/SwiftLint`.

Swift Package Manager is integrated directly into Xcode and allows developers to add dependencies by merely specifying a repository URL and rule for version handling.

In the existing framework version, we use CocoaPods to import code. Client projects list their dependencies in a `Podfile`[3], while framework modules use a `Specfile`[4] to make their code importable using CocoaPods.

While a Podspec can be made available publicly through the CocoaPods Spec Repository, since our framework is closed source, we opted to use a private Spec Repo to publish our code internally. We host this repository on our private GitLab instance.

Developers can also choose to import a dependency hosted on their local machine by specifying the source for a Pod using a relative local path. This mechanism is especially useful for framework development because the dependency's source code can be edited directly from the IDE, as opposed to importing the dependency from a remote repository, in which case only the header files are visible and the code cannot be edited.

A.3 Build Pipelines

We use *fastlane* to simplify our build process with automated scripts.[5] This Ruby tool can be used for iOS, Android, or cross-platform mobile development. Developers use a `Fastfile` in which they specify actions to automate various tasks like installing dependencies, linting, building, and deploying the project.

Using this approach, we can build multiple workflows, or *lanes*, for different build environments. For example, we can distribute a new version to testers, or upload an app – including metadata and screenshots – to the App Store. This setup is especially useful because the same script can be used both on the developers' machines and on build runners controlled by a Continuous Integration system. There are hundreds of actions and integrations available; for example, *fastlane* can directly send notifications to Slack.

An abridged version of the `Fastfile` in our framework repository looks like this:

[3]See also `https://guides.cocoapods.org/syntax/podfile.html`.
[4]See also `https://guides.cocoapods.org/syntax/podspec.html`.
[5]See also `https://docs.fastlane.tools`.

```
desc "Run static analysis/linting of framework"
lane :lint do
  cocoapods
  scan
  sh("bash xccov-to-sonarqube-generic.sh
    ../Build/Logs/Test/*.xcresult > out/cov.xml")
  swiftlint(
    reporter: "json",
    output_file: "./fastlane/out/swiftlint.results.json"
  )
  sonar
end

desc "Trigger integration build in projects"
lane :trigger_integration do |params|
  triggers = YAML.load_file('triggers.yml')
  triggers.each do |t|
  sh("curl -X POST -F token=#{t[:token]} -F ref=#{t[:ref]}
    <gitlab-url>/api/v4/projects/#{t[:id]}/trigger/pipeline")
  puts "Triggered Build { #{t[:name]}"
  end
end
```

A `Fastfile` in a project requires different actions than in the framework. Depending on whether we want to deliver the project to testers, or just run static tests, we run a different lane.

```
desc "Build and run integration tests"
lane :integration_build
  match(readonly: true)
  # Load dependencies using latest framework version
  cocoapods(podfile: "./IntegrationTests")
  # Run tests
  scan(scheme: "MyProject", build_for_texting: true)
end

desc "Build and deploy to testers"
lane :deploy_updraft do
  build
  updraft
  post_build_notification
end

lane :build do
  # Load required values from keychain
  match(readonly: true)
  cocoapods # Import dependencies
  gym # Build project
end
```

A.4 Continuous Integration

We have set up continuous integration through our version management system, *GitLab*. This system interacts directly with the Git repository and is easily set up by adding a `.gitlab-ci.yml` file at the project root. This file sets the build actions and the branches or merge requests for which they are triggered.

An example configuration file looks like this:

```
stages:
  - deploy_to_testers
  - run_integration_tests

updraft:
  stage: deploy_to_testers
  script:
    - fastlane deploy_updraft --env updraft-staging
  tags:
    - ios-dev
  only:
    - testing
  except:
    - triggers
    - schedules
  when: always

integration:
  stage: run_integration_tests
  script:
    - pod deintegrate && pod clean
    - fastlane integration_build --env updraft-staging
  tags:
    - ios-dev
  only:
    - triggers
    - schedules
  when: always
```

We see two build pipelines here. The `updraft` pipeline would be used to distribute a new build to testers through our app delivery tool, *Updraft*, described in Section A.7. We added the `integration` pipeline as part of our work on this thesis. This stage is exclusively run through triggers or schedules. For example, a webhook can trigger a project build once developers push a new version of the framework to GitLab. A schedule is used to build the project nightly against whichever is the latest framework version.

This setup ensures that changes in the framework are tested with client projects quickly and that developers get notified if a change they made in their project is incompatible with the latest framework version.

To enable continuous integration for iOS projects, the build runners (machines that are used for building) must support the Xcode Command Line Tools, for example, the `xcode-build` command. This toolset is only available for macOS, restricting us to using Macs to run these builds. In our setup, we run two Mac Minis as build runners. Both host the GitLab macOS runner application and are registered in GitLab with the `ios-dev` tag so that iOS projects are only assigned to these machines.

A.5 Pipeline Feedback

In paid tiers of GitLab, a feature called multi-project pipelines allows build processes to trigger builds in other projects. The build status of these downstream pipelines is then fed back to the original trigger, giving developers feedback about these dependent builds. This feature would neatly enable our project build integration functionality, as described in Section 4.3. Unfortunately, since we are using the open-source GitLab Community Edition, we do not have access to this feature.

As a workaround, we perform an additional action during the framework CI build to trigger project builds using the GitLab API. This means that project builds will now run automatically as part of the framework build process. However, developers do not get any feedback about the build status using this approach.

We extended our implementation with *Danger*,[6] a tool that can interact with merge requests to perform additional checks and that can add feedback in GitLab as comments. For each project, GitLab provides static URLs to an SVG file indicating the build status.[7] Developers commonly use this in README documents to show CI results in a project overview.

Using a simple `markdown` step in Danger, we add a comment with the badges for triggered projects to the merge request that triggered the build process. This way, developers can see the build status for any downstream pipelines in their merge request, as well as jump directly to any involved project by clicking the badge image.

[6] See also `https://danger.systems`.

[7] See also `https://docs.gitlab.com/ee/user/project/badges.html`.

A.6 Static Analysis

We previously described our usage of *SonarQube* as static analyzer in Section 3.1.2. Since Objective-C and Swift projects can only be analyzed using the paid Developer Edition, we set up a new instance using our existing Docker infrastructure.

Integrating the tool into the build pipeline was a two-step process. A basic integration needs a `sonar-project.properties` file placed in the project root directory, which sets basic properties such as server URL, but also custom settings like excluded directories. An analysis is triggered through the fastlane `sonar` action in the build script. SonarQube should now run an analysis every time the CI pipeline runs.

In a second step, we added code coverage analysis. Since we have a mixed Swift and Objective-C project, SonarQube relies on an abstracted, generic representation of coverage. We achieve this by adding a *build wrapper* to our build steps in the `lint` fastlane. This step produces a file that can be interpreted by SonarQube.

A.7 Continuous Delivery

As the last step in our build process, we distribute project builds to our testers via a custom tool named *Updraft*.[8] This system allows testers to install new app versions directly using a web link instead of having to go through the App Store review process, which can take hours or even days. An integration with fastlane[9] allows developers to direct uploads of newly built app binaries.

[8]See also `https://getupdraft.com`.
[9]See also `https://github.com/appswithlove/fastlane_tools`.

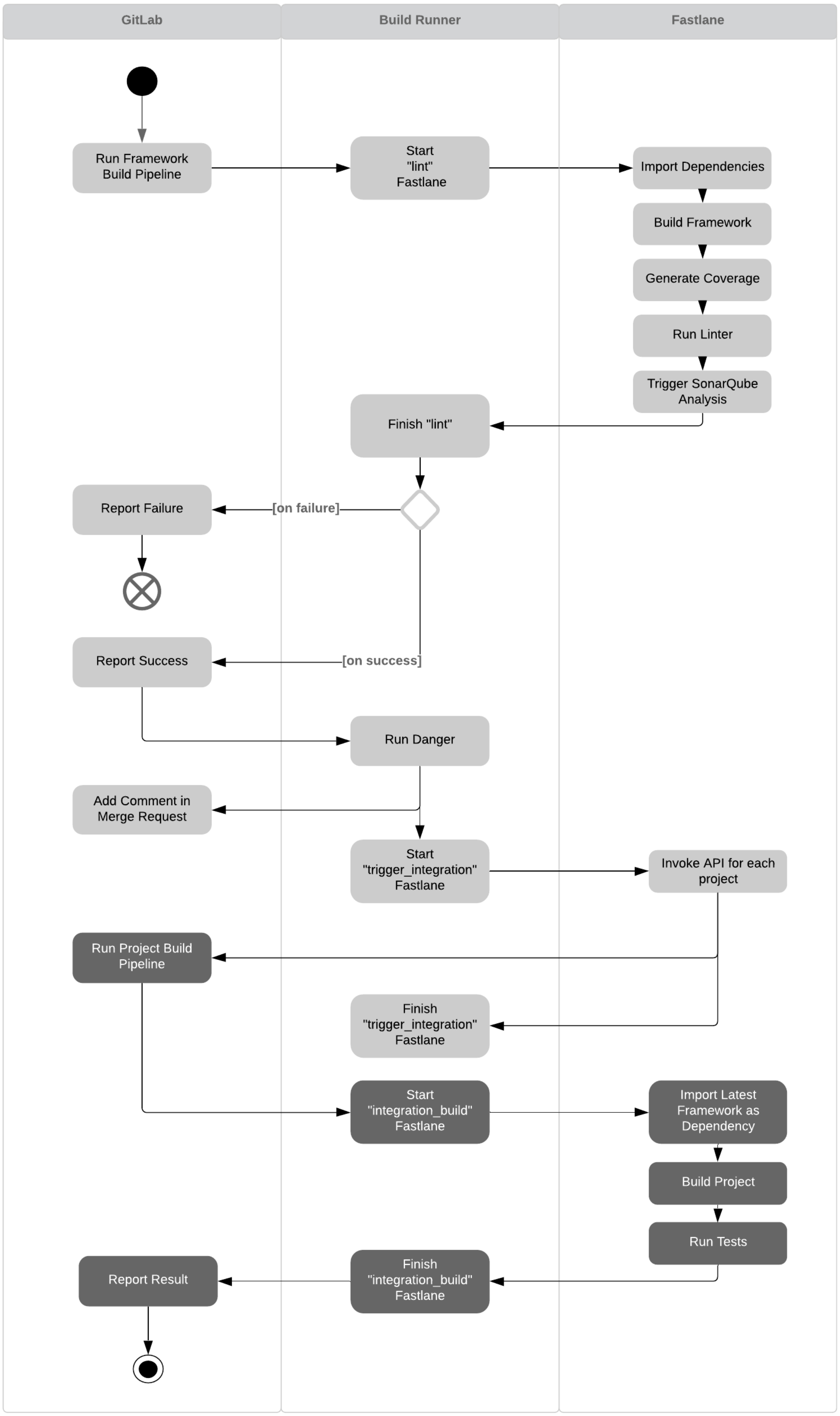

GitLab
Build Runner
Fastlane
Run Framework Build Pipeline
Start "lint" Fastlane
Import Dependencies
Build Framework
Generate Coverage
Run Linter
Trigger SonarQube Analysis
Finish "lint"
Report Failure
[on failure]
Report Success
[on success]
Run Danger
Add Comment in Merge Request
Start "trigger_integration" Fastlane
Invoke API for each project
Run Project Build Pipeline
Finish "trigger_integration" Fastlane
Start "integration_build" Fastlane
Import Latest Framework as Dependency
Build Project
Run Tests
Report Result
Finish "integration_build" Fastlane

Figure A.1: Complete Framework Build Workflow. Light backgrounds represent framework build steps. Dark backgrounds represent project-level steps.

B

Additional Data

B.1 Dependency Visualization

We used a tool to list and visualize our framework's dependencies as CocoaPods imported them. For this, we prepared a test project and added all available framework modules in the project's Podfile.

A prerequisite for the following steps is the installation of the Graphviz software.[1]

We used an open-source tool named `cocoapods-dependencies`[2] to traverse through the dependency tree and create a DOT[3] file, which contains the resolved dependencies.

[1]Open-source, available at `https://graphviz.org`.

[2]Available at `https://github.com/segiddins/cocoapods-dependencies`.

[3]See also: `https://graphviz.gitlab.io/_pages/doc/info/lang.html`.

An optional step we performed is reducing the dependency graph to omit *transitive dependencies*. This step removed any connections from $A \rightarrow C$ where $A \rightarrow B$ and $B \rightarrow C$ also applies, reducing the total number of links to make the figure more readable.

Finally, we rendered the DOT file into a PNG image. The commands we used, sequentially, were:

```
pod dependencies --graphviz

tred Podfile.gv > Podfile_reduced.gv

dot -T png -O Podfile_reduced.gv
```

Using this approach, we created the image shown in Figure C.4. We also show the graph with transitive dependencies in Figure C.5.

B.2 Change Frequency

We analyzed the framework's repository to find the files where developers made changes most often. To achieve this, we used a method described on StackOverflow.[4]

By running the command described in the linked answer, we found the files changed most frequently were `ABNetworkEngine` (63), `ABAppDelegate` (51), `ABBusinessEngine` (50), `ABDataAccessLayer` (39), and `ABFAttachmentsModule` (33).

B.3 Swift Package Manager

This tool for dependency management is closely integrated with the Swift Programming Language and, therefore, modern iOS development. Developers can import Swift packages by specifying a repository URL, or a local path, in their Xcode project. SPM resolves the correct library version, as well as any additional dependencies.

To make a library available for usage with SPM, maintainers simply need to add a `Package.swift` file and structure their code into a *Sources* and a *Tests* folder.

The following is an example of a `Package.swift` file from the `Info` module, which specifies a dependency on the `Core` module, which is locally available:

[4]Available at `https://stackoverflow.com/a/18594249` (Jan. 2020).

```
// swift-tools-version:5.1

import PackageDescription

let package = Package(
    name: "Info",
    platforms: [.iOS(.v11)],
    products: [.library(name: "Info", targets: ["Info"])],
    dependencies: [.package(path: "../Core")],
    targets: [
        .target(
            name: "Info",
            dependencies: ["Core"]),
        .testTarget(
            name: "InfoTests",
            dependencies: ["Info"]),
    ]
)
```

C
Figures and Tables

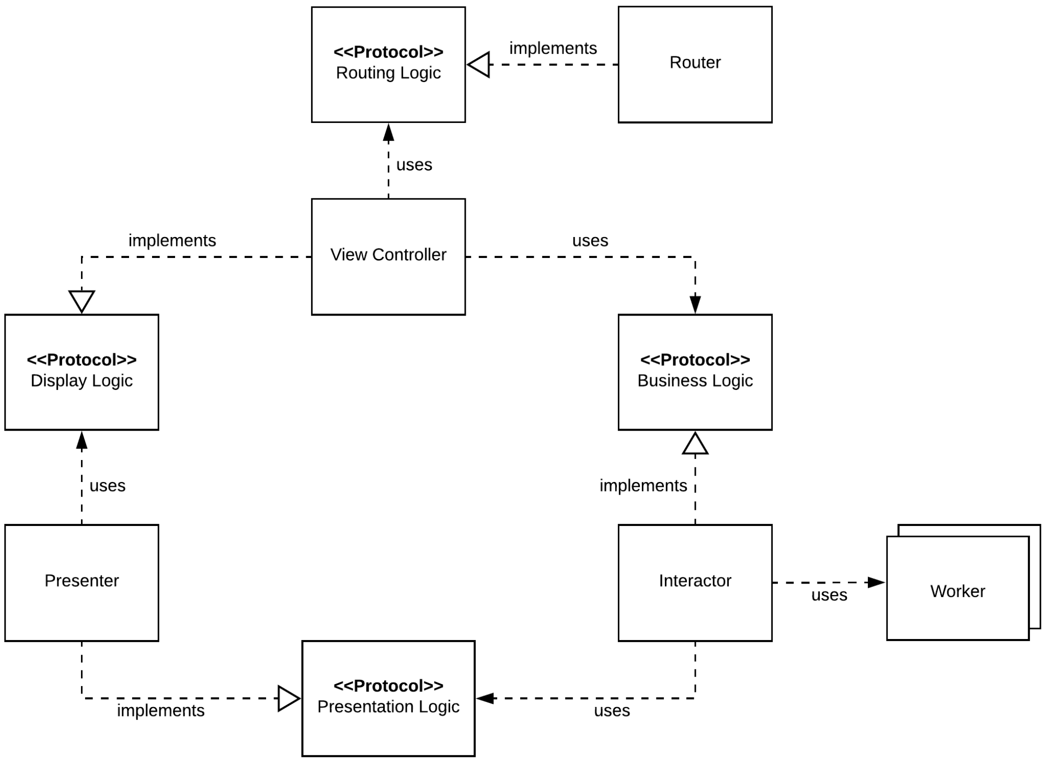


Figure C.1: VIP cycle with Router and Worker

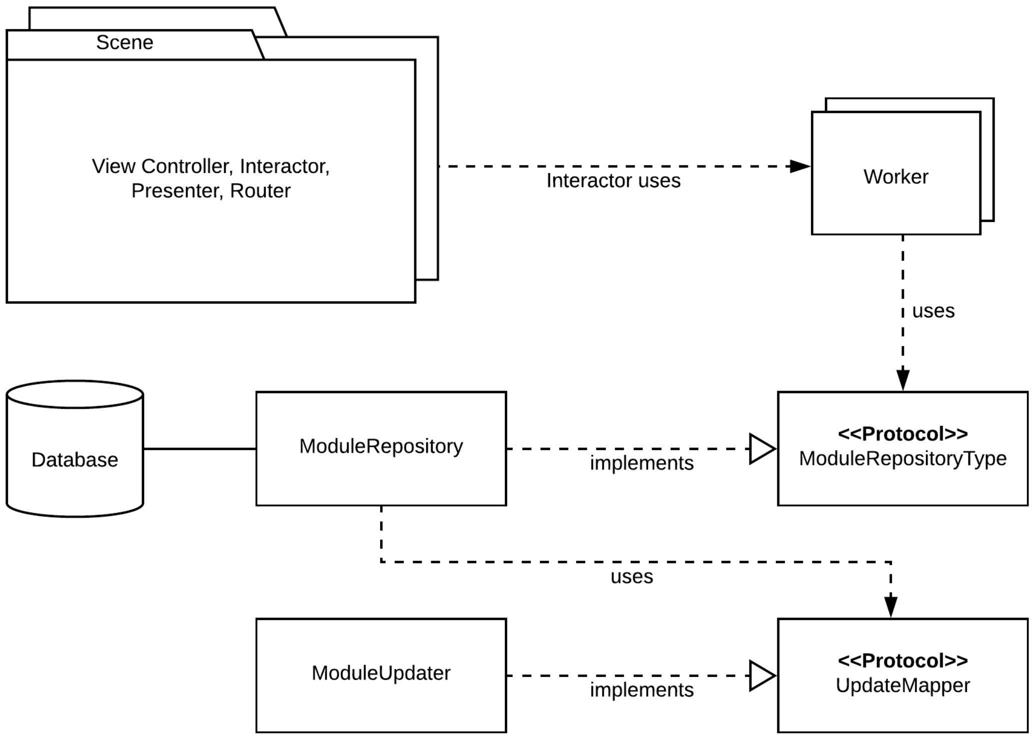

Scene
View Controller, Interactor, Presenter, Router
Interactor uses
Worker
uses
Database
ModuleRepository
implements
<<Protocol>> ModuleRepositoryType
uses
ModuleUpdater
implements
<<Protocol>> UpdateMapper

Figure C.2: Architecture of a Data Module

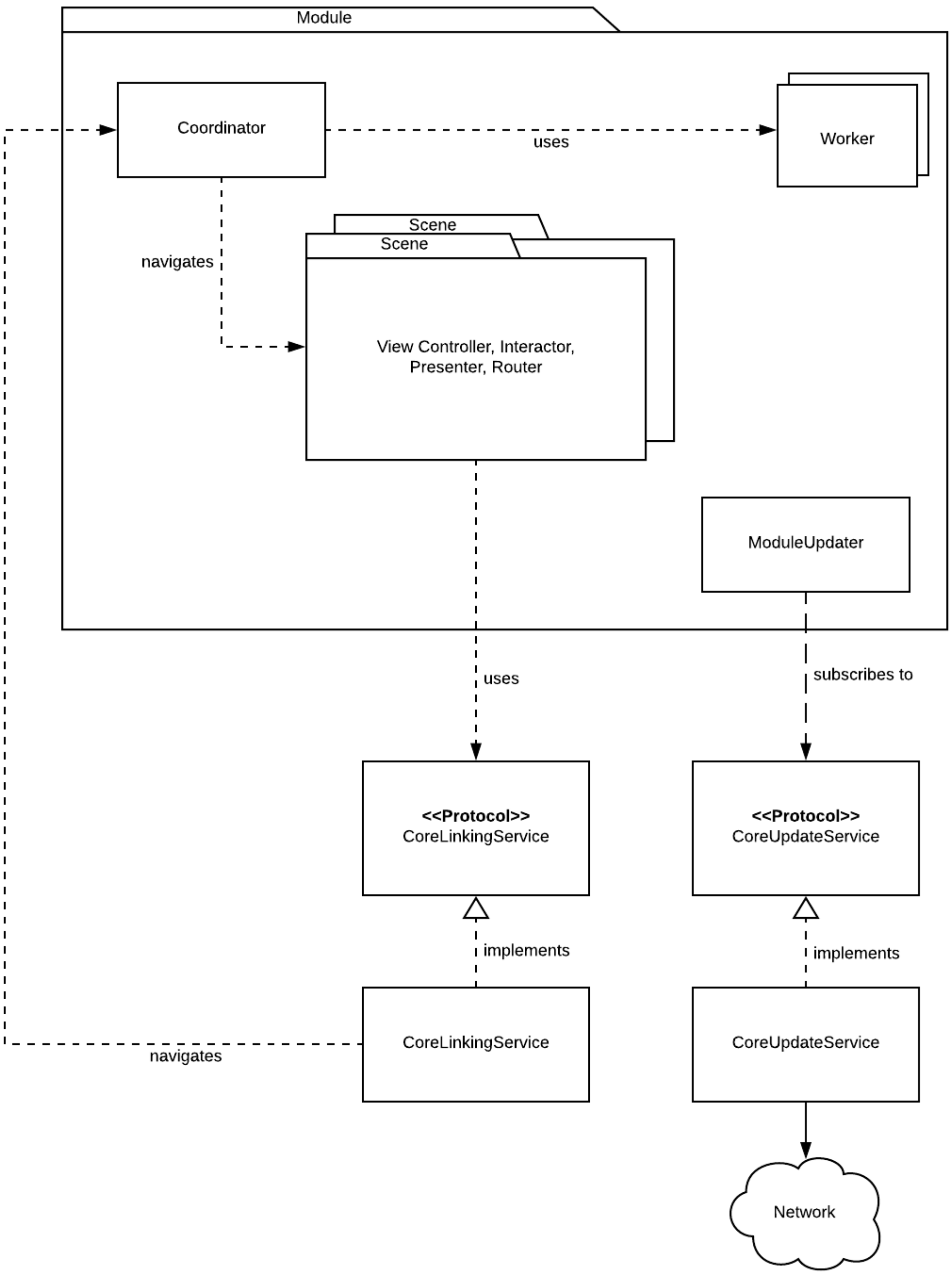


Figure C.3: App architecture with Core and relation between scenes

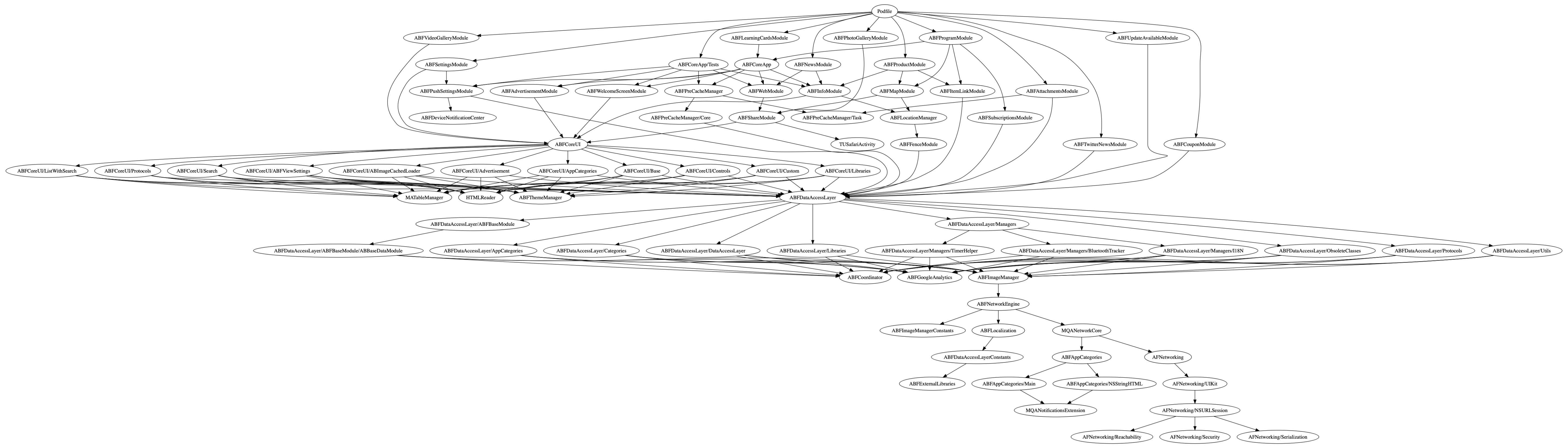


Figure C.4: Reduced dependency tree of framework module imports

(Also available at `https://raw.githubusercontent.com/ast3150/bt-public/master/images/Podfile_reduced.png`)

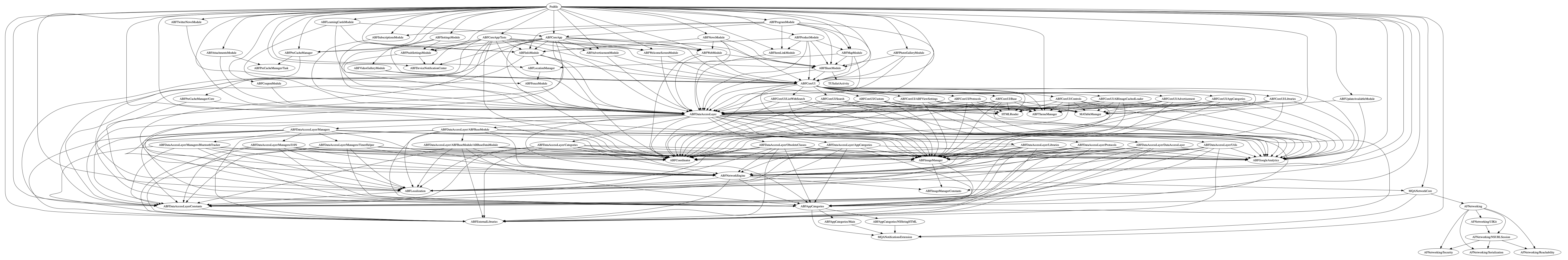


Figure C.5: Full dependency tree of framework module imports

(Also available at `https://raw.githubusercontent.com/ast3150/bt-public/master/images/Podfile_full.png`)

Table C.1: Problems mentioned by developers and possible solution approaches, ordered by origin and estimated effort to implement.

#	Origin	Problem	Possible Solutions
1	Code	Many warnings are generated at compile-time because code needs to be updated to use newer language features (notably, code annotations regarding nullability need to be added)	Add Nullability Annotations, fix other warnings
2	Code	Unintended consequences when making changes	Set up a test project and continuous integration, add integration tests
3	Code	Missing class comments and code comments in complex methods	Refactor complex methods and add comments
4	Code	Outdated code which is marked as deprecated but cannot be removed due to internal dependencies	Remove dependencies and deprecated code
5	Code	Customizing screens is hard, changing shared screen implementations can have unintended side effects	Change architecture and feature scope of framework
6	Framework	Missing Documentation of architecture	Add documentation; instruct developers and enforce through quality gates
7	Framework	Usage of Objective-C over Swift	Migrate or rewrite components
8	Framework	Too much project-specific or outdated logic in framework	Move some logic to projects, remove code that is no longer needed
9	Organization	No code reviews are conducted during development, impacting code quality	Set up quality tools, enforce the use of merge requests and reviews
10	Organization	No versioned releases, projects reference directly to commits	Change tools used for dependency management, establish development roadmap
11	Organization	Multiple versions and branches in production use and need to be maintained	Define branching conventions, instruct developers, migrate older projects
12	Organization	Too few know-how exchanges between Moqod and Apps with love	Enforce regular exchange through periodic team meetings
13	Organization	Old projects are hard to migrate to newer framework versions	Establish versioned releases, set up regular project maintenance schedules in SLAs

Table C.2: Quality attributes (high and medium priority) with subjective assessments, possible metrics and matching developer feedback.

Attribute	Assessment	Possible Metrics	Matching Feedback
Repairability	Fixes often involve working in multiple modules and projects; it's hard to figure out where a defect comes from and which consequences a fix has.	Time to fix, Coverage, Coupling, # Integration Tests, Warnings	2, 8, 10, 11, 13
Evolvability	Needs to be improved; the framework is falling behind technologically because migrations need a lot of work.	Deprecated Usages, Stale Branches	2, 5, 7, 8, 10, 11, 13
Verifiability	Automated tests are missing and integration with projects can't be tested easily.	Coverage, # Integration Tests	1, 2, 4, 8, 9, 10, 11, 13
Understandability	Complex system and missing documentation means the system is hard to understand.	Cyclomatic Complexity, Comment Density, Linting Violations	3, 6, 7, 8, 9, 10, 11
Robustness	Unexpected error states are often not handled. Objective-C is less robust than Swift.	Crash Logs	1, 4, 7
Productivity	Creators are more productive than newer developers, but could be improved for all.	Time Spent	2, 3, 5, 6, 7, 9, 11