Modular Architecture on iOS

Building Large Scale Modular iOS Apps And Frameworks

Dedication

"To my Mom and Dad, because they really tried."

&&

"To all my non-tech friends whom I am fixing problems with not enough disk space, printers, forgotten passcodes etc."

&&

"To all passionate engineers who are solving tough problems on a daily basis with a smile, it is great pleasure for everyone to work with you!"

&&

"Finally, to whoever my current girlfriend is…"

About the author

Cyril Cermak is a software engineer by heart and author of this book. Most of his professional career was spent by building iOS apps or iOS frameworks. Beginning by Skoda Auto Connect App in Prague, continuing building iOS platform for Freelancer Ltd in Sydney and as of now being an iOS TechLead in Stuttgart for Porsche AG. In this book, Cyril describes different approaches for building modular iOS architecture so as some mechanisms and essential knowledge that should help you, the reader, to decide which approach would fit in the best or should be considered on your project.

Reviewers

Joerg Nestele
… Ask Joerg whether he wants be the reviewer.

Table Of Contents

- CI/CD
    - Scripts execution
- GitFlow
- Profesional Experience
    - Joerg Nestele: iOS Tech Lead - Porsche AG
    - Majd Alfhailly: Senior iOS Engineer - Blinkst
    - Daniel Williams: Senior iOS Engineer - Canva
    - Aldrich Co: iOS Tech Lead - Freelancer Ltd.
    - Gleb Arkhipov: iOS Tech Lead - STRV

Introduction

In the software engineering field, people are going from project to project, gaining different kind of experience out of it. Especially, on iOS mostly the monolithic approaches are used. In some cases it makes totally sense, so nothing against it. However, scaling up the team, or even better, team of teams on monolithically built app is horrifying and nearly impossible without some major time impacts on a daily basis. Numerous problems will rise up, that are limiting the way how iOS projects are built itself or even on the organisational.

Scaling up the monolithic approach to a team of e.g 7+ developers will most likely result in hell. By hell, I mean, resolving xcodeproj issues, where in the worst case both parties renamed or edited and deleted the same file, touched the same {storyboard|xib} file, or typically both worked on the same file which would resolve in classic merge conflicts. Somehow, those issues we all got used to live with…

The deal breaker will come when your PO/PM/CTO will come to the team to announce that they are planning to release a new flavour of the app or to divide the current app into two separate parts. Afterwards, the engineering decision needs to be made. Either, to continue with this monolithic approach, create different targets, assign files towards the new flavour of the app and continue living in multiplied hell and hoping that some requirement such as shipping core components of the app to a subsidiary or open-sourcing it as a framework will not come.

Not surprisingly, a better approach would be to start refactoring the app into a modular approach, where each team can be responsible for particular frameworks (parts of the app) that are then linked towards those apps. That will take time as it will not be easy to transform but the future of company's mobile engineering will be faster, scalable, maintainable and even ready to distribute or open source some parts of it to the outer world.

Another scenario could be, that you are already working on an app which is done in a modular way but your app takes at around 20 mins to compile. As it is a huge legacy codebase that was in development past 8 or so years and linked every possible 3rd party library along the way. The decision was made to modularise it with Cocoapods therefore, you cannot link easily already pre-compiled libraries with Carthage and every project clean you can take double shot of espresso. I had been there, trust me, it is another type of hell, definitely not a place where anyone would like to be. I described the whole migration process of such project here. Of course, in this book you will read about it in more detail.

Nowadays, as an iOS tech lead, I am often getting asked some questions all over again from new teams or new colleagues regards those topics. Thereafter, I decided to sum it up and tried to get the whole subject covered in this book. I hope it will help developers working on such architectures to gain the speed, knowledge and understanding faster.

I hope this introduction gave enough motivation to deep dive further into this book.

What you Need

The latest version of Xcode for compiling the demo examples, brew to install some mandatory dependencies, Ruby and Bundler for running scripts and downloading some Ruby gems.

//TODO: Provide a tutorials for installing those softwares

What is this book about

This book describes essentials about building modular architecture on iOS. You will find examples of different approaches, framework types, pros and cons, common problems and so on. By the end of this book you should have a very good understanding of what benefits will bring such architecture to your project, whether it is necessary at all or which would be the best way for modularising the project.

At the end of this book, you can read experiences from the top notch iOS engineers working across numerous different projects from different countries and continents.

What is this book NOT about

SwiftUI.

Modular Architecture

Modular

`adjective - employing or involving a module or modules as the basis of design or construction: "modular housing units"`

In the introduction, I briefly touched the motivation for building the project in the modular way. To summaries it, modular architecture will give you much more freedom when it comes to the product decisions that will influence the overall app engineering. Such as, building another app for the same company, open-sourcing some parts of the existing codebase, scaling the team of developers and so on. With the already existing mobile foundation, it will be done way faster and cleaner.

To be fair, maintaining such software foundation of a company might be also really difficult. By maintaining, I mean, taking care of the CI/CD, old projects developed on top of the foundation that was heavily refactored in the meantime, legacy code, keeping it up-to date with the latest development tools and so on. No need to say, that on a very large project, this could be a work of one standalone team.

This book describes different approaches of building such large scalable project by an example: The software foundation for the International Space Station.

Design

In this book, I chose to use the architecture that I think is the most evolved. It is a five layer architecture that consists of those layers:

- Application
- Domain
- Service
- Core
- Shared

Each layer is explained in the following chapter.

Nevertheless, the same principles can be applied for example for feature oriented architecture, where the layers could be defined as:

- Application

- Feature
- Core
- Shared

Now let us have a look on what those layers will consist of.

Layers

Let us have a look now on each layer and its purpose. Modules within the layers are then demonstrated with the example in the following chapter.

Application Layer

Application layer consists of the final customer facing products; applications. Applications are linking domains and via configurations and Scaffold module glueing all the different parts together. In such architecture, the App is basically a container that puts pieces together.

Nevertheless, App might also contain some necessary Application implementations like receiving push notifications, handling deep linking, permissions and so on.

Patterns, that will help achieve such goals will be described later.

For example an app in an e-commerce business could be `The Shop` for online customer and `Cashier` for the employees of that company.

Domain Layer

Domain layer links services and other modules from layers below and uses them to implement business domain needs of the company or the project. Domains will contain for example the users flow within the particular domain part of the app. So as, the necessary components for it like; view controllers, views, models and view models. No need to say, that it depends on the teams preferences and technical experience which pattern will be used for creating screens. Personally, the reactive MVVM+C is my favourite but more on that later.

For example a domain in an e-commerce app could be a `Checkout` or `Store Items`.

Service Layer

Services are modules supporting domains. Each domain, can link several services in order to achieve desired outcomes. Such service will most likely talk to the backend, get the data out of it, persist them in its own storage and expose them to domains.

For example a service in an e-commerce app could be a `Checkout Service` which will handle all the necessary communication with the backend so as proceeding with the credit card payments etc.

Core Layer

Core layer is the enabler for the whole app. Services will link the necessary modules out of it for e.g communicating with the backend or persisting the data. Domains will link e.g ui components for easier implementation o screens and so on.

For example a core module in an e-commerce app could be `Network` or `UIComponents`

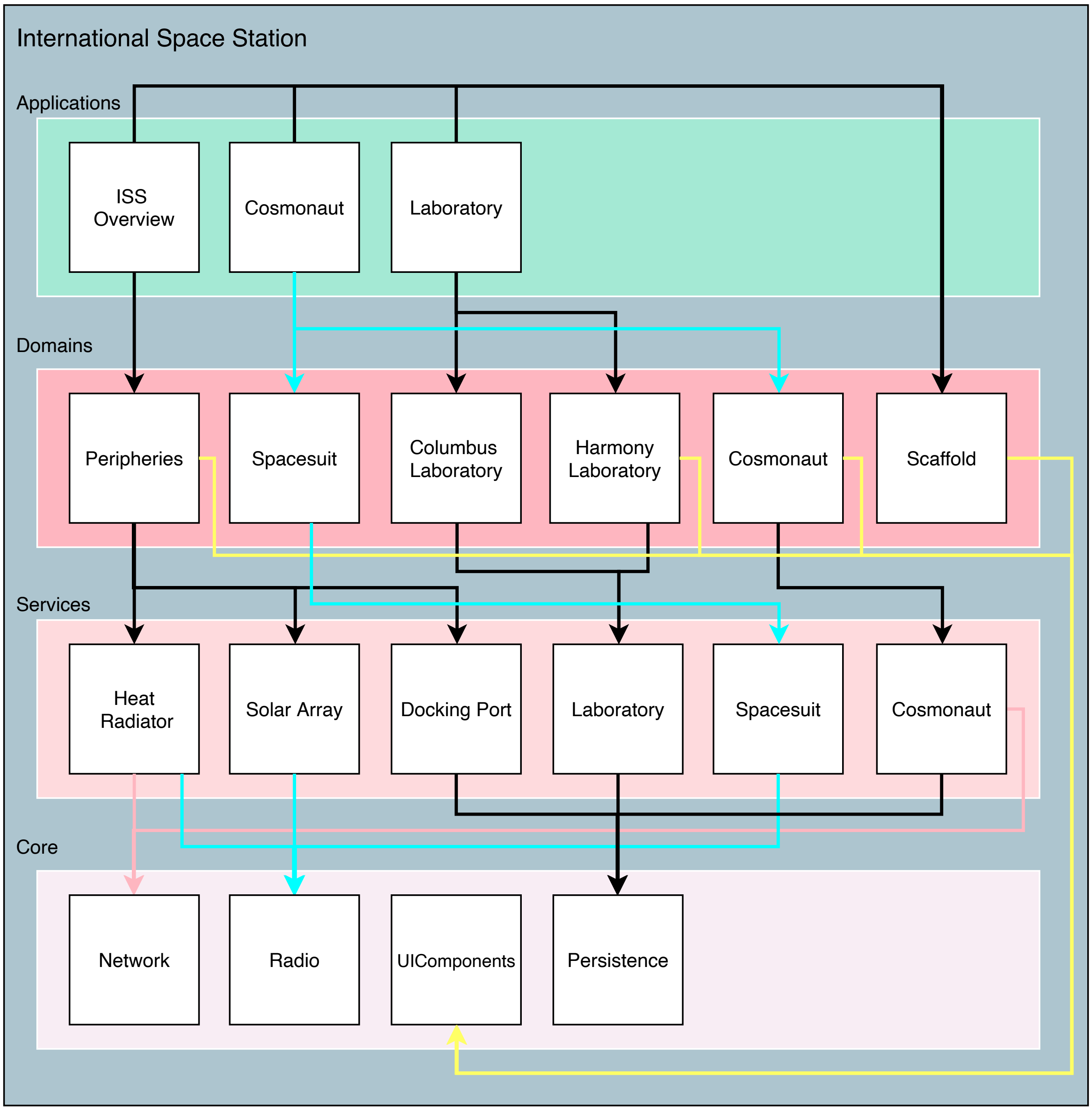
Shared Layer

Shared layer is a supporting layer for the whole framework. It can happen that this layer might not need to exist. However, a perfect example for the shared layer is logging. Where even core layer modules will want to log some output. That would potentially lead to duplicates, which could be solved by the shared layer.

For example a shared module in an e-commerce app could be `Logging` or `AppAnalytics`

Example: International Space Station

Now in this example, let us have a look at how such architecture could really look like for International Space Station. Diagram below shows the four layer architecture with the modules and linking.



The example has three applications.

- `ISS Overview` : app that shows astronauts the overall status of the space station
- `Cosmonaut` : app where Cosmonaut can control his spacesuit so as his supplies and personal information
- `Laboratory` : app from which the laboratories on the space station can be controlled

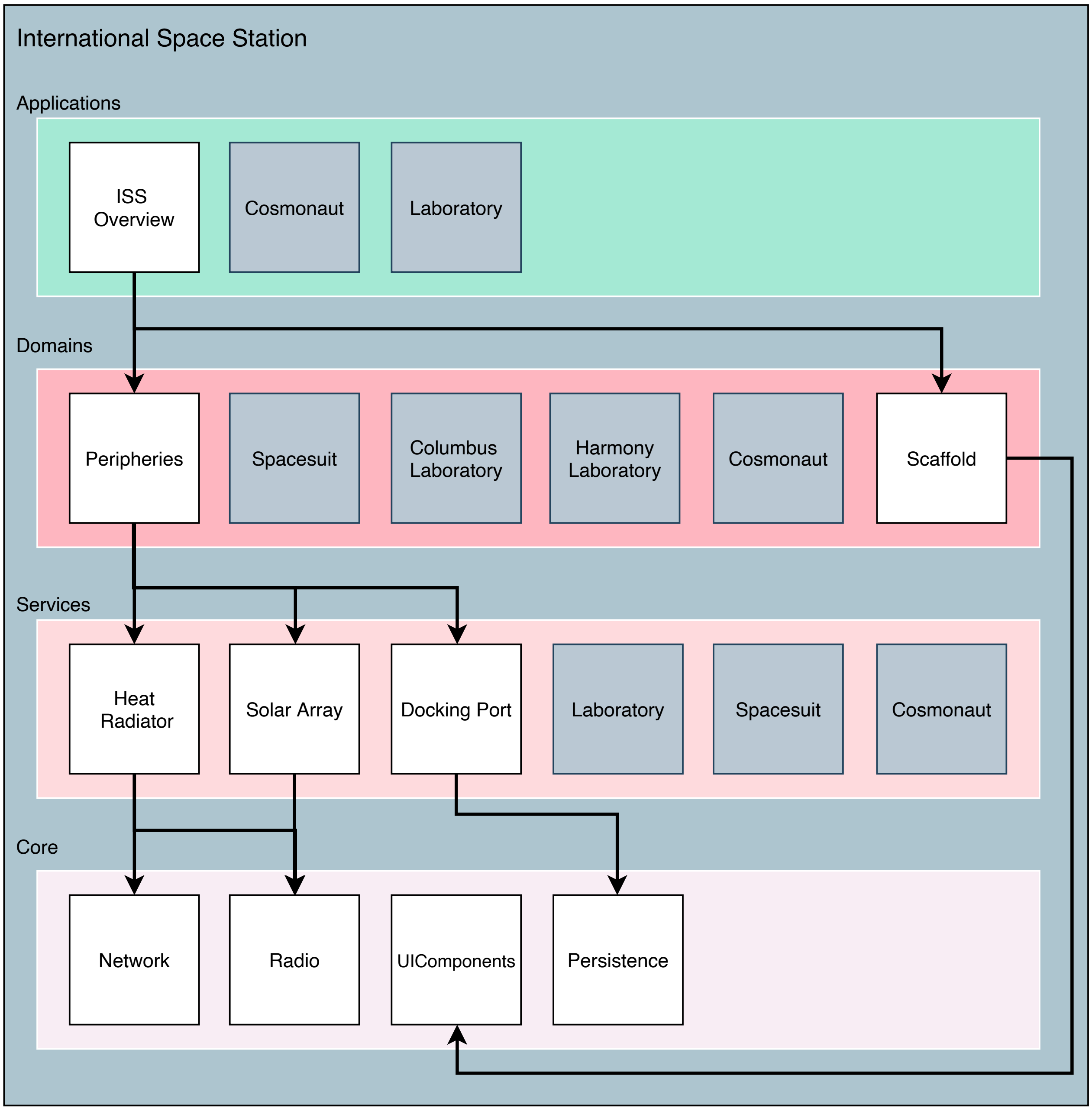
As described above, all apps are linking Scaffold module which provides the bootstrapping for the app as the app itself is the container.

ISS Overview

`ISS Overview` app links `Peripheries` domain which implements logic and screens for peripheries of the station.

The `Peripheries` domain links `Heat Radiator`, `Solar Array` and `Docking Port` services from whom the data about those peripheries are gathered so as `UIComponents` for bootstrapping the screens development.

Linked services are using `Network` and `Radio` core modules which are providing the foundation for the communication with other systems via network protocols. `Radio` in this case could implement some communication channel via BLE or other technology which would connect to the solar array or heat radiator. Diagram below describes the concrete linking of modules for the app.

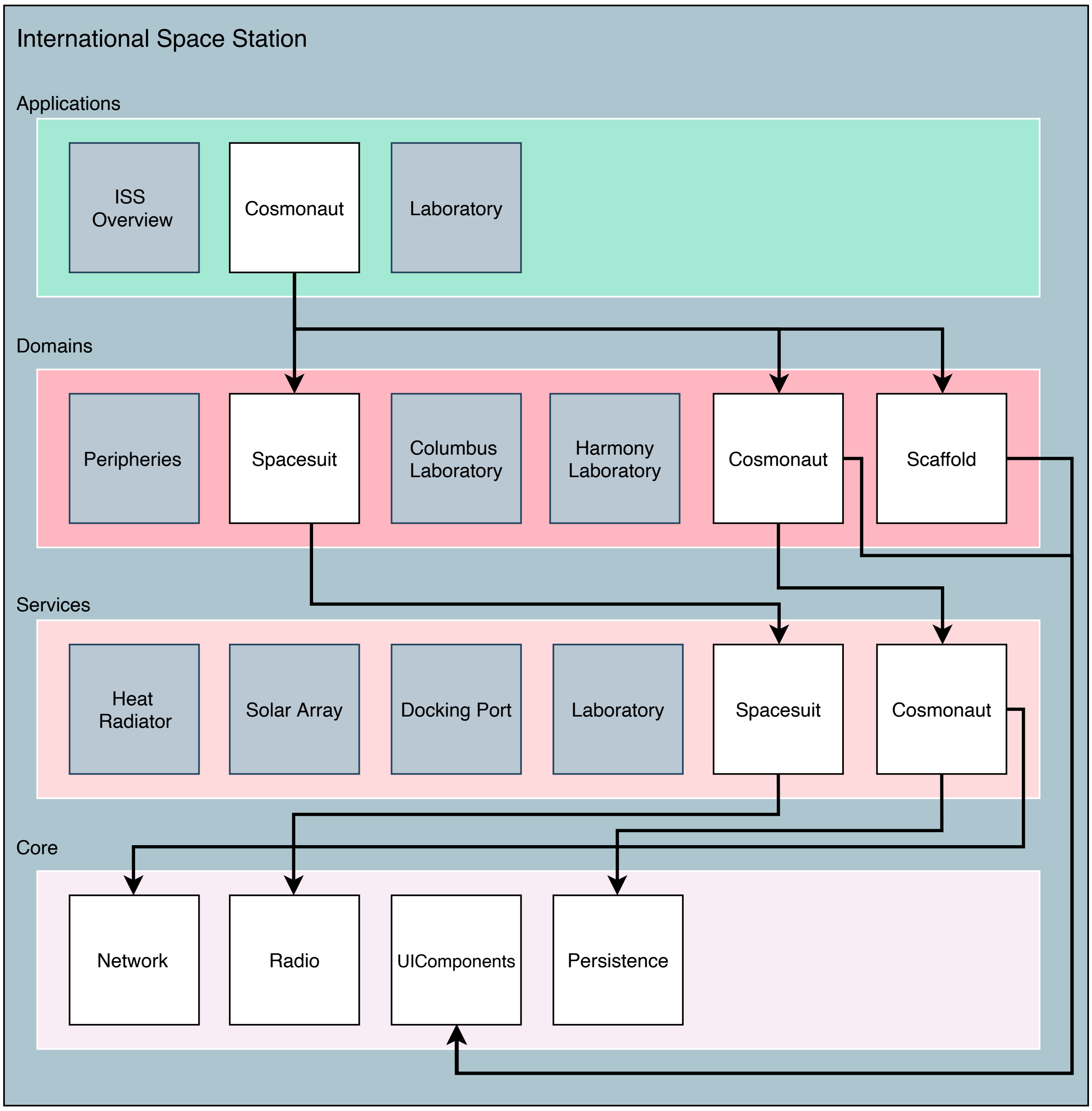


Cosmonaut

`Cosmonaut` app links `Spacesuit` and `Cosmonaut` domains. Same as for every other domain, each module is responsible for screens and users flow through the part of the app.

`Spacesuit` and `Cosmonaut` domains link `Spacesuit` and `Cosmonaut` services that are providing data for domain defined screens so as `UIComponents` who are providing the UI parts.

`Spacesuit` service is using `Radio` for communication with cosmonauts spacesuit via BLE or other type of radio technology. `Cosmonaut` service is using `Network` for updating Huston about the current state of the `Cosmonaut` so as `Persistence` for storing the date of the cosmonaut for offline usage.



Laboratory

I will leave this one for the reader to figure it out.