Transferleistung 4

Noah Peeters

April 2020

Contents

List of Figures

List of Tables

1 Introduction

One of the basic best practices in software engineering is reusing code. This can be done on multiple levels: extracting reused logic into a function which can be called from multiple places to extracting whole packages. Package managers try to simplify managing those extracted packages because this can be a lot of manual work and a complicated process with a growing number of dependent packages. Especially when working in bigger teams or using continuous integration, it is crucial to automate this process.

On the one hand, it helps users of the reusable code to easily install and even more important to update the dependencies throughout the project's lifetime. This makes it easier for users to include dependencies especially if the dependencies have many transitive dependencies and keep the project maintainable. On the other hand, creators of reusable code are forced to provide a package which is understandable by the package manager. This often includes versioning the code and providing useful metadata. This makes it easier for creators to share their reusable code in a usable way. Therefore, package managers can be seen as a two sided market where the two user groups have a huge overlap in many cases because package creators can rely on other packages as well.

First, the theoretical background of package managers is presented as it serves as the foundation for the following sections. After that, a method from another paper to evaluate software is presented and adapted to evaluate package managers. Then, the three package managers CocoaPods, Carthage and the Swift Package Manager, which are the different options for package managers on iOS currently, are analyzed using the presented method. Finally, the results are used to evaluate the different options. The findings show that there is not one package manager which should be used in all cases as each have advantages and disadvantages. Therefore, a structured approach to choose one of the package managers considering the different requirements of a project is developed.

2 Package Manager

Package managers can be used in different areas. They can be divided into *language-specific* and *language-agnostic* package managers. However, as stated in [22], the name language-specific is no longer fitting because many package managers often support multiple languages. Instead, they are restricted to ecosystems which support multiple languages. The same is true for the package managers for iOS as they do support multiple languages like Swift and Objective-C. To overcome this naming issue, on the internet other terms like *system-level* and *application-level* dependency managers are sometimes used[24]. As those are currently not used in research, this paper continues to use *language-specific* and *language-agnostic*.

Environment Packages are always installed in an environment. Packages can interact with other packages in the same environment but are separated from other environments. Different package manager systems define the environment differently. For example, some systems use the whole computer system while others work on a project level.

Packages The most important concept of Packages are different versions. During the life cycle of software, it is often released multiple times which in terms of packages is called a version of that package. To refer to a version of a package, the term package is often used.

Each version of a package consists of metadata and the artifact. Additionally, versions can depend on specific versions or one of multiple possible version of another package. This is called a *dependency*. Besides dependencies, a package can have *transitive dependencies*. Transitive dependencies are all dependencies required by a package either as a direct dependency of that package or as a dependency of a dependency. The opposite direction is called *reverse dependency* or *transitive reverse dependency*.

Dependency Solving Due to the different constraints of the versions of dependencies, not all versions of different packages can be installed simultaneously in one environment. Dependency solving is also known as *planning an upgrade path*[1].

For example, when version 1 of *Package A* requires version 1 of *Package C* while version 1 of *Package B* requires version 2 of *Package C*, it is not possible to install version 1 of both *Package A* and *Package B*. In that case, one of the packages must be removed or a different version of *Package A* and/or *Package B* must be found.

Package Managers There are multiple terms to describe packages managers. The most common ones are *package manager* and *dependency manager*. Package managers have multiple functions related to the management of packages. These can be categorized using the following approach[1]:

- Package managers must retrieve packages with metadata and artifacts. Sources can be remote repositories on servers, or locally stored packages.
- Package managers must be able to perform dependency solving
- Package managers might provide options to fine-tune the final result

- Package managers must be able to deploy the packages by adding them to or removing them from the environment.

3 Methods

In [23], an approach to evaluate simulation software based on hierarchical criteria is presented. The highest level of criteria is the vendor, the user and the software. Not all of the presented criteria can be used to evaluate different package managers as some of the criteria are specific to simulation software. However, the approach can be adapted by replacing the simulation software specific criteria with package manager specific criteria. All of the other criteria can be reused.

The package manager specific criteria can be derived from the different requirements and classifications for package managers outlined in section 2. The resulting hierarchy is visualized in Figure 1. New criteria added to the original criteria are <u>underlined</u>.

This framework includes three types of criteria. Quantifiable criteria like the price, non-quantifiable criteria like the documentation, and non-comparable criteria like the required operating system[23]. The type of the criteria determine how they influence the final result. On the one hand, non-comparable criteria are exclusion criteria when it comes to choosing one of the options. For example, if the operating system is not compatible with the user's requirements, the package manager cannot be used. On the other hand, quantifiable and non-quantifiable criteria can be both used to find the best of multiple options and can serve as an exclusion criterion by defining acceptable values.

4 Package Manager for iOS

For iOS app development, multiple incompatible package managers exist. They can be used together but they aren't aware of the packages installed by other package managers and can therefore not include this during the dependency solving.

The three available package managers for iOS are *CocoaPods*, *Carthage* and *the Swift Package Manager*. In the following sections, those will be analyzed based on the criteria described in section 3.

4.1 Vendor

Both CocoaPods and Carthage are open source project developed by voluntary teams and many other contributors. The Swift Package Manager is also an open source project which is mainly

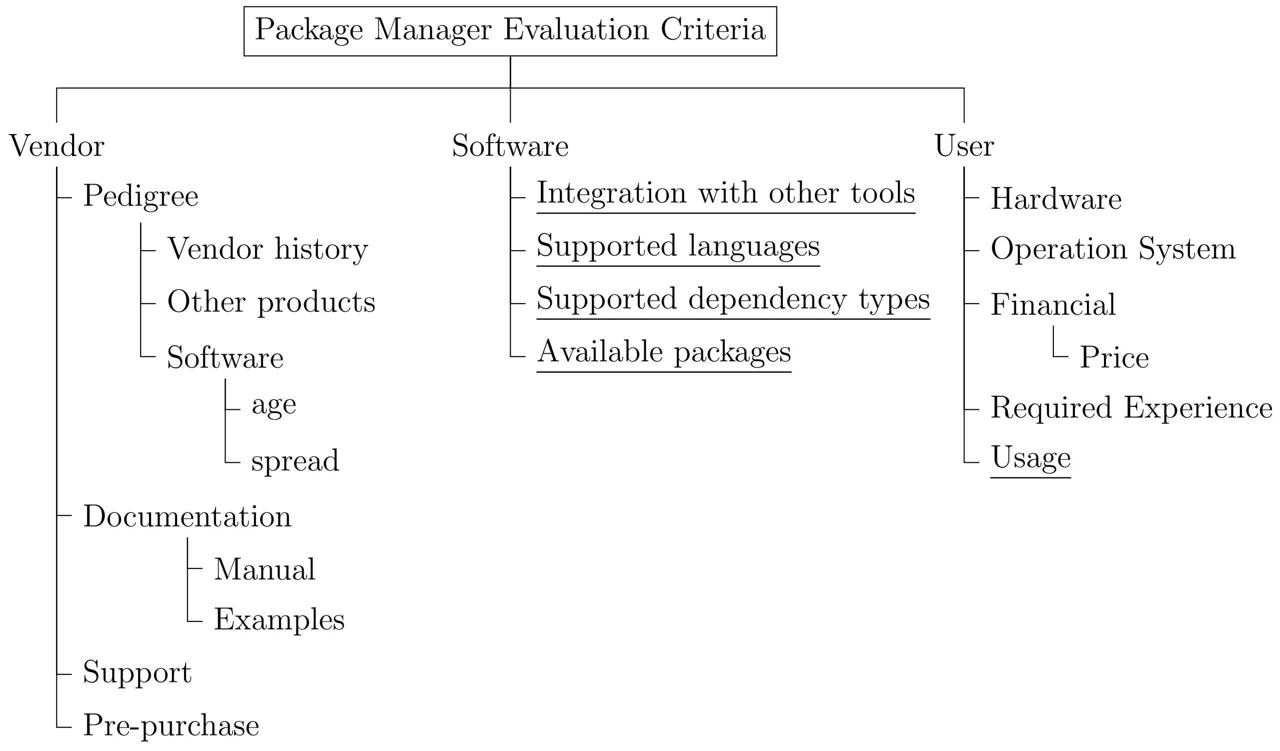


Figure 1: Hierarchical criteria for package manager evaluation

developed by Apple Inc. employees. However, there are many other contributors as well.

4.1.1 Pedigree

The pedigree of the vendor can be used to evaluate the reliability of the vendor and the software itself[23].

Vendor History and Other Products As CocoaPods and Carthage are developed by voluntary teams, there are no other products or a history of a vendor. The Swift Package Manager was created by Apple Inc., a well-known and the most valuable brand worldwide[21], together with the Swift Programming Language, which it was designed for. Apple is also the vendor of iOS and iOS compatible devices as well as other developer tools like Xcode which are required for the iOS development.

Age The first version of CocoaPods was release in September 2011 with the version number 0.0.1[18]. However, the first release with a description and the first release listed in the change log was released in October 2014 with the version number 0.3.0[19], roughly three years after the initial release. The first version of Carthage was released in November 2014 with the version number 0.1[25]. The first version of the Swift Package Manager was release in March 2016 for Swift 2.2[3]. Therefore, the oldest package manager for iOS is CocoaPods while the Swift Package Manager was created the most recent.

Spread Only CocoaPods publishes information about the number of the apps using it as the package manager. The official website states that "over 3 million apps"[13] use CocoaPods. The other package managers do not publish any data about the number of users.

Instead, the watches, stars and forks in Table 1 published by GitHub for the official repositories can be used to estimate the popularity of the package managers[8], [11], [14]. Additionally, big companies use different package managers to distribute their packages. Google uses CocoaPods[20] for their packages while Apple uses the Swift Package Manager[9].

Overall, the Swift Package Manager seems to have the least spread while CocoaPods seems to be a bit more popular than Carthage. However, these are only estimates gathered from unreliable data.

Package Manager	Watches	Stars	Forks
CocoaPods	548	12389	≈ 2200
Carthage	382	13628	≈ 1400
the Swift Package Manager	386	7663	942

Table 1: Watches, Stars, and Forks of the package managers on GitHub (April 16, 2020)

4.1.2 Documentation

All of the package managers come with an online documentation. However, they greatly differ in extend. While Carthage only provides 4 markdown files of which only 2 provide usage information for the user, CocoaPods has detailed descriptions of every supported features with examples and guides. The documentation of the Swift Package Manager is somewhere in between.

Package Manger	Documentation
CocoaPods	`https://guides.cocoapods.org`
Carthage	`https://github.com/Carthage/Carthage/tree/master/Documentation`
The Swift Package Manager	`https://github.com/apple/swift-package-manager/tree/master/Documentation`

4.1.3 Support

None of the presented package managers have an official support, neither free nor paid, and it is not possible to get an Service Level Agreement (SLA). In case one needs help, one can try to get help from the community or from someone of the developers. CocoaPods additionally has a mailing list which is "occasionally [used] for support"[16].

4.2 Software

In this section, the different features of the package managers are analyzed.

4.2.1 Integration with other tools

Xcode is the Integrated development environment provided by Apple for the development of iOS app which also bundles the required system libraries.

Carthage has almost no integration for Xcode. The list of required dependencies and where to get them from must be provided with a file called `Cartfile`, similar to the following one:

```
github "apple/swift-nio" ~> 2.16.0
github "apple/swift-nio-ssl" ~> 2.7.1
github "apple/swift-nio-http2" ~> 1.11.0
```

The package manager must be started in the command line each time the `Cartfile` is changed. The compiled frameworks must be added to the project manually including the linking and embedding in the final application. Carthage only offers a command to help with embedding "necessary bitcode-related files and dSYMs"[11] to enable debugger symbols.

CocoaPods uses a file called `Podfile` which is similar to the `Cartfile`:

```
target "MyApp" do
    use_frameworks!
    pod "SwiftNIO", "~> 2.16"
    pod "SwiftNIOSSL", "~> 2.7"
    pod "SwiftNIOHTTP2", "~> 1.11"
end
```

The package manager must be started in the command line or with the optional graphical user interface each time the `Podfile` is changed. CocoaPods creates a Xcode workspace file from the `Podfile` which wraps the original project file. The workspace is configured to handle all of the steps to use the dependencies in the project. From that point on, the workspace must be used instead of the project file so that the compiler uses the installed dependencies. If the additional workspace file is not desired, CocoaPods can be instructed to only install the dependencies and the integration into the project must be done manually.

The Swift Package Manager uses a file called `Package.swift`:

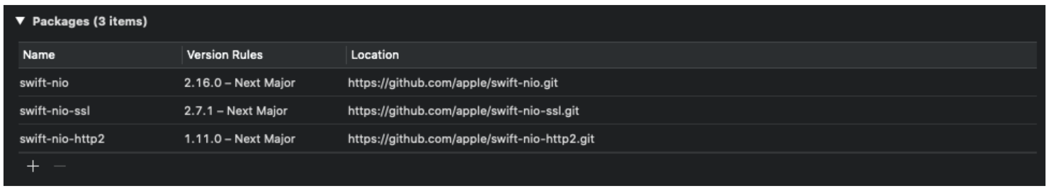
```
// swift-tools-version:5.1
import PackageDescription

let package = Package(
    name: "MyApp",
    products: [
        .executable(name: "MyApp", targets: ["MyApp"])
    ],
    dependencies: [
        .package(url: "git@github.com:apple/swift-nio.git",
                 from: "2.16.0"),
        .package(url: "git@github.com:apple/swift-nio-ssl.git",
                 from: "2.7.1"),
        .package(url: "git@github.com:apple/swift-nio-http2.git",
                 from: "1.11.0")
    ],
    targets: [
        .target(name: "MyApp",
                dependencies: ["NIO", "NIOSSL", "NIOHTTP2"])
    ]
)
```

In contrast to the other package managers, Xcode has built in support for the Swift Package Manager. For iOS and macOS projects, Xcode provides a GUI to interact with as shown in Figure 2. With the GUI, dependencies can be added, removed and modified. The Swift Package Manager is automatically triggered by Xcode whenever this is required.

4.2.2 Supported Languages

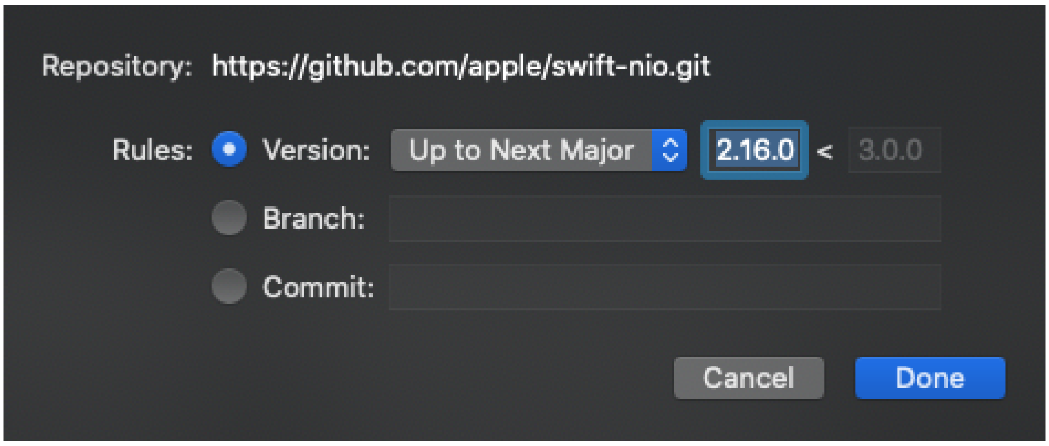
All of the presented package managers can be classified as a language-specific package manager. As already discussed in section 2, this does not mean only one language is supported. In the case of CocoaPods and Carthage, any Xcode project with an importable artifact can be turned into



▼ Packages (3 items)

Name	Version Rules	Location
swift-nio	2.16.0 – Next Major	https://github.com/apple/swift-nio.git
swift-nio-ssl	2.7.1 – Next Major	https://github.com/apple/swift-nio-ssl.git
swift-nio-http2	1.11.0 – Next Major	https://github.com/apple/swift-nio-http2.git

(a) Overview



Repository: https://github.com/apple/swift-nio.git

Rules: [✓ Version:] Up to Next Major 2.16.0 < 3.0.0
Branch:
Commit:

(b) Details

Figure 2: Integration of the Swift Package Manager in Xcode

a package[12], [15]. In practice, packages contain Swift and C-Languages (C, C++, Objective-C, and Objective-C++). The Swift Package Manager supports the same languages with the limitation that one module can either contain Swift or C-Languages. Mixing them is currently not possible[2]. However, this is only relevant when creating packages as the iOS project can still contain mixed languages.

4.2.3 Supported Dependency Types

On iOS, there are multiple ways to include code from dependencies which can be used by package managers. The first most straight forward solution would be to add the dependency's code directly to one of the existing targets of the app. However, this has many disadvantages. Dependencies with many files are hard to handle especially if they contain additional resources. Additionally, this can lead to naming conflicts when adding the code of multiple dependencies. Therefore, this is not used by any of the package managers.

There are four methods to include dependencies by combining *static* and *dynamic* together with *library* and *framework*. Libraries can only contain source code. Resources cannot be added to libraries[1]. Frameworks may contain both source code and resources[4]. Static means,

[1]Of course it is possible to include raw data in the source code. However, it is not possible to have a file in the file system in the final app bundle which is the default way to ship resources with iOS apps.

the dependency is added directly to the executable but is handled as a separate module while dynamic dependencies are placed next to the executable[2]. Dynamic dependencies increase the load time of the app compared to their static counterparts because the kernel loads the dynamic loader which first has to find the dynamic dependencies in the file system[5]. Another limitation of dynamic dependencies is that they only work on iOS 8 and newer although that should not be a significant as the adoption rate of iOS 8 and newer already reached 98% in September 2017 and likely increased since then[6]. CocoaPods support all 4 of these methods. Carthage supports static and dynamic frameworks though some additional steps are required for static frameworks[10].

The Swift Package Manager uses swift modules to organize the dependencies. Regarding the features swift modules support, they are similar to static libraries.

4.2.4 Available Packages

CocoaPods main central repository is located at `https://github.com/CocoaPods/Specs`. It contains the package specification files of most publicly available packages. As of April 2020, there are over 70 thousand packages listed. For private packages, it is possible to use a custom package repository additionally. Apart from package repositories, there are some other possibilities to include packages like from a local path for development. However, the recommended approach is to use package repositories.

Carthage and the Swift Package Manager do not have a central repository to manage the available packages. Instead, each package has its own git repository. To add a package as a dependency, a URL to the repository must be provided[3]. Therefore, it is difficult to tell how many packages are publicly available.

To get an idea of the magnitute of the number of packages, one can compare the number of tagged repositories on GitHub. As of April 2020, there are 2080 repositories tagged with "cocoapods", 901 repositories tagged with "carthage" and 644 repositories tagged with "swift-package-manager".

Additionally, the Swift Package Manager requires every package (both dependencies and projects which are using the Swift Package Manager to manage dependencies) to have a file called `Package.swift`. When searching on GitHub for files with that name using the query `filename:Package.swift`, about 5800 files can be found. All of these hints suggest that Co-

[2]Dynamic dependencies can be placed at other places in the system as well and be shared by multiple executables. However, on iOS only the system frameworks can be installed outside of an apps bundle. Custom dependencies can only be accessed by the app which included it.

[3]CocoaPods allows to only specify the owner and repository name for packages hosted on `https://github.com`.

coaPods has significantly more packages. However, there is currently not enough data available to make a reliable statement about the exact numbers.

4.3 User

In these sections the packages are analyzed regarding specific user needs and circumstances.

4.3.1 Hardware and Operating System

For iOS development, it is necessary to use *macOS* which is only allowed to run on "Apple-branded Systems"[7]. Therefore, CocoaPods and Carthage only run on macOS as well because iOS development is the only use case they are designed for. On the contrary, the Swift Package Manager is not limited to iOS development and can be used for any Swift project on macOS and Linux. However, using Linux is only an option to develop iOS independent libraries as the iOS toolchain is only available for macOS. This can be helpful when creating a library which should be used both in an iOS app and in a backend to prevent developing the library twice.

4.3.2 Financial

All of the presented package managers are open source. CocoaPods and Carthage are released under the MIT License while the Swift Package Manager is released under the Apache License 2.0. Therefore, all of the package managers are free to use.

4.3.3 Required Experience

When developing iOS apps, the required knowledge depends on the integration with Xcode as discussed in subsubsection 4.2.1.

Both CocoaPods and the Swift Package Manager have a very good integration in Xcode and required almost no knowledge about dependencies as those package managers only require the user to list the packages required and they will configure the project. For Carthage, more knowledge is required as some manual steps are required throughout the process.

Overall the Swift Package Manager requires the least experience as the integration with Xcode provides a *GUI* (Graphical User Interface) and GUIs are generally easier to understand.

4.3.4 Usage

When developing an iOS app, a package manager can be used only to include third party dependencies or to develop and distribute custom dependencies as well. This can be useful when parts of the app's code can be either reused in other apps or be shared with a backend. Of course, all of the presented package managers support both use cases as they are the fundamental use cases of package managers in general as discussed in section 1. While Carthage and CocoaPods do not differ between developing an app or a package except for the additional requirement that packages must be versioned in a git repository, the Swift Package Manager has some additional constraints when it comes to develop a package. In particular, the GUI of Xcode cannot be used and a `Package.swift` must be used instead as discussed in section 4.2.1, and the Swift and C-Languages cannot be mixed as discussed in section 4.2.2.

5 Evaluation

The analysis of the different package managers shows that each one has some strengths and some weaknesses. For example, there are more packages with support for CocoaPods while the Swift Package Manager can be used to develop cross-platform libraries. Therefore, it is not possible to select one of the package managers to use for all projects.

However, it is possible to give recommendations under which conditions which package manager should be used. On approach proposed by [23] is to use the Analytic Hierarchy Process. However, this requires to define priorities for all of the different criteria from section 4 first[17]. As prioritizing the different criteria must be done on a per-project basis, this paper will use the criteria to first derive a list of questions which only exclude some of the package manages under certain answers. This removes the necessity to prioritize and already leads to good result as there are only three package managers.

Pedigree, Documentation, Support:

What kind of support is desired?

Integration in Xcode, Required Knowledge:

Do the developers have knowledge about dependency management for iOS apps or do they prefer to do some manual steps during the setup?

Supported Languages, Usage:

Is it required to develop a package with mixed languages?

Hardware and Operating System, Usage:

Is it required to develop a cross platform package?

Available Packages:

Should as many packages as possible be available in case new packages will be required in the future?

These questions can now be visualized by chaining them in a flowchart as in Figure 3.

Companies might have additional constraints like reducing the number of vendors of third-party software. The company specific policy constraints are not considered in this section and the flowchart. These can be applied to the result of the flowchart to get a final decision.

In some cases, multiple packages managers can be used. In that situation, it is necessary to choose one of those by using another method like the Analytic Hierarchy Process as suggested by [23]. In other cases, no package manager is fitting the constraints of the project. In that situation, one either has to reduce the constraint or use the closest fit and manage some of the dependencies manually.

6 Conclusion

The different package managers presented all have their advantages and disadvantages in certain situation. This paper adapted the hierarchical framework presented by [23] so that it can be used for package manages and applied it to the package managers available for iOS development. After that, it uses a simple approach to propose a method to decide for one of the package managers. Using this method, it is possible to make the decision for a package manager on a per-project basis in an efficient way.

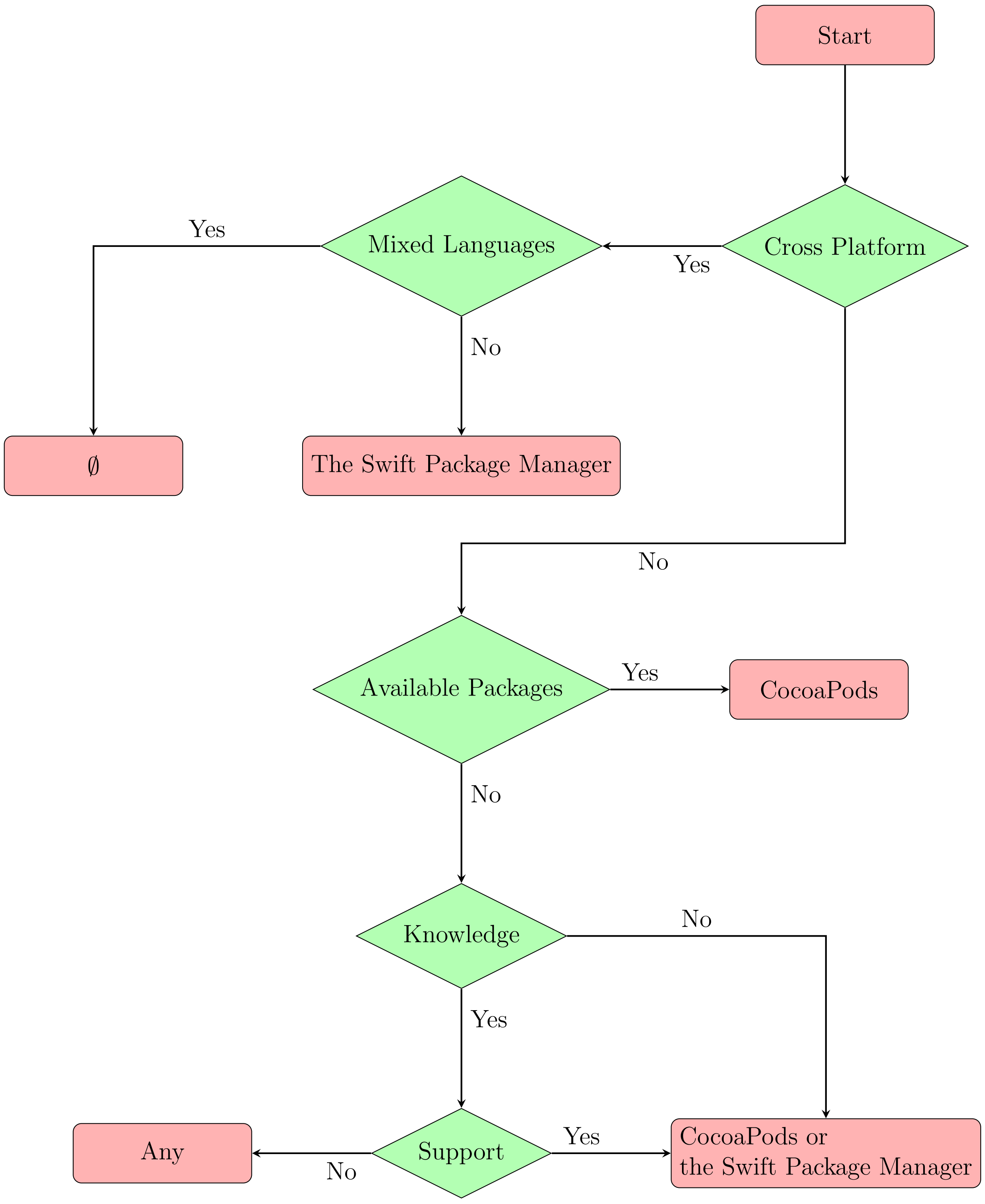


Figure 3: Decision Flowchart

References

[1] P. Abate, R. DiCosmo, R. Treinen, and S. Zacchiroli, "Mpm: A modular package manager," in *Proceedings of the 14th International ACM Sigsoft Symposium on Component Based Software Engineering*, ser. CBSE '11, Boulder, Colorado, USA: Association for Computing Machinery, 2011, pp. 179–188, ISBN: 9781450307239. DOI: `10.1145/2000229.2000255`. [Online]. Available: `https://doi.org/10.1145/2000229.2000255`.

[2] Apple Inc. (Oct. 15, 2019). Creating c language targets, [Online]. Available: `https://github.com/apple/swift-package-manager/blob/master/Documentation/Usage.md#creating-c-language-targets` (visited on 04/30/2020).

[3] ——, (Apr. 16, 2020). Download swift, [Online]. Available: `https://swift.org/download/#releases` (visited on 04/16/2020).

[4] ——, (Feb. 4, 2009). Loading code at runtime, [Online]. Available: `https://developer.apple.com/library/archive/documentation/DeveloperTools/Conceptual/MachOTopics/1-Articles/loading_code.html` (visited on 04/08/2020).

[5] ——, (Jul. 23, 2012). Overview of dynamic libraries, [Online]. Available: `https://developer.apple.com/library/archive/documentation/DeveloperTools/Conceptual/DynamicLibraries/100-Articles/OverviewOfDynamicLibraries.html` (visited on 04/08/2020).

[6] ——, (Feb. 27, 2020). Share of apple devices by ios version worldwide from 2016 to 2019, [Online]. Available: `https://www.statista.com/statistics/565270/apple-devices-ios-version-share-worldwide/` (visited on 04/09/2020).

[7] ——, *Software license agreement for macos catalina*, en, Apple Inc., Oct. 2019. [Online]. Available: `https://www.apple.com/legal/sla/docs/macOSCatalina.pdf` (visited on 04/28/2020).

[8] ——, (Apr. 16, 2020). Swift package manager repository, [Online]. Available: `https://github.com/apple/swift-package-manager/` (visited on 04/16/2020).

[9] ——, (Feb. 18, 2020). Swiftnio readme, [Online]. Available: `https://github.com/apple/swift-nio/blob/master/README.md#getting-started` (visited on 04/28/2020).

[10] Carthage. (Jun. 3, 2018). Build static frameworks to speed up your app's launch times, [Online]. Available: `https://github.com/Carthage/Carthage/blob/master/Documentation/StaticFrameworks.md` (visited on 04/29/2020).

[11] ——, (Apr. 11, 2020). Carthage repository, [Online]. Available: `https://github.com/Carthage/Carthage/` (visited on 04/16/2020).

[12] ——, (Mar. 18, 2020). Supporting carthage for your framework, [Online]. Available: `https://github.com/Carthage/Carthage#supporting-carthage-for-your-framework` (visited on 04/30/2020).

[13] CocoaPods. (Apr. 16, 2020). Cocoapods homepage, [Online]. Available: `https://cocoapods.org` (visited on 04/16/2020).

[14] ——, (Apr. 13, 2020). Cocoapods repository, [Online]. Available: `https://github.com/CocoaPods/CocoaPods/` (visited on 04/16/2020).

[15] ——, (Apr. 2020). Making a cocoapod, [Online]. Available: `https://guides.cocoapods.org/making/making-a-cocoapod.html` (visited on 04/30/2020).

[16] ——, (Apr. 2020). Where can i ask questions? [Online]. Available: `https://guides.cocoapods.org/making/making-a-cocoapod.html#where-can-i-ask-questions` (visited on 04/29/2020).

[17] L. Davis and G. Williams, "Evaluating and selecting simulation software using the analytic hierarchy process," in *Integrated Manufacturing Systems*, vol. 5, MCB UP Ltd, 1994, pp. 23–32.

[18] E. Durán. (Sep. 17, 2011). Release 0.0.1, [Online]. Available: `https://github.com/CocoaPods/CocoaPods/releases/tag/0.0.1` (visited on 04/08/2020).

[19] S. Giddins. (Oct. 29, 2014). Release 0.3.0, [Online]. Available: `https://github.com/CocoaPods/CocoaPods/releases/tag/0.3.0` (visited on 04/08/2020).

[20] Google. (Apr. 16, 2020). Install a google sdk using cocoapods, [Online]. Available: `https://developers.google.com/ios/guides/cocoapods` (visited on 04/16/2020).

[21] Interbrand. (Sep. 2019). Best global brands 2019 rankings, [Online]. Available: `https://www.interbrand.com/best-brands/best-global-brands/2019/ranking/#?listFormat=ls` (visited on 04/27/2020).

[22] H. Muhammad, L. C. V. Real, and M. Homer, "Taxonomy of package management in programming languages and operating systems," in *Proceedings of the 10th Workshop on Programming Languages and Operating Systems*, ser. PLOS'19, Huntsville, ON, Canada: Association for Computing Machinery, 2019, pp. 60–66, ISBN: 9781450370172. DOI: `10.1145/3365137.3365402`. [Online]. Available: `https://doi.org/10.1145/3365137.3365402`.

[23] J. Nikoukaran, V. Hlupic, and R. J. Paul, "Criteria for simulation software evaluation," in *Proceedings of the 30th Conference on Winter Simulation*, ser. WSC '98, Washington, D.C., USA: IEEE Computer Society Press, 1998, pp. 399–406, ISBN: 0780351347.

[24] Sonatype. (Apr. 8, 2020). Out of the wild: A beginner's guide to package and dependency management, [Online]. Available: `https://guides.sonatype.com/foundations/devops/out-of-wild-pkg-dep-mgmt/` (visited on 04/08/2020).

[25] J. Spahr-Summers. (Nov. 18, 2014). Release 0.1, [Online]. Available: `https://github.com/Carthage/Carthage/releases/tag/0.1` (visited on 04/08/2020).