Department of Computer Science
Faculty of Science
Palacký University Olomouc

BACHELOR THESIS

Build-time Dependency Resolution In Swift Language

2022

Tadeáš Kříž

Supervisor:
Mgr. Roman Vyjídáček

Study field: Applied Computer Science, combined form

Bibliografické údaje

Autor:	Tadeáš Kříž
Název práce:	Vyhodnocování stromu závislostí při kompilaci programu v jazyce Swift
Typ práce:	bakalářská práce
Pracoviště:	Katedra informatiky, Přírodovědecká fakulta, Univerzita Palackého v Olomouci
Rok obhajoby:	2022
Studijní obor:	Aplikovaná informatika, kombinovaná forma
Vedoucí práce:	Mgr. Roman Vyjídáček
Počet stran:	59
Přílohy:	1 CD
Jazyk práce:	anglický

Bibliograhic info

Author:	Tadeáš Kříž
Title:	Build-time Dependency Resolution In Swift Language
Thesis type:	bachelor thesis
Department:	Department of Computer Science, Faculty of Science, Palacký University Olomouc
Year of defense:	2022
Study field:	Applied Computer Science, combined form
Supervisor:	Mgr. Roman Vyjídáček
Page count:	59
Supplements:	1 CD
Thesis language:	English

Anotace

Při použití dependency injection frameworků může docházet k chybám za běhu programu v případě nesprávně deklarovaných závislostí. Proto tyto frameworky kontrolují graf závislostí při startu. Slabá reflexe v jazyce Swift nicméně omezuje možnosti frameworků za běhu programu. Některé Swift dependency injection frameworky obsahují pomocný program, spouštěný při kompilaci, pro překonání těchto překážek. Bohužel však často omezují přímo vývojáře, či jinak zhoršují developer experience.

Cílem této práce je tato omezení zmírnit, a to pomocí dvojího přístupu k závislostem. A to podporou implicitních závislostí filtrovaných regulárními výrazy nad názvy tříd, a typově bezpečné API v jazyce Swift pro explicitní deklarování a konfiguraci závislostí.

Vytvořený program analyzuje zdrojový kód v jazyce Swift aby našel chybějící závislosti a cykly závislostí. Ty jsou vývojáři během kompilace nahlášeny jako chyby, což vede ke zvýšení bezpečnosti při kompilaci a celkově lepšímu komfortu pro vývojáře.

Synopsis

Using a dependency injection framework can result in run-time crashes when dependencies aren't declared correctly. Therefore these frameworks verify the dependency graph during startup. Swift's minimal reflection limits what dependency injection frameworks can do at run-time. Some Swift dependency injection frameworks include a companion program, ran during compilation, to circumvent those limitations. However, they are often placing restrictions on the developer, or sacrificing developer experience.

This thesis aims to alleviate these restrictions, using a two way approach to dependencies. Supporting implicit dependencies using regex matching of the class' name, and a type-safe Swift API for explicitly declaring and configuring dependencies.

The program analyzes Swift code to find missing dependencies and dependency cycles. Those are reported as errors during compilation to the developer, resulting in improved compile-time safety and overall better developer experience.

Klíčová slova: injekce závislostí; generátor pro Dip; verifikace závislostí

Keywords: dependency injection; Dip generator; dependency verification

I want to thank my supervisor for his help and guidance. I also want to thank my family and colleagues for their support and understanding during my studies.

Contents

List of Figures

List of Tables

List of theorems

List of Source Codes

1 Introduction

Dependency injection is a popular and recommended component of application development. But with *Swift* still being a young language, there wasn't enough time for the language mature enough to compare with the likes of JVM languages and robust reflection. Therefore compile-time safe solutions for dependency injection are needed to catch developer mistakes and avoid run-time crashes. Catching those mistakes during compilation leads to reduction of time required between making a change in code and seeing results. It can significantly improve developer experience. Compile-time safety also reduces how many errors are encountered by users.

This thesis is intended for Swift developers who already use dependency injection in Swift and recognize the limitations of current solutions, helping them alleviate some or most of the limitations. For those who haven't gotten into dependency injection yet because of the same limitations, this thesis describes dependency injection and why it should be used. With the program resulting from this thesis making it easier to declare dependencies in a simple way, the barrier of entry is significantly lowered.

As an experienced Swift developer, I often need to declare dependencies in a type-safe manner without using a hierarchical dependency injection which brings along a different set of limitations. This thesis then solves the problem and so I'm investing in the quality and enjoyment of my future work as well as the Swift community as a whole.

This thesis introduces a build-time dependency resolver that scans the source code of an application written in Swift. From classes in the code, a dependency graph will be constructed and its validity verified. Upon successful verification, a dependency container initialization code will be generated, ready to be used by the developer of said application. The generator is accompanied by a run-time library that developers can add to their projects, and then declare dependencies explicitly with the provided API. To help developers use the generator and explicit declarations, a user guide is provided.

The next section serves to introduce the concept of dependency injection as a whole – how it works and why use it. In the third section analyzes three main problems that are slowing down developers using dependency injection in *Swift*. The proposed solution to these problems is then described in the fourth section. The last two sections describe the implementation process and possibilities for future improvements.

2 Dependency Injection

Dependency Injection (DI) is a practice of providing dependencies to objects from an outer scope. It's commonly used with Object-Oriented Programming (OOP), which is guided by five major principles represented by the *SOLID* acronym. This set of principles was first introduced by Robert C. Martin in the paper *Design Principles and Design Patterns*[1]. These principles are:

- The Single Responsibility Principle (SRP): There should never be more than one reason for a class to change.
- The Open-Closed Principle (OCP): A module should be open for extension but closed for modification.
- The Liskov Substitution Principle (LSP): Subclasses should be substitutable for their base classes.
- The Interface Segregation Principle (ISP): Many client specific interfaces[1] are better than one general purpose interface.
- The Dependency Inversion Principle (DIP): Depend upon Abstractions. Do not depend upon concretions.

Implementing these principles in development leads to a better code quality[2]. Although not explicitly mentioned, Dependency Injection is a crucial component of *SOLID*. SRP guides us to separate functionality into separate classes. Such separation can make one of the classes depend on another. If that happens, the dependent class needs a reference to the class it depends on. One way is for the dependent class to create a new instance of the dependency class itself. However, when creating the new instance itself, DIP along with ISP are not honored.

The other option to obtain the reference is Dependency Injection. That means the instance (dependency) is passed in (injected) from the outside of the dependent class. The dependent class doesn't need to know and shouldn't care about where the instance came from.

Let's look at an example to show the difference between the two approaches and how they interact with the *SOLID* principles. Let's begin with two classes, `class House` and `class Plot` shown in Figure 1. `House` depends on `Plot`, because a house is built on a plot. So any instance of `House` needs a reference to a `Plot` instance.

In the listing 2.1 the `class House` creates a new instance of `class Plot` on the highlighted line. Even with such simple example, it becomes evident it's breaking the SRP, OCP and DIP principles. Let's now compare with the same

[1]In this thesis terms *protocol* and *interface* are used interchangeably. Although their semantics aren't equal, for dependency injection purposes the role of *protocols* takes on the same meaning as *interface*.

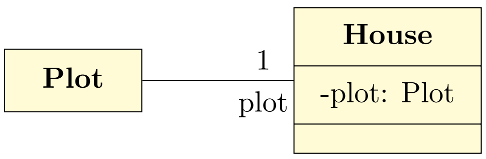

Figure 1: A plot and a house.

example, but with Dependency Injection in place. The listing 2.2 shows the `class House` expects a `Plot` instance as an argument to its initializer. On the highlighted line, the new instance of `class Plot` (dependency) is being created and then passed (injected) into the initializer for `class House`.

```swift
class House {
    private let plot = Plot()
}

class Plot {}

let house = House()
```

Listing 2.1: Example code without DI.

```swift
class House {
    private let plot: Plot

    init(plot: Plot) {
        self.plot = plot
    }
}

class Plot {}

let house = House(plot: Plot())
```

Listing 2.2: Example code with DI.

Let's dive deeper. Suppose the requirements change and instead of a single `class Plot` class, we need three kinds of land. So `class Plot` becomes a `protocol Plot` and is implemented by `class FarmLand`, `class CityLand`, and `class TownLand` (Figure 2).

In Listing 2.3 these changes are implemented without DI. Since *Plot* is no longer a `class`, the *House* can no longer create an instance of *Plot*. However, *House* still needs an instance of *Plot*. In this case, *House* creates an instance of *CityLand*, clearly breaking the Dependency Inversion Principle. The transition between Listing 2.1 and 2.3 also shows the Single Responsibility Principle being

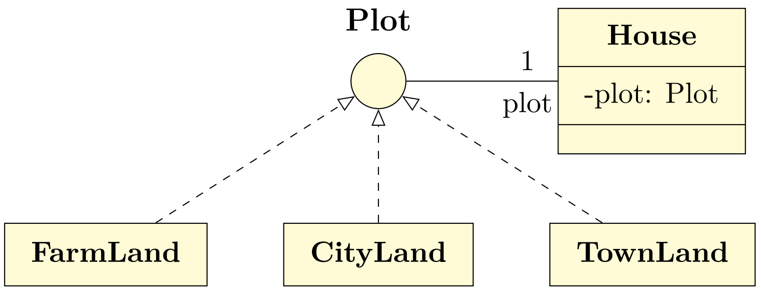

Figure 2: A plot, a house, and three land types.

broken. The only reason for the class *House* to change should be its own implementation, not a dependency changing from a `class` to `protocol` (as long as the dependency's API stays the same).

Now, in Listing 2.4 the *Plot* becomes a `protocol` with the three new implementations. With DI, the code for the *House* class doesn't change at all. What changes is the initialization of *House* on the highlighted lines. Thanks to DI, *House* doesn't decide what land it uses, and can be reused with any of the implementations of *Plot*.

```
class House {
    private let plot: Plot = CityLand()
}

protocol Plot {}

class FarmLand: Plot {}

class CityLand: Plot {}

class TownLand: Plot {}

let house = House()
```

Listing 2.3: Example code without DI **with DIP broken**.

To summarize, Dependency Injection is a practice of providing instances of dependencies from a scope outside of the object itself. DI makes code easier to test, maintain, reuse and extend.

```swift
class House {
    private let plot: Plot

    init(plot: Plot) {
        self.plot = plot
    }
}

protocol Plot {}

class FarmLand: Plot {}

class CityLand: Plot {}

class TownLand: Plot {}

let farmHouse = House(plot: FarmLand())
let cityHouse = House(plot: CityLand())
let townHouse = House(plot: TownLand())
```

Listing 2.4: Example code with DI with multiple implementations.

2.1 Dependency Injection Methods

There are multiple ways to inject dependencies into a class. If the Dependency Inversion Principle is being honored and each implementation has one or more interfaces it implements, those interfaces should never reveal which injection method is used. Otherwise the API of those interfaces might change when a class needs to switch to a different method (e.g. trying to resolve a dependency cycle 3.2).

2.1.1 Constructor Injection

When the object's constructor declares all the object's dependencies as its parameters, it is called a constructor injection (see Listing 2.5). Doing so makes it clear which dependencies are required by the class. It's a good practice to use constructor injection as much as possible.

Requiring all of the dependencies up front is helpful in both tests and the program itself. Combined with a type-safe language that checks the parameters in compile-time, developers are notified about missing dependencies right away during the compilation. This leads to a shorter development loop[2] as the compiler prints all the errors and the developer can fix them in one go.

[2]Development loop is the process of changing code, building it, deploying it, and seeing the changes. The longer it takes between making the change and seeing it, the less efficient the developer is.

```swift
class Person {}

class Car {
    private let owner: Person

    init(owner: Person) {
        self.owner = owner
    }
}

let car = Car(owner: Person())
```

Listing 2.5: Example of constructor injection.

2.1.2 Setter/Property Injection

Setter injection (in some languages also property injection, see Listing 2.6) is good for dependencies that might not be required for the object to function. Although such use-case could be fulfilled by marking a dependency in constructor optional, property injection allows for changing the dependency. That makes it easier to replace a dependency of an existing object without the need for a proxy. The main downside is the need for a two-step object initialization. When an object is created, and the dependency isn't optional, the created object cannot be used right away and has to have its properties injected first. This method also makes it easier to make mistakes and forget to inject a dependency, leading to a longer development loop. This is because the crash will only happen during run-time and not at compile-time or startup. More importantly, the crash would only happen when the property is accessed, so it may slip under the QA's radar and get deployed to production.

```swift
class Person {}

class Car {
    var owner: Person?
}

let car = Car()
car.owner = Person()
```

Listing 2.6: Property injection of an optional dependency.

2.1.3 Interface Injection

Interface injection is similar to the setter injection, but there is one major difference. The setter is declared as a single method of an interface. The interface has a single purpose - to be injected. This method allows for an easy grouping of multiple implementations of the interface in a collection and injecting them all at once (see Listing 2.7).

An alternative to creating an interface for each possible dependency is using function references in languages supporting such feature. Developers can then reference the setters directly without the need for an interface, as the signature of the function will behave as the interface.

```swift
class Person {}

protocol OwnerInjectable {
    func inject(owner: Person)
}

class Car: OwnerInjectable {
    private var owner: Person!

    func inject(owner: Person) {
        self.owner = owner
    }
}

let car = Car()
let ownerInjectables: [OwnerInjectable] = [car]
for ownerInjectable in ownerInjectables {
    ownerInjectable.inject(owner: Person())
}
```

Listing 2.7: Injecting a group of interface-injection enabled classes.

2.2 Dependency Graph

Before classes with DI can be injected, a dependency graph has to be constructed. A dependency graph is a directed graph[3], where vertices represent classes and arcs represent dependencies between the classes. A vertex at the tail of an arc *depends on* a vertex at the head of the arc. A dependency graph D can be described by a *vertex set* $V(D)$ and an *arc set* $A(D)$. A *vertex set* contains all classes that are part of the graph. An *arc set* contains tuples (u, v) where u *depends on* v.

Such description for the example graph in Figure 3 would be:

$$V(D) = \{a, b, c, d, e, f\}, A(D) = \{(a, b), (a, d), (b, c), (b, e), (c, d)\} \tag{1}$$

The *order* $|D|$ of the graph tells us how many classes there are in total. For each class $u \in V(D)$, the *out-degree* $d_D^+(u)$ tells us how many dependencies the class has. Classes with the *out-degree* equal to zero ($d_D^+(u) = 0$) have no dependencies and can be constructed directly. For each class $v \in V(D)$, the *in-degree* $d_D^-(v)$ tells us how many classes depend on the class v. Classes with the *in-degree* equal to zero ($d_D^-(v) = 0$) have no class depend on them.

Analyzing the graph in Figure 3 gives us detailed information about the graph and dependencies:

- $|D| = 6$ (*there are 6 classes in total*),
- $d^+_D(d) = d^+_D(e) = d^+_D(f) = 0$ (*d, e and f don't depend on any other class*),
- $d^-_D(a) = d^-_D(f) = 0$ (*no class depends on a and f*),
- $d^+_D(f) = 0 \wedge d^-_D(f) = 0$ (*no class depends on f and it has no dependencies*),
- the graph is acyclic (*no class has direct nor transitive dependency on itself, see 3.2*).

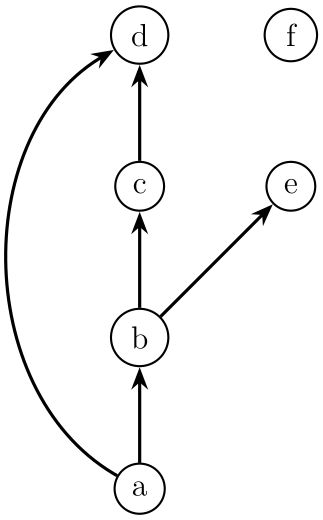

Figure 3: A dependency graph D.

Listing 2.8 contains a basic example of a simple dependency graph with classes using constructor injection (see 2.1.1). The goal is to create an instance of the `DefaultGreetingController` and call the `greet(name:)` method on it.

2.2.1 Static Configuration

Static configuration means the dependency graph never exists in run-time of the application. Instead, dependencies are declared statically - either by creating instances directly, providing them with instances they depend on, or by declaring factory lambda functions, which create new instances of dependencies. It's usually done without a DI framework. The main advantage of such approach is not depending on a third party software (see 3.3). Static configuration is preferred on small projects to keep them simple, or in libraries to minimize number of transitive dependencies. From Listing 2.9, it's clear the implementations have to be instantiated in a specific order. The `DefaultGreetingRepository` has to be first, as it doesn't depend on any other protocol. The second is the

```swift
protocol GreetingRepository {
    func greeting() -> String
}

protocol GreeterService {
    func composeGreeting(name: String) -> String
}

protocol GreeterController {
    func greet(name: String)
}

class DefaultGreetingRepository: GreetingRepository {
    func greeting() -> String {
        return "Hello, "
    }
}

class DefaultGreeterService: GreeterService {
    private let greetingRepository: GreetingRepository

    init(greetingRepository: GreetingRepository) {
        self.greetingRepository = greetingRepository
    }

    func composeGreeting(name: String) -> String {
        return greetingRepository.greeting() + name
    }
}

class DefaultGreeterController: GreeterController {
    private let greeterService: GreeterService

    init(greeterService: GreeterService) {
        self.greeterService = greeterService
    }

    func greet(name: String) {
        print(greeterService.composeGreeting(name: name))
    }
}
```

Listing 2.8: Example code for static vs dynamic demonstration.

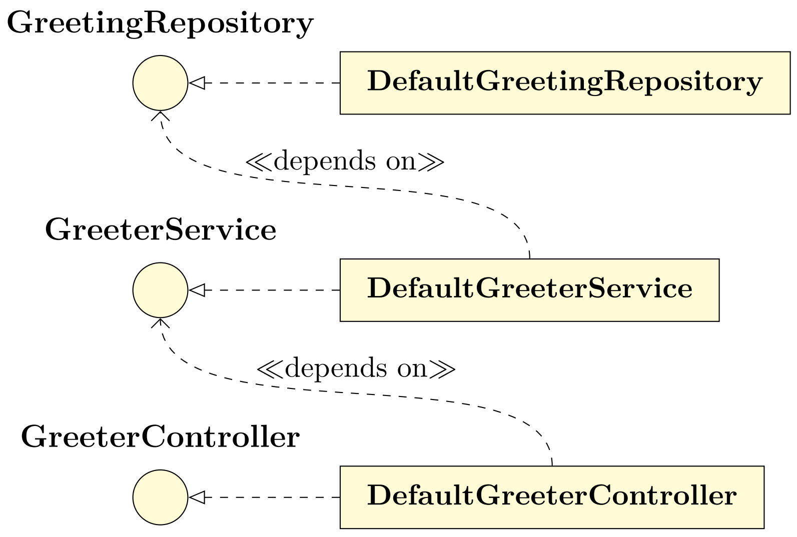

Figure 4: Caption

`DefaultGreeterService`, which depends on the repository. After the two are successfully instantiated, an instance of `DefaultGreeterController` is created.

The strict order is required by the compiler, since the dependent classes can't be instantiated without providing all their dependencies to the initializer. This behavior is the main advantage of the static configuration. If the program compiles, developers can rest assured all the required dependencies are configured. However, once the project grows, maintaining the configuration takes more development time, especially if multiple modules are needed.

```swift
let greetingRepository: () -> GreetingRepository
    = { DefaultGreetingRepository() }
let greeterService: () -> GreeterService
    = { DefaultGreeterService(greetingRepository:
     ↪  greetingRepository()) }
let greeterController: () -> GreeterController
    = { DefaultGreeterController(greeterService: greeterService()) }

greeterController().greet(name: "World") // prints "Hello, World"
```

Listing 2.9: Example of static injection.

2.2.2 Dynamic Configuration

For dynamic configuration, using a framework is recommended. Listing 2.10 shows dynamic configuration, using the Dip dependency container (see 2.5.1) to register and resolve dependencies. Main differences between dynamic and static configuration are registration order and compile-time safety. When using dynamic configuration, the registration order usually doesn't matter, which makes adding new dependencies straightforward. However, dynamic configuration loses the validation of the dependency graph by the compiler. Most DI frameworks support startup verification. These checks look for missing dependencies and dependency cycles. If found, the check fails and crashes the application during startup. Unfortunately, depending on how many registrations are missing, fixing the issues can take quite a while. As the framework will usually fail on the first missing dependency, multiple runs are required to ensure all dependencies are properly registered.

Depending on the selected injection framework, dynamic configuration brings additional features, such as:

- dependency scoping,
- debug information about dependencies,
- composing multiple configurations into a single one,
- dependency overrides.

Although these features can be achieved with static configuration, switching to dynamic configuration results in way less work. The most useful is dependency scoping. It allows for dependency lifetime management. In a typical application, some classes are supposed to only be instantiated once. Those are called *singletons* and the DI framework should keep such instance alive and in-memory until the application exits. In Spring, the backend development framework, scopes can be generally used to have a single instance per user session, or per web request.

```swift
import Dip

let container = DependencyContainer { container in
    container.register {
        DefaultGreeterService(
            greetingRepository: try container.resolve()
        ) as GreeterService
    }
    container.register {
        DefaultGreeterController(
            greeterService: try container.resolve()
        ) as GreeterController
    }
    container.register {
        DefaultGreetingRepository() as GreetingRepository
    }
}

let controller = try! container.resolve() as GreeterController
controller.greet(name: "World") // prints "Hello, World"
```

Listing 2.10: Example of dynamic injection with Dip.

2.3 Benefits Of Dependency Injection

Projects using Dependency Injection usually benefit from it in multiple ways, such as:

- easier maintenance,
- testability,
- better extensibility,
- improved reusability.

Additionally, splitting a project into submodules is usually easier when using DI. The dependency graph can be used to decide which interfaces and classes can be split into a separate module. As long as the visibility of classes in the original module is permissive enough to be used from the new module, creating a new module is effortless.

2.3.1 Testability

Software tests should be reliable, repeatable and parallelizable. However software itself is full of side-effects like updating state of a database, or sending requests to a third party service. If left unhandled, such side-effects lead to unreliable tests.

Since DI is built on providing implementations to objects from the outside, a different implementation can be injected when running tests than when running in production. Injecting so called mock dependencies, allows the developer to isolate the functionality being tested and ensure reliable tests.

Additionally, the developer can make the mock behave in a way that would be too difficult to setup with a real implementation, or even in invalid way to ensure the tested functionality handles it as expected. A good use-case of mocks is replacing communication with hardware, allowing tests to run without the real hardware and only simulating the communication. Simulating the communication saves a lot of time during tests as the communication can be instantaneous. Also it improves the test reliability as hardware can be unreliable.

When coupled with constructor injection (see 2.1.1), the developer, who is writing tests, has to provide all the dependencies of the component under test. When the tested component is modified to require an additional dependency, the test will fail to compile, indicating a problem and possibly a new and untested functionality.

Let's demonstrate how DI can be used to inject a mocked dependency. Suppose a program needs a component providing caching to filesystem. To conform with SRP it's split into two classes:

1. *Cache* containing an algorithm deciding when to keep and when to delete the cached values

2. *DataStorage* containing logic to save and load data to disk.

Without Dependency Injection, the `Cache` would create a new instance of the `DataStorage` in its constructor and use it for storing the cached values. This brings two problems to testing. Firstly the tests are testing two functionalities, the code of the caching algorithm and the data saving code, making it more difficult to find which of the two is working incorrectly. Secondly, the tests have to access the disk to cache and make sure the cached data is deleted after the testing is done.

With DI, the `Cache` would receive an instance of the `DataStorage` as an constructor parameter and use it for storing the cached values. In the production application, a `DataStorage` implementation that stores the data onto the filesystem would be injected into the `Cache`. In tests however, it makes more sense to inject an implementation of `DataStorage` that stores the data in memory. Storing the data in memory results in a clean cache storage each time the tests run without the need to manually delete any files.

2.3.2 Extensibility

Injecting an implementation allows developers to create multiple different implementations and then inject them based on the required functionality. For example:

- Database connection interface with implementations for different databases,
- fake implementation providing nice data for marketing materials,
- service implementation performing method invocations as remote procedure calls.

Another category of use-cases is decorating[4]. Before injecting a dependency, the developer can wrap it inside a different implementation of the same interface. This way an additional functionality can be layered on top of an existing implementation. For example:

- Logging before and after calling each method in the interface,
- timing how long a call to an interface's method took for performance testing,
- adding security by checking if an interface's method can be called,
- adding caching to long-running networking calls to improve responsiveness.

All of the above can be achieved without Dependency Injection, but using DI allows deciding which implementation is used in runtime and doesn't require any changes to the implementation to use a different dependency.

Let’s extend the example set in the previous section. A useful extension might be a logging decorator for `Cache`, that would be used in a development environment. Such decorator, let’s call it `CacheLogger` would wrap a `Cache` instance and implement the *Cache* interface itself. That way any consumers of the `Cache` instance would call methods on the `CacheLogger`. It would log debug information about caching and call the respective method in the wrapped `Cache` instance it holds. With DI, developers can even have multiple such decorators wrapping each other, providing additional functionality, without changing a single line in the original implementation.

```swift
protocol Cache {
    func retrive(name: String): Data?
}

class CacheLogger: Cache {
    private let cache: Cache

    init(cache: Cache) {
        self.cache = cache
    }

    func retrieve(name: String): Data? {
        print("Retrieving cached value for name: \\(name)")
        let result = cache.retrieve(name: name)
        print("Value for name: \\(name) was cached: \\(result !=
        ↪  nil)")
        return result
    }
}
```

2.3.3 Reusability

Along with extensibility, Dependency Injection can help with code reusability. Since each component has a final set of dependencies, it can be instantiated multiple times with different dependency implementations that perform the same work, but in a different way based on the dependencies provided. This leads to unexpected amount of reusable components.

Let’s look at the example from testability section 2.3.1. Suppose the original purpose of `Cache` was to cache files that are used the most often. During the development, the developer decided to also cache results of a complex computation. Now the developer needs two instances of `Cache`, one storing data in memory, one on the disk. Without DI, they would have to either change the `Cache` to support both, or duplicate the `Cache`, one for most used files, one for the computation results. With DI though, a second instance of `Cache` can be created with a different `DataStorage`.

2.4 Alternatives To Dependency Injection

Dependency Injection isn't the only option for objects to obtain instances of their dependencies from an outside scope.

2.4.1 Globally Accessible Instances

With globally accessible instances, each class may have a static property, containing the instance of the class. Listing 2.11 shows how a simple code with a global instance might look like. In this example, the `RepairShop` is the singleton, and can be accessed by the static property `shared` (usage shown on the highlighted line).

It can be tempting to have each dependency accessible in any place in the program. However, the disadvantages are way worse than advantages. The main disadvantage is an uncertainty what part of the program is being accessed and/or mutated. It also makes modularization a way more difficult and time consuming process, compared to a project that's using DI.

```swift
class Car {}

class RepairShop {
    static let shared = RepairShop()

    private init() {}

    func repair(car: Car) {
        println("Car: \\(car) repaired")
    }
}

class Person {
    let car = Car()

    func repairMyCar() {
        RepairShop.shared.repair(car: car)
    }
}

let person = Person()
person.repairMyCar()
```

Listing 2.11: Example of globally accessible singleton.

2.4.2 Service Locators

Service locators, described in [5] are a suitable alternative for Dependency Injection. Service locators are half-way between globally accessible singletons and dependency injection. They are objects providing access to implementations of interfaces that other implementations might need. Service locators are usually static, but configured during the program start-up by an *Assembler*.

In Figure 5 is an example of references between classes when using a service locator. The important thing is the `Cache` doesn't depend on an implementation of the `DataStorage`. Instead the `Assembler`'s responsibility is to configure the `ServiceLocator` before it's used by the `Cache`. Since the locator can be configured during start-up, tests can replace implementations with mocks if needed. Listing 2.12 shows a possible implementation of service locator in *Swift*.

Since the service locator is used from inside of a class, like in the `Cache`, it's not clear what dependencies are used by a class. Compared to constructor injection, this makes writing and maintaining tests more difficult.

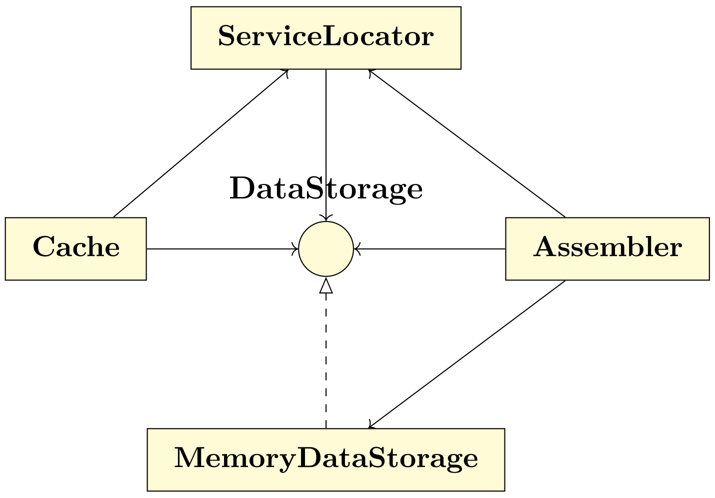

Figure 5: Basic service locator setup.

```swift
class ServiceLocator {
    private(set) static var shared: ServiceLocator!

    let dataStorage: DataStorage

    init(dataStorage: DataStorage) {
        self.dataStorage = dataStorage
    }

    class func load(locator: ServiceLocator) {
        shared = locator
    }
}

protocol DataStorage {
}

class MemoryDataStorage: DataStorage {
}

class Cache {
    private let dataStorage = ServiceLocator.shared.dataStorage
}

class Assembler {
    class func assembleLocator() {
        ServiceLocator.load(locator:
            ServiceLocator(dataStorage: MemoryDataStorage())
        )
    }
}
```

Listing 2.12: Example of service locator usage.

2.5 Prior Art

Swift language, like many other languages, doesn't have a dependency injection framework built-in. Therefore it needs to be added via an external dependency. There are many frameworks to choose from, one of them being *Dip*[6].

2.5.1 Dip

Dip[6] is a dependency injection container for *Swift*. It supports scopes, named definitions and arguments.

Dip features its own code generator, but its main disadvantage is relying on code comments for information about the dependency graph. Annotating the code with comments loses IDE auto-completion and refactoring support.

2.5.2 Spring Beans

In the *Java* framework *Spring*[7], Dependency Injection is a first-class citizen. It's dependencies can be configured using annotations or XML files. The framework is fully featured with support for scoping, dependency visibility, tagging and more. Since configuration is dynamic, it checks the dependency graph for missing dependencies, when the *Spring* application starts. When missing a dependency, the application terminates during the startup phase.

When configuring dependencies using annotations, any class annotated using `@Component` annotation gets added to the dependency graph. For third party dependencies, or where more control over the class instantiating is needed, *Spring* supports an explicit configuration method. Each class annotated with `@Configuration` is scanned for methods annotated with `@Bean`, called bean definitions. Bean definitions declare their dependencies as method parameters. The return type of these methods declares the interface provided. And the method body returns an instance of a class implementing the promised interface. Listing 2.13 contains an example of such configuration, with three bean definitions `greetingRepository`, `greeterService` and `greeterController`.

```kotlin
@Configuration
class GreetingConfiguration {

    @Bean
    fun greetingRepository(): GreetingRepository {
        return DefaultGreetingRepository()
    }

    @Bean
    fun greeterService(repository: GreetingRepository):
     ↪  GreeterService {
        return DefaultGreeterService(repository)
    }

    @Bean
    fun greeterController(service: GreeterService):
     ↪  GreeterController {
        return DefaultGreeterController(service)
    }
}
```

Listing 2.13: Configuration for classes from 2.8 in Spring.

3 Problem Analysis

Because *Swift* doesn't have a complex reflection, dependency injection containers in *Swift* require registering implementations for protocols manually. Doing so is error-prone as the errors appear only at run-time. Over time as the program's code-base grows, it becomes easier to make mistakes and the errors become more frequent. Dependency containers have checks to verify that all dependencies are satisfied and without cycles. However, it's usually a run-time check requiring a fresh build and program startup. Fixing missing dependency errors then slows down development.

To combat this, a program analyzing the dependency graph at compile-time and generating the supporting code is needed. Some injection container frameworks include their take on compile-time generators. Some generators require definition in code comments, foregoing all type-safety and automatic IDE refactoring capabilities. Others use *Swift's* property wrappers feature, requiring a single dependency container and disallow initializer injection.

3.1 Missing Dependencies

The easiest mistake to make is forgetting to register a new implementation to the dependency container. When the application is then ran, unless there's a class that depends on the new implementation, nothing out of the ordinary happens. It's when the implementation is declared as a dependency of another class, when the problems arise. When the application is run, it will crash because of the missing dependency. The developer trying to use it has to register it to the dependency container for it to work.

Because of the missing reflection in *Swift*, the dependency container can't use meta information about the program to construct the dependency graph and find all missing dependencies. Instead it can only report the first encountered. When there's a chain of dependencies not registered to the container, it can take a while for the developer to fix it.

To improve the situation, I need to support implicit dependencies, so that there's less need to declare new implementations manually. Additionally, a dependency graph has to be created during compilation and verified for missing dependencies all at once. That way developers get as much information right away and don't have to wait for the application to deploy and start.

3.2 Constructor Injection Cycles

When using constructor injection (see 2.1.1), all dependencies are required before a new instance of an implementation can be created. In some cases, one implementation, let's call it `AImpl`, implementing an interface `A` might depend on an interface `B`. When the implementation `BImpl` of the interface `B` requires an instance of the interface `A`, it results in a *dependency cycle* (also called *circular dependency*). This scenario is shown in Figure 6.

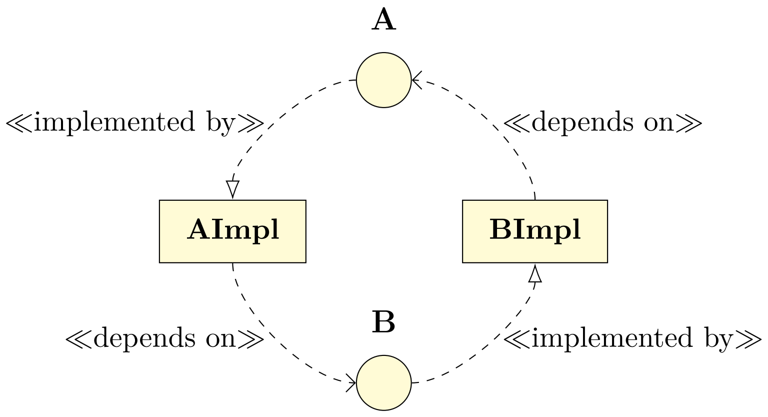

Figure 6: Dependency cycle

There are multiple ways to resolve such cycles, however most of them require changes in one of the implementations. Listing 3.1 is a simplified real-world example of a hierarchy of sections and items. Each item can either be a single text, or a whole new sub-section. When it comes to displaying the data, it needs to be converted to view models (classes with the *VM* suffix). However, the view model factories depend on one another. That's caused by the apparent circular dependency present in the data structures. There a section contains multiple items, and a sub-section item contains a section structure, that represents the inner section. In the example, the circular dependency is resolved using property injection (see 2.1.2). The assignment is shown on the highlighted lines.

```swift
struct Section {
  let items: [Item]

  enum Item {
    case row(text: String)
    indirect case subSection(section: Section)
  }
}
class SectionVM {
    let items: [ItemVM]

    class Factory {
        var itemFactory: ItemVM.Factory!

        func create(entity: Section) -> SectionVM {
            return SectionVM(items:
            ↪  entity.items.map(itemFactory.create))
        }
    }
}
class ItemVM {
    class SubSection: ItemVM {
        let section: SectionVM
    }
    class Row: ItemVM {
        let text: String
    }
    class Factory {
        var sectionFactory: SectionVM.Factory!

        func create(entity: Section.Item) -> ItemVM {
            switch entity {
            case .row(let text):
                return Row(text: text)
            case .subSection(let section):
                return SubSection(section:
                ↪  sectionFactory.create(entity: section))
            }
        }
    }
}

let itemFactory = ItemVM.Factory()
let sectionFactory = SectionVM.Factory()
itemFactory.sectionFactory = sectionFactory
sectionFactory.itemFactory = itemFactory
```

Listing 3.1: Example of a dependency cycle resolved using property injection.[3]

Listing 3.2 contains an example of resolving a dependency cycle using property injection and the *Dip* dependency framework. Two classes, `class A` and `class B`, are registered to the container. Each is dependent on the other, resulting in a dependency cycle. However, since the `class A` can be created without requiring an instance of `B`, the container is able to break the dependency cycle. The process that happens in the container when `container.resolve() as B` is called consists of (simplified):

1. The container knows that an instance of the class A is required to initialize an instance of B,
2. the container initializes a new instance of A, A_0,
3. the container initializes a new instance of B, B_0,
4. the container runs the closure passed to `resolvingProperties` for instance A_0.

```swift
class A {
  var b: B?
}

class B {
  private let a: A

  init(a: A) {
    self.a = a
  }
}

container.register() { A() }
  .resolvingProperties { container, service in
    service.b = try container.resolve() as B
  }

container.register() { B(a: container.resolve()) }
```

Listing 3.2: Example of dependency cycle resolution with Dip.

Dependency cycles should be avoided wherever possible as they lead to a confusing and hard to maintain code. Problems with dependency cycles between packages are described by Robert C. Martin in the paper *Granularity*[8], where he defines Acyclic Dependencies Principle (ADP). Although it describes dependency cycles between packages, the principle applies to classes as well. In *Swift* dependency cycles can lead to memory leaks if the two classes hold strong references to one another. This creates a so-called strong reference cycle, which won't be automatically freed[9].

Due to the added complexity and requirements for the implementations, I will implement dependency cycle check, which will result in an error during build if a cycle is found. This will make the developers restructure their code to get rid of the cycle.

3.3 Dependence On A DI Framework

Manual dependency management is tedious especially when advanced injection features like scoping and tagging is required. Inevitably developers decide to use a DI framework. Some of these frameworks offer unique features, or work in a different way than the rest of the frameworks. This can create a dependence on such framework, as migrating to a different solution can require substantial rewrites of the code using it.

The aforementioned support for implicit dependencies should reduce the dependency on a run-time DI framework. It could be argued it will be replaced by a dependency on the generator itself, but the generator doesn't place any special requirements on the code, other than disallowing circular dependencies. Therefore moving away from the generator and writing the registration code manually shouldn't pose a problem for the development.

4 Proposed Solution

Based on the problem analysis, I need to develop an API that developers can put into their applications and use it to declare dependencies. I also need to make a command-line program, that developers will run before compilation, to generate dependency registration for a dependency injection container.

4.1 Dependency Declaration API

The API has to be a suitable replacement for the missing meta programming in *Swift*. It has to allow declaring and tagging dependencies. Developers should be able to choose a scope of the dependency using the API too. Tagging dependencies will allow having a single interface available with multiple implementations, so consumers can choose which one to use.

4.1.1 Spring Beans Inspiration

I decided to take inspiration from the explicit configurations in *Spring Beans* (see 2.5.2). With *Spring Beans* developers declare so called configuration classes. These classes are annotated using `@Configuration`. Methods declared in such class can be annotated with `@Bean` to become a dependency provider. The return type of the method declares the interface being provided. If the method has any parameters, these are considered dependencies of the returned interface. The configuration class can have dependencies of its own and they are implicitly added as dependencies of each of the providers declared in the class.

For tagging dependencies, *Spring Beans* uses string names. They can be applied to `@Bean` methods using the `@Named` annotation. It accepts a string value which is the name for the dependency. Adding the name to a dependency makes it available only when the name is specified in a dependency declaration. So a `@Bean` method with a parameter accepting the dependency has to have the `@Named` annotation with the same value as the original declaration.

Similar to the dependency naming, assigning scopes to dependencies is done by annotating the provider method with `@Scope` annotation. The annotation accepts a string value, the name of the scope. To make a dependency a singleton, we'd put `@Scope("singleton")` above the provider method.

Listing 4.1 shows how these features would be used together using *Spring Beans*. The configuration class, `SpringConfiguration` has a single dependency on `SomeDependency`. It has two dependency providers, *firstDependency* and *secondDependency*. The *firstDependency* provider depends on an instance of `OtherDependency` and implicitly on `SomeDependency`, which is brought in by the configuration class. The *firstDependency* provider provides an instance of `FirstDependency`, named as *a name*. The other provider, *secondDependency*, depends on the `FirstDependency`. Notice the use of `@Named("a name")` applied to the type of the method's parameter. That's required for the dependency to be resolved, as there is no provider of an unnamed `FirstDependency`. An instance of `SecondDependency` is provided by *secondDependency*, this time with no name. But the method is annotated with the `@Scope` annotation, resulting in the `SecondDependency` being a singleton.

```kotlin
@Configuration
class SpringConfiguration(
    private val someDependency: SomeDependency,
) {
    @Named("a name")
    @Bean
    fun firstDependency(otherDependency: OtherDependency):
    ↪  FirstDependency {
        return FirstDependencyImpl(otherDependency)
    }

    @Scope("singleton")
    @Bean
    fun secondDependency(firstDependency: @Named("a name")
    ↪  FirstDependency): SecondDependency {
        return SecondDependencyImpl(firstDependency, someDependency)
    }
}
```

Listing 4.1: Spring Beans Configuration for inspiration.

4.1.2 Generics Composition API

Unfortunately the closest feature *Swift* has to annotations are property wrappers, which cannot be applied on methods and classes. So an alternative way to describe the dependencies is needed. The `@Configuration` annotation can simply be replaced by a marker protocol, that the configuration class will have to conform to.

Replacing the `@Scope` and `@Named` annotations isn't as easy. I drafted three possibilities for the API. The first one, shown in Listing 4.2 would use generics and composition instead of the annotations. To tag a dependency, developers would create a generic structure conforming to `DependencyTag` with a property for the dependency instance. The *firstDependency* provider would then return the `FirstDependency` wrapped in the `AName` tag structure. Same as in the *Spring Beans* configuration, the *secondDependency* provider depends on the tagged `FirstDependency`. It does so by accepting it wrapped in the `ATag` structure.

To scope a dependency, developers would use one of the provided scoping structures. The *secondDependency* provider shows how declaring a `Singleton` would work. It's similar to the tagging, except the scopes are predefined and new ones can't be added. Unfortunately, when using both scopes and tags, it becomes too verbose. The two structures need to be composed, with the type resulting in `Singleton<AName<AnotherDependency>>`.

Additionally, using generic dependencies results in ambiguity. Suppose a provider requires `Cache<Data>`, is the `Cache` supposed to be a tag for `Data`, or is it a class of its own? The generator could decide based on the tag conforming to the `DependencyTag` protocol, but that could lead to confusion. Since the generator can only scan source codes in one module a structure could conform to the `DependencyTag`, but the generator wouldn't know it.

```swift
struct AName<D>: DependencyTag {
    let dependency: D
}

class SwiftConfiguration: DependencyConfiguration {
    private let someDependency: SomeDependency

    init(someDependency: SomeDependency) {
        self.someDependency = someDependency
    }

    func firstDependency(otherDependency: OtherDependency) ->
    ↪ AName<FirstDependency> {
        AName(
            dependency: FirstDependencyImpl(
                otherDependency: otherDependency
            )
        )
    }

    func secondDependency(firstDependency: AName<FirstDependency>)
    ↪ -> Singleton<SecondDependency> {
        Singleton(
            dependency: SecondDependencyImpl(
                firstDependency: firstDependency.dependency,
                someDependency: someDependency
            )
        )
    }

    func thirdDependency(secondDependency: SecondDependency) ->
    ↪ ThirdDependency {
        ThirdDependencyImpl(secondDependency: secondDependency)
    }
}
```

Listing 4.2: Draft of the API using generics composition.

4.1.3 Flag Functions API

The second draft, shown in Listing 4.3, doesn't include the generics and the providers return provided types directly. To configure the dependencies, flag functions would be used. To tag a dependency, the function *tagged* would be used, with the tag's type as parameter. To depend on a tagged dependency, a new structure `Tagged` is introduced with two generic parameters, one for the dependency type, the other for the tag. Compared to the first draft, using tags is no longer ambiguous, as the *Swift* compiler is checking the conformance to `DependencyTag` protocol instead of the generator. Scoping would be a set of flag functions, like *singleton* that developers would wrap the dependency initialization with.

The generator would look for the flag functions and consider them a configuration of the dependency. The main downside is, that the code seems to be doing nothing and would require looking in the documentation to see why that's happening. Another issue arises when combining scopes and tags. It requires a conscious choice whether to apply the *tagged*, or the *singleton* flag function first. Additionally a new type of ambiguity is introduced, where developers could wrap a dependency initialization in two *tagged* function calls. There would have to be clear rules for which tag gets used.

4.1.4 Registration Structure API

The last draft in Listing 4.4 makes the dependency declaration more explicit. In *Spring Beans*, methods providing dependencies have to be annotated with the `@Bean` annotation. Here methods have to return the `DependencyRegistration` structure to be collected by the generator. The structure would have a single generic parameter for the dependency type. The structure's initializer accepts *tag* and *scope* parameters. Similarly to the *tagged* flag function, the *tag* parameter accepts a type of a tag instead of using an instance of it. This allows using the `Tagged` structure to depend on a tagged dependency. The scope would be an `enum` of possible values, so providing it to the initializer can be done the short-hand way without naming the type. This option has the least ambiguity and guides the developer as much as possible through declaring a dependency.

```swift
enum ATag: DependencyTag { }

class SwiftConfiguration: DependencyConfiguration {
    private let someDependency: SomeDependency

    init(someDependency: SomeDependency) {
        self.someDependency = someDependency
    }

    func firstDependency(otherDependency: OtherDependency) ->
    ↪  FirstDependency {
        tagged(AName.self) {
            FirstDependencyImpl(
                otherDependency: otherDependency
            )
        }
    }

    func secondDependency(firstDependency: Tagged<FirstDependency,
    ↪  AName>) -> SecondDependency {
        singleton {
            SecondDependency(
                firstDependency: firstDependency.dependency,
                someDependency: someDependency
            )
        }
    }
}
```

Listing 4.3: Draft of the API using flag functions.

```swift
enum ATag: DependencyTag { }

class SwiftConfiguration: DependencyConfiguration {
    private let someDependency: SomeDependency

    init(someDependency: SomeDependency) {
        self.someDependency = someDependency
    }

    func firstDependency(otherDependency: OtherDependency) ->
    ↪  DependencyRegistration<FirstDependency> {
        DependencyRegistration(tag: AName.self) {
            FirstDependencyImpl(
                otherDependency: otherDependency
            )
        }
    }

    func secondDependency(firstDependency: Tagged<FirstDependency,
    ↪  AName>) -> DependencyRegistration<SecondDependency> {
        DependencyRegistration(scope: .singleton) {
            SecondDependency(
                firstDependency: firstDependency.dependency,
                someDependency: someDependency
            )
        }
    }
}
```

Listing 4.4: Draft of the API using registration structures.

4.2 Command-Line Generator

The command line program should traverse source code of an application written in *Swift*. It should note all protocols and classes it encounters and construct a dependency graph from them. It should also be able to load explicit dependencies declared using the *api* library (see above 4.1). Once constructed, the program should run analysis on the graph to find any missing dependencies and dependency cycles. If it finds some, the program should terminate and inform the developer about it. Otherwise, a *Swift* file should be generated creating a dependency container instance and registering all the dependencies.

The program should also support printing the dependency graph as an output. This will allow developers to look at it and see the relationships between classes. For future-proofing, a command running just the analysis without generating an output should be provided. It can also be used by developers to check the graph validity without changing any code, making it perfect to use from a *pre-commit*[4] script.

The program should use the *Dip* dependency container (see 2.5.1) as I have prior experience with it and it's one of the popular ones. It has support for both dependency tagging and scopes so it's a good fit.

[4]Version Control software like *Git* support running scripts before submitting changes to it. Those scripts can verify the code passes predefined check, like having a valid dependency tree or passing tests.

5 Implementation

Before I began working on the generator, I needed to decide which of the API proposals would be the best for developers to give information about dependencies. Since the program is supposed to make developer's lives easier, I went with the most explicit, the least ambiguous variant (see 4.1.4). I implemented the run-time library first to support the structure from Listing 4.4. Then I drafted an example to test the next development on. The finished examples are available on the CD in the *Example/* directory.

Next step was defining the main structures the program will work with. Each dependency needs a unique identifier for the program, and later on for the dependency container. A module-name and a type-name identifies a class uniquely. Since the program runs for a single module at a time, type-name is enough to identify a class in this context. Because dependencies can have an optional tag, it has to be included as part of the identifier - `DependencyKey`.

Moving on, `DependencyFactory` is a description of a function, that takes dependencies as parameters and returns an instance of a provided dependency. The associations and properties are shown in Figure 7.

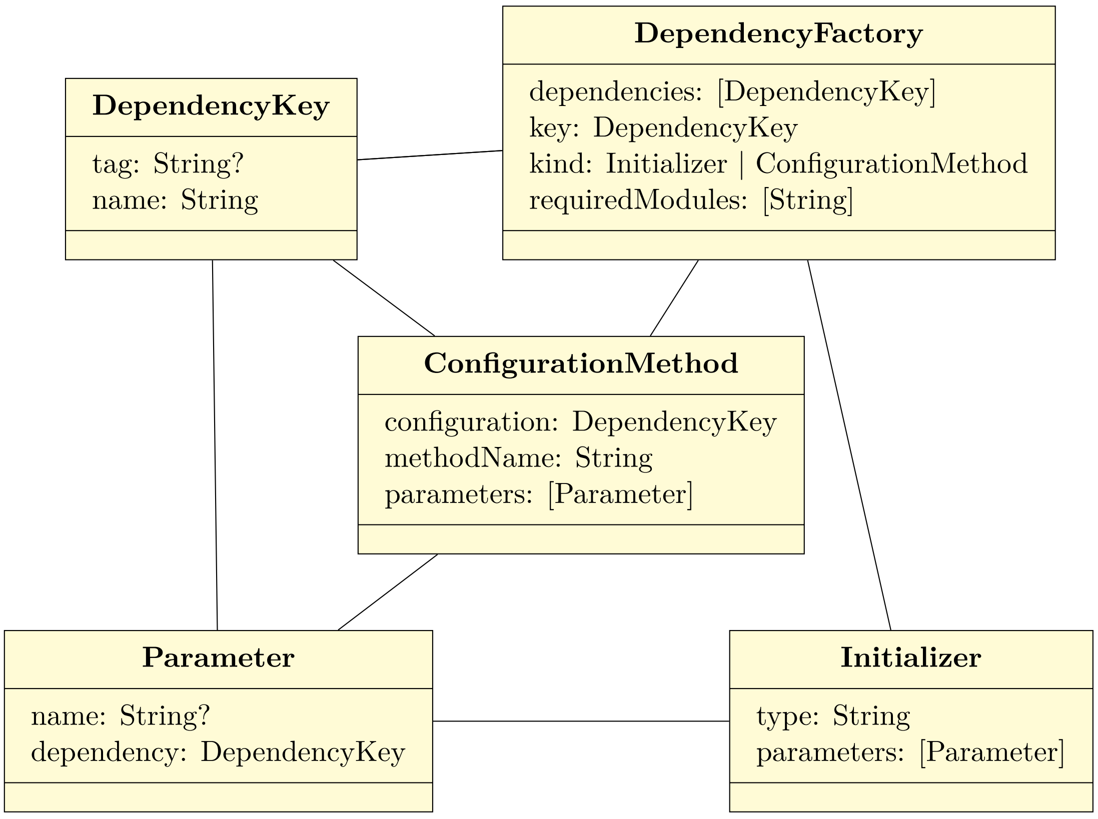

Figure 7: Collecting phase structures.

5.1 Modes of Operation

When I tried to use the generator in a real project that was modular, I realized I had to change the program to have two modes of operation:

- when running for a library target – *library mode*,
- when running for an executable target – *application mode*.

The main difference is in the program's output. In *library mode* it's an intermediate representation, containing the information about all dependencies provided by the library. The *application mode* would then use these files as sources of dependency providers when generating the dependency container registrations. For the sake of simplicity, I decided the mode will be selected automatically based on the file extension of the output file. If it's a file with *.dpendmodule* extension, it runs in the *library mode*, otherwise it runs in an application mode.

5.2 Collecting

I began developing the generator phase by phase, so first I tackled the *collecting* phase. This phase expects a directory and returns all dependency declarations it finds.

It begins with `FactoryLoader` and `DependencyCollector` classes. The `FactoryLoader` is provided a collection of paths and it's recursively going through the directory tree to find all *Swift* files. Then the files are passed into *SwiftSyntax*[5] which outputs an AST for each of the *Swift* files. Then it provides the collection of those ASTs to the `DependencyCollector`.

The `DependencyCollector` first walks a visitor through each of the ASTs. When the visitor encounters a class conforming to the `protocol DependencyConfiguration`, it walks a child visitor on the children to find all dependency provider declarations. Otherwise it notes the class as a possible implicit dependency and moves on.

After visiting all of the ASTs, the `DependencyCollector` filters out all of the possible implicit dependencies that don't match one of the provided regular expressions[6]. Once filtered, initializers of implicit dependencies are transformed into dependency factories. Each combination of an initializer and a protocol conformance of the class[7] results in a unique `DependencyFactory`. Collecting factories from the code in Listing 5.1, would result in the total of **7** `DependencyFactory` instances. Table 1 shows all of these instances.

[5] *SwiftSyntax*[10] is a library from *Apple* for *Swift* code introspection. Using visitor pattern, it allows traversing any *Swift* code easily, in a type-safe way.

[6] Later on, I also had to filter out classes that aren't public when running in the *library mode*.

[7] The program considers only directly declared conformances, not those declared using extensions.

```
protocol A { }
protocol B { }
protocol C { }
protocol D { }

class AImpl: A { }
class BImpl: B { }
class CDImpl: C, D {
  init(a: A) { }

  init(b: B) { }
}
```

Listing 5.1: Example code for implicit collecting.

Dependency Factories		
Interface	Dependencies	Factory
A	$\emptyset$	AImpl.init
B	$\emptyset$	BImpl.init
B	$\emptyset$	BImpl.init
C	A	CDImpl.init(a:)
C	B	CDImpl.init(b:)
D	A	CDImpl.init(a:)
D	B	CDImpl.init(b:)

Table 1: Result of collecting code in listing 5.1.

Similar process is applied to explicit configurations, with a couple of differences. The configuration class becomes a dependency itself, with its initializers having the same treatment as implicit dependency initializers. Then each method that returns `DependencyRegistration` results in a unique `DependencyFactory` with the configuration class as one of its dependencies. Collecting factories from the code in Listing 5.2, would result in the total of **3** `DependencyFactory` instances. Table 2 shows all of these instances.

Dependency Factories		
Interface	Dependencies	Factory
ExampleConfiguration (EC)	B	EC.init
A	EC	EC.provideA
C	EC, A	EC.provideC(a:)

Table 2: Result of collecting code in listing 5.2.

```swift
protocol A { }
protocol B { }
protocol C { }
protocol D { }

class AImpl: A { }
class BImpl: B { }
class CDImpl: C, D {
    init(a: A, b: B) { }
}

class ExampleConfiguration: DependencyConfiguration {
    private let b: B

    init(b: B) {
        self.b = b
    }

    func provideA() -> DependencyRegistration<A> {
        DependencyRegistration {
            AImpl()
        }
    }

    func provideC(a: A) -> DependencyRegistration<C> {
        DependencyRegistration {
            CDImpl(a: a, b: b)
        }
    }
}
```

Listing 5.2: Explicit dependency configuration.

5.3 Graph Building

The `class DependencyGraphBuilder` takes the output of the collecting phase and constructs a dependency graph. Each node in the graph is represented using the `DependencyGraphNode` structure. This structure stores the factory and all nodes that depend on it. The result of this phase is a collection of satisfied dependency nodes, and a map of unsatisfied dependency keys. The map of unsatisfied dependency keys has an associated collection of dependency nodes that depend on it. These structures are shown in Figure 8.

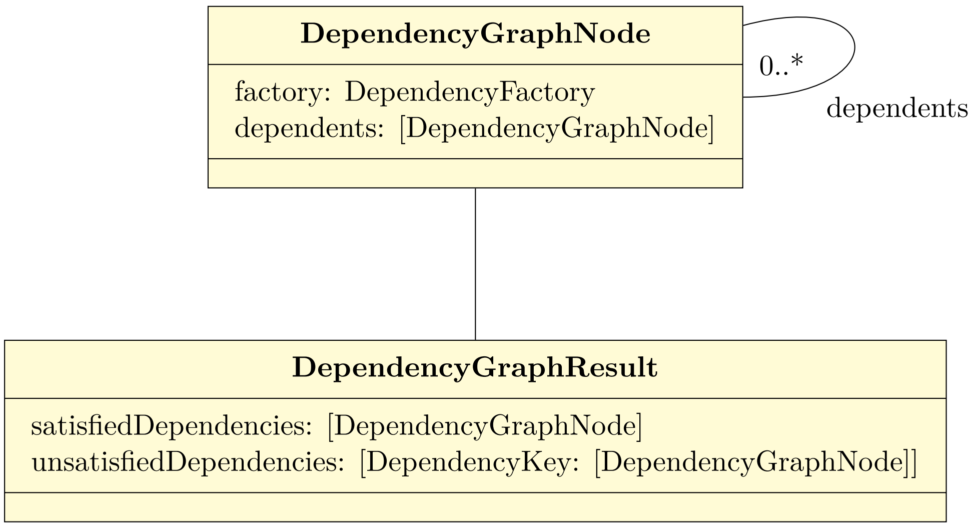
DependencyGraphNode
factory: DependencyFactory
dependents: [DependencyGraphNode]
0..*
dependents
DependencyGraphResult
satisfiedDependencies: [DependencyGraphNode]
unsatisfiedDependencies: [DependencyKey: [DependencyGraphNode]]

Figure 8: Graph building phase structures.

5.4 Analyzing

For the analysis phase I've chosen the visitor pattern, to walk each of the the dependency trees and analyze them. The core of the analysis is the `class DependencyAnalyzer`, which accepts result from the previous phase and returns all issues found by the analysis. To find the issues, the analyzer is provided with a list of visitors that will walk the trees and accumulate errors. There's two types of errors:

- unsatisfied dependency,
- dependency cycle.

The first type is constructed by the analyzer directly, by wrapping each of the unsatisfied dependencies from the chaining result into an instance of the error. The other type is constructed by the `DependencyCycleFinder`. Walking with a visitor is essentially a depth-first search on a dependency graph. So the cycle finder keeps a set of *seen nodes*. It check each visited node if it's already present in the *seen nodes* set. If not, it puts the node into the set and continues the traversal. If it is present, a dependency cycle error is constructed and the traversal is interrupted.

Figure 9 shows a traversal of a graph with a transitive cycle in it. The traversal proceeds in the opposite direction of the arrows from A through B and C to D. The current node and edge are highlighted in blue. Nodes that were visited once are highlighted in green. In the fifth step, the analysis visits the node B which is depends on the node D. However, the node B was already visited in the current traversal. This means to resolve the node B we need an instance of the node B. The analysis phase stores the error and continues traversing the tree, in this case it traverses from the node A to nodes E and F is visited and the analysis completes. Once done, the analysis reports all errors to the user. An analysis of the example in Figure 9 will result in a dependency cycle error with path $A \to B \to C \to D \to B$.

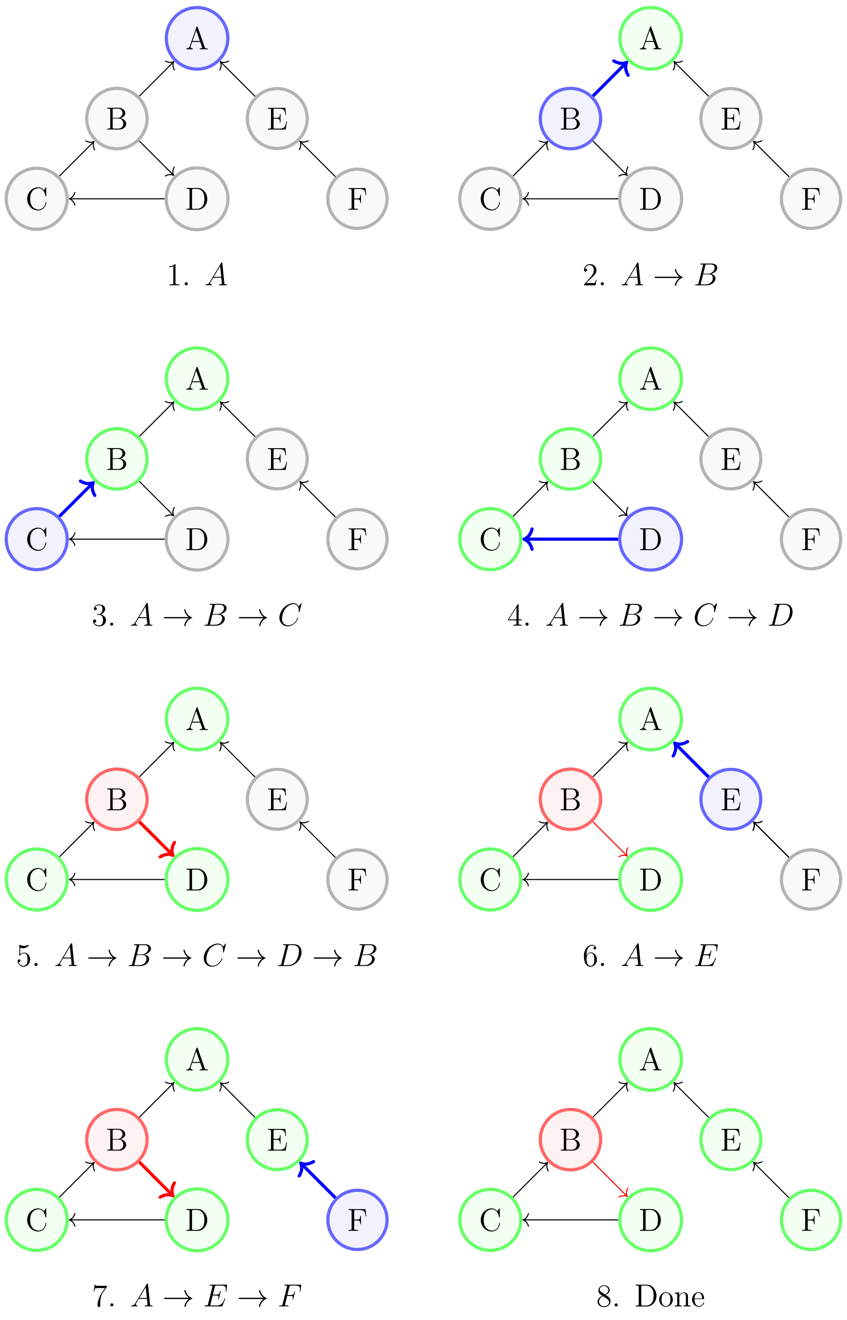

Figure 9: Dependency cycle analysis steps.

5.5 Generating

Once the analysis is complete, the program can generate its output. When running in the *library mode*, the result is a *JSON* file containing an array of `DependencyFactory` declarations. In the *application mode*, it generates a *Swift* file declaring the dependency container and dependency registrations.

Best way to showcase the difference between the two modes is with an example. Listing 5.3 has three classes, A, B and C. The class A has no dependencies, B depends on A, and C depends on both A and B. Let's look at the *library mode* output first shown in Listing 5.4. It contains all the necessary information about the classes A and B. The class C is missing in the *JSON* because of its `internal` visibility. However, looking at the output when running *application mode* shown in Listing 5.5, all three classes have a dependency registration generated.

```swift
public class A {
    public init() { }
}

public class B {
    public init(a: A) { }
}

class C {
    init(b: B, a: A) { }
}
```

Listing 5.3: Example code for showcasing the generator's output.

```
[
  {
    "dependencies" : [],
    "key" : {
      "name" : "A"
    },
    "kind" : {
      "initializer" : {
        "parameters" : [],
        "type" : "A"
      }
    },
    "requiredModules" : []
  },
  {
    "dependencies" : [
      {
        "name" : "A"
      }
    ],
    "key" : {
      "name" : "B"
    },
    "kind" : {
      "initializer" : {
        "parameters" : [
          {
            "dependency" : {
              "name" : "A"
            },
            "name" : "a"
          }
        ],
        "type" : "B"
      }
    },
    "requiredModules" : []
  }
]
```

Listing 5.4: *Library mode* output, when running for code in Listing 5.3.

```swift
import Dip

let dependencyContainer = DependencyContainer { container in
    container.register(.unique) { A() as A }
        container.register(.unique) { B(a: try container.resolve()
        ↪  as A) as B }
        container.register(.unique) { C(b: try container.resolve()
        ↪  as B, a: try container.resolve() as A) as C }
}
```

Listing 5.5: *Application mode* output, when running for code in Listing 5.3.

5.6 Test setup

Collecting, chaining and analysis are all tested separately using the Quick[11] and Nimble[12] frameworks. Since the generator's output is a Swift code, I decided against implementing automated tests for it. The structure of the code is not part of the API, so writing automated tests comparing the generated sources with expected sources would only add work needed when improving the generator's output and adding features. A proper testing setup for the generator would require a separate application that would run the generator on its sources and then run its own tests to ensure the generated container declares all of the expected dependencies. Such setup is complex and outside of the scope for this thesis. Instead, I've been testing the program by running it against the examples in the *Example/* directory and verifying the output manually.

6 Possible Improvements

The generator works well for the example project included with the generator's source code. I also had the opportunity to use it in a production application project. Since the project used *Dip* framework and constructor injection, the implicit configuration worked quite well. However, in the future, the following improvements will have to be added to cover more use-cases.

6.1 Support For More Frameworks

In its current version, the generator only supports generating code for the *Dip* framework. Adding support for other dependency containers is mostly straightforward and should require nothing more than changing the generator's output. This is a blocker to wider adoption as only projects using *Dip* can use the generator now. Support for hierarchical injectors (see Needle[13] and Cleanse[14]) would require more work, as these injectors require declaring the dependencies in a hierarchy.

6.2 Support For Static Configuration

Since the generator knows the entire dependency graph when running in the application mode, it could generate static code for the whole graph. This would remove the dependency on a run-time dependency injection framework. Implementing the static configuration would require changing the generator and figure out a way to support different scopes.

6.3 Performance Optimization

In a larger codebase, the dependency graph can get enormous. Each of the phases needs to traverse most of the dependency graph and does so serially. Scanning, analysis and generating can be parallelized to make use of modern multi-core CPUs. Additionally, the scanning phase runs *SwiftSyntax* which is still in development on the codebase, and it's expected to perform better with future releases.

6.4 Support For Property Injection

Some teams prefer the property injection instead of the constructor injection. Support for the property injection could improve Developer Experience by generating a factory method for the object, accepting all required dependencies as parameters. This generated method would be especially helpful in tests because it'd be easy to see required dependencies to have a properly initialized object. As such it would make the property injection behave similarly to the constructor injection as far as test codebase is concerned, alleviating one of the main disadvantages of the property injection.

6.5 Change Detection and Incremental Compilation

In its current state, the generator has to be run during each build. Doing so inevitably makes the build time longer. However, each build where the dependency graph didn't change runs the generator unnecessarily.

A possible solution is to store the inputs and their modified time at each successful generator run. Then each next run would check its inputs with the previous inputs. If none were added, removed or changed between the runs, the run is skipped.

Závěr

Cílem této práce bylo analyzovat a zmírnit omezení dependency injection ve Swiftu a zlepšit developer experience. Bylo důležité přijít s řešením, které by vývojáře co nejméně zatěžovalo. Vývojář by měl mít možnost deklarovat nové třídy a používat je jako závislosti, aniž by se musel starat o registraci do dependency injection containeru.

Během implementace jsem se musel několikrát vrátit zpět k návrhu, většinou kvůli omezením jazyka Swift. Hlavní překážkou byla absence metaprogramování, která omezuje strukturu API a zhoršuje developer experience. Druhou překážkou bylo, když jsem si uvědomil, že program musí podporovat dva režimy běhu, jeden pro knihovny a druhý pro spustitelné programy. Implementace je nyní připravena na budoucí rozšíření. Jako první bych přidal podporu pro další frameworky pro dependency injection a také bych přidal podporu pro generování kódu, který žádný dependency injection framework nebude ke své funkci potřebovat.

Vyvinutý program sestaví graf závislostí z kódu napsaném v jazyce Swift a ověří zda neobsahuje cykly závislostí a chybějící závislosti. Po ověření je vygenerován Swift kód připravený k použití v aplikaci. Implicitní a explicitní deklarace závislostí dohromady umožňují vývojářům psát co nejméně kódu, a přitom mít v případě potřeby plnou kontrolu.

Conclusions

This thesis was supposed to analyze and alleviate limitations of dependency injection in *Swift* and improve the developer experience. It was important to come up with a solution that would get out of the developer's way as much as possible. The developer should be able to declare new classes and use them as dependencies without worrying about registration to an injection container.

During implementation, I had to get back to the drawing board multiple times, mostly due to limitations of the *Swift* language. The main hurdle being the absence of meta-programming which limits the API's structure and the developer experience. The other was when I realized the program has to support two modes of operation, one for libraries and one for executables. The implementation is ready to be extended in the future. I'd add support for other dependency injection frameworks as well as making dependency injection frameworks optional altogether.

The developed program constructs a dependency graph from *Swift* code-base as intended and verifies its validity by checking for dependency cycles and missing dependencies. Once verified, *Swift* code is generated, ready to be used in an application. The two options for declaring dependencies, implicit and explicit, work great together, allowing developers to write as little code as possible, while still having full control when needed.

A User Guide

The program *dpend* is a command line tool used to generate a compile-time safe dependency injection module. It does this by:

1. constructing a dependency graph,
2. analyzing it for cycles and unsatisfied dependencies,
3. generating *Swift* file for the dependency module with a binary file describing provided and required dependencies.

The recommended way of running the program is using *Swift* Package Manager's *run* command, like so: `swift run dpend <subcommand>`.

A.1 Help Subcommand

The *help* subcommand prints information how to use the program. When run without any parameters (`swift run dpend help`), it prints a list of available subcommands.

```
OVERVIEW: A utility for analyzing dependencies and generating a
 ↪  dependency graph.

USAGE: dpend <subcommand>

OPTIONS:
  -h, --help              Show help information.

SUBCOMMANDS:
  analyze                 Analyze the dependency graph to find
  ↪  cycles and unsatisfied dependencies.
  tree                    Construct and print the dependency graph
  ↪  of a module.
  generate (default)      Construct dependency graph, analyze it and
  ↪  generate needed files.

  See 'dpend help <subcommand>' for detailed help.
```

A.2 Analyze Subcommand

Running the *analyze* subcommand is intended for quickly checking the module for issues, like missing dependencies and dependency cycles. Once advanced plugin support is added to *Xcode*, this subcommand should be run by an Xcode plugin to show dependency analysis in real-time during development. The documentation for the command can be printed by running `swift run dpend help analyze`.

```
OVERVIEW: Analyze the dependency graph to find cycles and
 ↪  unsatisfied dependencies.
```

```
USAGE: dpend analyze [<input-files> ...] [--implicit-filter
 ↪  <implicit-filter> ...]

ARGUMENTS:
  <input-files>           Paths to files and directories.
   ↪  Directories are traversed recursively to find all .swift
   ↪  files.

OPTIONS:
  --implicit-filter <implicit-filter>
                          Regex used to choose protocols and classes
                           ↪  to be part of the dependency graph.
                           ↪  Multiple filters behave like OR. When
                           ↪  no filters are provided, no implicit
                           ↪  dependencies will be analyzed.
  -h, --help              Show help information.
```

A.3 Tree Subcommand

The tree subcommand is useful for debugging and learning the app structure. It constructs the dependency graph and prints it in a human readable way. The documentation for the command can be printed by running `swift run dpend help tree`.

```
OVERVIEW: Construct and print the dependency graph of a module.

USAGE: dpend tree [<input-files> ...] [--implicit-filter
 ↪  <implicit-filter> ...]

ARGUMENTS:
  <input-files>           Paths to files and directories.
   ↪  Directories are traversed recursively to find all .swift
   ↪  files.

OPTIONS:
  --implicit-filter <implicit-filter>
                          Regex used to choose protocols and classes
                           ↪  to be part of the dependency graph.
                           ↪  Multiple filters behave like OR. When
                           ↪  no filters are provided, no implicit
                           ↪  dependencies will be analyzed.
  -h, --help              Show help information.
```

A.4 Generate Subcommand

The most important subcommand is *generate*. It supports two modes of operation:

- library mode,
- application mode.

The *library mode* doesn't generate any *Swift* code. A JSON file with *.dpendmodule* extension is generated, containing all available dependency providers.

The *application mode*'s output is a single *Swift* file, configuring the *Dip* dependency injection container. The *application mode* should be used for an executable module which is supposed to be compiled and linked with all library dependencies. The *application mode* accepts one or more `--library <library>` arguments, to provide the application with dependencies declared in libraries.

The documentation for the command can be printed by running `swift run dpend help generate`.

```
OVERVIEW: Construct dependency graph, analyze it and generate needed
 ↪  files.

USAGE: dpend generate [<input-files> ...] --output-file
 ↪  <output-file> [--implicit-filter <implicit-filter> ...]
 ↪  [--library <library> ...]

ARGUMENTS:
  <input-files>           Paths to files and directories.
   ↪  Directories are traversed recursively to find all .swift
   ↪  files.

OPTIONS:
  --output-file <output-file>
                          Path where the output is generated to. It
                           ↪  also selects the generator's mode. If
                           ↪  the output file has .dpendmodule
                           ↪  extension, the generator will produce
                           ↪  description of the library's
                           ↪  dependencies.
  --implicit-filter <implicit-filter>
                          Regex used to choose protocols and classes
                           ↪  to be part of the dependency graph.
                           ↪  Multiple filters behave like OR. When
                           ↪  no filters are provided, no implicit
                           ↪  dependencies will be analyzed.
  --library <library>     Path to a .dpendmodule file for a linked
   ↪  library.
  -h, --help              Show help information.
```

A.5 DpendRuntime Library

To get the explicit configuration support from *dpend*, add the *DpendRuntime* library to your application or library targets. It's lightweight with no transitive dependencies.

To declare an explicit configuration, begin by creating a new class implementing the `DependencyConfiguration` marker protocol. We'll call it `FooConfiguration`. Don't forget to put the `import DpendRuntime` at the top of the file.

```swift
import DpendRuntime

```

```swift
class FooConfiguration: DependencyConfiguration {
}
```

This configuration will be picked up automatically by the *dpend* program (as opposed to implicit dependencies that require explicit filter argument). If the configuration class is declared in a library, it's required to have `public` modifier, so that it can be used from a dependent module.

Now let's add our first explicit dependency declaration. Add a new method, `fooService` with no parameters, returning the generic structure `DependencyRegistration<D>`. The generic type `D` specifies the type of the provided dependency. In our case we'll also declare a `protocol FooService` and use it.

```swift
import DpendRuntime

class FooConfiguration: DependencyConfiguration {
    func fooService() -> DependencyRegistration<FooService> {

    }
}

protocol FooService { }
```

Next step is deciding what scope will the dependency be in and whether or not it'll be tagged. Supported scopes are:

- `.unique` – each time a class needs this dependency, a new instance is created (default),
- `.shared` – during a top-most container `resolve` call a same instance is used,
- `.singleton` – once created, an instance of the class is retained and reused until the application terminates,
- `.eagerSingleton` – same as `.singleton`, but an instance is created along with the DI container,
- `.weakSingleton` – same as `.singleton`, but an instance is stored using weak reference, so once deallocated a new instance is created when needed.

Tag can be any type conforming to the `DependencyTag` marker protocol. Suppose our `FooService` will be `.singleton` scoped and we'll create a new tag `enum Bar` for it (using an `enum` to declare tags is recommended as they cannot be instantiated or overriden).

```swift
import DpendRuntime

public enum Bar: DependencyTag { }

class FooConfiguration: DependencyConfiguration {
    func fooService() -> DependencyRegistration<FooService> {
        DependencyRegistration(tag: Bar.self, scope: .singleton) {

        }
    }
}

protocol FooService { }
```

Our FooService dependency declaration is almost done. Last step is creating an instance of a class conforming to the protocol FooService in the DependencyRegistration lambda. In our case, we'll create a new class DefaultFooService and return a new instance of it.

```
import DpendRuntime

public enum Bar: DependencyTag { }

class FooConfiguration: DependencyConfiguration {
    func fooService() -> DependencyRegistration<FooService> {
        DependencyRegistration(tag: Bar.self, scope: .singleton) {
            DefaultFooService()
        }
    }
}

protocol FooService { }
class DefaultFooService: FooService { }
```

Let's declare a protocol BarService and a class DefaultBarService conforming to the BarService protocol. This class will need an instance of FooService before it can be instantiated. Let's add it to our configuration.

```
import DpendRuntime

public enum Bar: DependencyTag { }

class FooConfiguration: DependencyConfiguration {
    func fooService() -> DependencyRegistration<FooService> {
        DependencyRegistration(tag: Bar.self, scope: .singleton) {
            DefaultFooService()
        }
    }

    func barService(fooService: Tagged<FooService, Bar>) ->
     ↪  DependencyRegistration<BarService> {
        DependencyRegistration {
            DefaultBarService(fooService: fooService.dependency)
        }
    }
}

```

```swift
protocol FooService { }
class DefaultFooService: FooService { }

protocol BarService {}
class DefaultBarService: BarService {
    init(fooService: FooService) { }
}
```

Notice the parameter of the new `barService` method. Adding parameters to dependency declarations makes the declaration depend on the parameter types. In this case, it depends on `FooService` so that the `DefaultBarService` can be instantiated. However, since we registered the `FooService` with a tag `Bar`, wrapping the type in `Tagged` is required. This structure tells *dpend* which dependency is required.

Let's add our last declaration, a `protocol BazService` with a `class DefaultBazService: BazService`. This new class will depend on `BarService` and we'll accept an instance of it as a parameter. This time without being wrapepd in `Tagged`, as the declaration doesn't specify a tag.

```swift
import DpendRuntime

public enum Bar: DependencyTag { }

class FooConfiguration: DependencyConfiguration {
    func fooService() -> DependencyRegistration<FooService> {
        DependencyRegistration(tag: Bar.self, scope: .singleton) {
            DefaultFooService()
        }
    }

    func barService(fooService: Tagged<FooService, Bar>) ->
    ↪  DependencyRegistration<BarService> {
        DependencyRegistration {
            DefaultBarService(fooService: fooService.dependency)
        }
    }

    func bazService(barService: BarService) ->
    ↪  DependencyRegistration<BazService> {
        DependencyRegistration {
            DefaultBazService(barService: barService)
        }
    }
}

protocol FooService { }
class DefaultFooService: FooService { }

protocol BarService {}
class DefaultBarService: BarService {
    init(fooService: FooService) { }
}
```

```

protocol BazService {}
class DefaultBazService: BazService {
    init(barService: BarService) { }
}
```

B Developer Guide

This guide describes how to get the project set up for local development using *Xcode*. The project is using *Swift* Package Manager[15], so it can be opened using *Xcode*. However, to run the program from *Xcode*, an extra environment variable must be added to the run scheme:

```
DYLD_LIBRARY_PATH=/Applications/Xcode.app/Contents/Developer/Toolch⌋
↪  ains/XcodeDefault.xctoolchain/usr/lib/swift/macosx/
```

Without it, the program crashes as the *dyld* cannot find the *libSwiftSyntax* library. Another option is to run the program using SwiftPM from the terminal: `swift run dpend`.

The code-base is split into three main directories:

- Example – contains examples to run the program against,
- Sources – contains modules of the program itself,
- Tests – contains tests for the program.

B.1 Examples

Three examples are currently implemented:

- Example/Library,
- Example/Application,
- Example/DependencyCycle.

The first two are happy path examples to show how *dpend* can be used. The third shows, as name suggests, how *dpend* behaves when it encounters a dependency cycle.

B.1.1 Library

The example in *Example/Library* showcases the explicit configuration support. To generate the *.dpendmodule* file, run the following from the *dpend* project root directory:

```
swift run dpend generate --output-file
↪  Example/Library/Derived/ExampleLibrary.generated.dpendmodule
↪  Example/Library
```

Although the generated file has the *.dpendmodule* extension, it's a JSON file, so it can be opened in any text editor and its contents inspected. The JSON is pretty-printed and the contents should be stable, so it's recommended to commit the file to version control. That way, changes in the dependencies provided by the library can be tracked throughout version history. It also removes the need to run the generator to build the project, and only becomes needed when making changes.

B.1.2 Application

The example in *Example/Application* shows the implicit dependencies support. Since it's an executable, the generator is run in application mode and generates a single *Swift* file. To generate it, run the following from the *dpend* project root directory:

```
swift run dpend generate --output-file Example/Application/Derived/⌋
 ↪  ExampleApplicationModule.generated.swift --library
 ↪  Example/Library/Derived/ExampleLibrary.generated.dpendmodule
 ↪  --implicit-filter ".+\\.Factory" Example/Application
```

The command passes in the *ExampleLibrary* module as *–library* argument, since the *ExampleApplication* module depends on the *ExampleLibrary* module. To enable implicit dependencies, a filter argument *–implicit-filter ".+*
.Factory" is passed in. This filter will match any inner class called *Factory*. Similar to the *.dpendmodule* file, the generated *Swift* file should be stable and adding it to version control is recommended.

B.1.3 Dependency Cycle

The example in *Example/DependencyCycle* uses an explicit configuration that creates a dependency cycle, to end-to-end test the program against it.

The following three commands can be run to test the program:

```
swift run dpend tree Example/DependencyCycle

swift run dpend analyze Example/DependencyCycle

swift run dpend generate --output-file
 ↪  Example/DependencyCycle/Derived/DI.swift Example/DependencyCycle
```

B.2 Sources

The *Sources* directory contains source files for the *dpend* program and for a run-time library to be included in user programs.

B.2.1 dpend

The *dpend* directory contains the *main.swift* executable, along with the *Dpend* program and its subcommands. It's using *Apple*'s *Swift* argument parser[16] to parse terminal arguments and print helpful information about the programs usage. *dpend* executable depends on the *DpendKit* target for the main logic.

B.2.2 DpendKit

For future-proofing, the *dpend* program is split and most of the logic lies in the *DpendKit* module. In the future, this module could be used from an IDE plugin, or from a command-line linter tool.

There are two structures used throughout the module, these are `DependencyKey` and `DeependencyFactory`. The former is used as a unique identifier of dependency declarations. It has an optional *tag* and a required *name*. `DependencyFactory` describes how a dependency can be obtained. It's identified by a `DependencyKey` and declares its dependencies using an array of `DependencyKey`. For the generated *Swift* code to be compilable, the factory also retains all required imports in the *requiredModules* property. Last but not least, there's currently two factory kinds:

- initializer factory – instance is obtained by initializing a type,
- configuration method factory – instance is obtained by calling a method on an instance of explicit dependency configuration.

The first kind is used for implicit dependencies and for initializing explicit configurations. It has an array of parameters, so the generator know which initializer to use. The second kind is used for explicit dependencies. As such it keeps the `DependencyKey` to obtain an instance of the configuration. Then the generator can generate code that resolves the configuration instance first and then calls the method on it which returns the dependency.

Each of the program's phases has its own directory:

- Collecting,
- Chaining,
- Analysis,
- Generating.

SwiftSyntax[10] is being used for collecting information about the *Swift* code in users' code-base. It relies heavily on the visitor pattern.

The dependency graph is constructed in the *Chaining* phase, after which the `class DependencyAnalyzer` can be used to traverse the graph and analyze the dependencies. The `DependencyAnalyzer` takes a list of visitor factories, which are invoked when analysis starts. The visitors are then run on the graph nodes. Each visitor can decide whether to visit dependent nodes or not. The analysis visits dependent nodes recursively as long as any visitors remain.

Currently *dpend* has a single analysis visitor implementation, which is the `class DependencyCycleFinder`. Use it as a reference when developing new analysis visitors.

B.2.3 DpendRuntime

Developers who want to use the *dpend* program to its full potential have to include *DpendRuntime* library in their project. This library has to be kept lightweight with no transitive dependencies. It contains the explicit configuration API, which is needed to specify dependency scopes and tags. Any new API to be added into the *DpendRuntime* sources has to be reviewed thoroughly, as it becomes public API and changes in it might break users' code-bases.

As new versions of *Swift* are released, new features introduced in these versions should be added to the *DpendRuntime* library in a backward compatible manner, possibly deprecating an old method if a vastly better new method is added.

B.3 Tests

Tests are written using a *BDD* library *Quick*[11]. Currently `DependencyAnalyzer`, `DependencyChainng` and `DependencyCollector` are being unit tested. Library *Nimble*[12] is used for assertions as recommended by the authors of *Quick*.

C Contents of attached CD

The attached CD contains the following files and directories.

`bin/`
Contains the *dpend* binaries, one compiled for x86_64 and one for arm64. It's macOS only and doesn't support other operation systems. Compatible with Swift 5.5.2 toolchain, which needs to be downloaded from https://www.swift.org/download/.

`doc/`
The thesis in PDF format and LaTeX source files used to compile it.

`src/`
Source code for the *dpend* program. It can be opened in *Xcode*.

`src/README.md`
Short instructions for running the program. For more detailed instructions see the user guide.

Acronyms

ADP	Acyclic Dependencies Principle
API	Application Programming Interface
AST	Abstract Syntax Tree
BDD	Behavior-Driven Development
DI	Dependency Injection
DIP	Dependency Inversion Principle
DX	Developer Experience
IDE	Integrated Development Environment
ISP	Interface Segregation Principle
LSP	Liskov Substitution Principle
OCP	Open-Closed Principle
OOP	Object-Oriented Programming
QA	Quality Assurance
SRP	Single Responsibility Principle

References

[1] MARTIN, Robert Cecil. *Design Principles and Design Patterns*. 2000. Available from WWW: ⟨https://web.archive.org/web/20150906155800/http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf⟩.

[2] SINGH, Harmeet; HASSAN, Syed Imtiyaz. Effect of SOLID Design Principles on Quality of Software: An Empirical Assesment. *International Journal of Scientific & Engineering Research*. 2015, vol. 6, no. 4 pp. 1.6:1–1.6:64. Available also from WWW: ⟨https://www.ijser.org/researchpaper/Effect-of-SOLID-Design-Principles-on-Quality-of-Software-An-Empirical-Assessment.pdf⟩. ISSN 2229-5518.

[3] BANG-JENSEN, Jørgen; GUTIN, Gregory Z.; GUTIN, Gregory. *Digraphs: theory, algorithms and applications*. 2. print. London Berlin Heidelberg: Springer, 2006. ISBN 9781852336110.

[4] GAMMA, Erich (ed.). *Design patterns: elements of reusable object-oriented software*. Reading, Mass: Addison-Wesley, 1995. Addison-Wesley professional computing series. ISBN 9780201633610.

[5] FOWLER, Martin. *Inversion of Control Containers and the Dependency Injection pattern*. 2004. Available from WWW: ⟨https://martinfowler.com/articles/injection.html⟩.

[6] HALLIGON, Olivieer. *Dip* [online]. 2022-1-7. Available from WWW: ⟨https://github.com/AliSoftware/Dip⟩.

[7] *Spring makes Java simple.* [online]. 2022-1-7. Available from WWW: ⟨https://spring.io/⟩.

[8] MARTIN, Robert Cecil. *Granularity*. Available from WWW: ⟨https://condor.depaul.edu/dmumaugh/OOT/Design-Principles/granularity.pdf⟩.

[9] *Automatic Reference Counting — The Swift Programming Language (Swift 5.5).* [online]. 2022-1-7. Available from WWW: ⟨https://docs.swift.org/swift-book/LanguageGuide/AutomaticReferenceCounting.html⟩.

[10] *SwiftSyntax*. [online]. 2022-1-7. Available from WWW: ⟨https://github.com/apple/swift-syntax⟩.

[11] *Quick*. [online]. 2022-1-7. Available from WWW: ⟨https://github.com/Quick/Quick⟩.

[12] *Nimble*. [online]. 2022-1-7. Available from WWW: ⟨https://github.com/Quick/Nimble⟩.

[13] *Needle*. [online]. 2022-1-7. Available from WWW: ⟨https://github.com/uber/needle⟩.

[14] *Cleanse*. [online]. 2022-1-7. Available from WWW: ⟨https://github.com/square/Cleanse⟩.

[15] *Package Manager — The Swift Programming Language (Swift 5.5)*. [online]. 2022-1-7. Available from WWW: ⟨https://www.swift.org/package-manager⟩.

[16] *Swift Argument Parser*. [online]. 2022-1-7. Available from WWW: ⟨https://github.com/apple/swift-argument-parser⟩.

[17] MARTIN, Robert Cecil. *DIP: The Dependency Inversion Principle*. Available from WWW: ⟨http://www.labri.fr/perso/clement/enseignements/ao/DIP.pdf⟩.

[18] BOENDER, Ferry. *Dependency Resolving Algorithm*. 2010. Available from WWW: ⟨https://www.electricmonk.nl/docs/dependency_resolving_algorithm/dependency_resolving_algorithm.html⟩.

[19] PUCHKA, Ilya. *dipgen* [online]. 2022-1-7. Available from WWW: ⟨https://github.com/ilyapuchka/dipgen⟩.