VRIJE
UNIVERSITEIT
AMSTERDAM

~~Bachelor Thesis~~ Research Proposal

A zero-cost dependency injection system

Author: Tiziano Coroneo [2736905]

1st supervisor: Atze van der Ploeg
daily supervisor: Atze van der Ploeg
2nd reader: TBD

~~A thesis submitted in fulfillment of the requirements for the VU Bachelor of Science degree in Computer Science~~

May 26, 2024

Introduction

Consider the following piece of Swift code:

```swift
class APIClient {
  init() {}
  func request(url: String) async throws -> String { /*[...]*/ }
}

class CheckoutService {
    let apiClient = APIClient()
    init() {}
    func performCheckout() async throws {
        let response = try await apiClient.request(url: "https://example.com/
checkout")
        print("Data fetched successfully! Response: \(response)")
    }
}
```

If you need an instance of `CheckoutService` to perform the checkout operation, you can simply instantiate the class and checkout away. But what if you have multiple classes that require a reference to the same `APIClient`? What if you need multiple instances of the `CheckoutService`, each with a slightly differently configured `APIClient`? What if you want to unit-test the `CheckoutService` class? In all these cases, you want to *inject* the specific `APIClient` instance that the `CheckoutService` *depends on*.

For example, you may pass the specific instance of the `APIClient` to use in the initializer of the `CheckoutService` class:

```swift
class CheckoutService {
    let apiClient: APIClient
    init(apiClient: APIClient) {
      self.apiClient = apiClient
    }
    func performCheckout() async throws {
        let response = try await apiClient.request(url: "https://example.com/
checkout")
        print("Data fetched successfully! Response: \(response)")
    }
}

let checkout = CheckoutService(apiClient: APIClient())
```

Another option is to assign the instance to a property on the `CheckoutService` after initialization is complete:

```swift
class CheckoutService {
    var apiClient: APIClient?
    init() {}
    func performCheckout() async throws {
        let response = try await apiClient?.request(url: "https://example.com/
checkout")
        print("Data fetched successfully! Response: \(response ?? "")")
    }
}

let checkout = CheckoutService()
checkout.apiClient = APIClient()
```

Or even pass the instance directly to the method that requires it:

```
class CheckoutService {
    init() {}
    func performCheckout(apiClient: APIClient) async throws {
        let response = try await apiClient.request(url: "https://example.com/
checkout")
        print("Data fetched successfully! Response: \(response)")
    }
}

let checkout = CheckoutService()
checkout.performCheckout(apiClient: APIClient())
```

These seem very simple solutions to a simple problem: how can a class be decoupled from the creation of the objects that it depends on? As we will see, this apparent simplicity hides a world of complexity that no ambitious library author can ignore.

Modern mobile apps grow larger and larger over time, adding new features and extending existing capabilities. Some large-scale apps like Uber can have ~300 modules, amounting to ~1,000,000 lines of code [1]. As they grow, their internal structures also grow so complex that a substantial amount of engineering effort is spent making systems to manage said complexity, be they abstract sets of principles like SOLID [2] or support systems for everyday operations.

A category of these support systems is “Dependency Injection” (DI) [3].

DI solves the problem of isolating sections of code from their direct dependencies so that unit testing is as easy as possible while not overcomplicating the code base with additional functionality only meant to support testing. DI systems solve this issue by splitting building objects from using them and removing strong coupling between components [4]. They typically do so by defining some kind of *object container* that builds an object graph of interconnected dependencies while maintaining loose coupling between each node of the graph; in addition to that, they also provide a convenient way to access each node in the graph, so that such accesses can easily be modified in unit tests to verify the implementation of a feature.

DI is a common feature of many ecosystems: in Android apps, `Dagger` [5], [6] is the most common library, while in `.NET` there is an integrated DI system in the `IServiceCollection` API. In iOS there is no standard, first-party solution, but many different libraries are available with different APIs, capabilities, and tradeoffs. The most popular library is `Swinject`, with ~6200 stars on GitHub, followed by Square’s `Cleanse` with 1800 stars, Uber’s `needle`[7] with 1700 stars, `Factory`[8] with 1600 stars, and the latest entry, `swift-dependencies`[9] with 1400 stars. Uber’s implementation is particularly interesting, as it offers unique tradeoffs: while most DI systems resolve their dependency graph representation at run-time, their application is so large that they found the need to write their own high-performance, compile-time safe DI system, doing their best to keep the impact on compilation times low. Their choices in designing this library offer insights into the different costs associated with dependency injection: a run-time solution is easier to implement than a compile-time build plugin, but it offers less type safety and impacts an application’s startup time. A compile-time solution provides better safety guarantees (“if it compiles, it works!”), and higher performance at run-time, but it risks slowing down the development cycle by increasing build times locally and in continuous integration.

All these frameworks provide similar capabilities while offering different performance tradeoffs. The first sub-research question explores these differences (**SQ1**):

"What are the tradeoffs in the design of a DI system?"

We will benchmark these frameworks to evaluate their performance impact in different categories compared to regular Swift code that uses no external library to manage complexity. Presenting the results constitutes the answer to the second research sub-question (**SQ2**):

"What is the performance impact of dependency injection systems in iOS?"

Then, we will present a new library to optimize the performance of the *graph-building* part of a dependency injection system, and I will benchmark its performance compared to the mainstream open-source alternatives previously mentioned, (hopefully) achieving a *zero-cost dependency injection system*, and providing an answer to the main research question (**MQ**):

"Is a zero-cost dependency injection system possible?"

Related work

"Zero-cost abstractions" is a description of some abstractions in the C++ ecosystem that follow the "zero-overhead principle", first described by Bjarne Stroustrup [10].

Some works in the literature refer to DI systems with the more generic name of "Inversion of Control containers," like the paper "Inversion-of-control layer" [11] by Sobernig and Zdun, which presents the pattern and one of its possible evolutions from a multi-modular architecture point of view.

While developing the `needle` library, Uber also made another library called `Poet`[1] to generate iOS projects with different module configurations. They used this code generation mechanism to generate test projects and benchmark the library's capabilities in different environments.

Methodology

We conducted a series of benchmarks and analyses to investigate the tradeoffs and performance impact of dependency injection (DI) systems in iOS. This section outlines the procedures and criteria used to evaluate the frameworks.

We developed a codegen tool that generates projects using each library under test, creating a large object graph; then we integrated this tool as a build-time plugin inside a benchmark suite that measures three different procedures:

1. How long does it take to create the object graph?

For the largest object graphs, this is typically done once, at the start of the lifetime of a mobile application. The typical app has to initialize an API client, maybe a local database, deeplink handlers, push notification handlers, some kind of secure storage for passwords and sensitive data, a logging system… and most of these objects are interdependent and never deallocated for the lifetime of the application. Measuring the creation of the dependency injection system's object graph captures how long it takes for each library to create this first network of objects, which, from here on, we will call the "DI container".

2. How long does it take to access all objects in the dependency injection container?

This represents a proxy for common usage of the above-mentioned objects. Apps are making API calls, saving things to databases and caches, and accessing all these kinds of dependencies all the time. We approximate the typical usage of the objects in the dependency graph by simulating a high count of sequential accesses to the objects in the DI container.

3. How long does it take to perform both tasks?

This value is used as a total score to compare the performance of each library.

Benchmarking Setup

The hardware used for benchmarking is a MacBook Pro equipped with an Apple M1 chip and 16GB of RAM. Software-wise, we used macOS Sonoma 14.5, Xcode 15.3, and Swift 5.10. The project setup involves a set of 3 benchmarks, each corresponding to one of the tasks above, the codegen tool that generates standardized projects, and a series of "project templates" used in combination with the codegen tool.

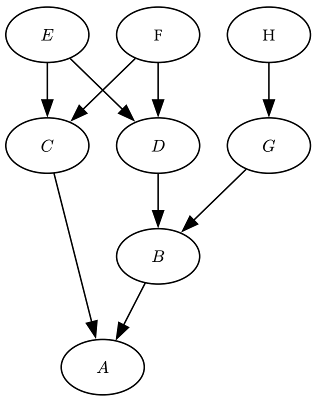
We identified four key metrics to evaluate the performance impact of each DI system: startup time, access time, memory usage, and instructions count:

- `Startup time` is the time used to generate the object graph container.
- `Access time` is the time used to access the different objects within the container.
- `Memory usage` represents how much RAM the system uses while performing the tasks (in megabytes).
- `Instruction count` represents how many instructions the generated project executes to perform the tasks.

These metrics capture the critical aspects of developer experience and runtime performance. To capture these metrics, we use a very recent benchmarking library called `SwiftBenchmark` [12]. This library allows us to record and compare different measurements and easily set up the project using the Swift Package Manager [13].

The baseline measurement provided a performance benchmark without any DI framework. This would serve as our control to understand the overhead introduced by each DI system. This baseline framework simply constructs each object in the correct order, like in Listing 1, and stores all of them in a dictionary keyed by the `ObjectIdentifier` of the type of object, shown in Listing 2. This allows for an easy retrieval of the object by type.

```swift
let a = A()
let b = B(a: a)
let c = C(a: a)
let d = D(b: b)
let e = E(c: c, d: d)
let f = F(c: c, d: d)
let g = G(b: b)
let h = H(g: g)
```



Listing 1: This piece of code initializes a number of objects in topological order. The arrows in the graph indicate a "depends on" relation.

```swift
[
    ObjectIdentifier(A.self): a as Any,
    ObjectIdentifier(B.self): b as Any,
    ObjectIdentifier(C.self): c as Any,
    ObjectIdentifier(D.self): d as Any,
    ObjectIdentifier(E.self): e as Any,
    ObjectIdentifier(F.self): f as Any,
    ObjectIdentifier(G.self): g as Any,
    ObjectIdentifier(H.self): h as Any
]
```

Listing 2: This is how the built objects graph is stored in the baseline project. ObjectIdentifier is a unique pointer to the metadata of the type in the Swift runtime, guaranteeing that we have a unique value for each type in the dictionary.

Development process

The process began by creating a small project generator able to take a generic graph, generate a class for each node and a property for each edge. For an edge e from A to B, the codegen would generate the following code:

```swift
class A {
  let b: B
  init(b: B) { self.b = b }
}
class B {
  init() {}
}
```

The next step was to add more generators, one for each library, according to each library documentation and best practices. Most libraries are very similar in their setup and operation, so that didn't take long, with one glaring exception: Uber's `needle` library also contained a code generation step. Figuring out how to chain two build plugins that depend on each other in the Swift Package Manager has been an interesting journey in itself.

The implementation of the benchmarks on top of the generated project didn't take long, as all generated projects adhere to the same protocol, so that we can use polymorphism to reuse the benchmark code regardless of which library is used internally. Specifically, the projects implement the following protocol:

```swift
protocol GeneratedProject {
    associatedtype Container
    // Create an object graph and store it in a box
    func makeContainer() -> Container
    // Access every object in the box
    func accessAllInContainer(_ container: Container)
}
```

To perform the benchmark we used the facilities provided by the `SwiftBenchmark` library: running the following script from the root folder of the project is enough to download dependencies, compile everything, generate the template projects, run the benchmarks, and generate a report in JMH format for later analysis:

```
swift package --allow-writing-to-package-directory benchmark --format jmh
```

The report contains informations for each of the three benchmarks: “Access all”, “Create container”, and “Complete run”.

To make sure that the benchmark operations were implemented correctly, we analyzed the behavior of the generated projects using Apple’s `Instruments` profiling tool. This required the addition of a new target to the project (named, creatively, “Profiler target”), specifically for running the generated projects in the profiler with all optimizations. For each integration, we repeated the performance tests using Instruments.

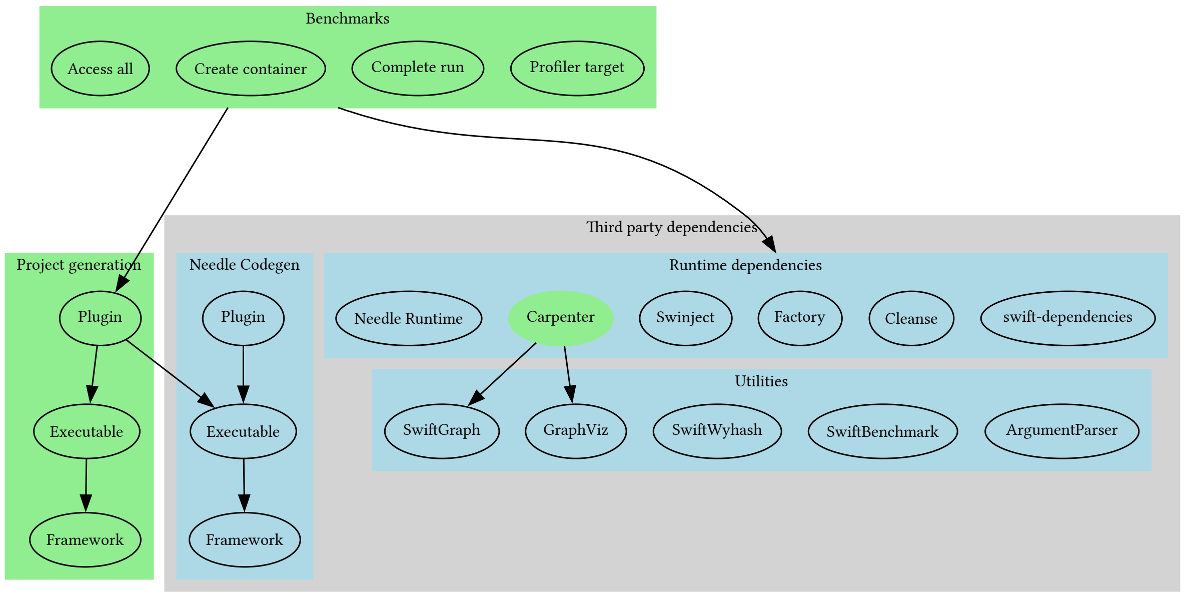


Figure 1: The green elements represent the contributions in this paper.

Benchmark results

Developing a Zero-Cost Dependency Injection System

Inspired by the zero-cost abstraction philosophy in C++, we embarked on designing our DI system with a focus on compile-time safety and minimal runtime overhead. The goal was to leverage Swift’s powerful type system to build a dependency graph at compile-time, thus eliminating the need for runtime resolution and associated overhead.

We began by implementing a mechanism to build the dependency graph using Swift's generics and protocols. This compile-time graph-building aimed to ensure that if the code compiled, the dependency graph was correctly constructed.

Next, we developed a code generation tool to automate the creation of boilerplate code for dependency resolution. This tool was designed to integrate seamlessly with the build process, minimizing any additional complexity for developers.

Finally, we focused on performance tuning, optimizing the generated code and the DI system for performance. Techniques such as caching and lazy initialization were employed to ensure minimal runtime overhead.

Evaluation

To evaluate the effectiveness of our zero-cost DI system, we conducted the same set of benchmarks. This provided a direct comparison against the mainstream DI frameworks, allowing us to see if our system truly achieved zero-cost abstraction.

Additionally, we conducted case studies on existing iOS projects. These case studies aimed to validate the practical applicability of our DI system, focusing on ease of integration, impact on developer productivity, and real-world performance.

Results and Discussion

The results of our benchmarks revealed fascinating insights into the tradeoffs and performance impacts of each DI framework.

Swinject, while highly versatile, introduced significant runtime overhead, especially in startup time and memory usage. Cleanse provided excellent compile-time guarantees, but its compile-time graph-building increased overall compilation time. Needle, optimized for large-scale applications, demonstrated impressive runtime performance with minimal overhead, though it required more complex setup and integration. Factory, with its lightweight functional approach, offered a good balance between simplicity and performance but lacked some advanced features. Swift-dependencies showed promise with modern Swift features but still had room for optimization.

In contrast, our zero-cost DI system demonstrated near-baseline performance across all metrics. Compilation times were slightly higher due to the code generation process, but startup times, memory usage, and execution times were nearly identical to the baseline. This validated our hypothesis that a zero-cost dependency injection system is achievable.

Case Study Findings

The case studies provided further insights into the real-world applicability of our DI system. Integration into existing projects was straightforward, with minimal changes required to the codebase. Developers reported increased productivity due to the compile-time guarantees and reduced debugging time. Real-world performance observations aligned with our benchmark results, confirming minimal runtime overhead.

Conclusion

This paper has explored the design, implementation, and evaluation of dependency injection systems in iOS. Through rigorous benchmarking and real-world case studies, we have demonstrated that a zero-cost dependency injection system is not only possible but also practical. Our findings provide a valuable tool for iOS developers seeking to manage complexity in large-scale applications without compromising performance.

Future Work

Future work could explore further optimizations to the zero-cost DI system, such as integrating with SwiftUI and other modern Swift features. Additionally, extending the benchmarking to other platforms and languages could provide a more comprehensive understanding of DI systems' performance impacts. Expanding the adoption of the zero-cost DI system within the iOS developer community could also yield valuable feedback and improvements.

Bibliography

[1] Uber, "Introducing Uber Poet, an Open Source Mock App Generator for Determining Faster Swift Builds." [Online]. Available: https://www.uber.com/en-NL/blog/uber-poet/?uclick_id=76466083-c911-4bb1-9435-322d5fe3156c

[2] R. C. Martin, *Agile software development: principles, patterns, and practices*. Prentice Hall PTR, 2003.

[3] M. Fowler, "Inversion of control containers and the dependency injection pattern," 2004, [Online]. Available: http://www.martinfowler.com/articles/injection.html

[4] M. Seemann, *Dependency Injection in .NET*. Manning, 2011. [Online]. Available: https://books.google.nl/books?id=lnOqcQAACAAJ

[5] Square, "Dagger 1." [Online]. Available: https://github.com/square/dagger

[6] Google, "Dagger 2." [Online]. Available: https://github.com/google/dagger

[7] Uber, "needle." [Online]. Available: https://github.com/uber/needle

[8] M. Long, "Factory." [Online]. Available: https://github.com/hmlongco/Factory

[9] S. Celis and B. Williams, "swift-dependencies." [Online]. Available: https://github.com/pointfreeco/swift-dependencies

[10] B. Stroustrup, "Foundations of C++," in *Programming Languages and Systems*, H. Seidl, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 1–25.

[11] S. Sobernig and U. Zdun, "Inversion-of-control layer," in *Proceedings of the 15th European Conference on Pattern Languages of Programs*, in EuroPLoP '10. Irsee, Germany: Association for Computing Machinery, 2010. doi: 10.1145/2328909.2328935.

[12] ordo-one, "Swift Package Benchmark." [Online]. Available: https://github.com/ordo-one/package-benchmark

[13] Apple, "Swift Package Manager." [Online]. Available: https://www.swift.org/documentation/package-manager/