Combating Technical Debt with the Power of Dependency Inversion in Swift

Daniel Zhang
June 30, 2023

Contents

1 Dependency Inversion Principle Summary

1. **Concrete Dependencies:** Creating concrete dependencies can go against SOLID principles and lead to technical debt.
2. **Dependency Inversion Principle:** The dependency inversion principle matches the "D" in SOLID and goes by "DIP" for short.
3. **Reducing Technical Debt:** Nipping embedded concrete dependencies in the bud can save time and money in the long run.

1.1 A Real-World Example

At the outset of a new project, it can be challenging to anticipate the costs associated with allowing concrete dependencies to become deeply embedded in the codebase – particularly when the primary challenge is simply getting the project off the ground. However, over time, as the use of concrete types

increases, technical debt can accumulate and compound. It can drag down the development process, making adding new features, fixing bugs, and delivering customer value more difficult.

This article will explore how to address technical debt from the outset of development using the power of dependency inversion. To illustrate these concepts, I will draw from my work experience with a real-world example to illustrate these concepts.

1.2 Abstracting a Dependency for Reusable Charting

We use a charting library, Charts by Daniel Gindi (Swift package version 4.1.0), to display line chart data in our iOS and macOS apps for visualizing heart rate records in HealthKit. It is open-source, available on GitHub, and has equivalent functionality to an Android library for the same purpose.

Chart data consists of arrays of ChartDataEntry. It is a concrete type representing a single data point on a chart. It has three properties we use: x, y, and data.

- **x:** The x property is a Double that represents the x-value of the data point.
- **y:** The y property is a Double that represents the y-value of the data point.
- **data:** The data property is an optional "Any" representing additional data associated with the data point.

To prevent concreting our use of chart data, we can create an abstract interface that represents the data we need to display on a chart by first creating a protocol.

```
1 protocol ChartDataProtocol
2 {
3     associatedtype MeasurementDate
4     var x: Double { get set }
5     var y: Double { get set }
```

```
    var data: MeasurementDate { get set }
}
```

The power of the associated type in the protocol lets us elegantly specify the type of the data property. We can then create a concrete implementation of this interface that conforms to the protocol.

```
struct ChartDataProvider: ChartDataProtocol
{
    typealias MeasurementDate = Date
    var x: Double
    var y: Double
    var data: MeasurementDate
}
```

Finally, in the few places where we need to depend on the imported concrete type, we can transform our custom type into the concrete type for the dependency with a one-line function in an extension.

```
import Charts

extension ChartDataProvider
{
    /// - Returns: A ChartDataEntry object with values from the struct.
    func toChartDataEntry() -> ChartDataEntry
    {
        ChartDataEntry(x: x, y: y, data: data)
    }
}
```

Retrieving chart data from the data source and converting it into a ChartDataEntry object has been streamlined and optimized for efficiency.

- **Simplified Procedure:** We can now easily achieve what we need by invoking the .toChartDataEntry() method on instances of the ChartDat-

aProvider struct.

- **Optimized Transformation:** With a time complexity of O(n), converting an array of ChartDataProvider instances into a corresponding array of ChartDataEntry objects stays efficient.

```
let timeBins = TimeBinCalculator
    .createTimeBins(for: entries as [ChartDataProvider])
let concreteTimeBins: ([ChartDataProvider]) -> [ChartDataEntry] =
{ entries in
    entries.map
    { entry in
        entry.toChartDataEntry()
    }
}
let timeBinSet = LineChartDataSet(entries: concreteTimeBins(timeBins))
```

This small effort allows us to work abstractly with chart data entries without repeatedly importing the Charts library into our codebase. Making abstraction a habit can help us avoid technical debt and increase flexibility over the long term.

1. **Library Swapping:** It also allows us to swap out one library for another, which can benefit multiplatform development.
2. **Streamlining Code Dependencies:** When we reduce the number of concrete dependencies in our codebase by a factor of ten or more, we can reap significant benefits.
3. **UI Dependencies:** We free ourselves from tightly-coupled UI dependencies in our view models.

The benefits go on and on. Let's wrap things up by detailing the key concepts demonstrated in this article.

2 Navigating Dependency Inversion with Protocols and Polymorphism in Swift

Investing in minor upfront planning to abstract dependencies can pay off by reducing the cost of change. It is particularly true when the project is likely to undergo significant changes in the future. In the context of a charting library, the primary purpose is presentation.

As UI tends to undergo dramatic changes over time, like with SwiftUI after UIKit and AppKit, keeping it as decoupled from the rest of the codebase as possible is essential.

Swift protocols and polymorphism are two powerful tools for implementing the Dependency Inversion Principle (DIP), one of the foundational principles of SOLID software architecture.

1. **Protocols as Abstractions**: In Swift, protocols act as a blueprint of methods, properties, and other requirements. They provide a layer of abstraction between high-level and low-level modules in a system. High-level modules define protocols that low-level modules conform to. This way, high-level modules are not directly dependent on low-level modules; they both depend on abstractions.
2. **Polymorphism**: Polymorphism allows us to treat different types as the same general type. In Swift, this means that a function or method can accept parameters of a protocol type, working with any type that conforms to the protocol. This allows high-level modules to remain decoupled from the specific type of low-level modules.
3. **Protocol Extensions**: Swift allows default implementations of methods and computed properties in protocols using protocol extensions. This helps reduce the amount of boilerplate code and allows for better abstraction of behavior that might be common to several classes.

4. **Protocol Composition**: Swift supports combining multiple protocols together. A type can conform to multiple protocols, and a function or method can work with the parameters of a composite protocol type. This provides high flexibility and allows a class to be built from reusable components.
5. **Associated Types**: Protocols in Swift support associated types, providing a way to use a placeholder type in protocols that can be defined when the protocol is adopted. This gives us another tool for creating flexible and reusable code components.

Through these features, Swift protocols and polymorphism offer an elegant and efficient way to implement the Dependency Inversion Principle, allowing us to write more flexible, reusable, and maintainable code.

3 Closing Thoughts and Engagement

As we end this discourse, it's essential to note that the Dependency Inversion Principle (DIP) can greatly influence your projects' maintainability, flexibility, and scalability. Swift provides us with powerful tools to employ it and effectively combat technical debt before it happens. However, it's ultimately up to us as developers to consciously incorporate these practices into our workflows.

Now, it's over to you. I would love to hear how you have been able to apply the principles discussed in this article to your projects. Has this piece helped you understand and implement DIP better? Or maybe you've encountered challenges or questions that were not covered? Please feel free to share your thoughts, experiences, and suggestions for improvement in the comments section below. Your feedback is invaluable and will help in refining future articles and discussions.

Remember, the goal is continuous improvement and learning. Let's embark on this journey together, one codebase at a time.