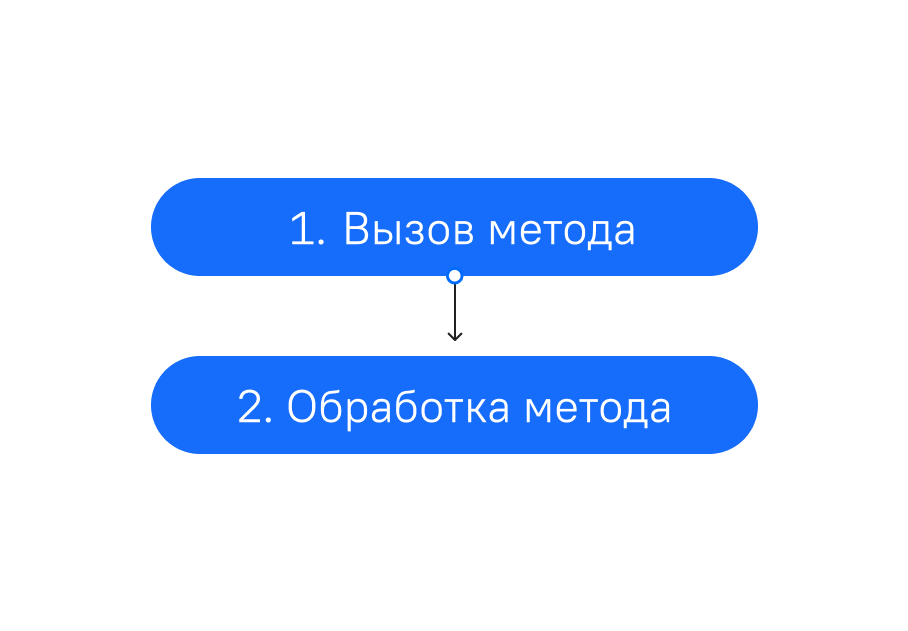
Все iOS-разработчики так или иначе сталкиваются с диспетчеризацией (**Method Dispatch**), но далеко не каждый понимает, как это работает. Зная, как проходит процесс диспетчеризации под капотом программы, вы можете повысить производительность своего кода.

В этой статье мы разберем типы диспетчеризации, их плюсы и минусы, а также затронем один из распространённых багов.

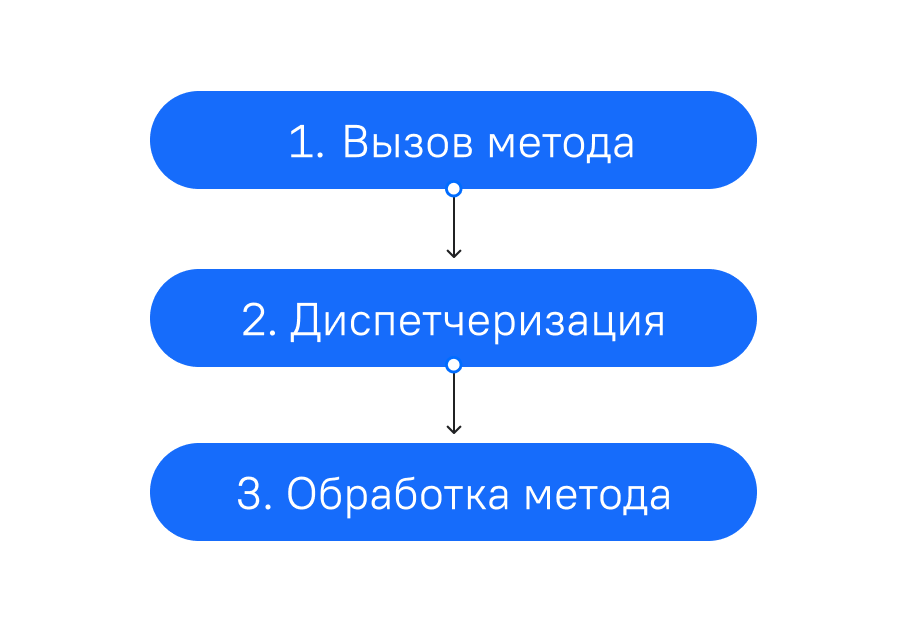
Материал будет полезен для всех iOS-разработчиков, которые хотят улучшить производительность своего кода. Кроме того, этот материал поможет при подготовке к собеседованию, где вопросы о Method Dispatch встречаются достаточно часто.

### **Что такое диспетчеризация в Swift?**

Это процесс, при котором программа выбирает, какие инструкции выполнить при вызове метода. Диспетчеризация происходит каждый раз, когда вызывается метод. Многим кажется, что это происходит так:

Рис 1. Неправильное понимание отработки метода.

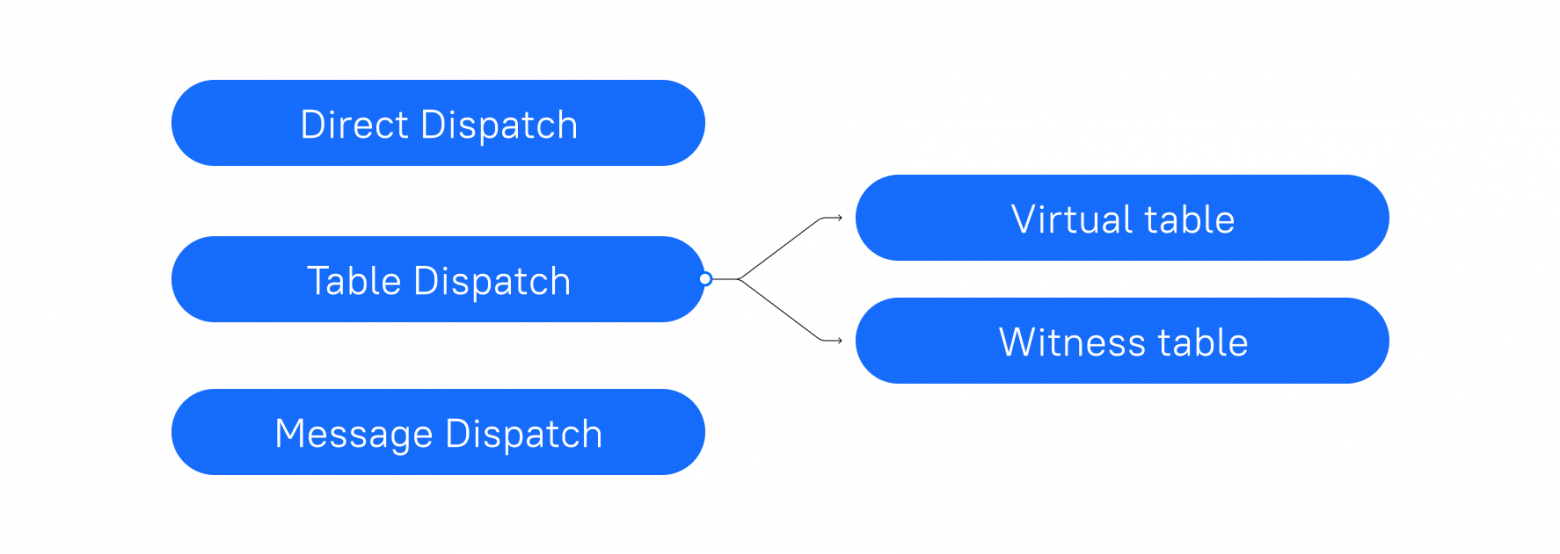
Это заблуждение, потому что есть промежуточный пункт — это диспетчеризация.

Рис 2. Правильное понимание отработки метода.

Цель диспетчеризации состоит в том, чтобы программа сообщила центральному процессору, где в памяти он может найти исполняемый код для вызова конкретного метода.

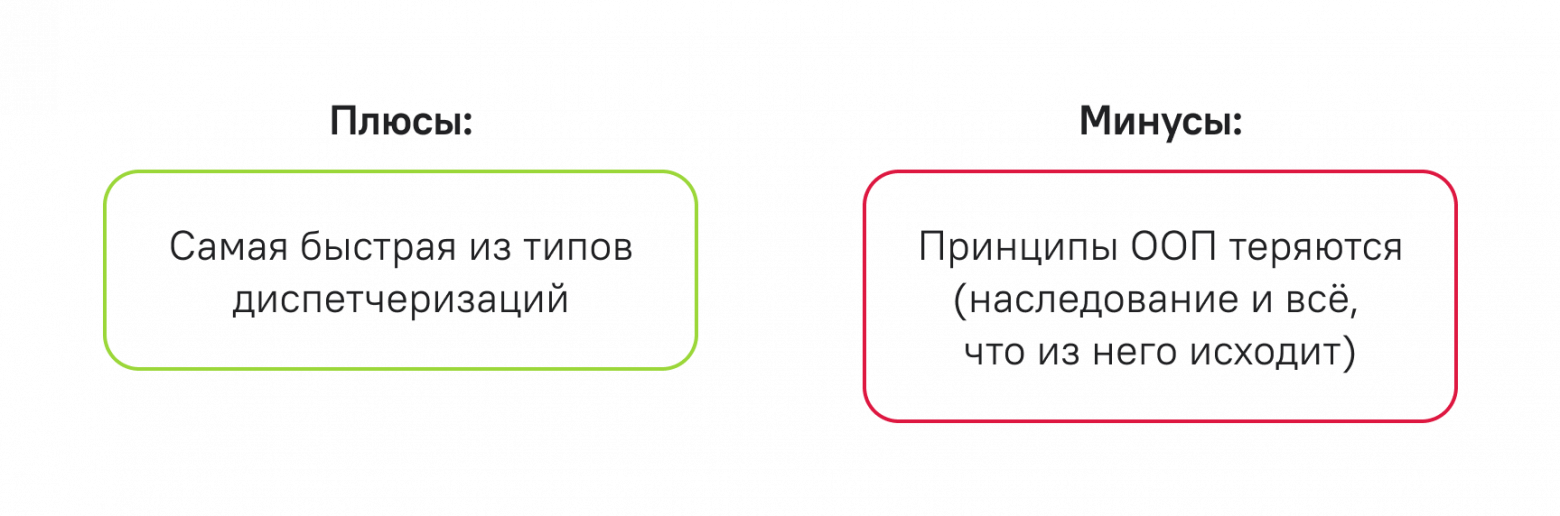
### **Типы диспетчеризации**

Перейдем к типам диспетчеризации на языке Swift. Существует три вида: **Direct Dispatch**(статическая), **Table Dispatch**(динамическая, в свою очередь, делится на Virtual Table и Witness Table) и **Message Dispatch**(самая динамическая диспетчеризация).

Рис 3. Типы диспетчеризации.

#### **1. Direct Dispatch**

Direct Dispatch — это **самый быстрый**способ отправки метода, часто называют статической диспетчеризацией. Однако прямая отправка наиболее ограничивающая с точки зрения программирования и недостаточно динамична для ООП. У всех value-объектов (например, структуры) используется Direct Dispatch.

Рис 4. Плюсы и минусы Direct Dispatch.

Рассмотрим практические примеры Direct Dispatch:

**Final Class**

// MARK: Example 1 - Final Class

**final** **class** **ClassExample1** {

// MARK: Direct Dispatch

**func** **doSomething**() {

print("Example 1 - Final Class")

}

}

При добавлении final становится недоступным наследование этого класса, соответственно, метод тоже нельзя оверрайднуть. Возможно, вы часто видели такое и не понимали, для чего это делается. А это один из плюсов — изменение диспетчеризации на Direct Dispatch и, соответственно, улучшение производительности кода. Ну и защита от наследования, когда оно не нужно, например, при создании класса ViewController, единственного в приложении.

**Protocol Extension**

// MARK: Example 2 - Protocol Extension

**protocol** **SomeProtocol** {

**func** **doSomething**()

}

**extension** **SomeProtocol** {

// MARK: Direct Dispatch

**func** **doSomething**() {

print("Example 2 - Protocol Extension")

}

}

**class** **ClassExample2**: **SomeProtocol** {}

**let** classExample2 = ClassExample2()

classExample2.doSomething()

При реализации дефолтного метода протокола с помощью его расширения, диспетчеризация c Witness Table (об этом больше информации представлено ниже) меняется на Direct Dispatch.

**Class Extension**

// MARK: Example 3 - Class Extension

**class** **ClassExample3** {}

**extension** **ClassExample3** {

// MARK: Direct Dispatch

**func** **doSomething**() {

print("Example 3 - Class Extension")

}

}

**let** classExample3 = ClassExample3()

classExample3.doSomething()

При написании метода класса в его расширении данный метод нельзя будет оверрайднуть, и он меняет свою диспетчеризацию на Direct Dispatch.

**Access Control**

// MARK: Example 4 - Access Control

**class** **ClassExample4** {

**func** **doSomething**() {

doSomethingPrivate()

}

// MARK: Direct Dispatch

**private** **func** **doSomethingPrivate**() {

print("Example 4 - Access Control")

}

}

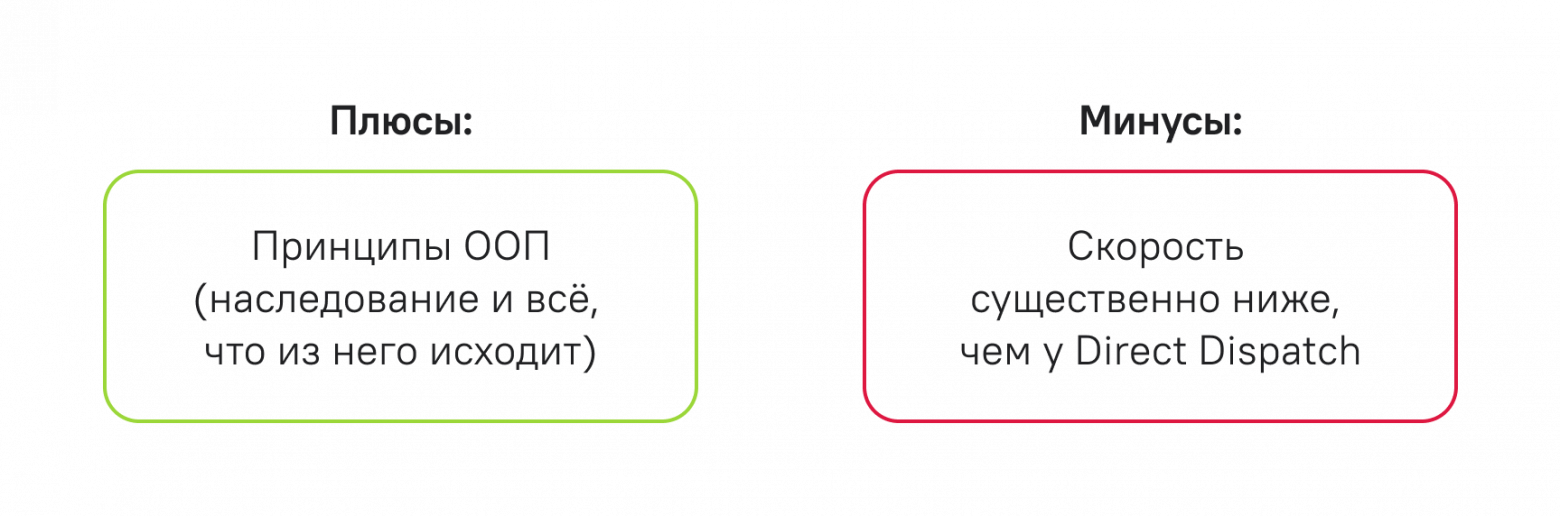
К приватному методу нет доступа, чтобы его как-то переписать у наследника (сабкласса), и соответственно он меняет свою диспетчеризацию на Direct Dispatch.

Данные практические примеры Direct Dispatch используются при разработке приложений для улучшения производительности кода, а также для защиты участков кода от переписывания сабклассами. Они помогут повысить скорость вашего кода и сделать его более профессиональным, старайтесь не забывать об этом.

#### **2. Table Dispatch**

**Virtual Table**

Virtual Table используется в наследовании. Для каждого класса и его наследника (сабкласса) создается виртуальная таблица (пример приведён ниже), по которой центральный процессор понимает, где искать нужную ссылку на метод для его выполнения. Главный минус динамической диспетчеризации в том, что ее **скорость существенно ниже, чем у статической**.

Рис 5. Плюсы и минусы Virtual Table.

Рассмотрим Virtual Table на практическом примере:

// MARK: Example 5 - Virtual Table

**class** **ClassExample5** {

**func** **doSomething**() {

print("Example 5 - Virtual Table")

}

}

**class** **SubclassExample5**: **ClassExample5** {

**override** **func** **doSomething**() {

print("Override for subclass")

}

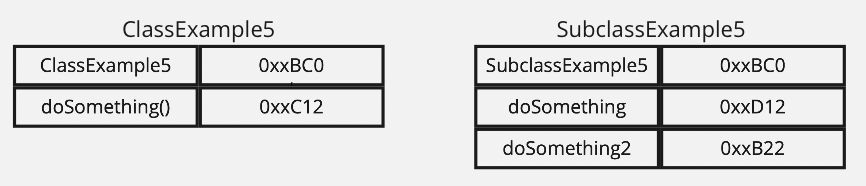
**func** **doSomething2**() {

print("Method of subclass")

}

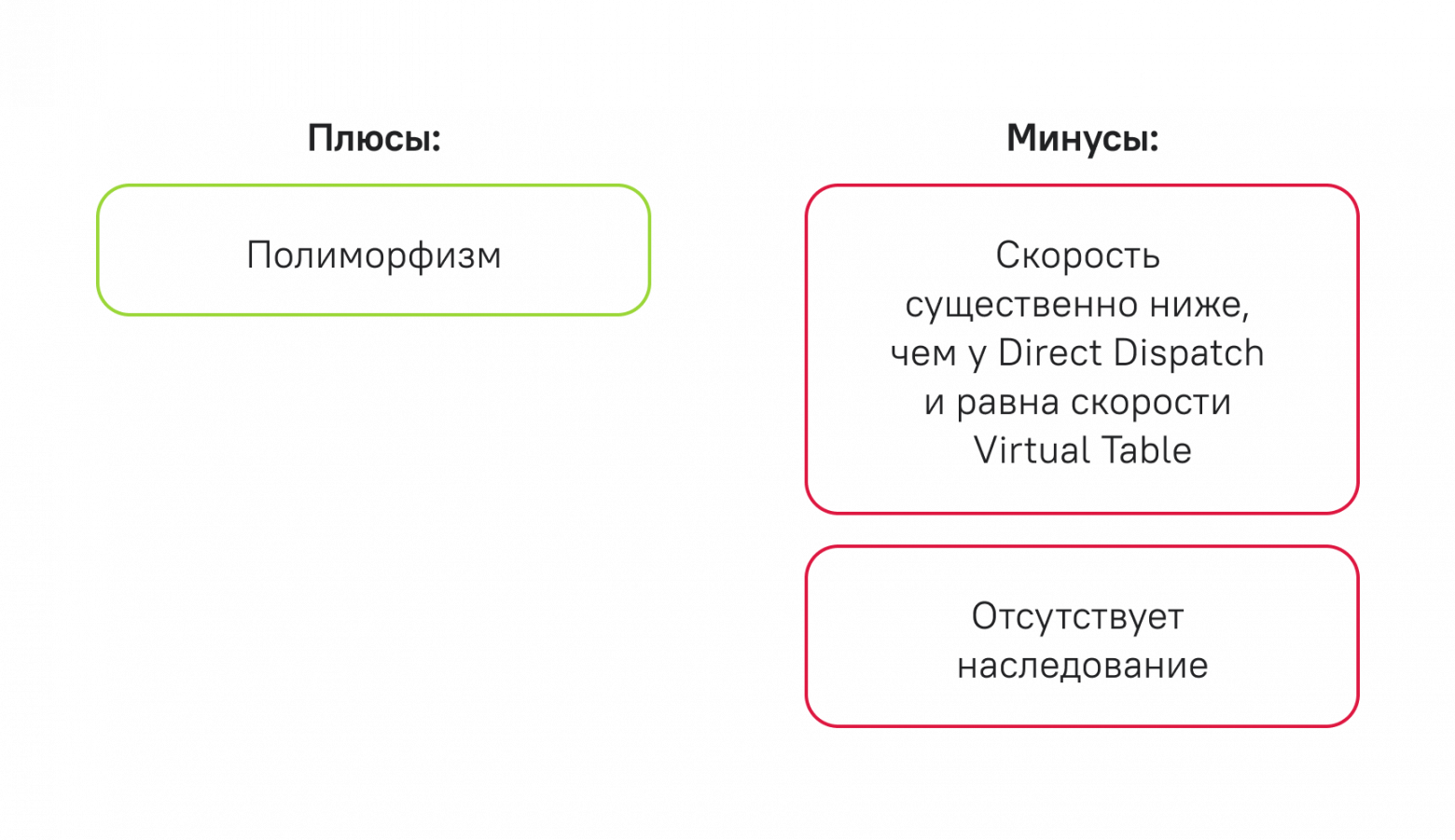
}

Создается класс и его сабкласс, для каждого из этих классов создается отдельная виртуальная таблица.

Рис 6. Virtual Table для данного кода.

**Witness Table**

Witness Table используется для реализации протоколов и создается для каждого класса, реализовавшего протокол. По этой таблице центральный процессор понимает, где искать нужную ссылку на метод для его выполнения. Главный минус Witness Table такой же, как и у Virtual Table — **скорость существенно ниже, чем у Direct Dispatch**.

Рис 7. Плюсы и минусы Witness Table.

Рассмотрим Witness Table на практическом примере:

// MARK: Example 6 - Witness Table

**protocol** **ProtocolExample6** {

**func** **doSomething**()

}

**class** **ClassExample6**: **ProtocolExample6** {

**func** **doSomething**() {

print("Example 6 - Witness Table")

}

}

**class** **AnotherClassExample6**: **ProtocolExample6** {

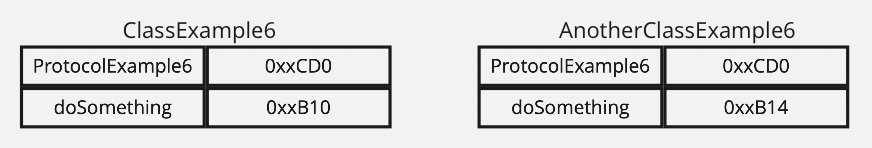
**func** **doSomething**() {

print("Hello World")

}

}

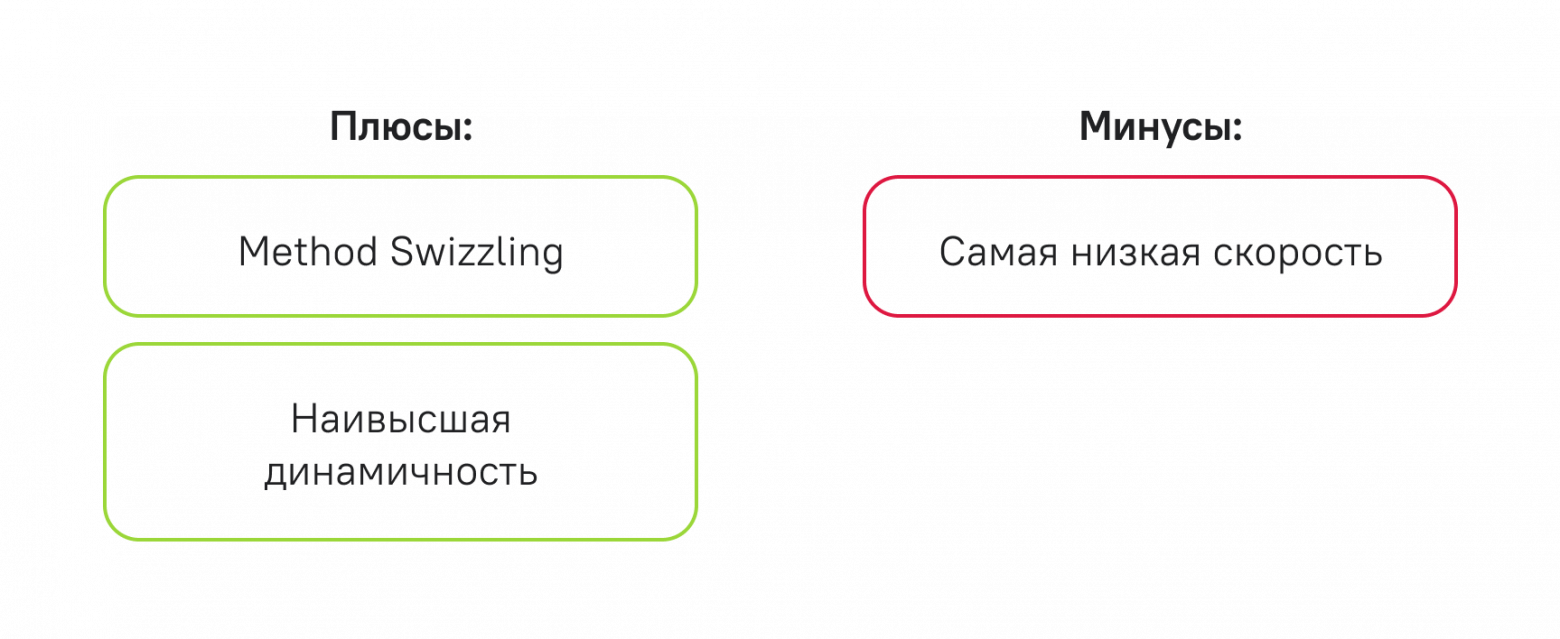
Создается протокол и два класса, которые реализуют этот протокол, для каждого из этих классов создается Witness Table.

Рис 8. Witness Table для данного кода.

#### **3. Message Dispatch**

Message Dispatch — это **самый динамичный**вызов метода с помощью Objective-C. Message Dispatch работает в рантайме и показывает, какой метод вызывать, то есть проверяет это в реальном времени. Message Dispatch лежит в основе KVO (и соответственно в реактивном программировании), UIAppearance, CoreData. Так как Message Dispatch работает в рантайме, соответственно, можно подменять реализацию методов — это называется **Method Swizzling**. Method Swizzling позволяет подменить метод вашим в рантайме, оставляя оригинальную имплементацию доступной. А также в рантайме можно менять экземпляры класса.

Message Dispatch часто используется для тестирования кода. Редко его можно встретить в проде, так как это не очень безопасный вызов и относится к **самой медленной диспетчеризации**. Но иногда он встречается и в проде, когда существующая библиотека не может обеспечить нам нужный результат и разработчик делает замену метода на свой в рантайме с помощью Method Swizzling.

Рис 9. Плюсы и минусы Message Dispatch.

Для реализации Message Dispatch требуется префикс "@objc dynamic", или можно добавить @objcMembers перед классом, тогда все его методы станут с префиксом @objc по дефолту.

Рассмотрим пример из того, что разобрали выше:

// MARK: Example 7 - All types of dispatching

**protocol** **ProtocolExample7** {

**func** **doSomethingWithWitnessTable**()

}

**class** **ClassExample7**: **NSObject** {

@objc **dynamic** **func** **doSomething**() {

print("Example 7 - Message Dispatch")

}

}

**class** **SubclassExample7**: **ClassExample7**, **ProtocolExample7** {

**private** **func** **doSomethingWithDirectDispatch**() {

print("Direct Dispatch")

}

**func** **doSomethingWithVirtualTable**() {

print("Virtual Table")

}

**func** **doSomethingWithWitnessTable**() {

print("Witness Table")

}

@objc **override** **dynamic** **func** **doSomething**() {

print("Override with Message Dispatch")

}

}

Тут присутствуют все типы диспетчеризации. Так как остальные типы мы уже разобрали сверху, разберем только Message Dispatch. В рантайме при вызове метода doSomething сначала будет выполняться поиск в данном сабклассе, если имплементация метода не найдется, то будет поиск по родительскому классу. Если и там его не окажется, то поиск переходит в класс NSObject. Если метод не найдется, а такое может произойти при указании метода через функцию performSelector, то случится краш с ошибкой: "NSInvalidArgumentException: unrecognized selector sent to instance".

### **Баг с диспетчеризацией**

Вместе с тем в Swift существуют различные баги с диспетчеризацией от Apple. Многие их них были пофикшены с выходом новых версий языка. Ниже разберем **баг с протоколом**, который до сих пор присутствует при разработке, и приведём пример, как его пофиксить.

// MARK: Example 8 - First Bug

**protocol** **ProtocolExample8** {}

**extension** **ProtocolExample8** {

**func** **doSomething**() {

print("Default Implementation")

}

}

**class** **ClassExample8**: **ProtocolExample8** {

**func** **doSomething**() {

print("Required Implementation")

}

}

**let** first = ClassExample8()

**let** second: ProtocolExample8 = ClassExample8()

first.doSomething()

second.doSomething()

По логике, два раза в консоль должна выводиться строчка "Required Implementation", но реальный результат таков:

Required Implementation

Default Implementation

Это происходит, потому что во втором случае система выбирает дефолтную реализацию протокола и это Direct Dispatch, а мы осуществляем в нашем классе свою реализацию, и под капотом этот метод переходит в Witness Table.

Для того, чтобы программа поняла, что в нашем случае мы используем Witness Table, нам нужно добавить этот метод в протокол.

**protocol** **ProtocolExample8** {

**func** **doSomething**()

}

После добавления результат становится правильным:

Required Implementation

Required Implementation

### **Итоги**

Итак, мы разобрали, как работает диспетчеризация. Это обширная тема, изучив которую, вы поймете, как методы реализуются под капотом на самом деле. Например, сможете смело заменять многие методы на Direct Dispatch, где ранее использовалась другая диспетчеризация по умолчанию. От этого ваш код станет лучше и быстрее.