Профессиональное программирование на Swift

Урок 7

Рефакторинг

Подходы к поиску и устранению ошибок. Рефакторинг.

Отладка программы

Отладка — это этап разработки программы, на котором обнаруживают, локализуют и устраняют ошибки. Найти ошибку в программе — здорово, особенно до того, как заказчик увидел работу. Обнаружение багов на ранней стадии разработки означает, что мы делаем свое дело и качество нашей работы повышается. Если ошибка появляется после публикации приложения, это совсем не здорово.

Отладку программы начинают с составления плана тестирования. Его должен представлять себе любой программист. Составление плана опирается на понятие об источниках и характере ошибок. Основными источниками являются недостаточная проработка алгоритма реализации функциональности, неверная работа с данными, невнимательность при написании кода.

К сожалению, ошибки совершают все, даже профессиональные программисты. И не стоит биться об заклад, что ваш код безгрешен, — это утопия.

Предположим, при отладке мы обнаружили ошибку. Поговорим о том, что делать дальше — как найти ошибочный участок кода и правильно исправить баг.

Неэффективные подходы к устранению ошибки

Когда обнаруживается ошибочная ситуация, программисты часто начинают гадать: что случилось, где ошибка — в этом коде или в том методе? Ищут способ «по-быстрому» ее исправить и пытаются написать для этого невнятный код. В большинстве случаев это не проходит, а только больше запутывает. Код, написанный с опорой на необоснованные предположения, будет вносить дополнительную энтропию в программу.

Еще один неэффективный подход — **отказ от анализа проблемы**. Желание побыстрее исправить ошибку играет с нами плохую шутку. Мы отказываемся внимательно изучать код, анализировать проблему и пытаемся быстро решить ее. В результате — тратим больше времени на написание кода для наших предположений и затягиваем решение. Лучше спокойно и всесторонне изучить проблему и только потом выносить вердикт и писать уже обоснованный код.

Старайтесь избегать исправления ошибки самым очевидным способом. Только после изучения и бага, и решения для его устранения, начинайте воплощать его в коде.

Рассмотрим пример ошибки и ее решения очевидным, быстрым способом. Это функция расчета последовательности чисел Фибоначчи с заданным шагом:

```
func fibonacci(steps: Int) -> [Int] {
    var result = [0, 1]

    var num1 = result[0]
    var num2 = result[1]
    for _ in 0 ..< steps {
        let nextNum = num1 + num2
        num1 = num2
        num2 = num1 // Здесь ошибка, должно быть num2 = nextNum

        result.append(nextNum)
    }

    return result
}
```

В приведенном выше коде совершена ошибка, но для шага, равного 2 (steps=2), функция работает замечательно.

```
print(fibonacci(steps: 2))

// [0, 1, 1, 2]
```

При отладке было выяснено, что данная функция выдает неверные значения при шаге, равном 3 (steps=3):

```
print(fibonacci(steps: 3))

// [0, 1, 1, 2, 2] - Ошибка!
// вместо
// [0, 1, 1, 2, 3]
```

Было принято быстрое и ошибочное решение, которое все же исправило работу функции при шаге, равном 3 (steps=3).

```
func fibonacci(steps: Int) -> [Int] {
    var result = [0, 1]

    var num1 = result[0]
    var num2 = result[1]
    for step in 0 ..< steps {
        let nextNum = num1 + num2
        num1 = num2
        num2 = num1 // Здесь ошибка, должно быть num2 = nextNum

        // Заглушка! Ошибка будет проявляться для steps > 3
        if step == 2 {
            result.append(3)
        } else {
            result.append(nextNum)
        }
    }

    return result
}


print(fibonacci(steps: 3))

// [0, 1, 1, 2, 3]
```

Но ошибка работы функции **fibonacci()** сохранилась. И будет проявляться при шагах, превышающих 3 (steps > 3).

Эффективные подходы к устранению ошибки

Один из эффективных подходов — **сбор данных с помощью повторяющихся экспериментов.** Реализуем его в примере с функцией **fibonacci()**. Собираем данные о том, какое возвращаемое значение функции **fibonacci()** будет при различных шагах.

```
print(fibonacci(steps: 7))
// [0, 1, 1, 2, 2, 2, 2, 2]

print(fibonacci(steps: 4))
// [0, 1, 1, 2, 2, 2]

print(fibonacci(steps: 3))
// [0, 1, 1, 2, 2]

print(fibonacci(steps: 2))
// [0, 1, 1, 2]
```

По результатам сбора данных понимаем, что функция **fibonacci()** некорректно работает для всех шагов, начиная с 2. Принимать поспешное решение о заглушке с шагом 3 мы бы уже не стали и сэкономили время на исправление этой ошибки.

Далее **формируем гипотезу**, что явилось причиной бага — ошибка в функции **fibonacci()** проявляется для всех шагов больше 2.

При необходимости можем разработать дополнительные эксперименты для проверки и/или опровержения гипотезы.

```
print(fibonacci(steps: 6))
// [0, 1, 1, 2, 2, 2, 2, 2] - Не верно!

print(fibonacci(steps: 5))
// [0, 1, 1, 2, 2, 2, 2] - Не верно!
```

Можем провести многократный повтор экспериментов, чтобы удостовериться в повторяемости ошибки и в том, что определенный нами фрагмент — это и есть код, приводящий к ошибке.

Советы по поиску ошибки

Полезный прием для нахождения кода с ошибкой — **сократить подозрительную область**. Целенаправленно удаляем или комментируем блоки кода, пока он не будет работать правильно. Затем выполняем обратную операцию — возвращаем или раскомментируем блоки. Делаем так до тех пор, пока код снова станет выдавать ошибку. Так уменьшаем область поиска ошибки и в большинстве случаев останавливаемся на конкретной ошибочной команде.

Рассмотрим это на примере с функцией **fibonacci()**:

```
func fibonacci(steps: Int) -> [Int] {
    var result = [0, 1]

    var num1 = result[0]
    var num2 = result[1]
    for _ in 0 ..< steps {
        let nextNum = num1 + num2
```

```
        // Убираем код дальнейших итераций
        // num1 = num2
        // num2 = num1

        result.append(nextNum)
    }

    return result
}

print(fibonacci(steps: 4))
// [0, 1, 1, 1, 1, 1]

print(fibonacci(steps: 3))
// [0, 1, 1, 1, 1]
```

Видим, что после комментирования кода дальнейших итераций подсчет **nextNum** и формирование результирующего массива **result** отрабатывает правильно. Значит, ошибка скрыта в закомментированном коде.

Обычно причиной подобных ошибок в уже работающих функциях являются **недавние изменения** — либо в текущем коде, либо в зависимом. Поэтому проверяйте недавно реализованный код на баги. Особенно хорошо в этом помогают системы контроля версии.

Не стремитесь написать сразу весь код, и лишь потом озаботиться его проверкой. Это порочная практика программирования. Большой пласт кода тяжело проверять, особенно если это связанный код. К концу проверки подобных пластов кода накапливается усталость, от некоторых проверок просто отказываются, якобы уверенные в исправности фрагментов. Поэтому **вносите изменения постепенно, шаг за шагом**, с проверкой.

Бывают тупиковые случаи, когда нет идей, как исправить ошибку. С этим сталкивается большинство программистов. Несколько советов:

- **Метод утенка** — психологический метод решения задачи, делегирующий ее мысленному помощнику. Если часть кода не работает, ставим перед собой игрушечного утенка (или представляем его, или заменяем на любой другой предмет) и пытаемся описать суть проблемы, задать вопрос о решении — как живому человеку, словно игрушка действительно сможет ответить. Считается, что правильная формулировка вопроса содержит как минимум половину ответа. Также это дает толчок мыслям, помогая найти решение проблемы и двигаться дальше.

  Если работаете в команде, обсудите проблему с коллегой. Изложение проблемы может подтолкнуть к ее решению, или вы получите помощь и сэкономите время.

- Еще один действенный совет, которым многие программисты пренебрегают — **отдохните от проблемы**, переключитесь. Когда после отдыха вернетесь к вопросу, сможете рассмотреть его уже под другим углом, что может привести к решению.

Устранение ошибки

Прежде чем браться за решение проблемы — **поймите ее**. Зачастую простого анализа критической области кода недостаточно — необходимо **понять программу целиком,** взаимосвязи внутри нее.

Подтвердив диагноз, **не торопитесь** и вносите изменения шаг за шагом, чтобы не породить новую энтропию, ошибку в коде.

Устраните проблему, а не симптомы.

Вернемся к примеру с функцией **fibonacci()**. Мы определили проблемный участок кода и вывели гипотезу — ошибка кроется в команде **‘num2 = num1’** (должно быть ‘**num2 = nextNum**’). Исправим код функции **fibonacci()**.

```
func fibonacci(steps: Int) -> [Int] {
    var result = [0, 1]

    var num1 = result[0]
    var num2 = result[1]
    for _ in 0 ..< steps {
        let nextNum = num1 + num2
        num1 = num2
        num2 = nextNum

        result.append(nextNum)
    }

    return result
}
```

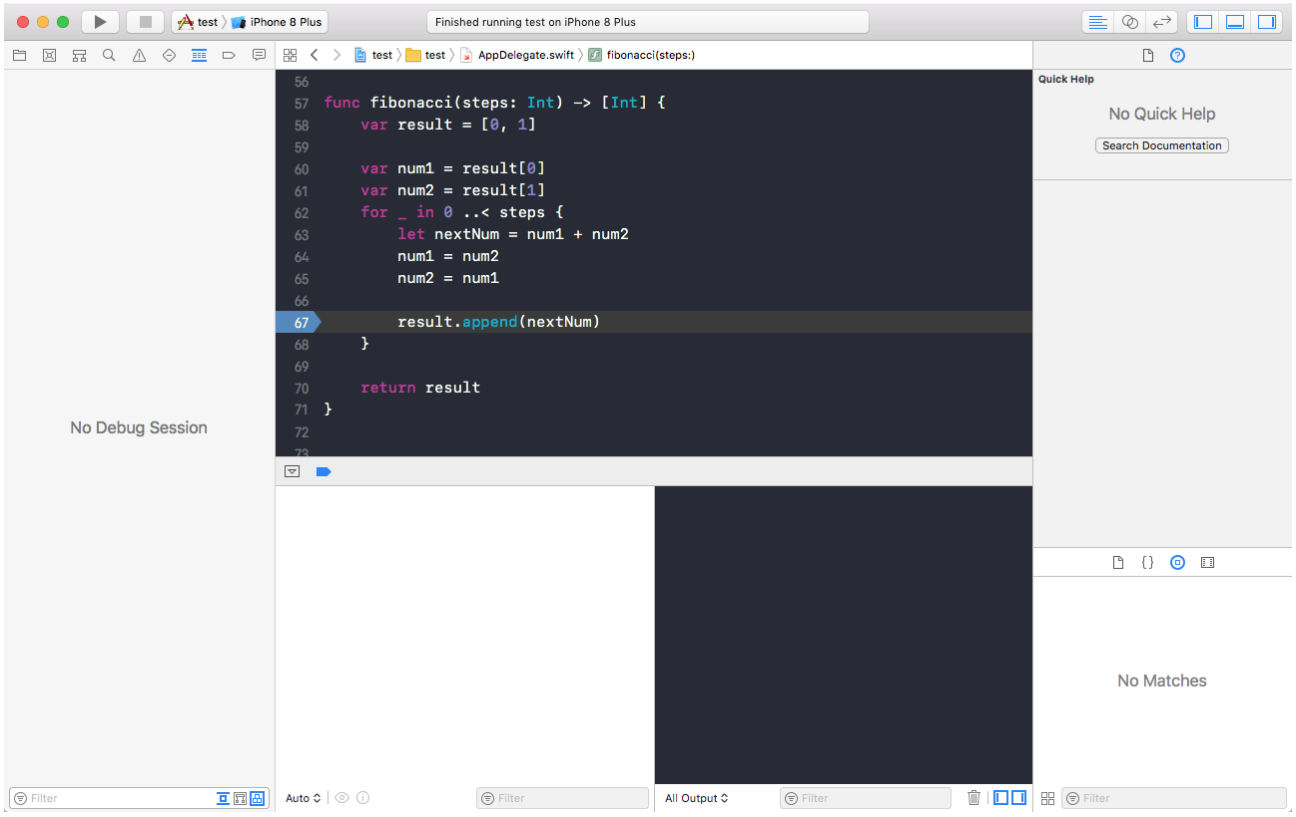
Когда ошибка найдена, а код вставлен, **обязательно проверьте внесенные исправления!** Только после этого можно быть уверенным, что ошибка устранена.

```
print(fibonacci(steps: 7))
// [0, 1, 1, 2, 3, 5, 8, 13, 21]

print(fibonacci(steps: 6))
// [0, 1, 1, 2, 3, 5, 8, 13]

print(fibonacci(steps: 5))
// [0, 1, 1, 2, 3, 5, 8]

print(fibonacci(steps: 4))
// [0, 1, 1, 2, 3, 5]

print(fibonacci(steps: 3))
// [0, 1, 1, 2, 3]

print(fibonacci(steps: 2))
// [0, 1, 1, 2]
```

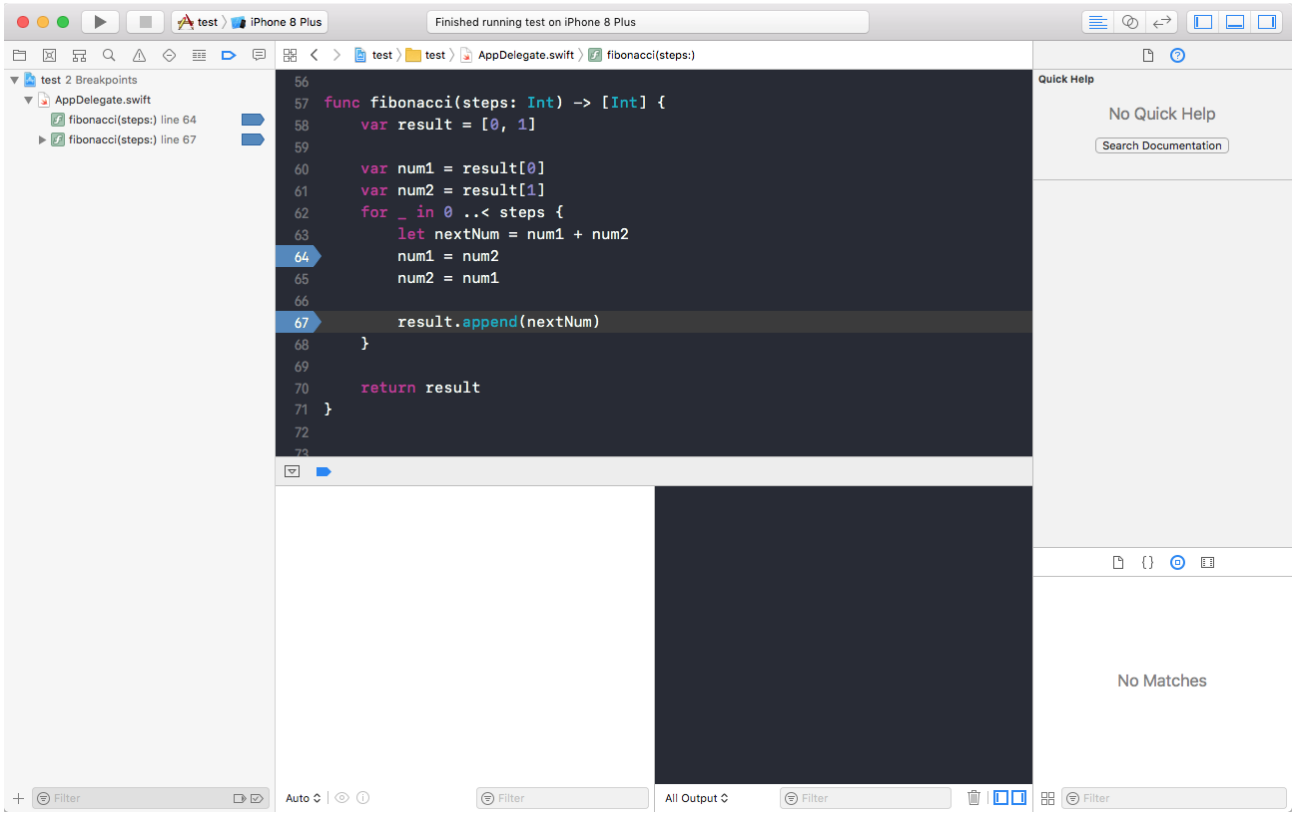
Инструмент отладки от Xcode Breakpoint

Breakpoint — это инструмент отладки, который позволяет приостановить выполнение программы до определенного момента. С его помощью сможем исследовать код программы — узнавать значения переменных в этом моменте, производить расчеты. Все это позволяет узнать, где возникают ошибки.

Чтобы установить точку остановки в Xcode, определите, где хотите приостановить выполнение кода (в какой строчке) и нажмите на левый столбец напротив номера строки, чтобы создать синий индикатор — точку остановки.

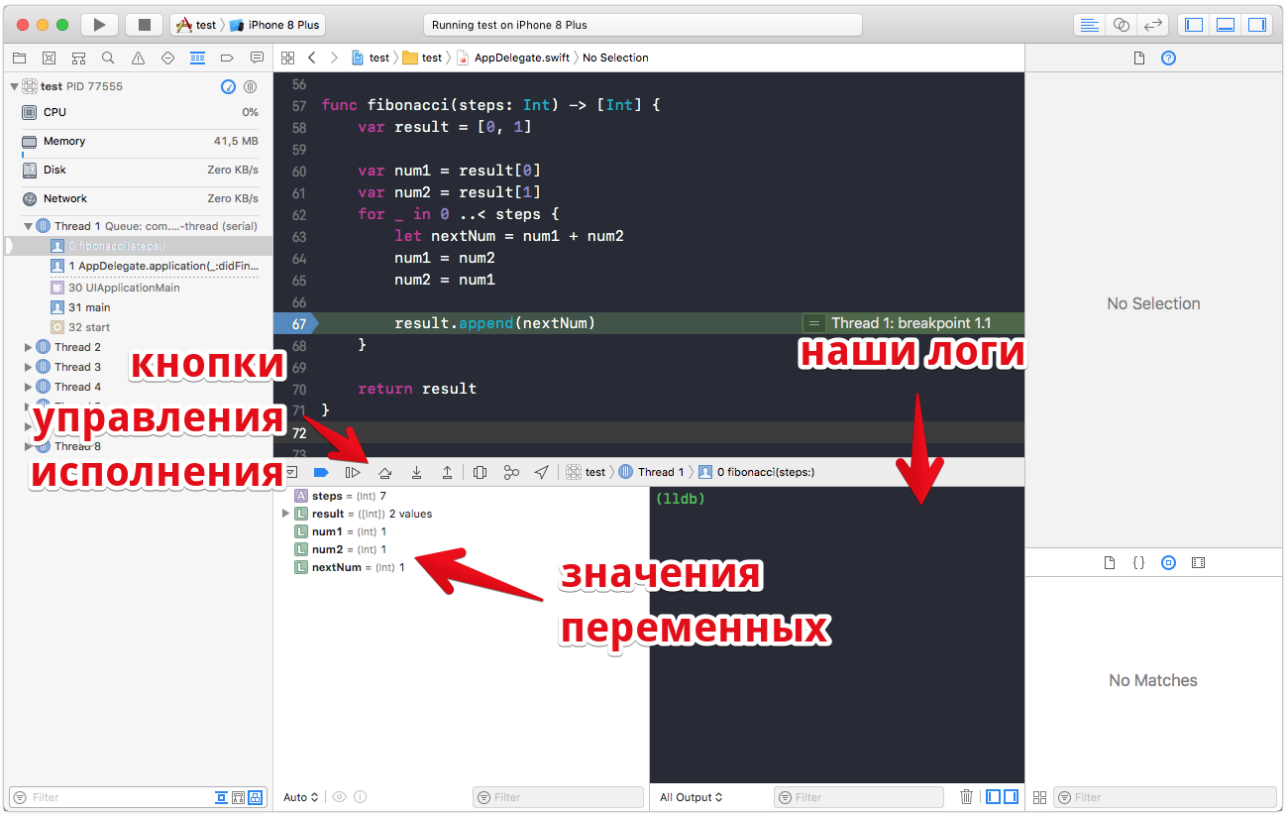


Весь список точек остановок можно посмотреть через **Breakpoint Navigator**.



Для управления точкой остановки выберите соответствующий пункт выпадающего меню, которое появляется после нажатия правой кнопки мыши на синий индикатор точки остановки.

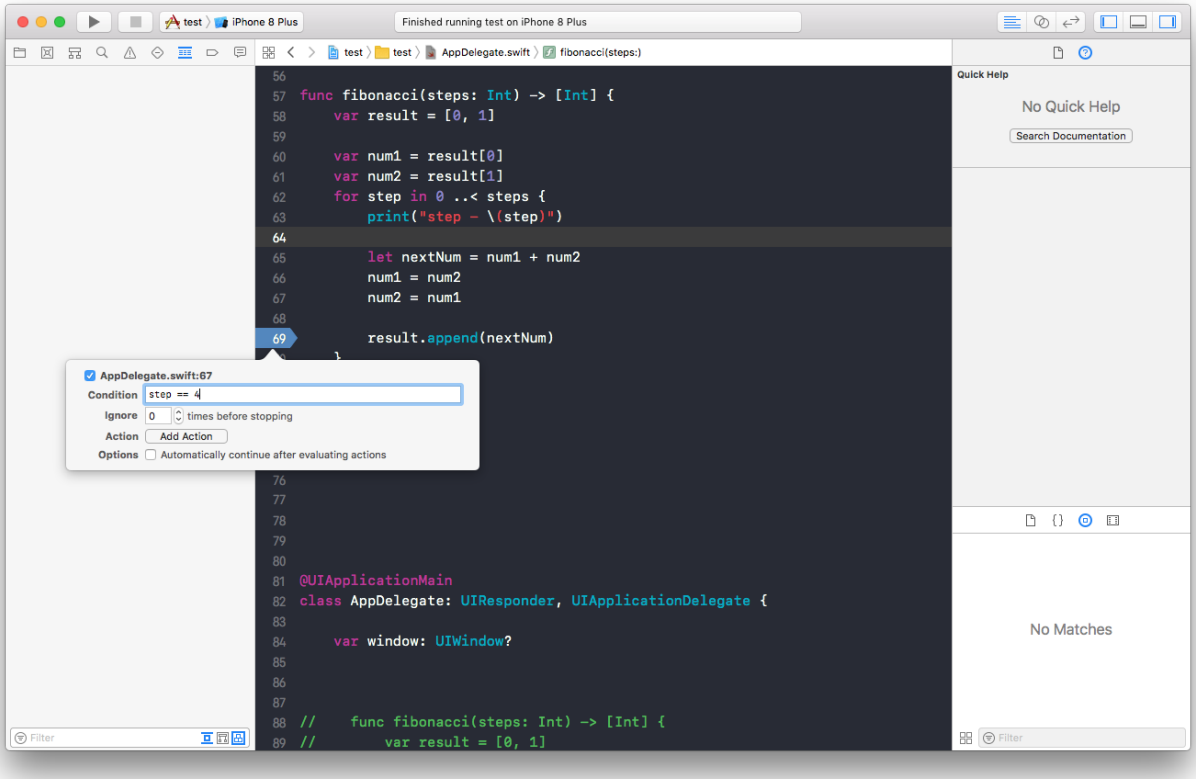
Посмотрим, какую информацию позволит проанализировать **Breakpoint**:

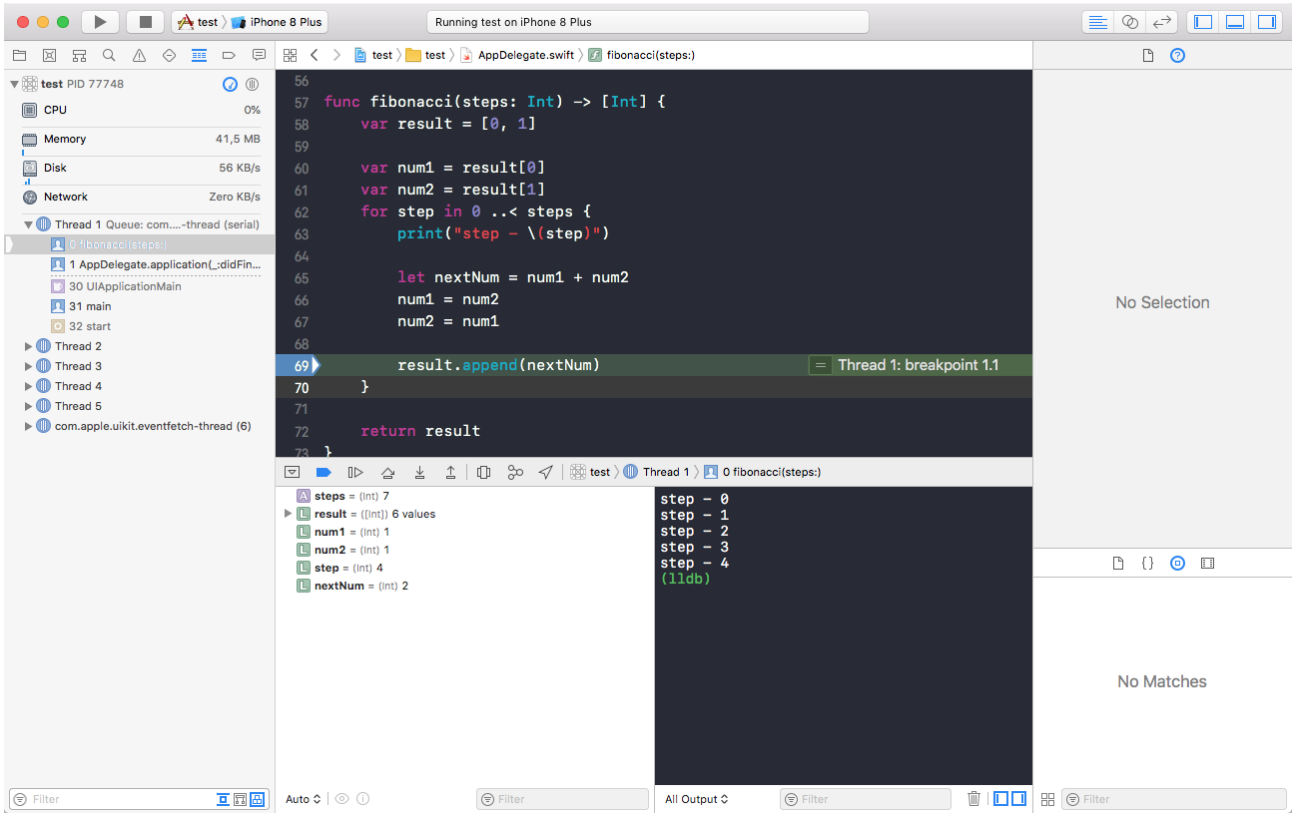


Доступны кнопки управления исполнением (**Step Over**, **Step Into...**), манипулируя которыми, осуществляем навигацию по коду. Можно анализировать значения переменных и работать с логами в **console output**.

При **Control-click** на индикаторе точки остановки отображается меню команд. Здесь можно установить дополнительные условия срабатывания точки остановки, добавить действия.

Поставим для точки остановки условие срабатывания. После запуска приложения программа остановит выполнение в данной точке, только когда переменная **step** примет значение 4.





Рефакторинг

Рефакторинг — крайне важная часть разработки. Писать код, «который просто работает», — не залог успешного приложения. Рефакторинг дает возможность привести код в надлежащий вид, что

позволит в дальнейшем этот код легко читать, использовать повторно, поддерживать и расширять приложение.

Рефакторинг — это улучшение кода без изменения его поведения.

Причины рефакторинга

Повторение кода. Хороший код не содержит повторений. Если приходится изменять участок кода с повторением, нужно не забыть внести эти же изменения и в других местах, где код аналогичен. Повторение может выражаться не только в области кода, в функции, но и в классах. Если классы очень похожи по функциональности — объединяйте, выносите базовый класс.

«Правило трех»:

- В первый раз просто пишем код;
- Пишем аналогичный код второй раз — делаем заметку, но все-таки повторяем;
- Пишем похожий код третий раз — начинаем рефакторинг.

Слишком длинный цикл и метод указывают на необходимость рефакторинга. Разбиваем подобные участки кода на более мелкие и понятные блоки.

Обычно большие фрагменты сопровождаются комментариями, ведь даже авторам такого кода становится сложно удержать в голове все моменты его работы. Множество комментариев в области кода также сигнализируют о необходимости рефакторинга.

Различают несколько уровней рефакторинга: уровень данных, уровень методов и уровень классов.

Рефакторинг на уровне данных

- **Замена магических литералов на константы**;
- **Ввод промежуточной переменной**. Особенно это актуально в математических расчетах, при формировании **CGRect** для графических компонентов;
- **Замена имени переменной на более информативное**;
- **Замена выражения на вызов метода**. Например, метод сильно разросся и начал включать несколько отдельных логик. Мы уже выделили эти блоки кода пустыми строками. Пора вынести выражение в отдельный метод;
- **Замена вызова метода на выражение**. Когда вынесенный метод сузился до нескольких строк (или даже до одной), выделение его в отдельный метод теряет всякий смысл и лишь загромождает код лишними строками отделения, описания, объявления. Усложняется навигация и понимание кода.

Применим данные правила к примеру с функцией **fibonacci()**:

```
func fibonacci(steps: Int) -> [Int] {
    let defaultFirstNumber = 0
    let defaultSecondNumber = 1

    var result = [defaultFirstNumber, defaultSecondNumber]

    while result.count < steps + 2 {
```

```
        result.append(result[result.count - 1] + result[result.count - 2])
    }

    return result
}
```

Рефакторинг на уровне методов

- **Вынос сложных выражений в логическую переменную**;
- **Вынос части кода из метода в отдельный метод**;
- **Отделение операции запроса данных от их изменения**;
- **Передача в метод объекта вместо нескольких полей**. Если количество входных параметров в методе возросло до 10, поддерживать и использовать его становится неудобно. Поэтому выносим эти 10 параметров в отдельную структуру или класс. А объект этой структуры/класса передаем как входной параметр в метод.
- **Передача в метод несколько полей вместо объекта класса**. Если методу нужно от класса из 10 свойств всего 2, передавать весь объект класса не нужно. В этом случае преобразуем метод — указываем два входных параметра и передаем только значения этих двух свойств объекта класса.

Применим данные правила к примеру с функцией **fibonacci()**:

```
func fibonacciRecursive(number1: Int, number2: Int, steps: Int, result: inout
[Int]) {
    if steps > 0 {
        let nextNumber = number1 + number2
        result.append(nextNumber)
        fibonacciRecursive(number1: number2, number2: nextNumber, steps:
steps-1, result: &result)
    }
}

func fibonacci(steps: Int) -> [Int] {
    let defaultFirstNumber = 0
    let defaultSecondNumber = 1
    var result = [defaultFirstNumber, defaultSecondNumber]

    fibonacciRecursive(number1: defaultFirstNumber, number2:
defaultSecondNumber, steps: steps, result: &result)

    return result
}

print(fibonacci(steps: 7))

// [0, 1, 1, 2, 3, 5, 8, 13, 21]
```

Рефакторинг классов

- **Разделение одного класса на несколько**. Простое всегда лучше сложного, плоское — лучше вложенного. При разделении класса на несколько мы улучшаем читабельность, делаем код понятнее;

- **Замена наследования на делегирование**. Выполняем данный рефакторинг, если наследование возникло только ради объединения общего кода, а не потому что подкласс будет расширением родительского класса. Такой рефакторинг уместен, если подкласс использует только часть методов родительского класса. Суть рефакторинга сводится к тому, чтобы разделить оба класса и сделать родительский класс помощником подкласса, а не его родителем. Вместо того чтобы наследовать все методы родительского класса, у подкласса будут только необходимые методы, которые будут делегировать выполнение методам объекта родительского класса;

- **Замена вызова конструктора на фабричный метод**. Есть код, в котором раньше создавался объект, куда передавалось значение какого-то типа. После применения рефакторинга появилось уже несколько подклассов, из которых нужно создавать объекты в зависимости от значения определенного типа. Изменить оригинальный конструктор так, чтобы он возвращал объекты подклассов, невозможно, поэтому мы создаем статический фабричный метод. Он будет возвращать объекты нужных классов, после чего заменит собой все вызовы оригинального конструктора.

- **Настройка области видимости.** Область видимости определяет, какие свойства и методы доступны снаружи класса. Если управление свойствами должно быть определено только внутри класса — произведите рефакторинг данного вида. Доступ к этим свойствам настройте, например, через методы **get***.

Практическое задание

1. Провести рефакторинг кода.

2. Реализовать экраны регистрации, авторизации и информации о пользователе (форма с возможностью редактировать данные).

3. Использовать наш сервисный слой. Сервисы и методы написаны ранее.

P.S. Не забудьте, что у вас есть тесты, которые помогут понять, что код не сломался после рефакторинга. Возможно, найдутся тесты, которые тоже придется переписать.

Дополнительные материалы

1. Метод утенка.

2. Числа Фибоначчи.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Стив Макконнелл. Совершенный код.

2. Debugging Tools.