Индивидуальный Проект
Объектно-Ориентированный Анализ и Проектирование
Виджая Стивен 932001

Описание проекта

Паттерн: Хранитель (Memento)
Язык программирования: Swift
Фреймворк: SwiftUI
Название проект: QuoteGenerator – приложение для генерирования случайных цитат и картинок для создания обои на телефон.

Определение

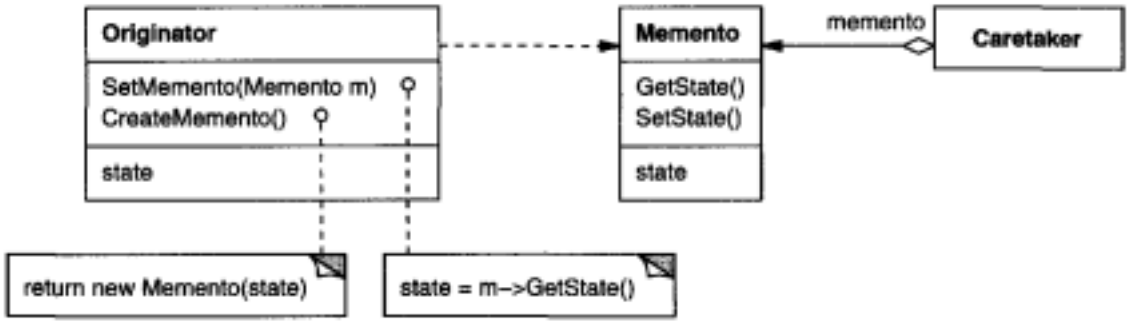
Memento – поведенческий шаблон проектирования, что означает, что они являются шаблонами проектирования, которые определяют, как объекты взаимодействуют друг с другом. Этот шаблон раскрывает частное внутреннее состояние объекта. Наиболее распространенной реализацией этого шаблона является функция «отмены» во многих приложениях, таких как Ms. Office, Google Docs, Music Player и т. д. Этот вид функциональности достижим путем восстановления объекта до его предыдущего состояния или путем управления версиями или через пользовательскую сериализацию.

Компоненты паттерна

Хранитель (memento) реализован с тремя основными объектами: *memento*, *originator*, и *caretaker*.

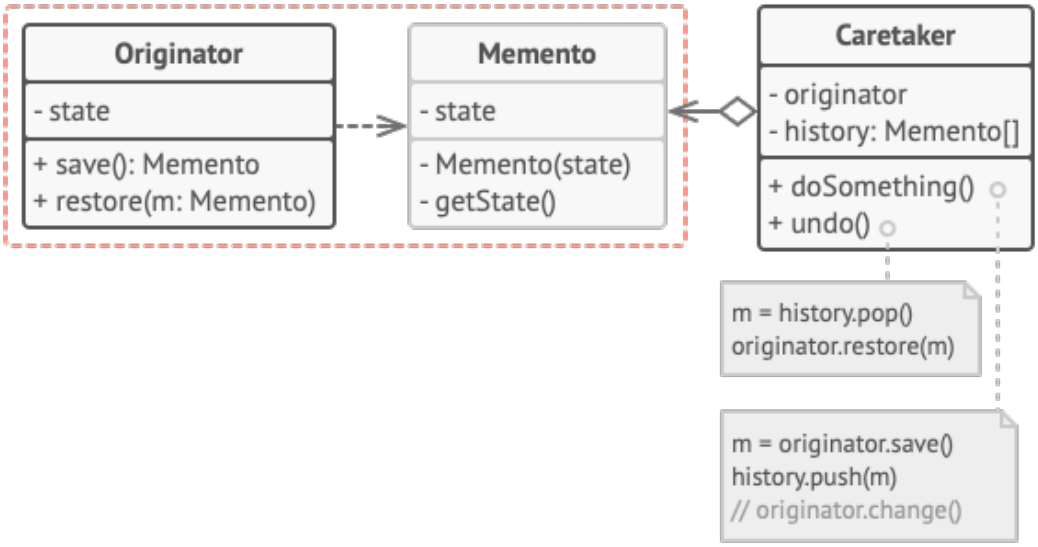
- Memento – Интерфейс/объект, которые сохраняют состояние объекта. Этот объект будет использоваться Создателем для создания воспоминаний (Memento объектов), которые сохраняются смотрителем.
- Originator – Объект, имеющий внутреннее состояние. Этот объект создаст объект Memento и восстановит свое внутреннее состояние с помощью Memento.
- Caretaker – Объект, который что-то делает с инициатором, но хочет иметь возможность отменить изменение. Обычно объект-смотритель сначала спрашивает создателя о памятном объекте, а затем выполняет ту операцию, которую собирался выполнить. Для отката/отката в предыдущее состояние (до операций) он возвращает памятный объект (состояние) создателю.

Примеры Диаграммы Классов



Существует несколько способов реализации паттерн хранитель (memento).

1. Реализация на основе вложенных классов.

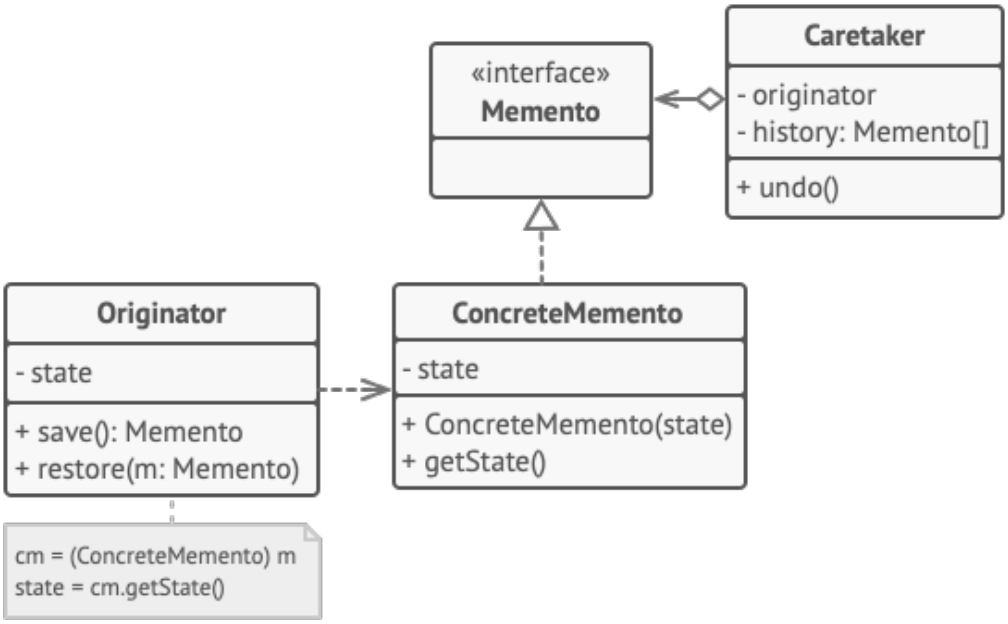


Класс Originator может создавать снимки своего состояния с помощью метода save(), а также восстанавливать свое состояние из снимков с помощью метода restore(m: Memento).

Класс Memento действует как снимок состояния объекта/создателя. На этой диаграмме все методы должны быть общедоступными, а не частными.

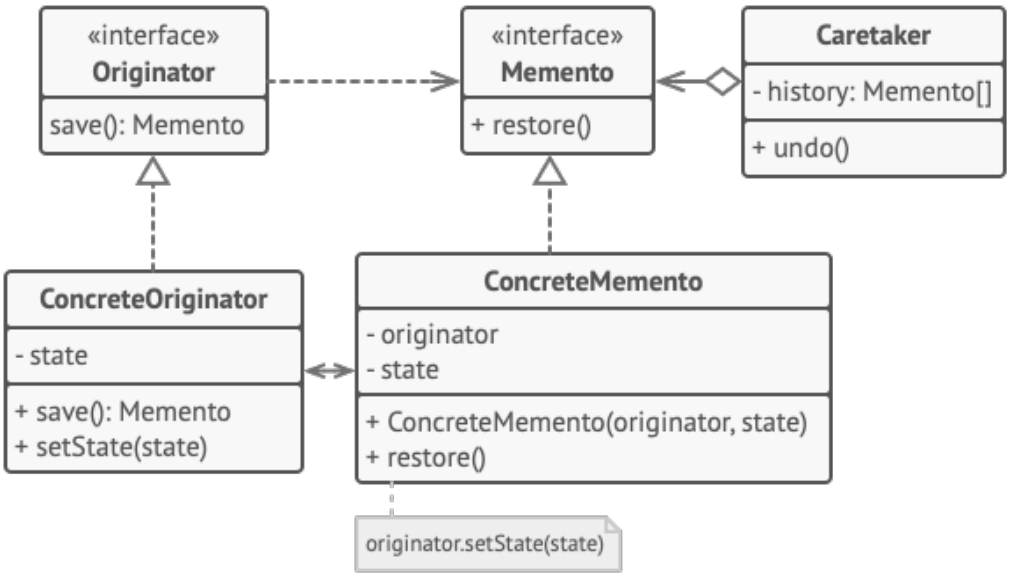
Caretaker — это класс, который отслеживает историю создателя, храня стопку Mementos. Он позаботится о том, когда захватывать и когда восстанавливать состояние отправителя.

2. Реализация на основе промежуточного интерфейса



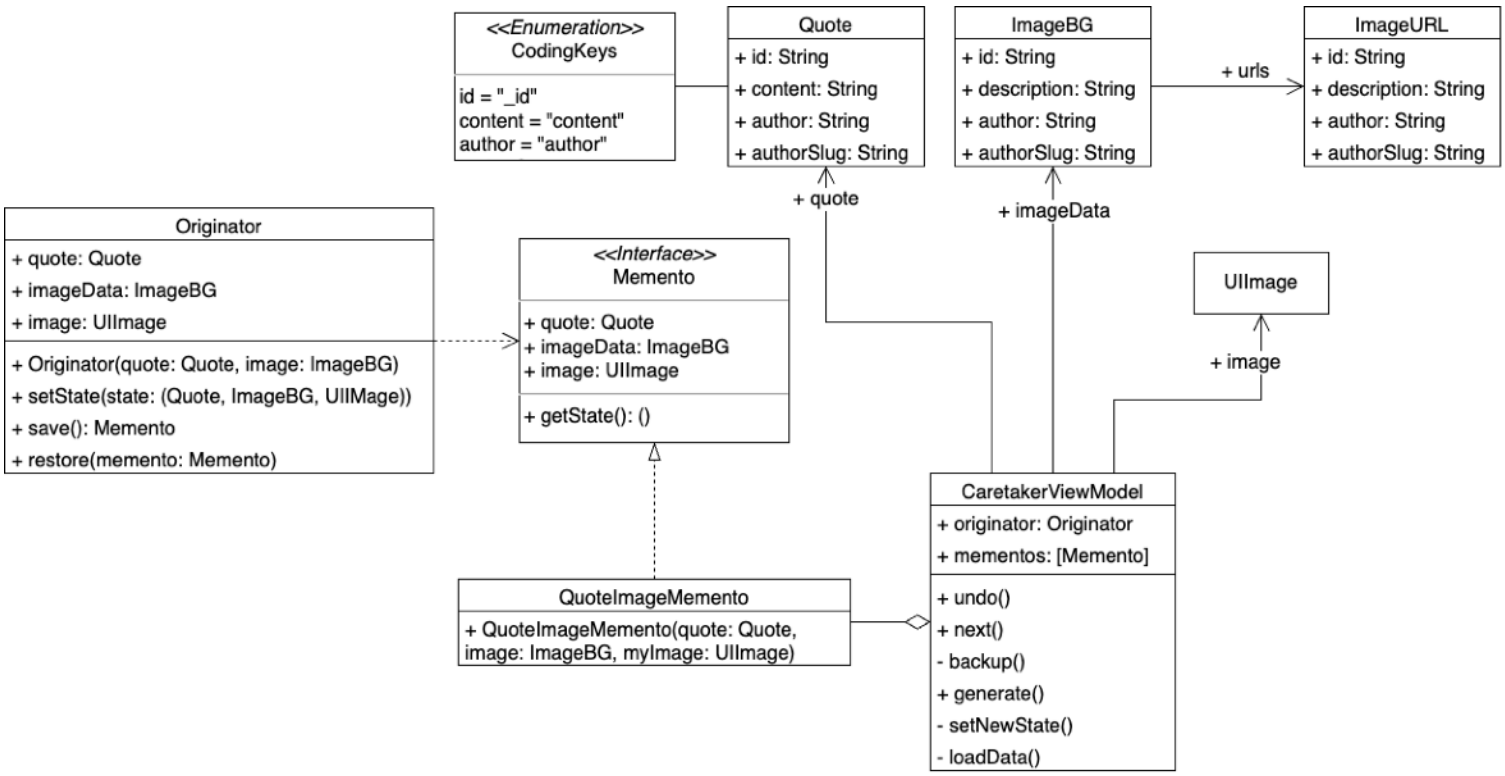
Здесь единственное отличие состоит в том, что мы используем класс интерфейса, поэтому мы можем создать несколько разных типов объектов, реализующих интерфейс Memento. На этой диаграмме (взято из Интернета) есть ошибка, которая представляет собой стрелку зависимости от Originator к ConcreteMemento, предполагается, что это стрелка зависимости от класса Originator к Memento (классу интерфейса), а не к его конкретному объекту memento, который реализовал интерфейс.

3. Реализация со строгой инкапсуляцией



Здесь Originator также будет реализован с использованием интерфейса. В этом случае у нас может быть несколько типов Originator и несколько типов Memento.

Диаграмма классов нашего приложения



Описание кода программы

В этом проекте мы используем реализацию на основе промежуточного интерфейса. Итак, есть 4 основных класса:

1. Memento (Интерфейс)
2. QuoteImageMemento (ConcreteMemento)
3. Originator
4. CaretakerViewModel

Здесь мы также используем архитектуру MVVM, и поскольку наш класс Caretaker отвечает за изменения состояния класса Originator, мы также будем использовать этот класс для нашего класса ViewModel. В этом проекте также есть еще один класс, который структурирует модель данных нашего приложения, и из этой модели наше представление может показывать пользователю полученные данные. Рассмотрим подробнее каждый компонент.

Models

ImageBG

```
struct ImageBG: Identifiable, Codable {
    let id: String
    let description: String?
    let alt_description: String
    let urls: ImageURL
}
```

Этот класс модели используется для сохранения данных изображения, взятых из внешнего API (https://unsplash.com/developers) (https://api.unsplash.com/photos/random/?client_id=[API_KEY]). Нам также нужен еще один класс/структура, который структурирует URL-адрес нашего изображения.

```
struct ImageURL: Codable {
    let raw: String
    let full: String
    let regular: String
}
```

Quote

```
struct Quote: Codable {
    let id: String
    let content: String
    let author: String
    let authorSlug: String

    enum CodingKeys: String, CodingKey {
        case id = "_id"
        case content = "content"
        case author = "author"
        case authorSlug = "authorSlug"
    }
}
```

Этот класс модели используется для сохранения данных наших котировок, которые также берутся из внешнего API (https://github.com/lukePeavey/quotable) (https://api.quotable.io/random). В этом классе у нас также есть перечисление, которое используется для сопоставления имени атрибута структуры с именами переменных, возвращаемыми API.

Memento

```
protocol Memento {
    var quote: Quote { get }
    var imageData: ImageBG { get }
    var image: UIImage { get }

    func getState() -> (Quote, ImageBG, UIImage)
}
```

Поскольку в Swift у нас на самом деле нет класса интерфейса, потому что Swift — это не полностью ООП-язык, а язык протокола. Итак, мы используем ключевое слово «protocol» вместо интерфейса. В этом классе интерфейса Memento мы сохранили три атрибута и один метод. Эти три атрибута сохранили состояние наших моделей данных.

QuoteImageMemento

```
class QuoteImageMemento: Memento {
    private (set) var quote: Quote
    private (set) var imageData: ImageBG
    private (set) var image: UIImage

    init(quote: Quote, image: ImageBG, myImage: UIImage) {
        self.quote = quote
        self.imageData = image
        self.image = myImage
    }

    func getState() -> (Quote, ImageBG, UIImage) {
        (quote, imageData, image)
    }
}
```

QuoteImageMemento — это класс, реализующий интерфейс Memento. В нем реализованы все методы, которые объявлены в интерфейсном классе. Здесь мы заставляем все наши атрибуты иметь приватный setter, но общедоступный геттер.

Почему мы не можем сделать все getter и setter приватными? Потому что в Swift всякий раз, когда мы используем протокол, нам нужно назначить общедоступным как минимум геттер или оба getter и setter. Таким образом, с помощью приватный setter мы можем гарантировать, что данные/состояния снимка (memento) не может быть изменена, а может быть доступна только для доступа к данным (get) или инициализации.

Здесь мы также создаем метод `getState()`, который возвращает кортеж всех типов данных атрибута, чтобы мы могли получить текущее состояние объекта Memento. Так что мы будем использовать не публичный геттер каждого атрибута, а именно из этого метода.

Originator

```
class Originator  {
    var quote: Quote
    var imageData: ImageBG
    var image: UIImage = UIImage()

    init(quote: Quote, image: ImageBG) {
        self.quote = quote
        self.imageData = image
        NetworkManager.loadData(url: URL(string: image.urls.full)!) { dImage in
            if let dImage {
                self.image = dImage
            }
        }
    }

    func setState(state: (Quote, ImageBG, UIImage)) {
        quote = state.0
        imageData = state.1
        image = state.2
    }

    func save() -> Memento {
        return QuoteImageMemento(quote: quote, image: imageData, myImage: image)
    }

    func restore(memento: Memento) {
        guard let memento = memento as? QuoteImageMemento else { return }
        self.quote = memento.quote
        self.imageData = memento.imageData
        self.image = memento.image
    }
}
```

Это класс Originator, в котором наше приложение будет устанавливать, восстанавливать и сохранять наше состояние. В инициализаторе мы видим, что он автоматически загружает данные «изображения» с изображением, загруженным с URL-адреса, сохраненного в атрибуте «imageData».

В этом классе мы также создаем метод `save()`, который возвращает Memento текущего состояния. Метод `restore(memento: Memento)` используется для восстановления нашего состояния из нашего memento. Метод `setState(state: (Quote, ImageBG, UIImage)` используется для установки нашего состояния Originator с состоянием из API (используется внутри одного из методов в классе CaretakerViewModel).

Caretaker (CaretakerViewModel)

```
final class CaretakerViewModel: ObservableObject {
    @Published var quote: Quote = DefaultQuote.quote
    @Published var imageData: ImageBG = DefaultImage.image
    @Published var image: UIImage = UIImage()

    private var mementos: [Memento] = [Memento]()
    private var originator: Originator

    init(originator: Originator) {
        self.originator = originator
        self.backup()
        NetworkManager.loadData(url: URL(string: originator.imageData.urls.full)!) { image in
            if let image {
                DispatchQueue.main.async {
                    self.image = image
                }
            }
        }
    }

    func undo() {
        guard !mementos.isEmpty else { return }
        let removedMemento: Memento = mementos.removeLast()

        originator.restore(memento: removedMemento)
        setNewState()
    }

    func generate() {
        backup()
        loadData()
    }

    private func backup() {
        mementos.append(originator.save())
        print(mementos)
    }

    private func setNewState() {
        let currentState = self.originator.save().getState()
        self.quote = currentState.0
        self.imageData = currentState.1
        self.image = currentState.2
    }
}
```

Вот как наш класс Caretaker должен быть реализован с использованием MVVM. В Swift ViewModel должен соответствовать протоколу ObservableObject, чтобы наше представление могло отслеживать изменения во всех опубликованных переменных. В этом случае он будет отслеживать изменения значений наших переменных `quote`, `imageData` и `image`. Основными компонентами нашего Смотрителя являются создатель и массив/стопка Mementos. Здесь оригинатор используется для обработки или сохранения нашего текущего состояния. В то время как массив mementos

используется для отслеживания всех наших предыдущих mementos/ моментальных снимков.

Здесь мы также видим, что есть метод `undo()`, который используется для возврата к предыдущему состоянию. Мы сначала проверяем, пуст ли наш массив mementos, если он пуст то ничего делать не будем, тогда возьмем и удалим последний элемент из нашего массива и сохраним его в переменную типа memento. Затем мы вызовем метод восстановления создателя, чтобы восстановить наше последнее состояние. Затем нам также нужно будет установить новое состояние наших опубликованных переменных с помощью метода `setNewState()`.

Чтобы сгенерировать новые данные и сохранить их состояние, мы будем использовать наш метод `generate()`, который вызовет метод `backup()` для сохранения снимка/воспоминания о нашем текущем состоянии в массиве воспоминаний, вызывая метод сохранения создателя, который вернуть памятный объект, который сохранил текущее состояние. После того, как мы обновим наш стек Memento, мы загружаем новые данные в наши переменные, используя эту функцию `loadData()`.

```
private func loadData() {
    var newImageData: ImageBG = DefaultImage.image
    var newQuote: Quote = DefaultQuote.quote
    var newImage: UIImage = UIImage()
    let dispatchGroup = DispatchGroup()

    dispatchGroup.enter()
    getData(getImageAPIURL(), ImageBG.self) { data in
        newImageData = data
        NetworkManager.loadData(url: URL(string: data.urls.regular)!) { image in
            if let image {
                newImage = image
                dispatchGroup.leave()
            }
        }
    }

    dispatchGroup.enter()
    getData(getQuoteAPIURL(), Quote.self) { data in
        newQuote = data
        dispatchGroup.leave()
    }

    dispatchGroup.notify(queue: .main) {
        self.originator.setState(state: (newQuote, newImageData, newImage))
        self.setNewState()
    }
}
```

Здесь мы используем `DispatchGroup()` для выполнения нескольких асинхронных функций для загрузки данных из двух разных API (один для получения данных котировок, другой для получения данных изображения). После каждой успешной загрузки данных мы покидаем группу отправки и переходим к другой группе. На заключительном этапе мы затем уведомляем наш основной поток об изменениях, которые здесь мы устанавливаем нашему инициатору с новым состоянием данных, которые мы получили с помощью метода setState инициатора. После этого мы вызываем метод `setNewState()`, чтобы изменить значение наших опубликованных переменных с текущим состоянием от нашего создателя.

Reverse Undo (Обратная Отмена)

Здесь мы не только применяем функциональность отмены, но и обращаем процессы отмены. Как мы это реализуем?

1. Мы создаем новый стек/массив nextMementos (объект Memento), чтобы сохранить все состояние, которое мы отменили.

```
private var nextMementos: [Memento] = [Memento]()
```

2. Затем всякий раз, когда мы отменяем наше состояние, нам нужно добавить это текущее состояние в наш новый массив.

```
func undo() {
    nextMementos.append(originator.save())
    guard !mementos.isEmpty else { return }
    let removedMemento: Memento = mementos.removeLast()

    originator.restore(memento: removedMemento)
    setNewState()
}
```

3. Затем мы создаем новую функцию с именем `next()` для этой функции обратной отмены. То же самое произошло здесь для метода отмены, мы сначала проверяем наш массив `nextMementos`, является ли он пустым или нет, если да, то ничего не делаем. Затем мы возвращаем и удаляем последний элемент из массива в переменную памяти. Который затем мы восстановим состояние нашего создателя, используя этот nextMemento. После этого мы можем установить все наши опубликованные переменные с новым состоянием, а также сделать резервную копию или

добавить текущее состояние в наш массив mementos для отмены, чтобы мы могли отменить нашу обратную отмену.

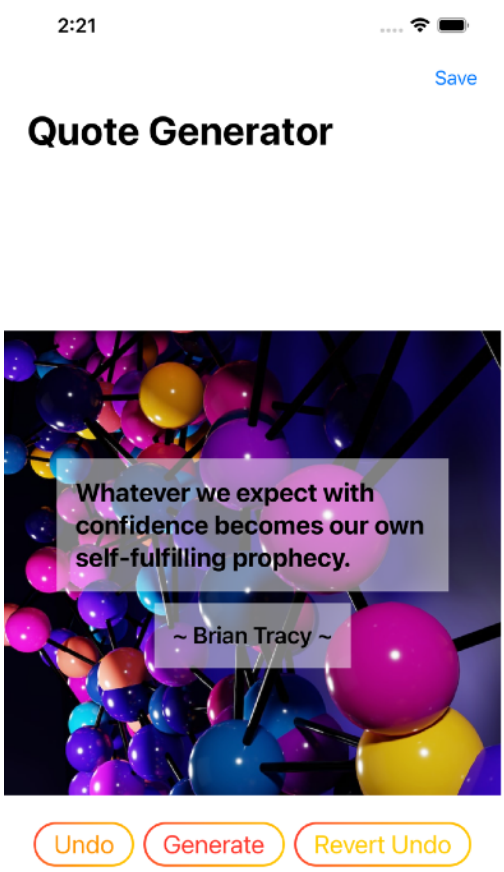
```
func next() {
    guard !nextMementos.isEmpty else { return }
    let removedMemento: Memento = nextMementos.removeLast()
    originator.restore(memento: removedMemento)
    setNewState()
    backup()
}
```

Скрины приложения

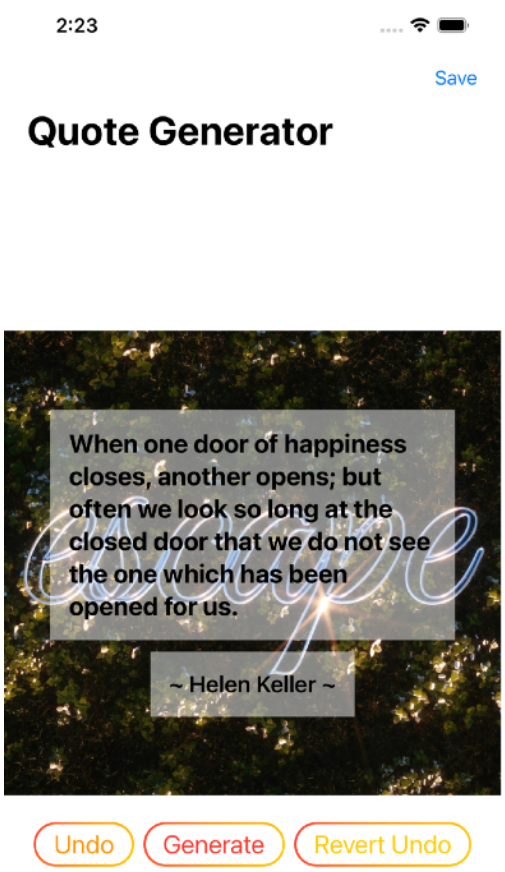
Ссылка на видео демонстрации приложения: https://drive.google.com/file/d/1aFbhYlL26vI_NUtZ2_XSHpMBtfSHjMC_/view?usp=share_link
Ссылка на GitHub: https://github.com/Steven2110/MementoQuotes

Главное меню с изображением по умолчанию, цитатой (ноль) и автором (ноль)

При нажатии кнопки «Generate»



Второе нажатие кнопки «Generate»



При нажатии кнопки «Undo»

При нажатии кнопки «Revert Undo»

При нажатии кнопки «Save», приложение сохраняет кадр изображения и цитату в в фототеку телефона (приложение «Фото».