

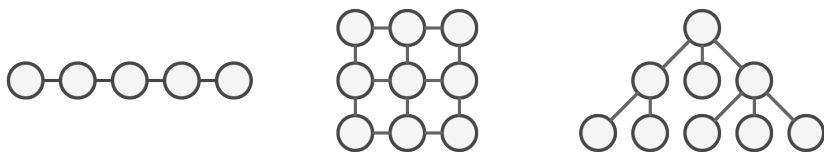
# ИТЕРАТОР

*Также известен как: Iterator*

**Итератор** — это поведенческий паттерн проектирования, который даёт возможность последовательно обходить элементы составных объектов, не раскрывая их внутреннего представления.

## ☹ Проблема

Коллекции — самая частая структура данных, которую вы можете встретить в программировании. Это набор объектов, собранный в одну кучу по каким-то причинам.

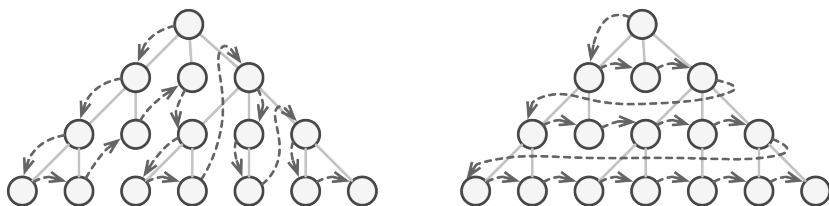


*Разные типы коллекций.*

Большинство коллекций выглядят как обычный список элементов. Но есть и экзотические коллекции, построенные на основе деревьев, графов и других сложных структур данных.

Но как бы ни была структурирована коллекция, пользователь должен иметь возможность последовательно обходить её элементы, чтобы проделывать с ними какие-то действия.

Но каким способом следует перемещаться по сложной структуре данных? Например, сегодня может быть достаточным обход дерева в глубину. Но завтра потребуется возможность перемещаться по дереву в ширину. А на следующей неделе, и того хуже, понадобится обход коллекции в случайном порядке.



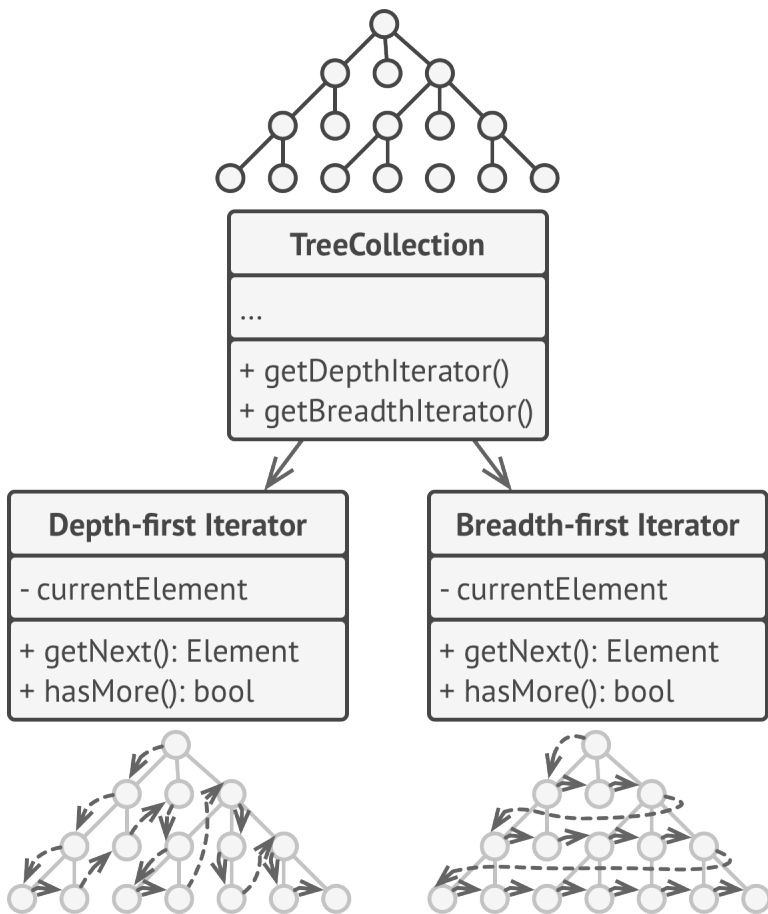
*Одну и ту же коллекцию можно обходить разными способами.*

Добавляя всё новые алгоритмы в код коллекции, вы понемногу размываете её основную, которой является эффективное хранение данных. Некоторые алгоритмы могут быть и вовсе слишком заточены под определённое приложение и смотреться дико в общем классе коллекции.

## 😊 Решение

Идея паттерна Итератор в том, чтобы вынести поведение обхода коллекции из самой коллекции в отдельный класс.

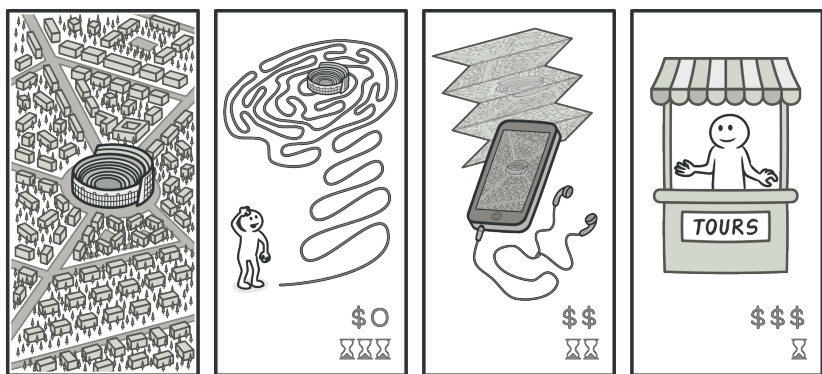
Объект итератор будет отслеживать состояние обхода, текущую позицию в коллекции и сколько элементов ещё осталось обойти. Одну и ту же коллекцию смогут одновременно обходить различные итераторы, а сама коллекция не будет даже знать об этом.



*Итераторы содержат код обхода коллекции. Одну коллекцию могут обходить сразу несколько итераторов.*

К тому же, если вам понадобится добавить новый способ обхода, вы создадите новый класс итератора, не изменяя существующий код коллекции.

## 🚗 Аналогия из жизни

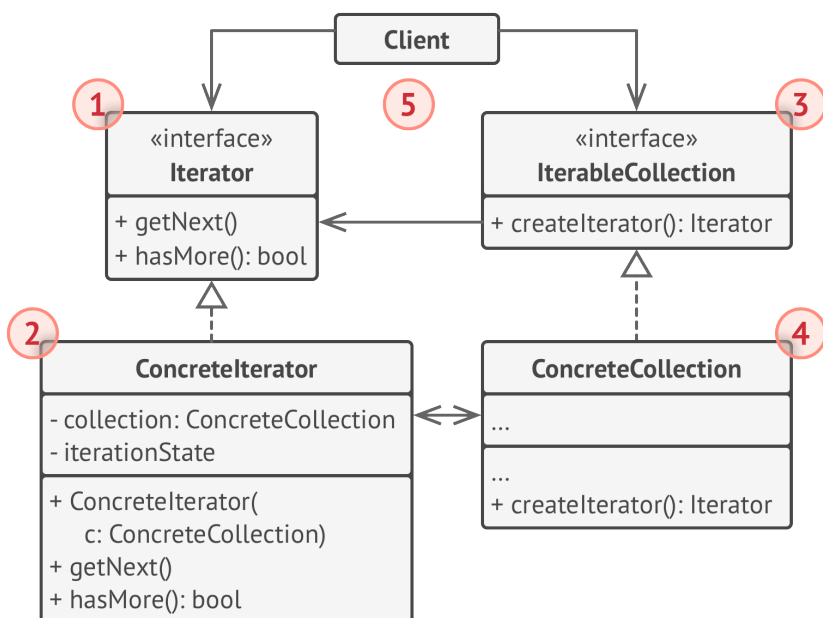


*Варианты прогулок по Риму.*

Вы планируете полететь в Рим и обойти все достопримечательности за пару дней. Но приехав, вы можете долго петлять узкими улочками, пытаясь найти Колизей. Если у вас ограниченный бюджет — не беда. Вы можете воспользоваться виртуальным гидом, скачанным на телефон, который позволит отфильтровать только интересные вам точки. А можете плюнуть и нанять локального гида, который хоть и обойдётся в копеечку, но знает город как свои пять пальцев и сможет посвятить вас во все городские легенды.

Таким образом, Рим выступает коллекцией достопримечательностей, а ваш мозг, навигатор или гид — итератором по коллекции. Вы, как клиентский код, можете выбрать один из итераторов, опираясь на решаемую задачу и доступные ресурсы.

## Структура



1. **Итератор** описывает интерфейс для доступа и обхода элементов коллекции.
2. **Конкретный итератор** реализует алгоритм обхода какой-то конкретной коллекции. Объект итератора должен сам отслеживать текущую позицию при обходе коллекции, чтобы отдельные итераторы могли обходить одну и ту же коллекцию независимо.
3. **Коллекция** описывает интерфейс получения итератора из коллекции. Как мы уже говорили, коллекции не всегда являются списком. Это может быть и база данных, и удалённое API, и даже дерево Компоновщика. Поэтому

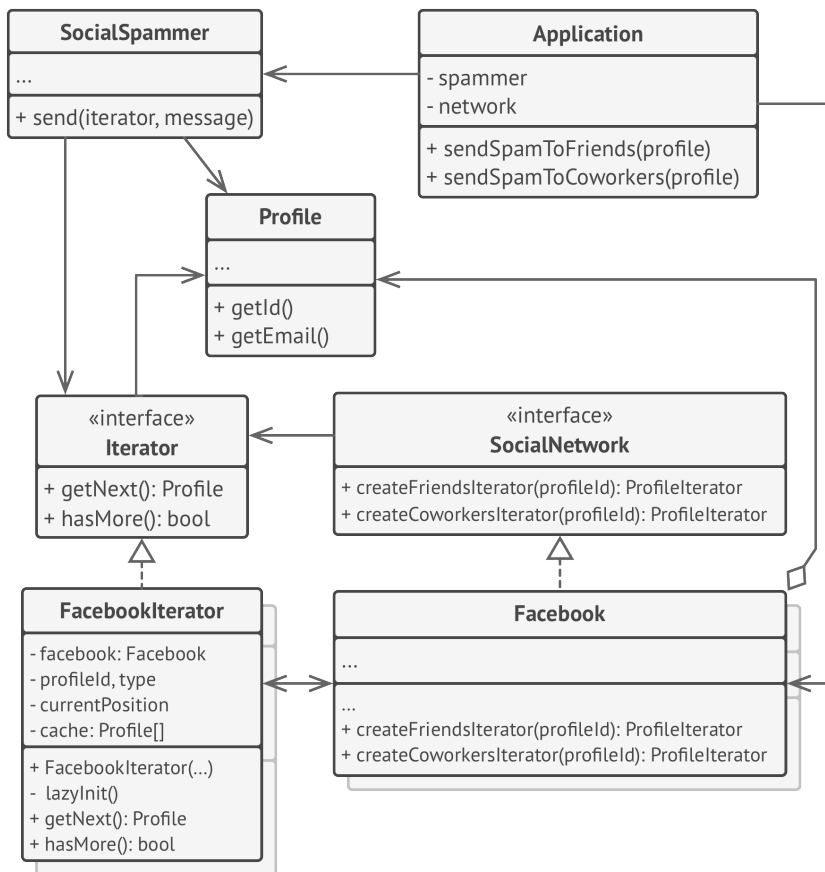
сама коллекция может создавать итераторы, так как она знает, какие именно итераторы могут с ней работать.

4. **Конкретная коллекция** возвращает новый экземпляр определённого конкретного итератора, связав его с текущим объектом коллекции. Обратите внимание, что сигнатура метода возвращает интерфейс итератора. Это позволяет клиенту не зависеть от конкретных классов итераторов.
5. **Клиент** работает со всеми объектами через интерфейсы коллекции и итератора. Так клиентский код не зависит от конкретного класса итератора, что позволяет применять различные итераторы, не изменяя существующий код программы.

В общем случае клиенты не создают объекты итераторов, а получают их из коллекций. Тем не менее, если клиенту требуется специальный итератор, он всегда может создать его самостоятельно.

## # Псевдокод

В этом примере паттерн **Итератор** используется для реализации обхода нестандартной коллекции, которая инкапсулирует доступ к социальному графу Facebook. Коллекция предоставляет несколько итераторов, которые могут по-разному обходить профили людей.



Пример обхода социальных профилей через итератор.

Так, итератор друзей перебирает всех друзей профиля, а итератор коллег — фильтрует друзей по принадлежности к компании профиля. Все итераторы реализуют общий интерфейс, который позволяет клиентам работать с профилями, не вникая в детали работы с социальной сетью (например, авторизацию, отправку REST-запросов и т.д.)

Кроме того, Итератор избавляет код от привязки к конкретным классам коллекций. Это позволяет добавить



поддержку другого вида коллекций (например, `LinkedIn`), не меняя клиентский код, который работает с итераторами и коллекциями.

```
1  // Общий интерфейс коллекций должен определить фабричный
2  // метод для производства итератора. Можно определить сразу
3  // несколько методов, чтобы дать пользователям различные
4  // варианты обхода одной и той же коллекции.
5  interface SocialNetwork is
6      method createFriendsIterator(profileId): ProfileIterator
7      method createCoworkersIterator(profileId): ProfileIterator
8
9
10 // Конкретная коллекция знает объекты каких итераторов
11 // нужно создавать.
12 class Facebook implements SocialNetwork is
13     // ... Основной код коллекции ...
14
15     // Код получения нужного итератора.
16     method createFriendsIterator(profileId) is
17         return new FacebookIterator(this, profileId, "friends")
18     method createCoworkersIterator(profileId) is
19         return new FacebookIterator(this, profileId, "coworkers")
20
21
22 // Общий интерфейс итераторов.
23 interface ProfileIterator is
24     method getNext(): Profile
25     method hasMore(): bool
26
27
28 // Конкретный итератор.
29 class FacebookIterator implements ProfileIterator is
```

```

30    // Итератору нужна ссылка на коллекцию, которую
31    // он обходит.
32    private field facebook: Facebook
33    private field profileId, type: string
34
35    // Но каждый итератор обходит коллекцию независимо от
36    // остальных, поэтому он содержит информацию о текущей
37    // позиции обхода.
38    private field currentPosition
39    private field cache: array of Profile
40
41    constructor FacebookIterator(facebook, profileId, type) is
42        this.facebook = facebook
43        this.profileId = profileId
44        this.type = type
45
46    private method lazyInit() is
47        if (cache == null)
48            cache = facebook.sendSophisticatedSocialGraphRequest(profileId,
49
50    // Итератор реализует методы базового
51    // интерфейса по-своему.
52    method getNext() is
53        if (hasMore())
54            currentPosition++
55            return cache[currentPosition]
56
57    method hasMore() is
58        lazyInit()
59        return cache.length < currentPosition
60
61
62
63




```

```

64 // Вот ещё полезная тактика: мы можем передавать объект
65 // итератора вместо коллекции в клиентские классы. При таком
66 // подходе, клиентский код не будет иметь доступа к
67 // коллекциям, а значит его не будет волновать подробности
68 // их реализаций. Ему будет доступен только общий
69 // интерфейс итераторов.
70 class SocialSpammer is
71     method send(iterator: ProfileIterator, message: string) is
72         while (iterator.hasNext())
73             profile = iterator.getNext()
74             System.sendEmail(profile.getEmail(), message)
75
76
77 // Класс приложение конфигурирует классы как захочет.
78 class Application is
79     field network: SocialNetwork
80     field spammer: SocialSpammer
81
82     method config() is
83         if working with Facebook
84             this.network = new Facebook()
85         if working with LinkedIn
86             this.network = new LinkedIn()
87         this.spammer = new SocialSpammer()
88
89     method sendSpamToFriends(profile) is
90         iterator = network.createFriendsIterator(profile.getId())
91         spammer.send(iterator, "Very important message")
92
93     method sendSpamToCoworkers(profile) is
94         iterator = network.createCoworkersIterator(profile.getId())
95         spammer.send(iterator, "Very important message")

```

## Применимость

-  **Когда у вас есть сложная структура данных, и вы хотите скрыть от клиента детали её реализации (из-за сложности или вопросов безопасности).**
-  Итератор предоставляет клиенту всего несколько простых методов перебора элементов коллекции. Это не только упрощает доступ к коллекции, но и защищает её данные от неосторожных или злоумышленных действий.
-  **Когда вам нужно иметь несколько вариантов обхода одной и той же структуры данных.**
-  Нетривиальные алгоритмы обхода структуры данных могут иметь довольно объёмный код. Этот код будет захламлять всё вокруг, если поместить его в класс коллекции или где-то посреди основной бизнес-логики программы. Применяв итератор, вы можете переместить код обхода структуры данных в собственный класс, упростив поддержку остального кода.
-  **Когда вам хочется иметь единый интерфейс обхода различных структур данных.**
-  Итератор позволяет вынести реализации различных вариантов обхода в подклассы. Это позволит легко

взаимозаменять объекты итераторов, в зависимости от того, с какой структурой данных приходится работать.

## Шаги реализации

1. Создайте интерфейс итераторов. В качестве минимума, вам понадобится операция получения следующего элемента. Но для удобства можно предусмотреть и другие методы, например, для получения предыдущего элемента, текущей позиции, проверки окончания обхода и прочих.
2. Создайте интерфейс коллекции и опишите в нём метод получения итератора. Важно, чтобы его сигнатура возвращала общий интерфейс итераторов, а не один из конкретных итераторов.
3. Создайте классы конкретных итераторов для тех коллекций, которые нужно обходить с помощью паттерна. Итератор должен быть привязан только к одному объекту коллекции. Обычно эта связь устанавливается через конструктор.
4. Реализуйте методы получения итератора в конкретных классах коллекций. Они должны создавать новый итератор того класса, который способен работать с данным типом коллекции. Коллекция должна передавать собственную ссылку в созданный итератор.

5. В клиентском коде и в классах коллекций не должно остаться кода обхода элементов. Клиент должен получать новый итератор из объекта коллекции каждый раз, когда ему нужно перебрать её элементы.



## Преимущества и недостатки

- ✓ Упрощает классы хранения данных.
- ✓ Позволяет реализовать различные способы обхода структуры данных.
- ✓ Позволяет одновременно перемещаться по структуре данных в разные стороны.
- ✗ Неоправдан, если можно обойтись простым циклом.



## Отношения с другими паттернами

- Вы можете обходить дерево Компоновщика, используя **Итератор**.
- **Фабричный метод** можно использовать вместе с **Итератором**, чтобы подклассы коллекций могли создавать подходящие им итераторы.

- **Снимок** можно использовать вместе с **Итератором**, чтобы сохранить текущее состояние обхода структуры данных и вернуться к нему в будущем, если потребуется.
- **Посетитель** можно использовать совместно с **Итератором**. *Итератор* будет отвечать за обход структуры данных, а *Посетитель* — за выполнение действий над каждым её компонентом.