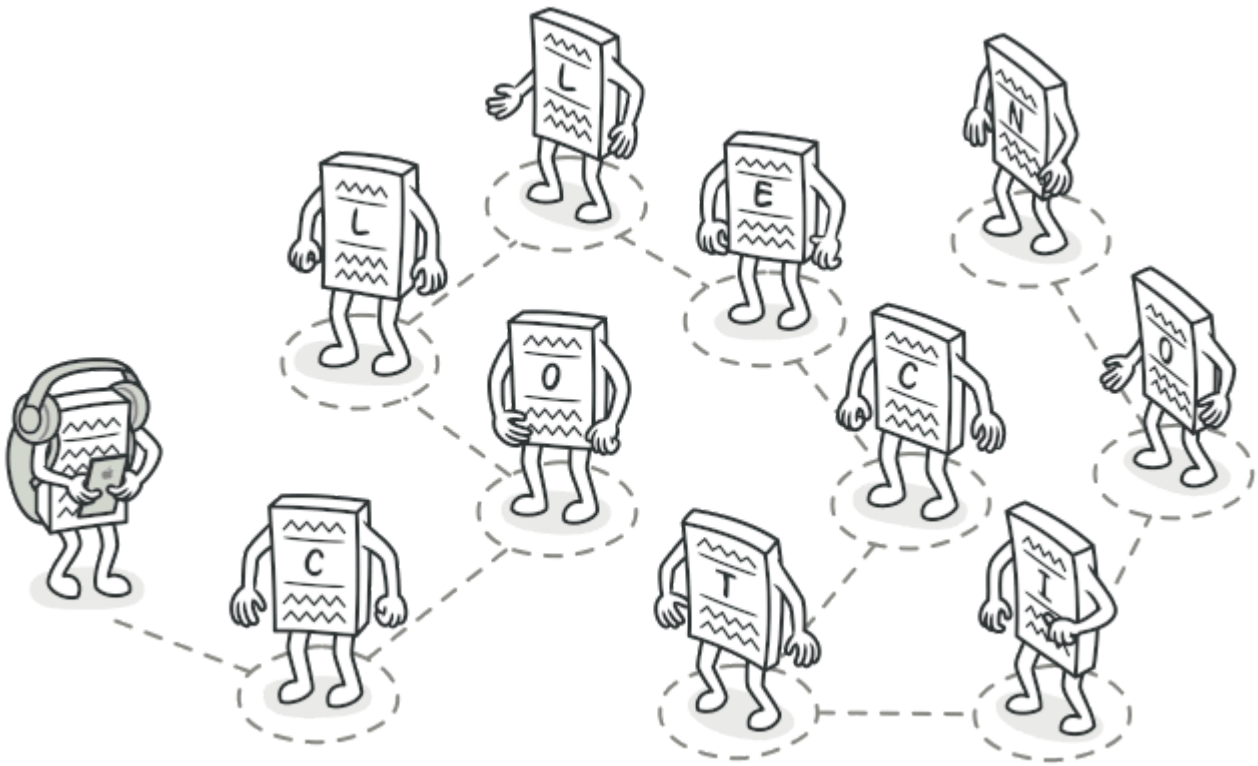
Итератор

Также известен как: Iterator

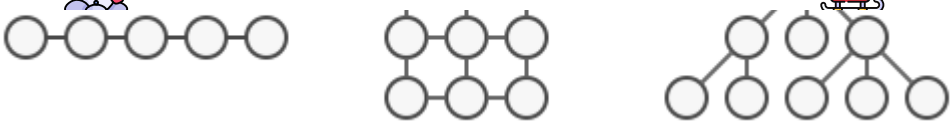
Суть паттерна

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



Проблема

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

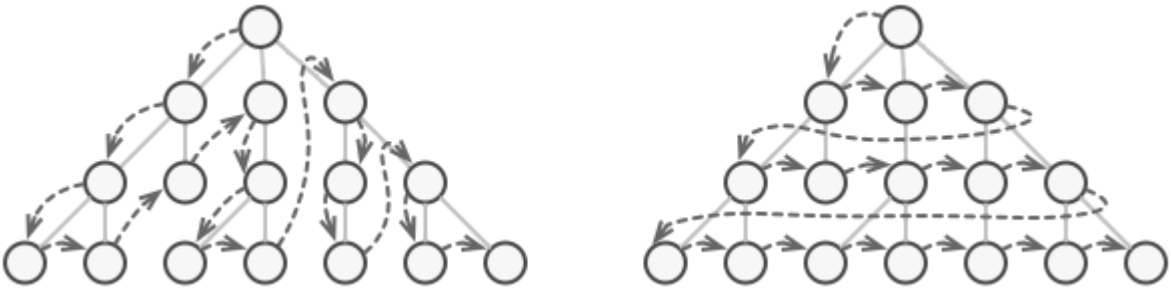


Разные типы коллекций.

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.

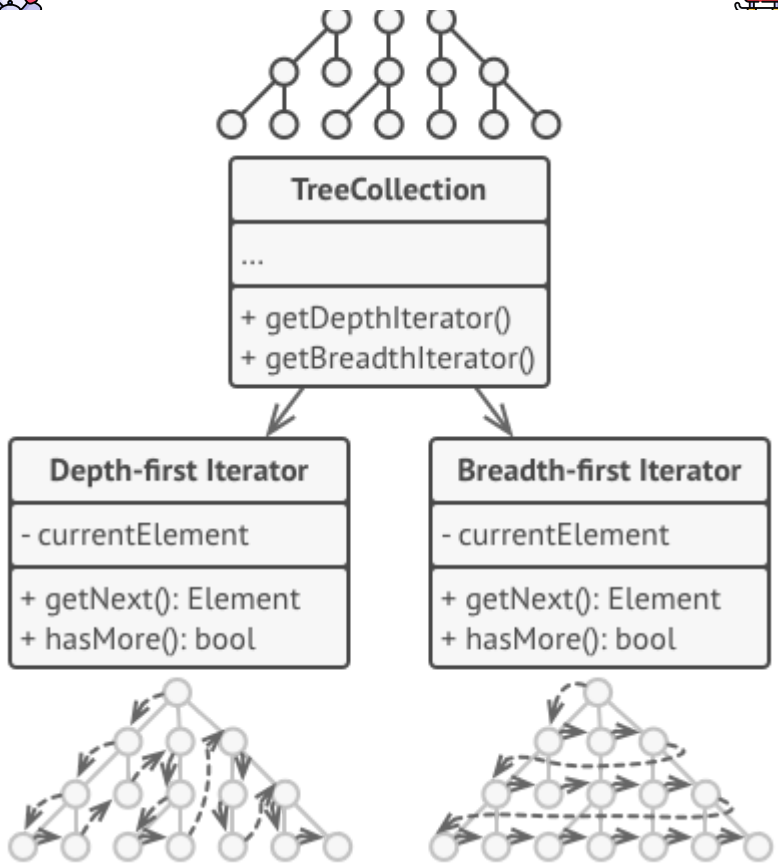


Одну и ту же коллекцию можно обходить разными способами.

Добавляя всё новые алгоритмы в код коллекции, вы понемногу размываете её основную задачу, которая заключается в эффективном хранении данных. Некоторые алгоритмы могут быть и вовсе слишком «заточены» под определённое приложение и смотреться дико в общем классе коллекции.

😊 Решение

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

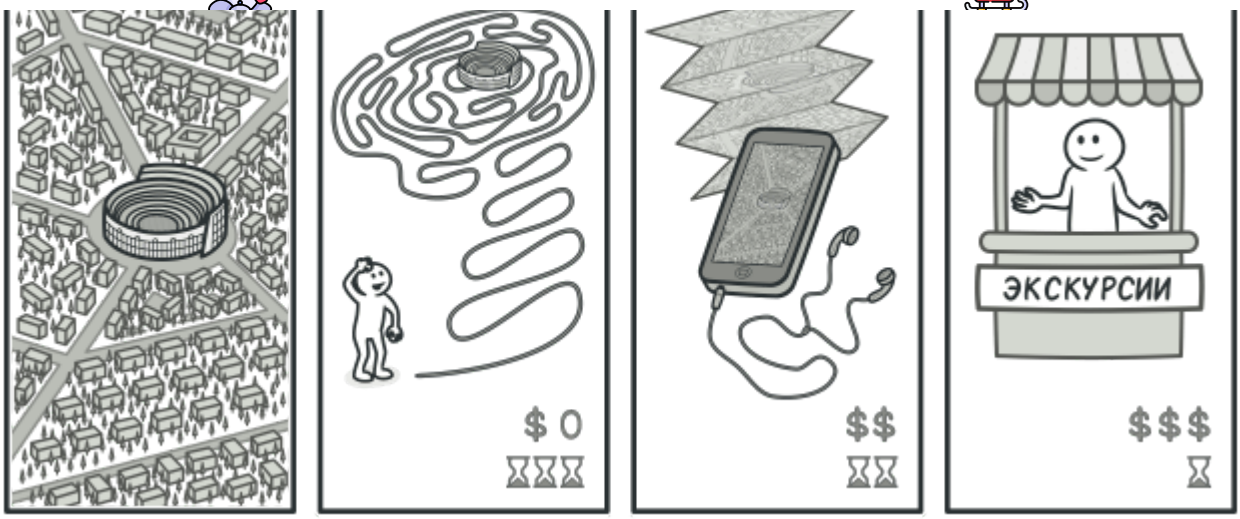


Итераторы содержат код обхода коллекции. Одну коллекцию могут обходить сразу несколько итераторов.

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.

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

Аналогия из жизни



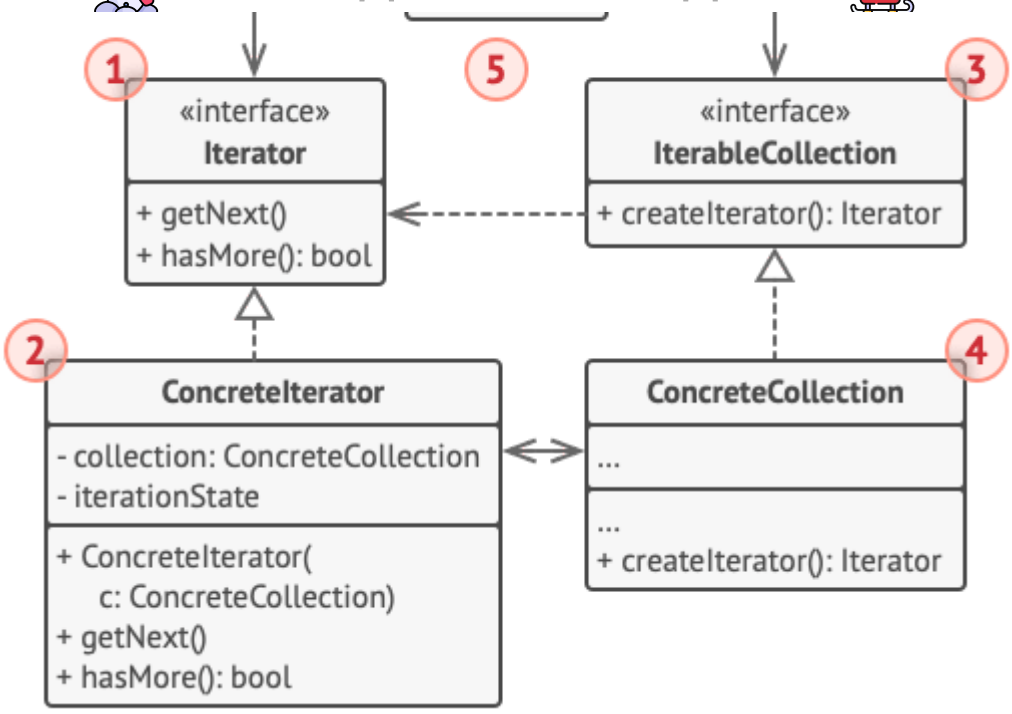
Варианты прогулок по Риму.

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, отталкиваясь от решаемой задачи и доступных ресурсов.

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.

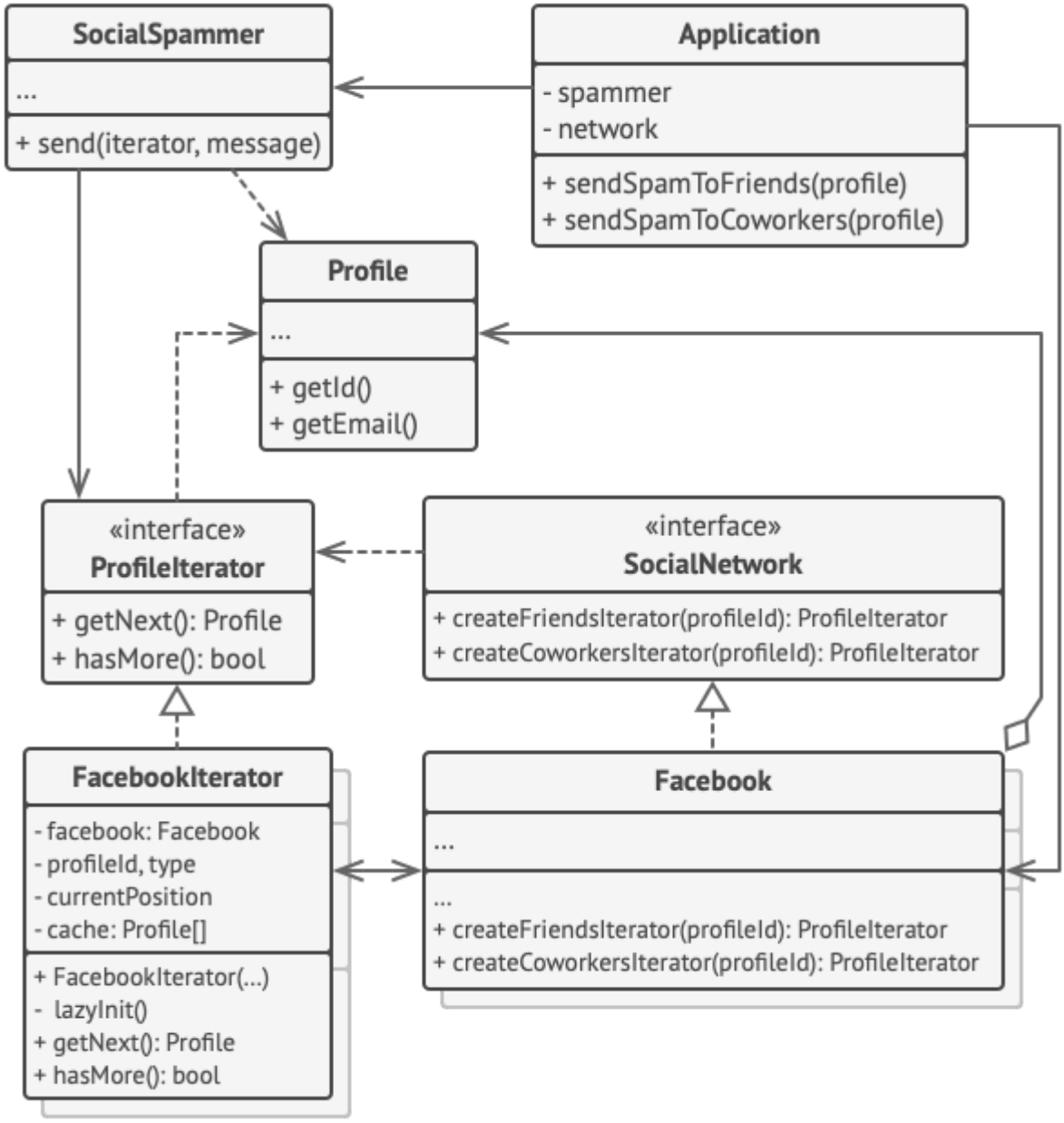
3. **Коллекция** описывает интерфейс получения итератора из коллекции. Как мы уже говорили, коллекции не всегда являются списком. Это может быть и база данных, и удалённое API, и даже дерево **Компоновщика**. Поэтому сама коллекция может создавать итераторы, так как она знает, какие именно итераторы способны 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 конкретных классов, 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.

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, в авторизацию, отправку REST-запросов и т. д.)

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

```
// Общий интерфейс коллекций должен определить фабричный метод
// для производства итератора. Можно определить сразу несколько
// методов, чтобы дать пользователям различные варианты обхода
// одной и той же коллекции.
interface SocialNetwork is
    method createFriendsIterator(profileId):ProfileIterator
    method createCoworkersIterator(profileId):ProfileIterator


// Конкретная коллекция знает, объекты каких итераторов нужно
// создавать.
class Facebook implements SocialNetwork is
    // ...Основной код коллекции...

    // Код получения нужного итератора.
    method createFriendsIterator(profileId) is
        return new FacebookIterator(this, profileId, "friends")
    method createCoworkersIterator(profileId) is
        return new FacebookIterator(this, profileId, "coworkers")


// Общий интерфейс итераторов.
interface ProfileIterator is
    method getNext():Profile
    method hasMore():bool


// Конкретный итератор.
class FacebookIterator implements ProfileIterator is
    // Итератору нужна ссылка на коллекцию, которую он обходит.
    private field facebook: Facebook
    private field profileId, type: string

    // Но каждый итератор обходит коллекцию, независимо от
    // остальных, поэтому он содержит информацию о текущей
    // позиции обхода.
    private field currentPosition
    private field cache: array of Profile

    constructor FacebookIterator(facebook, profileId, type) is
        this.facebook = facebook
        this.profileId = profileId
        this.type = type

    private method lazyInit() is
        if (cache == null)
```

```
        // Итератор реализует методы базового интерфейса по-своему.
        method getNext() is
            if (hasMore())
                result = cache[currentPosition]
                currentPosition++
                return result

        method hasMore() is
            lazyInit()
            return currentPosition < cache.length


// Вот ещё полезная тактика: мы можем передавать объект
// итератора вместо коллекции в клиентские классы. При таком
// подходе клиентский код не будет иметь доступа к коллекциям, а
// значит, его не будут волновать подробности их реализаций. Ему
// будет доступен только общий интерфейс итераторов.
class SocialSpammer is
    method send(iterator: ProfileIterator, message: string) is
        while (iterator.hasMore())
            profile = iterator.getNext()
            System.sendEmail(profile.getEmail(), message)


// Класс приложение конфигурирует классы, как захочет.
class Application is
    field network: SocialNetwork
    field spammer: SocialSpammer

    method config() is
        if working with Facebook
            this.network = new Facebook()
        if working with LinkedIn
            this.network = new LinkedIn()
        this.spammer = new SocialSpammer()

    method sendSpamToFriends(profile) is
        iterator = network.createFriendsIterator(profile.getId())
        spammer.send(iterator, "Very important message")

    method sendSpamToCoworkers(profile) is
        iterator = network.createCoworkersIterator(profile.getId())
        spammer.send(iterator, "Very important message")
```

Применимость

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 — будь то сам класс коллекции или часть бизнес-логики программы. Применив итератор, вы можете выделить 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 общий интерфейс итераторов. Обязательный минимум — это операция получения следующего элемента коллекции. Но для удобства можно предусмотреть и другое. Например, методы для получения предыдущего элемента, текущей позиции, проверки окончания обхода и прочие.

2. Создайте интерфейс коллекции и опишите в нём метод получения итератора. Важно, чтобы сигнатура метода 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 ссылку на собственный объект в конструктор итератора.

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.

✗ Не оправдан, 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.

</> Примеры реализации паттерна