



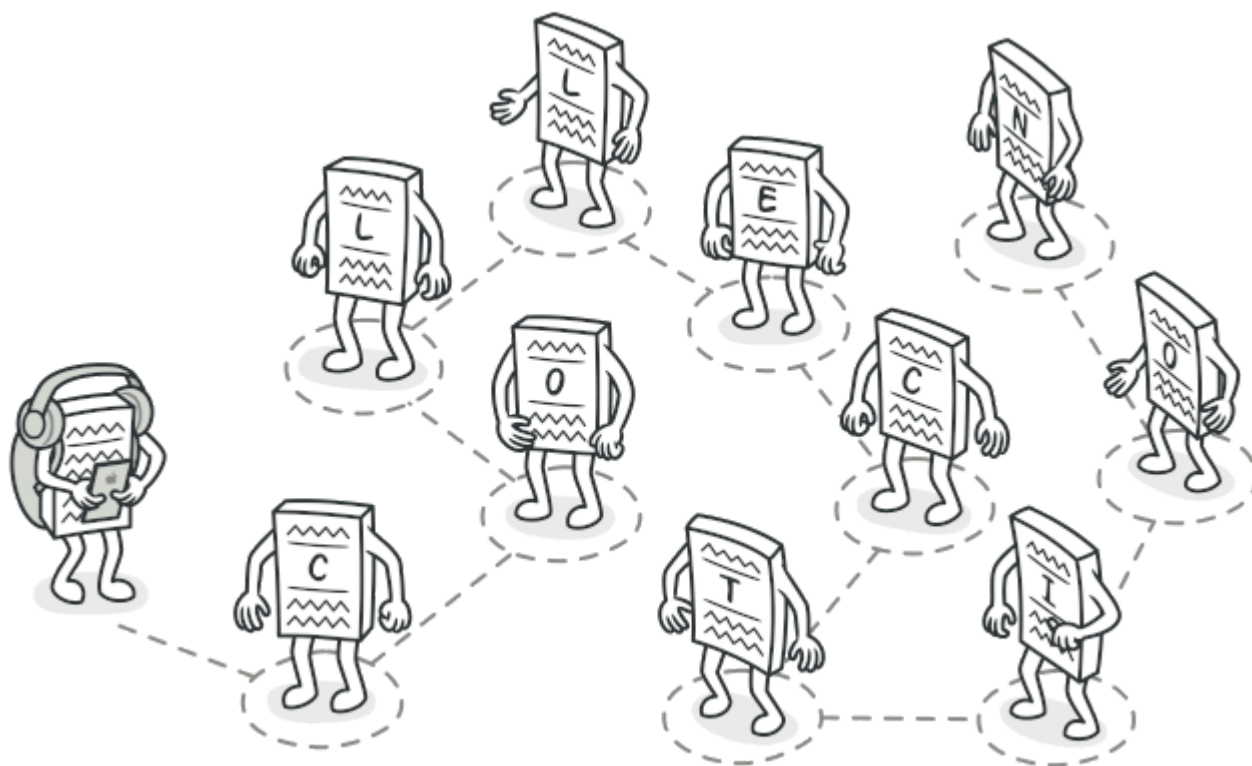
HELP UKRAINE STOP RUSSIA

[Home](#) / [Design Patterns](#) / [Behavioral Patterns](#)

Iterator

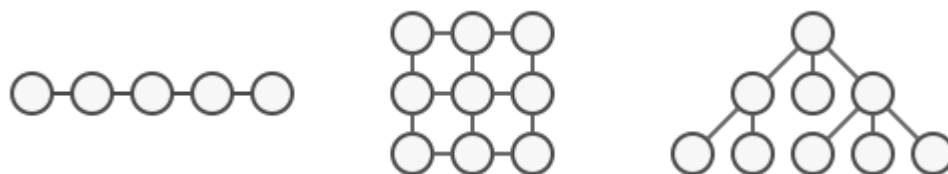
Intent

Iterator is a behavioral design pattern that lets you traverse elements of a collection without exposing its underlying representation (list, stack, tree, etc.).



Problem

Collections are one of the most used data types in programming. Nonetheless, a collection is just a container for a group of objects.



Various types of collections.

Most collections store their elements in simple lists. However, some of them are based on stacks, trees, graphs and other complex data structures.

But no matter how a collection is structured, it must provide some way of accessing its elements so that other code can use these elements. There should be a way to go through each element of the collection without accessing the same elements over and over.

This may sound like an easy job if you have a collection based on a list. You just loop over all of the elements. But how do you sequentially traverse elements of a complex data structure, such as a tree? For example, one day you might be just fine with depth-first traversal of a tree. Yet the next day you might require breadth-first traversal. And the next week, you might need something else, like random access to the tree elements.



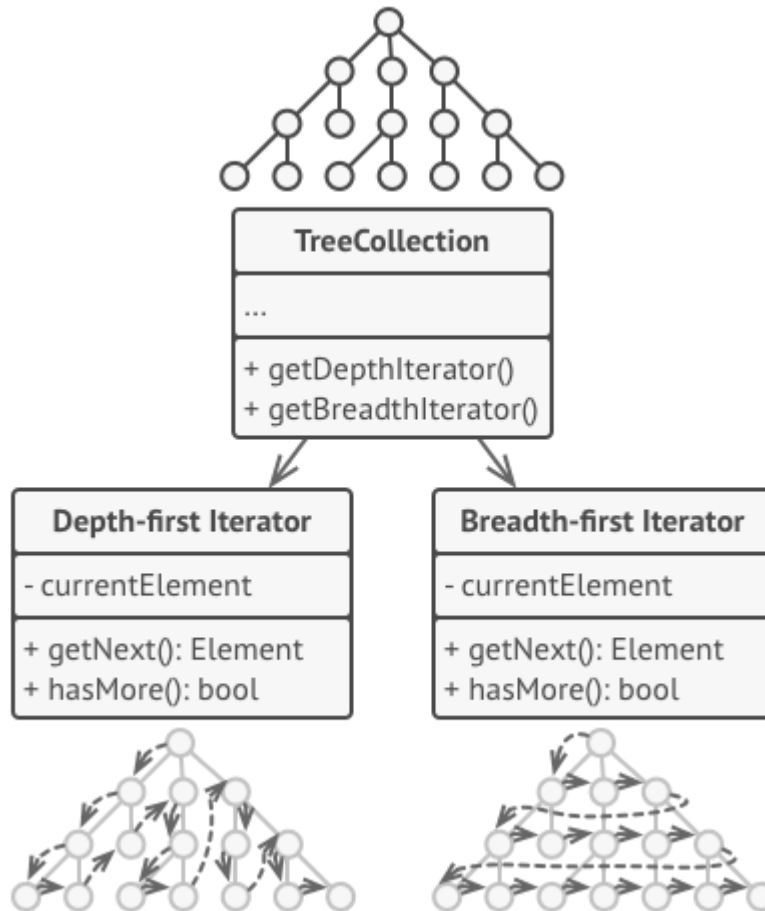
The same collection can be traversed in several different ways.

Adding more and more traversal algorithms to the collection gradually blurs its primary responsibility, which is efficient data storage. Additionally, some algorithms might be tailored for a specific application, so including them into a generic collection class would be weird.

On the other hand, the client code that's supposed to work with various collections may not even care how they store their elements. However, since collections all provide different ways of accessing their elements, you have no option other than to couple your code to the specific collection classes.

😊 Solution

The main idea of the Iterator pattern is to extract the traversal behavior of a collection into a separate object called an *iterator*.



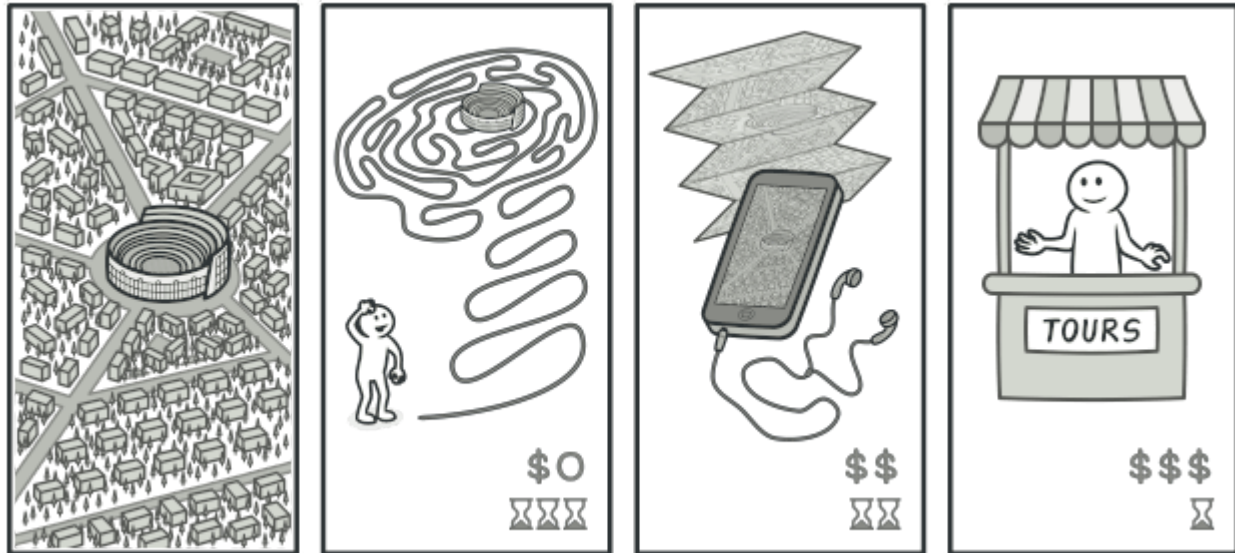
Iterators implement various traversal algorithms. Several iterator objects can traverse the same collection at the same time.

In addition to implementing the algorithm itself, an iterator object encapsulates all of the traversal details, such as the current position and how many elements are left till the end. Because of this, several iterators can go through the same collection at the same time, independently of each other.

Usually, iterators provide one primary method for fetching elements of the collection. The client can keep running this method until it doesn't return anything, which means that the iterator has traversed all of the elements.

All iterators must implement the same interface. This makes the client code compatible with any collection type or any traversal algorithm as long as there's a proper iterator. If you need a special way to traverse a collection, you just create a new iterator class, without having to change the collection or the client.

Real-World Analogy



Various ways to walk around Rome.

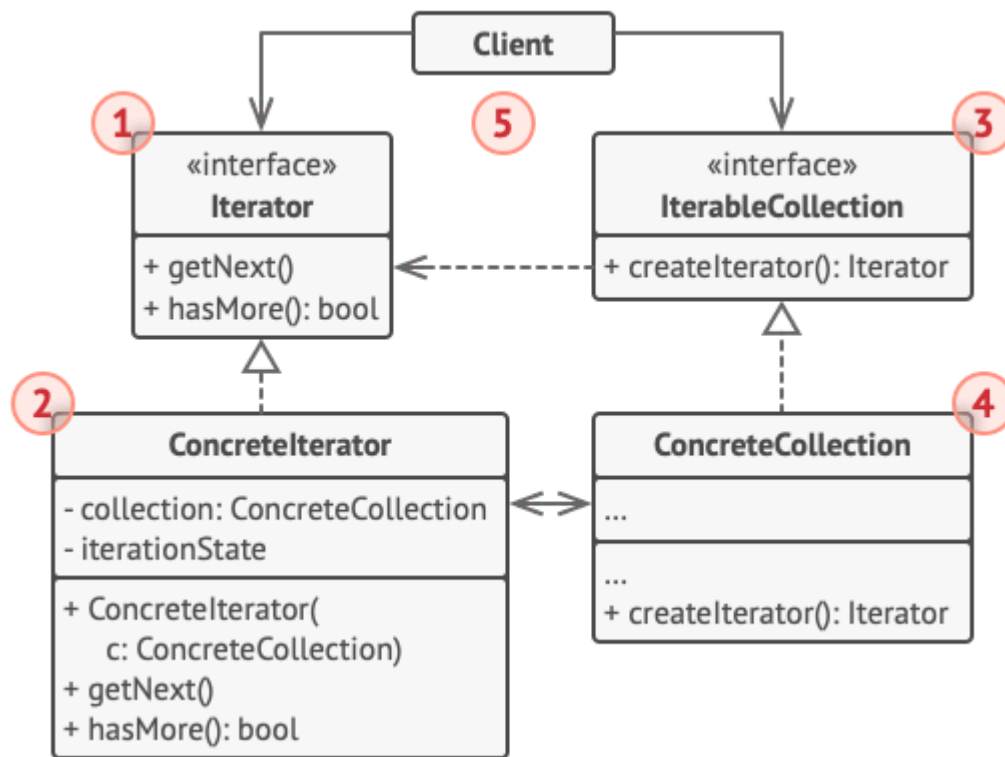
You plan to visit Rome for a few days and visit all of its main sights and attractions. But once there, you could waste a lot of time walking in circles, unable to find even the Colosseum.

On the other hand, you could buy a virtual guide app for your smartphone and use it for navigation. It's smart and inexpensive, and you could be staying at some interesting places for as long as you want.

A third alternative is that you could spend some of the trip's budget and hire a local guide who knows the city like the back of his hand. The guide would be able to tailor the tour to your likings, show you every attraction and tell a lot of exciting stories. That'll be even more fun; but, alas, more expensive, too.

All of these options—the random directions born in your head, the smartphone navigator or the human guide—act as iterators over the vast collection of sights and attractions located in Rome.

Structure



1. The **Iterator** interface declares the operations required for traversing a collection: fetching the next element, retrieving the current position, restarting iteration, etc.
2. **Concrete Iterators** implement specific algorithms for traversing a collection. The iterator object should track the traversal progress on its own. This allows several iterators to traverse the same collection independently of each other.
3. The **Collection** interface declares one or multiple methods for getting iterators compatible with the collection. Note that the return type of the methods must be declared as the iterator interface so that the concrete collections can return various kinds of iterators.
4. **Concrete Collections** return new instances of a particular concrete iterator class each time the client requests one. You might be wondering, where's the rest of the collection's code? Don't worry, it should be in the same class. It's just that these details aren't crucial to the actual pattern, so we're omitting them.
5. The **Client** works with both collections and iterators via their interfaces. This way the client isn't coupled to concrete classes, allowing you to use various collections and iterators with the same client code.

Pseudocode

```
classDiagram
    class SocialSpammer {
        ...
        + send(iterator, message)
    }
    class Application {
        - spammer
        - network
        + sendSpamToFriends(profile)
        + sendSpamToCoworkers(profile)
    }
    class Profile {
        ...
        + getId()
        + getEmail()
    }
    class ProfileIterator {
        <<interface>>
        + getNext(): Profile
        + hasMore(): bool
    }
    class SocialNetwork {
        <<interface>>
        + createFriendsIterator(profileId): ProfileIterator
        + createCoworkersIterator(profileId): ProfileIterator
    }
    class FacebookIterator {
        - facebook: Facebook
        - profileId, type
        - currentPosition
        - cache: Profile[]
        + FacebookIterator(...)
        - lazyInit()
        + getNext(): Profile
        + hasMore(): bool
    }
    class Facebook {
        ...
        + createFriendsIterator(profileId): ProfileIterator
        + createCoworkersIterator(profileId): ProfileIterator
    }

    Application --> SocialSpammer
    Application --> Profile
    Application --> SocialNetwork
    Application --> Facebook
    SocialSpammer ..> Profile
    Profile ..> ProfileIterator
    ProfileIterator <|.. FacebookIterator
    SocialNetwork <|.. Facebook
    ProfileIterator ..> SocialNetwork
    Facebook o--> FacebookIterator
```

The diagram illustrates the Iterator design pattern for a social spammer application. It consists of the following classes and interfaces:

- SocialSpammer**: Contains a `send(iterator, message)` method.
- Application**: Contains attributes `- spammer` and `- network`, and methods `+ sendSpamToFriends(profile)` and `+ sendSpamToCoworkers(profile)`. It has associations with `SocialSpammer`, `Profile`, `SocialNetwork`, and `Facebook`.
- Profile**: Contains attributes `...` and methods `+ getId()` and `+ getEmail()`. It has a dashed association with `ProfileIterator`.
- «interface» ProfileIterator**: Defines methods `+ getNext(): Profile` and `+ hasMore(): bool`. It is implemented by `FacebookIterator` and has a dashed association with `SocialNetwork`.
- «interface» SocialNetwork**: Defines methods `+ createFriendsIterator(profileId): ProfileIterator` and `+ createCoworkersIterator(profileId): ProfileIterator`. It is implemented by `Facebook`.
- FacebookIterator**: Implements `ProfileIterator`. It contains attributes `- facebook: Facebook`, `- profileId, type`, `- currentPosition`, and `- cache: Profile[]`. It has methods `+ FacebookIterator(...)`, `- lazyInit()`, `+ getNext(): Profile`, and `+ hasMore(): bool`. It has a bidirectional association with `Facebook`.
- Facebook**: Implements `SocialNetwork`. It contains attributes `...` and methods `+ createFriendsIterator(profileId): ProfileIterator` and `+ createCoworkersIterator(profileId): ProfileIterator`. It has a composition relationship with `FacebookIterator`.

6/12

The 'friends' iterator can be used to go over the friends of a given profile. The 'colleagues' iterator does the same, except it omits friends who don't work at the same company as a target person. Both iterators implement a common interface which allows clients to fetch profiles without diving into implementation details such as authentication and sending REST requests.

The client code isn't coupled to concrete classes because it works with collections and iterators only through interfaces. If you decide to connect your app to a new social network, you simply need to provide new collection and iterator classes without changing the existing code.

```
// The collection interface must declare a factory method for
// producing iterators. You can declare several methods if there
// are different kinds of iteration available in your program.
interface SocialNetwork is
    method createFriendsIterator(profileId):ProfileIterator
    method createCoworkersIterator(profileId):ProfileIterator
```

```
// Each concrete collection is coupled to a set of concrete
// iterator classes it returns. But the client isn't, since the
// signature of these methods returns iterator interfaces.
class Facebook implements SocialNetwork is
    // ... The bulk of the collection's code should go here ...

    // Iterator creation code.
    method createFriendsIterator(profileId) is
        return new FacebookIterator(this, profileId, "friends")
    method createCoworkersIterator(profileId) is
        return new FacebookIterator(this, profileId, "coworkers")
```

```
// The common interface for all iterators.
interface ProfileIterator is
    method getNext():Profile
    method hasMore():bool
```

```
// The concrete iterator class.
class FacebookIterator implements ProfileIterator is
    // The iterator needs a reference to the collection that it
    // traverses.
    private field facebook: Facebook
    private field profileId, type: string

    // An iterator object traverses the collection independently
    // from other iterators. Therefore it has to store the
    // iteration state.
```

```

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)
        cache = facebook.socialGraphRequest(profileId, type)

// Each concrete iterator class has its own implementation
// of the common iterator interface.
method getNext() is
    if (hasMore())
        currentPosition++
        return cache[currentPosition]

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

// Here is another useful trick: you can pass an iterator to a
// client class instead of giving it access to a whole
// collection. This way, you don't expose the collection to the
// client.
//
// And there's another benefit: you can change the way the
// client works with the collection at runtime by passing it a
// different iterator. This is possible because the client code
// isn't coupled to concrete iterator classes.
class SocialSpammer is
    method send(iterator: ProfileIterator, message: string) is
        while (iterator.hasMore())
            profile = iterator.getNext()
            System.sendEmail(profile.getEmail(), message)

// The application class configures collections and iterators
// and then passes them to the client code.
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")
```

Applicability


 Use the Iterator pattern when your collection has a complex data structure under the hood, but you want to hide its complexity from clients (either for convenience or security reasons).

 The iterator encapsulates the details of working with a complex data structure, providing the client with several simple methods of accessing the collection elements. While this approach is very convenient for the client, it also protects the collection from careless or malicious actions which the client would be able to perform if working with the collection directly.

 Use the pattern to reduce duplication of the traversal code across your app.

 The code of non-trivial iteration algorithms tends to be very bulky. When placed within the business logic of an app, it may blur the responsibility of the original code and make it less maintainable. Moving the traversal code to designated iterators can help you make the code of the application more lean and clean.

 Use the Iterator when you want your code to be able to traverse different data structures or when types of these structures are unknown beforehand.

 The pattern provides a couple of generic interfaces for both collections and iterators. Given that your code now uses these interfaces, it'll still work if you pass it various kinds of collections and iterators that implement these interfaces.

How to Implement

1. Declare the iterator interface. At the very least, it must have a method for fetching the next element from a collection. But for the sake of convenience you can add a couple of other methods, such as fetching the previous element, tracking the current position, and checking the end of the iteration.
2. Declare the collection interface and describe a method for fetching iterators. The return type should be equal to that of the iterator interface. You may declare similar methods if you plan to have several distinct groups of iterators.
3. Implement concrete iterator classes for the collections that you want to be traversable with iterators. An iterator object must be linked with a single collection instance. Usually, this link is established via the iterator's constructor.
4. Implement the collection interface in your collection classes. The main idea is to provide the client with a shortcut for creating iterators, tailored for a particular collection class. The collection object must pass itself to the iterator's constructor to establish a link between them.
5. Go over the client code to replace all of the collection traversal code with the use of iterators. The client fetches a new iterator object each time it needs to iterate over the collection elements.

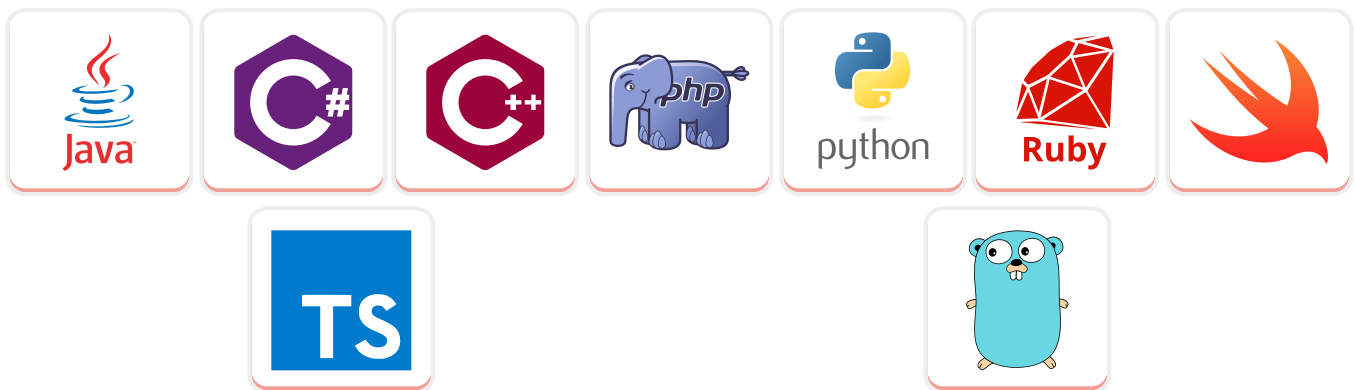
Pros and Cons

- ✓ *Single Responsibility Principle.* You can clean up the client code and the collections by extracting bulky traversal algorithms into separate classes.
- ✓ *Open/Closed Principle.* You can implement new types of collections and iterators and pass them to existing code without breaking anything.
- ✓ You can iterate over the same collection in parallel because each iterator object contains its own iteration state.
- ✓ For the same reason, you can delay an iteration and continue it when needed.
- ✗ Applying the pattern can be an overkill if your app only works with simple collections.
- ✗ Using an iterator may be less efficient than going through elements of some specialized collections directly.

↔ Relations with Other Patterns

- You can use **Iterators** to traverse **Composite** trees.
- You can use **Factory Method** along with **Iterator** to let collection subclasses return different types of iterators that are compatible with the collections.
- You can use **Memento** along with **Iterator** to capture the current iteration state and roll it back if necessary.
- You can use **Visitor** along with **Iterator** to traverse a complex data structure and execute some operation over its elements, even if they all have different classes.

</> Code Examples



Support our free website and own the eBook!

- 22 design patterns and 8 principles explained in depth.

- 409 well-structured, easy to read, jargon-free pages.
- 225 clear and helpful illustrations and diagrams.
- An archive with code examples in 9 languages.
- All devices supported: PDF/EPUB/MOBI/KFX formats.

[Learn more...](#)[Home](#)[Refactoring](#)[Design Patterns](#)[Premium Content](#)[Forum](#)[Contact us](#)

© 2014-2022 Refactoring.Guru. All rights reserved.

🏠 Khmelnytske shosse 19 / 27, Kamianets-Podilskyi, Ukraine, 32305

✉ Email: support@refactoring.guru

🖼 Illustrations by Dmitry Zhart

[Terms & Conditions](#)[Privacy Policy](#)[Content Usage Policy](#)