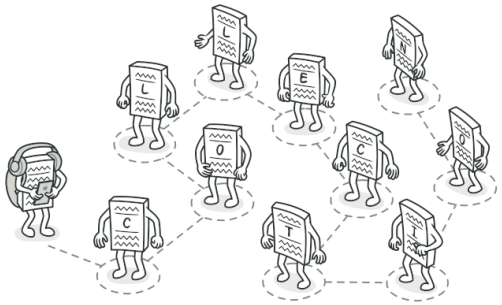
Padrões de Projetos:
ITERATOR / MEDIATOR

Produzido: Alexandre Bedin, Vitória Bomfim

Professor: Rogério Xavier

O que é o Iterator ?



- **Navegação Sequencial**
  - O Iterator fornece uma maneira uniforme de navegar por diferentes tipos de coleções, sejam elas listas, pilhas, árvores, ou outras estruturas de dados.
- **Abstração de Implementação**
  - Ele separa a lógica de navegação da implementação da estrutura de dados subjacente, tornando o código mais modular e reutilizável.
- **Redução de Complexidade**
  - O Iterator simplifica a interface de acesso aos elementos de uma coleção, escondendo os detalhes complexos da implementação.

Quando usar o Iterator?

- **Acesso Sequencial**
  - Quando você precisa acessar os elementos de uma coleção de forma sequencial, sem se preocupar com a estrutura interna.
- **Múltiplos Iteradores**
  - Quando você precisa ter vários iteradores ativos na mesma coleção, cada um com seu próprio estado de navegação.
- **Abstração de Implementação**
  - Quando você deseja ocultar os detalhes da implementação de uma coleção e fornecer uma interface simples e uniforme.

Problemas específicos resolvidos pelo Iterator

- **Manipulação de grandes conjuntos de dados sem exceder a memória:** Em sistemas onde os conjuntos de dados são muito grandes para caber na memória, o Iterator permite acessar um elemento de cada vez, sem a necessidade de carregar toda a coleção na memória.
- **Execução eficiente com lazy evaluation:** O Iterator pode implementar a técnica de **lazy evaluation**, que significa que os dados são calculados apenas quando são necessários.
- **Processamento contínuo de streams de dados:** Quando se trabalha com dados em tempo real (streams), como dados de sensores ou logs de servidor, o Iterator permite a leitura e processamento contínuo desses dados.
- **Geração dinâmica de sequências infinitas:** Com Iterator, é possível criar sequências que podem teoricamente ser infinitas, como a sequência de Fibonacci ou números primos.

Vantagens

- **Economia de Memória**: Iteradores geram itens um a um, sem precisar carregar toda a coleção na memória. Isso é especialmente útil para lidar com grandes conjuntos de dados.
- **Eficiência em Grandes Conjuntos de Dados**: Como iteradores acessam um elemento por vez, eles evitam o processamento desnecessário de grandes volumes de dados de uma só vez.
- **Lazy Evaluation**: Os elementos de um iterador são gerados apenas quando solicitados. Isso torna iteradores ideais para operações como streaming de dados ou leitura de grandes arquivos que podem ser processados gradualmente.
- **Flexibilidade em Loops**: Iteradores facilitam a implementação de loops que processam cada item de uma sequência sem precisar acessar a coleção diretamente.

Desvantagens

- **Iteradores são Descartáveis**: Depois de iterar até o fim de um iterador, ele não pode ser "resetado" ou reiniciado. Para iterar novamente, é necessário criar um novo iterador.

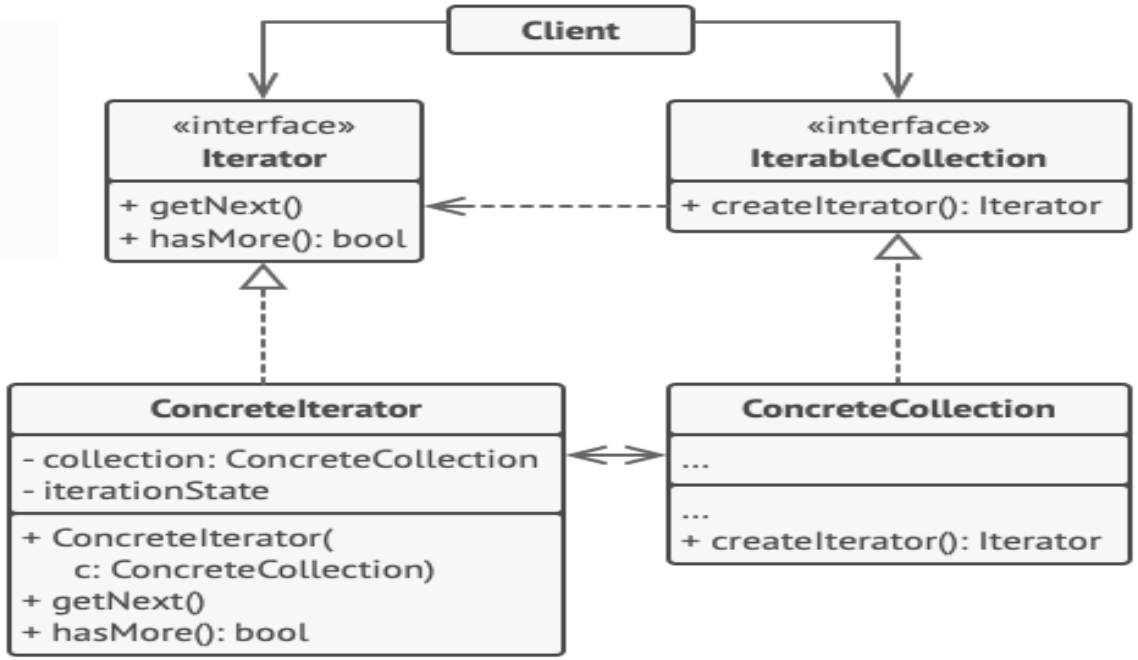
- **Sem Acesso Direto**: Diferente de listas e outras coleções, iteradores não permitem acessar um item específico por índice. A única forma de obter elementos é avançando sequencialmente.

- **Difícil de Depurar**: Como iteradores geram valores um a um, pode ser difícil verificar o estado do iterador ou retroceder caso algo dê errado durante o processamento.

- **Potencial para Exceções**: Quando se chega ao fim de um iterador, ele gera uma exceção StopIteration. Isso precisa ser tratado adequadamente, o que pode aumentar a complexidade do código.

Estrutura

1 A interface **Iterador** declara as operações necessárias para percorrer uma coleção: buscar o próximo elemento, pegar a posição atual, recomeçar a iteração, etc.



2 **Iteradores Concretos** implementam algoritmos específicos para percorrer uma coleção. O objeto iterador deve monitorar o progresso da travessia por conta própria. Isso permite que diversos iteradores percorram a mesma coleção independentemente de cada um.

3 A interface **Coleção** declara um ou mais métodos para obter os iteradores compatíveis com a coleção. Observe que o tipo do retorno dos métodos deve ser declarado como a interface do iterador para que as coleções concretas possam retornar vários tipos de iteradores.

4 **Coleções Concretas** retornam novas instâncias de uma classe iterador concreta em particular cada vez que o cliente pede por uma. Você pode estar se perguntando, onde está o resto do código da coleção? Não se preocupe, ele deve ficar na mesma classe. É que esses detalhes não são cruciais para o padrão atual, então optamos por omiti-los.

5 O **Cliente** trabalha tanto com as coleções como os iteradores através de suas interfaces. Dessa forma o cliente não fica acoplado a suas classes concretas, permitindo que você use várias coleções e iteradores com o mesmo código cliente.

Tipicamente, os clientes não criam iteradores por conta própria, mas ao invés disso os obtêm das coleções. Ainda assim, em certos casos, o cliente pode criar um diretamente; por exemplo, quando o cliente define seu próprio iterador especial.

```java
public interface Iterator<I> {
    boolean hasNext();
    I next();
}
```

Exemplo Prático

O que é o Mediator?

- Um padrão de design comportamental que centraliza as interações complexas entre objetos.

- Tem como **Função Principal** controla a comunicação entre objetos, conhecidos como "colaboradores", através de um mediador único. Com isso, evita-se a necessidade de cada objeto manter referências a todos os outros com os quais precisa se comunicar.

No padrão Mediator, os principais componentes são:

- **Mediator (Mediador)**
  - Ele encapsula a lógica de como os objetos interagem entre si, promovendo a comunicação indireta.

- **ConcreteMediator (Mediador Concreto)**
  - Contém a lógica específica de comunicação, referências entre os colaboradores,e decide como e quando interagir com eles.

- **Colleague (Colaborador)**
  - Cada Colleague é um objeto participante que depende de outros objetos, mas que não interage diretamente com eles. Em vez disso, comunica-se através do Mediator.

- **ConcreteColleague (Colaborador Concreto)**
  - Cada ConcreteColleague interage com o Mediator ao invés de acessar diretamente outros objetos.

Exemplo de Problema

Imagine um **sistema de chat em grupo** onde cada usuário (objeto User) pode enviar mensagens para outros usuários do mesmo grupo. Se cada usuário precisa conhecer todos os outros diretamente para enviar mensagens, o sistema rapidamente se torna complexo e confuso de manter. A adição de novos usuários exige atualizar referências em cada usuário existente, e a remoção de um usuário exige uma reconfiguração geral das dependências.

Solução com o Mediator

- O padrão Mediator resolve esse problema ao introduzir um objeto intermediário o ChatRoom , que centraliza a comunicação entre os usuários.

Interface Mediator

```
public interface ChatRoom {
    void sendMessage(String message, User user);
    void addUser(User user);
}
```

- Interface ChatRoom: Define o comportamento do mediador, com métodos para enviar mensagens e adicionar usuários.

Mediador Concreto (ConcreteMediator)

```java
import java.util.ArrayList;
import java.util.List;

public class GroupChat implements ChatRoom {
    private List<User> users = new ArrayList<>();

    @Override
    public void addUser(User user) {
        users.add(user);
    }

    @Override
    public void sendMessage(String message, User sender) {
        for (User user : users) {
            // Envia a mensagem para todos, exceto o remetente
            if (user != sender) {
                user.receive(message);
            }
        }
    }
}
```

GroupChat: Implementa o Mediator e gerencia a lista de usuários. Quando um usuário envia uma mensagem, o GroupChat encaminha essa mensagem a todos os outros usuários.

Classe Colleague (Usuário)

```
public abstract class User {
    protected ChatRoom chatRoom;
    protected String name;

    public User(ChatRoom chatRoom, String name) {
        this.chatRoom = chatRoom;
        this.name = name;
    }

    public abstract void send(String message);
    public abstract void receive(String message);
}
```

Classe User: É a abstração dos usuários do chat, com métodos para enviar e receber mensagens.

Colleague Concreto (Usuário Concreto)

```java
public class BasicUser extends User {
    public BasicUser(ChatRoom chatRoom, String name) {
        super(chatRoom, name);
    }

    @Override
    public void send(String message) {
        System.out.println(this.name + " envia: " + message);
        chatRoom.sendMessage(message, this);
    }

    @Override
    public void receive(String message) {
        System.out.println(this.name + " recebe: " + message);
    }
}
```

BasicUser: Implementa a lógica específica de envio e recebimento de mensagens.

Exemplo de Uso do Padrão Mediator

```java
public class ChatApplication {
    public static void main(String[] args) {
        ChatRoom chatRoom = new GroupChat();

        User user1 = new BasicUser(chatRoom, "Alice");
        User user2 = new BasicUser(chatRoom, "Bob");
        User user3 = new BasicUser(chatRoom, "Charlie");

        // Adiciona os usuários à sala de chat
        chatRoom.addUser(user1);
        chatRoom.addUser(user2);
        chatRoom.addUser(user3);

        // Envia mensagens
        user1.send("Oi, pessoal!");
        user2.send("Olá, Alice!");
        user3.send("Oi, Bob e Alice!");
    }
}
```

ChatApplication: Demonstra o uso do padrão Mediator. Aqui, três usuários são adicionados à sala de chat, e cada um envia uma mensagem que o Mediador, GroupChat, distribui para os outros usuários.