Refactoring to Patterns em Java 8

Eder Ignatowicz
Sr. Software Engineer
JBoss by Red Hat

Design Patterns +
Java +
Programação Funcional
<3

Antes uma história

:)


APUCARANA
Cidade Educação
CAPITAL DO BONÉ


UNICAMP

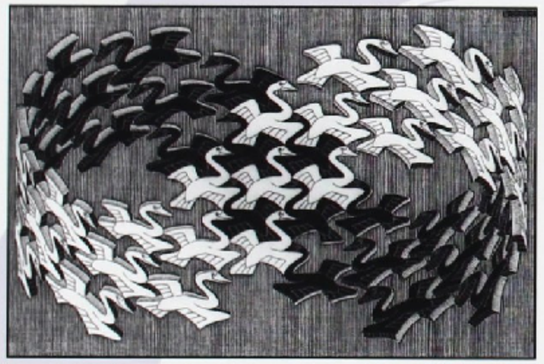
Me tornar um World Class Developer

Como virar um World Class Developer?

Design Patterns

Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides



Foreword by Grady Booch

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

:(

?

:)

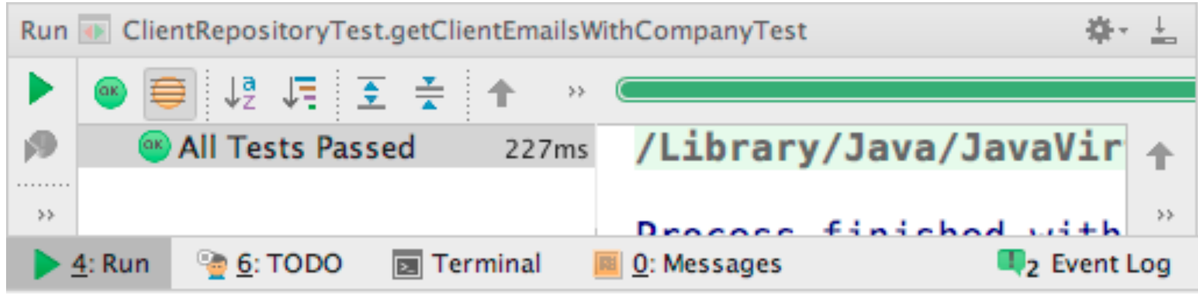
Refactoring to Patterns em Java 8

Refactoring Loops to Collection Pipelines

```
public class Client {

    private String name;
    private String email;
    private Company company;

    public Client( String name, String email, Company company ) {
        this.name = name;
        this.email = email;
        this.company = company;
    }

    public Client( String name ) {
        this.name = name;
    }

    public Client( String name, String email ) {
        this.name = name;
        this.email = email;
    }


 …
}
```

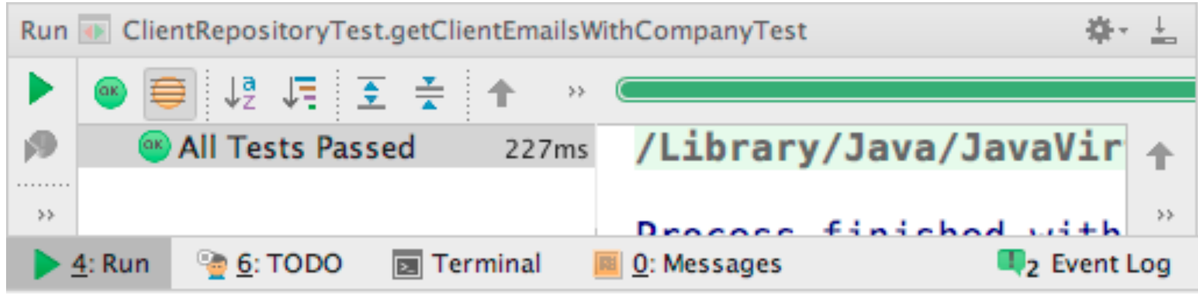
```java
public class ClientRepositoryTest {


    private ClientRepository repo;

    @Before
    public void setup() {
        Company empresa = new Company( "RedHat" );
        Client completo1 = new Client( "Completo1", "completo1@redhat.com", empresa );
        Client completo2 = new Client( "Completo2", "completo2@redhat.com", empresa );
        Client semEmpresa = new Client( "SemEmpresa", "semEmpresa@ederign.me" );
        Client somenteNome = new Client( "SomenteNome" );
        repo = new ClientRepository(
                            Arrays.asList( completo1, semEmpresa, completo2, somenteNome ) );

    }

    @Test
    public void getClientEmailsWithCompanyTest() {
        List<String> clientMails = repo.getClientMails();
        assertEquals( 2, clientMails.size() );
        assertTrue( clientMails.contains( "completo1@redhat.com" ) );
        assertTrue( clientMails.contains( "completo2@redhat.com" ) );
        assertTrue( !clientMails.contains( "semEmpresa@ederign.me" ) );
    }
}
```

```
public List<String> getClientMails() {
        ArrayList<String> emails = new ArrayList<>();

        for ( Client client : clients ) {
            if ( client.getCompany() != null ) {
                String email = client.getEmail();
                if ( email != null ){
                    emails.add( email );
                }
            }
        }

        return emails;
    }
```



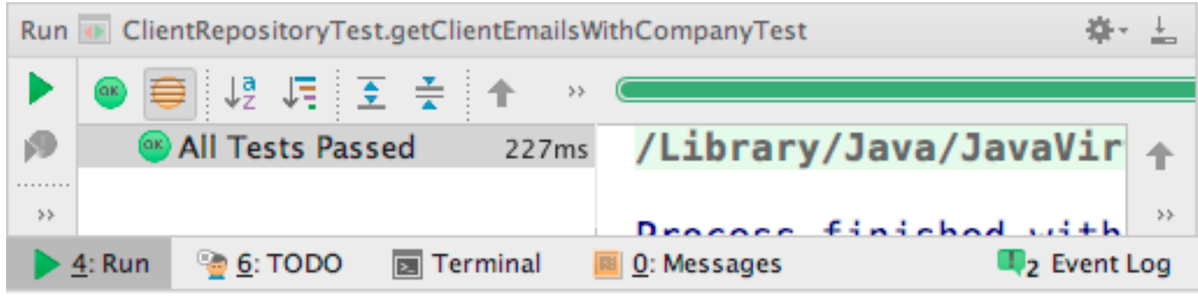
```
    public List<String> getClientMails() {
        ArrayList<String> emails = new ArrayList<>();
        List<Client> pipeline = clients;
        for ( Client client : pipeline ) {
            if ( client.getCompany() != null ) {
                String email = client.getEmail();
                if ( email != null ){
                    emails.add( email );
                }
            }
        }
        return emails;
    }
}
```

Extract Variable



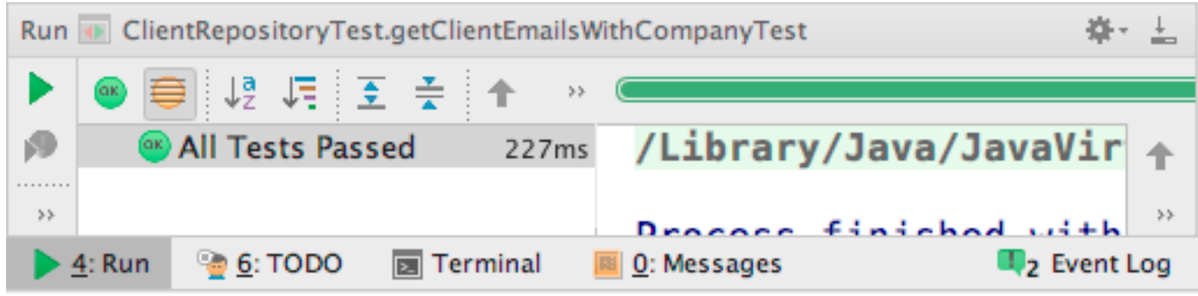
```
    public List<String> getClientMails() {
        ArrayList<String> emails = new ArrayList<>();
        List<Client> pipeline = clients
                .stream()
                .filter( c -> c.getCompany() != null )
                .collect( Collectors.toList() );
        for ( Client client : pipeline ) {
            if ( client.getCompany() != null ) {
                String email = client.getEmail();
                if ( email != null ) {
                    emails.add( email );
                }
            }
        }

        return emails;
    }
}
```

Filter Operation



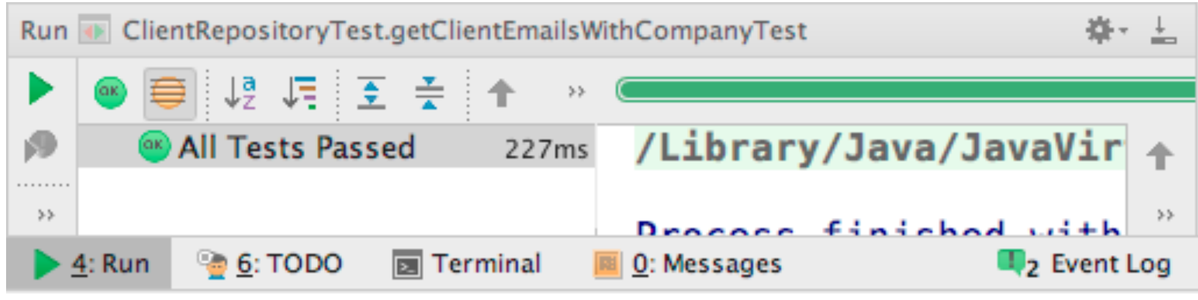
```
public List<String> getClientMails() {
      ArrayList<String> emails = new ArrayList<>();
      List<String> pipeline = clients
                .stream()
                .filter( c -> c.getCompany() != null )
                .map( c -> c.getEmail() )
                .collect( Collectors.toList() );
      for ( String mail : pipeline ) {
          String email = client.getEmail();
          if ( mail != null ) {
              emails.add( mail );
          }
      }

      return emails;
 }
```

Map Operation



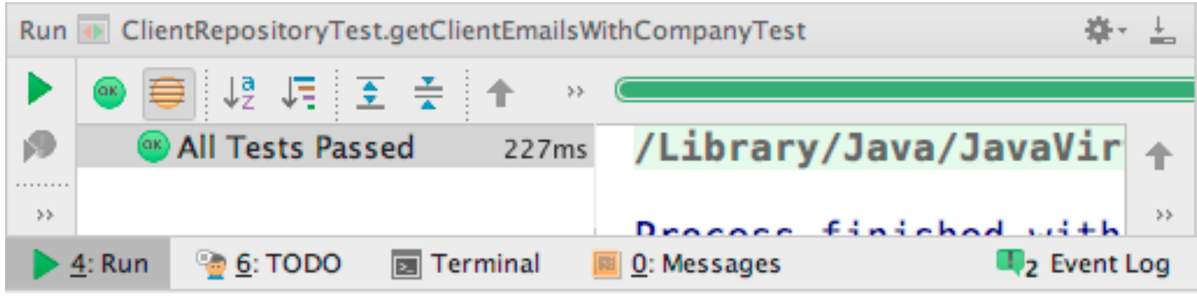
```java
public List<String> getClientMails() {
    ArrayList<String> emails = new ArrayList<>();
    List<String> pipeline = clients
            .stream()
            .filter( c -> c.getCompany() != null )
            .map( c -> c.getEmail() )
            .filter( m -> m != null )
            .collect( Collectors.toList() );
    for ( String mail : pipeline ) {
        if ( mail != null ) {
            emails.add( mail );
        }
    }

    return emails;
}
```

Filter Operation

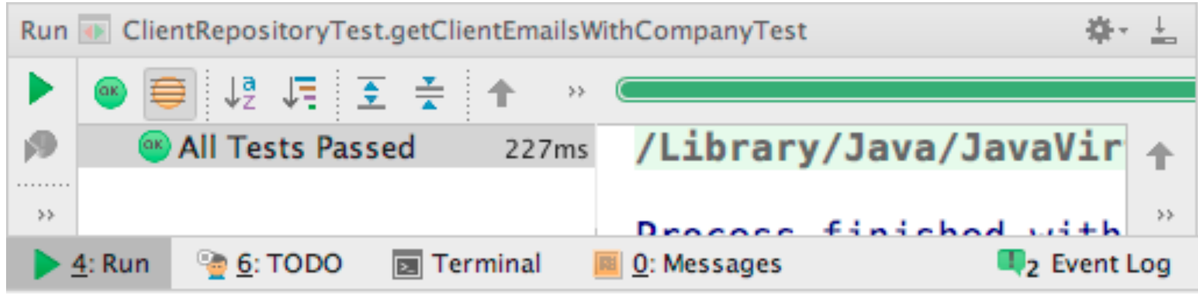


```
public List<String> getClientMails() {
    ArrayList<String> emails = new ArrayList<>();
    return clients
            .stream()
            .filter( c -> c.getCompany() != null )
            .map( c -> c.getEmail() )
            .filter( m -> m != null )
            .collect( Collectors.toList() );
    for ( String mail : pipeline ) {
        if ( mail != null ) {
            emails.add( mail );
        }
    }

    return emails;
}
```

Pipeline



```java
public List<String> getClientMails() {
    return clients
            .stream()
            .filter( c -> c.getCompany() != null )
            .map( c -> c.getEmail() )
            .filter( m -> m != null )
            .collect( Collectors.toList() );
}
```



Strategy

“Definir uma família de algoritmos, encapsular cada uma delas e torná-las intercambiáveis. Strategy permite que o algoritmo varie independentemente dos clientes que o utilizam” GAMMA, Erich et al.

```java
public class ShoppingCartTest {

    ShoppingCart cart;

    @Before
    public void setup() {
        Item item1 = new Item( 10 );
        Item item2 = new Item( 20 );
        cart = new ShoppingCart( Arrays.asList( item1, item2 ) );
    }

    @Test
    public void totalTest() {
        cart.pay( ShoppingCart.PaymentMethod.CREDIT ) );
    }
}
```

```
public class ShoppingCart {

   private List<Item> items;

    public ShoppingCart( List<Item> items ) {
        this.items = items;
    }

    public void pay( PaymentMethod method ) {
        int total = cartTotal();
        if ( method == PaymentMethod.CREDIT ) {
            System.out.println( “Pay with credit “ + total);
        } else if ( method == PaymentMethod.MONEY ) {
            System.out.println( “Pay with money “ + total );
        }
   }

    private int cartTotal() {
        return  items
                .stream()
                .mapToInt( Item::getValue )
                .sum();
    }
…
}
```

```
public class ShoppingCart {

    private List<Item> items;

    public ShoppingCart( List<Item> items ) {
        this.items = items;
    }

    public void pay( PaymentMethod method ) {
        int total = cartTotal();
        if ( method == PaymentMethod.CREDIT ) {
            System.out.println( “Pay with credit “ + total );
        } else if ( method == PaymentMethod.MONEY ) {
            System.out.println( “Pay with money “ + total );
        }
    }

    private int cartTotal() {
        return  items
                .stream()
                .mapToInt( Item::getValue )
                .sum();
    }

…
}
```

```java
public interface Payment {

    public void pay(int amount);

}
```

```java
public class CreditCard implements Payment {
    @Override
    public void pay( int amount ) {
        System.out.println( "Pay with Credit: “
                            + amount);
    }
}
```

```java
public class Money implements Payment {
    @Override
    public void pay( int amount ) {
        System.out.println( "Pay with Money: “
                                + amount);
}
```

```
public class ShoppingCart {
    …
    public void pay( Payment method ) {
        int total = cartTotal();
        method.pay( total );
    }

    private int cartTotal() {
        return  items
                .stream()
                .mapToInt( Item::getValue )
                .sum();
    }
}
```

Strategy

```java
public interface Payment {

    public void pay(int amount);

}
```

```java
public class CreditCard implements Payment {
    @Override
    public void pay( int amount ) {
        System.out.println( "make credit
                            payment logic" );
    }
}
```

```java
public class Money implements Payment {
    @Override
    public void pay( int amount ) {
        System.out.println( "make money
                            payment logic" );
    }
}
```

```java
public class DebitCard implements Payment {
    @Override
    public void pay( int amount ) {
        System.out.println( "make debit
                            payment logic" );
    }
}
```

```java
        public void totalTest() {
            assertEquals( 30, cart.pay( new CreditCard() ) );
            assertEquals( 30, cart.pay( new Money() ) );
            assertEquals( 30, cart.pay( new DebitCard() ) );
        }
    }
```

java.util.function

Interface Consumer<T>

Type Parameters:

`T - the type of the input to the operation`

All Known Subinterfaces:

`Stream.Builder<T>`

Functional Interface:

`This is a functional interface and can therefore be used as the assignment target for a lambda expression or method reference.`

All Methods	Instance Methods	Abstract Methods	Default Methods

Modifier and Type	Method and Description
`void`	**`accept(T t)`** Performs this operation on the given argument.
`default Consumer<T>`	**`andThen(Consumer<? super T> after)`** Returns a composed `Consumer` that performs, in sequence, this operation followed by the `after` operation.

```
public void pay( Payment method ) {
        int total = cartTotal();
        method.pay( total );
    }
```

```
public class ShoppingCart {
    …

    public void pay( Consumer<Integer> method ) {
        int total = cartTotal();
        method.accept( total );
    }
    …
}
```

```
public class ShoppingCart {
    …

    public void pay( Consumer<Integer> method ) {
        int total = cartTotal();
        method.accept( total );
    }
    …
}
```

```
public void totalTest() {
    cart.pay( amount -> System.out.println( "Pay with Credit: " + amount ) );
    cart.pay( amount -> System.out.println( "Pay with Money: " + amount ) );
    cart.pay( amount -> System.out.println( "Pay with Debit: " + amount ) );
}
```

```
public class PaymentTypes {

    public static void money( int amount ) {
        System.out.println( "Pay with Money: " + amount );
    }

    public static void debit( int amount ) {
        System.out.println( "Pay with Debit: " + amount );
    }

    public static void credit( int amount ) {
        System.out.println( "Pay with Credit: " + amount );
    }

}
```

```
        public void totalTest() {
            cart.pay( PaymentTypes::credit );
            cart.pay( PaymentTypes::debit );
            cart.pay( PaymentTypes::money );
        }
```

```
public class ShoppingCart {
    …
    public void pay( Consumer<Integer> method ) {
        int total = cartTotal();
        method.accept( total );
    }

    private int cartTotal() {
        return  items
                .stream()
                .mapToInt( Item::getValue )
                .sum();
    }
}
```

Strategy

Decorator

“Dinamicamente, agregar responsabilidades adicionais a objetos. Os Decorators fornecem uma alternativa flexível ao uso de subclasses para extensão de funcionalidades.”
GAMMA, Erich et al.

```
new BufferedReader(new FileReader(new File("some.file")));
```

```
public class Item {
    private int price;

    public Item( int price ) {
        this.price = price;
    }

    public int getPrice() {
        return price;
    }
}
```

Extras

Envio

Impostos

Embalagem

```java
public interface Item {
    int getPrice();
}
```

```java
public class Book implements Item {
    private int price;

    public Book( int price ) {
        this.price = price;
    }

    @Override
    public int getPrice() {
        return price;
    }
}
```

```
public abstract class ItemExtras implements Item {

    private Item item;

    public ItemExtras( Item item ) {
        this.item = item;
    }

    @Override
    public int getPrice() {
        return item.getPrice();
    }
}
```

```java
public class InternationalDelivery extends ItemExtras {

    public InternationalDelivery( Item item ) {
        super( item );
    }

    @Override
    public int getPrice() {
        return 5  + super.getPrice();
    }
}
```

```java
public class GiftPacking extends ItemExtras {

    public GiftPacking( Item item ) {
        super( item );
    }

    @Override
    public int getPrice() {
        return 15 + super.getPrice();
    }
}
```

```
public static void main( String[] args ) {

     Item book = new Book( 10 );
     book.getPrice(); //10

     Item international = new InternationalDelivery( book );
     international.getPrice(); //15

 }
```

```
public static void main( String[] args ) {

      Item book = new Book( 10 );
      book.getPrice(); //10

      Item internationalGift = new GiftPacking(
                                   new InternationalDelivery( book ) );
      internationalGift.getPrice(); //30

}
```

```
    public static void main( String[] args ) {

        Item book = new Book( 10 );
        book.getPrice(); //10

        Item internationalGiftWithTaxes = new InternacionalTaxes(
                                              new GiftPacking(
                                              new InternationalDelivery( book );
        internationalGiftWithTaxes.getPrice(); //80
    }
}
```

```
public static void main( String[] args ) {

        Item book = new Item( 10 );
        book.getPrice(); //10

        Function<Integer, Integer> giftPacking = value -> value + 15;
        giftPacking.apply( book.getPrice() ); //25
}
```

```
public static void main( String[] args ) {

        Item book = new Item( 10 );
        book.getPrice(); //10

        Function<Integer, Integer> giftPacking = value -> value + 15;
        giftPacking.apply( book.getPrice() ); //25

        Function<Integer, Integer> intTaxes = value -> value + 50;
        intTaxes.apply( book.getPrice() ); //60

}
```

```
public static void main( String[] args ) {

        Item book = new Item( 10 );
        book.getPrice(); //10

        Function<Integer, Integer> giftPacking = value -> value + 15;
        giftPacking.apply( book.getPrice() ); //25

        Function<Integer, Integer> intTaxes = value -> value + 50;
        intTaxes.apply( book.getPrice() ); //60


        giftPacking.andThen( intTaxes ).apply( book.getPrice() ); //75

}
```

```
public class Item {
    private int price;
    private Function<Integer, Integer>[] itemExtras = new Function[]{};

    public Item( int price ) {
        this.price = price;
    }

    public Item( int price, Function<Integer, Integer>... itemExtras) {
        this.price = price;
        this.itemExtras = itemExtras;
    }

    public int getPrice() {
        int priceWithExtras = price;
        for ( Function<Integer, Integer> itemExtra : itemExtras ) {
            priceWithExtras = itemExtra.apply( priceWithExtras );
        }
        return priceWithExtras;
    }

    public void setItemExtras( Function<Integer, Integer>... itemExtras ) {
        this.itemExtras = itemExtras;
    }
}
```

```
public static void main( String[] args ) {
    Item book = new Item( 10 );
    Function<Integer, Integer> giftPacking = value -> value + 15;
    Function<Integer, Integer> intTaxes = value -> value + 50;

    book.setItemExtras( giftPacking, intTaxes );

    book.getPrice(); //75

}
```

```
public static void main( String[] args ) {
    Item book = new Item( 10 );
    Function<Integer, Integer> giftPacking = value -> value + 15;
    Function<Integer, Integer> intTaxes = value -> value + 50;

    book.setItemExtras( giftPacking, intTaxes );

    book.getPrice(); //75

}
```

```apex
public class Packing {

    public static Integer giftPacking( Integer value ) {
        return value + 15;
    }
//other packing options here
}
```

```apex
public class Taxes {

    public static Integer internacional( Integer value ) {
        return value + 50;
    }
//other taxes here
}
```

```
public static void main( String[] args ) {
    Item book = new Item( 10, Packing::giftPacking,
                              Taxes::internacional );

    book.getPrice(); //75
}
```

```
public class Item {
    private int price;
    private Function<Integer, Integer>[] itemExtras = new Function[]{};

    public Item( int price ) {
        this.price = price;
    }

    public Item( int price, Function<Integer, Integer>... itemExtras) {
        this.price = price;
        this.itemExtras = itemExtras;
    }

    public int getPrice() {
        int priceWithExtras = price;
        for ( Function<Integer, Integer> itemExtra : itemExtras ) {
            priceWithExtras = itemExtra.apply( priceWithExtras );
        }
        return priceWithExtras;
    }

    public void setItemExtras( Function<Integer, Integer>... itemExtras ) {
        this.itemExtras = itemExtras;
    }
}
```

```
public class Item {
    private int price;
    private Function<Integer, Integer>[] itemExtras = new Function[]{};

    public Item( int price ) {
        this.price = price;
    }

    public Item( int price, Function<Integer, Integer>... itemExtras ) {
        this.price = price;
        this.itemExtras = itemExtras;
    }

     public int getPrice() {
         Function<Integer, Integer> extras =
                 Stream.of( itemExtras )
                 .reduce( Function.identity(), Function::andThen );
         return extras.apply( price );
     }

    public void setItemExtras( Function<Integer, Integer>... itemExtras ) {
        this.itemExtras = itemExtras;
    }
}
```

Template

“Definir o esqueleto de um algoritmo em uma operação, postergando alguns passos para as subclasses. Template Method permite que subclasses redefinam certos passo de um algoritmo sem mudar a estrutura do mesmo.”
GAMMA, Erich et al.

```
public abstract class Banking {

    public void processOperation( Operation op ) {
        preProcessing( op );
        process( op );
        postProcessing( op );
    }

    protected abstract void postProcessing( Operation op );

    protected abstract void preProcessing( Operation op );

    private void process( Operation op ) {
        //logic
        op.process( op );
    }
}
```

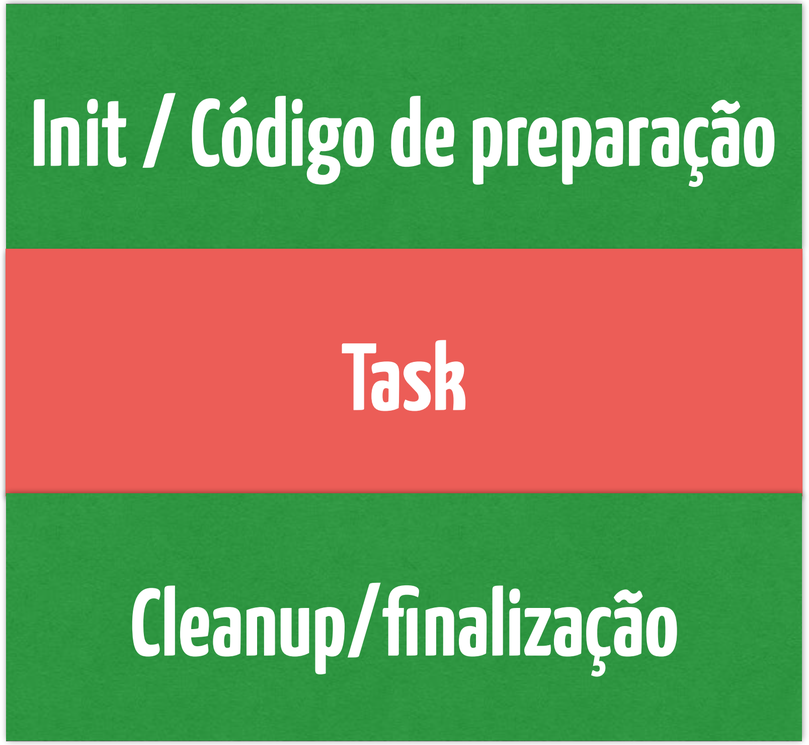
```java
public class VIPBanking extends Banking {

    @Override
    protected void preProcessing( Operation op ) {
        //pre processing vip logic
    }

    @Override
    protected void postProcessing( Operation op ) {
        //post processing vip logic
    }
}
```

```java
public class OnlineBanking extends Banking {

    @Override
    protected void preProcessing( Operation op ) {
        //pre processing online logic
    }

    @Override
    protected void postProcessing( Operation op ) {
        //post processing online logic
    }
}
```

```
public class Banking {

    public void processOperation( Operation op ) {
        process( op );
    }

    public void processOperation( Operation op,
                                  Consumer<Operation> preProcessing,
                                  Consumer<Operation> postProcessing ) {
        preProcessing.accept( op );
        process( op );
        postProcessing.accept( op );
    }

    private void process( Operation op ) {
        //logic
        op.process( op );
    }
}
```

Execute Around

```
public static void main( String[] args ) throws IOException {

    BufferedReader br = new BufferedReader( new FileReader( "dora.txt" ) );
    try {
        br.readLine();
    }  finally {
        br.close();
    }
}
```


Init / Código de preparação
Task
Cleanup/finalização

```
public static void main( String[] args ) throws IOException {

    BufferedReader br = new BufferedReader( new FileReader( "dora.txt" ) );
    try {
        br.readLine();
    }  finally {
        br.close();
    }
}
```

```
try ( BufferedReader br =
                new BufferedReader( new FileReader( "dora.txt" ) ) ) {

        br.readLine();
}
```

```
@Override
public ServerTemplate store( final ServerTemplate serverTemplate, final
                             final List<ServerTemplateKey> keys) {
    final Path path = buildPath( serverTemplate.getId() );
        try {
            ioService.startBatch(path.getFileSystem());
            ioService.write(path, serverTemplate);
            ioService.write(path, keys);
        } finally {
            ioService.endBatch();
        }
        return serverTemplate;
}
```

```
public void store( final ServerTemplate serverTemplate,
                   final List<ServerTemplateKeys> keys ) {

    try {
        ioService.startBatch( path.getFileSystem() );
        ioService.write( path, serverTemplate );
        ioService.write( path, keys );
    } finally {
        ioService.endBatch();
    }

}
```

```
public class IOService {
      …
    public void processInBatch( Path path, Consumer<Path> batchOp ) {
        try {
            startBatch( path.getFileSystem() );
            batchOp.accept( path );
        } finally {
            endBatch();
        }
    }
}
```

```
public void store( final ServerTemplate serverTemplate,
                   final List<ServerTemplateKeys> keys ) {

    try {
        ioService.startBatch( path.getFileSystem() );
        ioService.write( path, serverTemplate );
        ioService.write( path, keys );
    } finally {
        ioService.endBatch();
    }

}
```

```
public void store( final ServerTemplate serverTemplate,
                   final List<ServerTemplateKeys> keys ) {

    ioService.processInBatch( path, ( path ) -> {
        ioService.write( path, serverTemplate );
        ioService.write( path, keys );
    } );
}
```

```
public void delete( final ServerTemplate serverTemplate,
                    final List<ServerTemplateKeys> keys ) {

    ioService.processInBatch( path, ( path ) -> {
        ioService.delete( path, serverTemplate );
        ioService.delete( path, keys );
    } );
}
```

Chain of Responsibilities

“Evitar o acoplamento do remetente de uma solicitação ao seu receptor, ao dar a mais de um objeto a oportunidade de tratar a solicitação. Encadear os objetos receptores, passando a solicitação ao longo da cadeia até que um objeto a trate.”
GAMMA, Erich et al.

Chain of Responsibilities


Client
Payment
Payment Processor 1
Payment
Payment Processor 2
...
Payment Processor n

```
public static void main( String[] args ) {

    PaymentProcessor paymentProcessor = getPaymentProcessor();

    paymentProcessor.process( new Payment( 10 ) );

}

private static PaymentProcessor getPaymentProcessor() {
    PaymentProcessor g = new PaymentProcessorA();

    g.setNext( new PaymentProcessorB() );
    g.setNext( new PaymentProcessorC() );

    return g;
}
```

```
public abstract class PaymentProcessor {

    private PaymentProcessor next;

    public void setNext( PaymentProcessor processors ) {
        if ( next == null ) {
            next = processors;
        } else {
            next.setNext( processors );
        }
    }

    public Payment process( Payment p ) {
        handle( p );
        if ( next != null ) {
            return next.process( p );
        } else {
            return p;
        }
    }

    protected abstract void handle( Payment p );

}
```

```
public class PaymentProcessorA extends PaymentProcessor {

    @Override
    protected void handle( Payment p ) {
        System.out.println(
            "PaymentProcessorA for payment: " +    p.getAmount() );
    }
}


public class PaymentProcessorB extends PaymentProcessor {

    @Override
    protected void handle( Payment p ) {
        System.out.println(
            "PaymentProcessorB for payment: " +    p.getAmount() );
    }
}
```

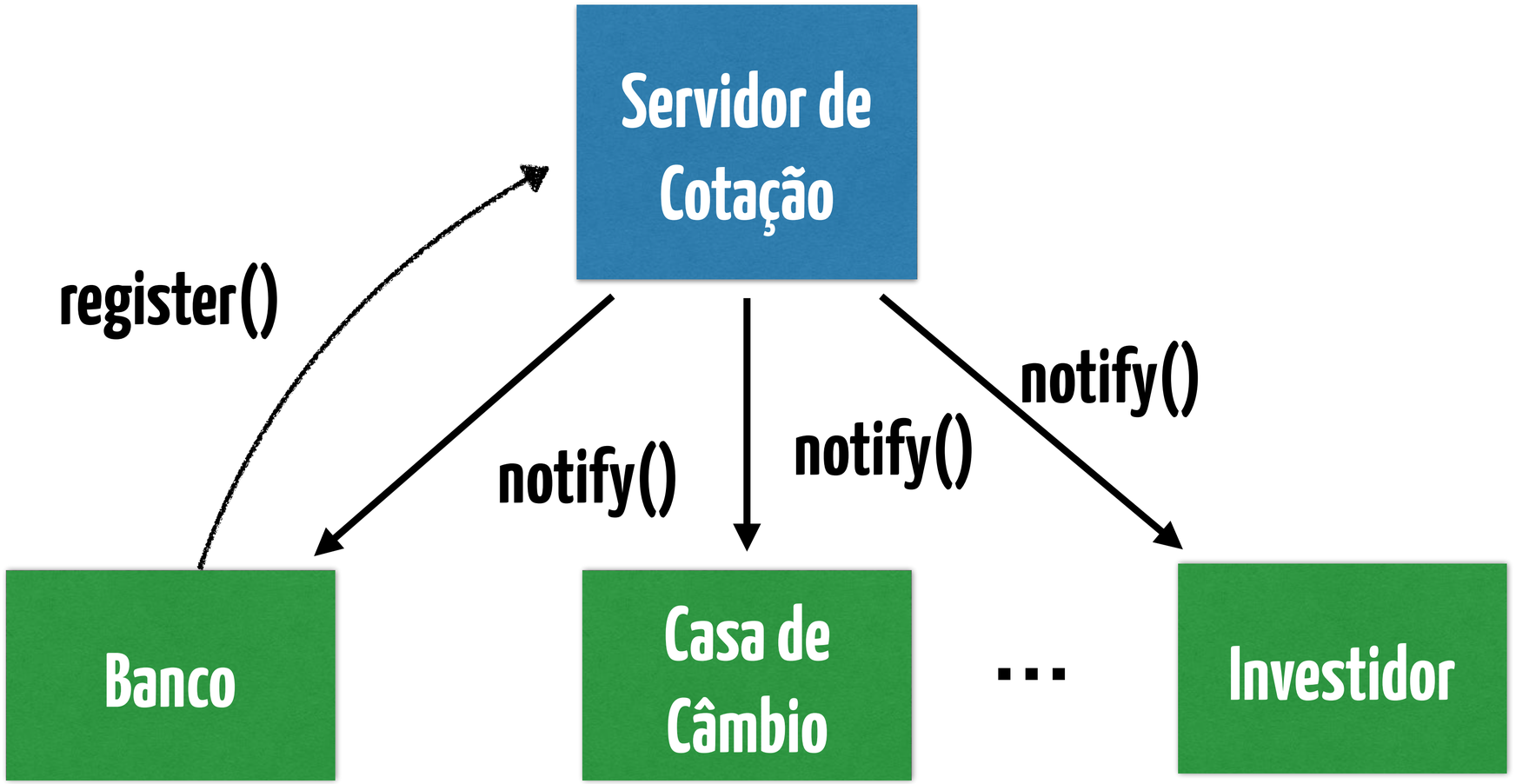
```
public static void main( String[] args ) {

    PaymentProcessor paymentProcessor = getPaymentProcessor();

    paymentProcessor.process( new Payment( 10 ) );
   //PaymentProcessorA for payment: 10
  //PaymentProcessorB for payment: 10
   //PaymentProcessorC for payment: 10
}

private static PaymentProcessor getPaymentProcessor() {
    PaymentProcessor g = new PaymentProcessorA();

    g.setNext( new PaymentProcessorB() );
    g.setNext( new PaymentProcessorC() );

    return g;
}
```

```
Function<Payment, Payment> processorA =
            p -> {
                System.out.println( "Processor A " + p.getAmount() );
                return p;
            };

  Function<Payment, Payment> processorB =
            p -> {
                System.out.println( "Processor B " + p.getAmount() );
                return p;
            };

  Function<Payment, Payment> processorC =
            p -> {
                System.out.println( "Processor C " + p.getAmount() );
                return p;
            };
```

```
  Function<Payment, Payment> processorA =
        p -> {
            System.out.println( "Processor A " + p.getAmount() );
            return p;
        };

Function<Payment, Payment> processorB =
        p -> {
            System.out.println( "Processor B " + p.getAmount() );
            return p;
        };

Function<Payment, Payment> processorC =
        p -> {
            System.out.println( "Processor C " + p.getAmount() );
            return p;
        };

  Function<Payment, Payment> chain =
          processorA.andThen( processorB ).andThen( processorC );

  chain.apply( new Payment( 10 ) );
//Processor A 10
//Processor B 10
//Processor C 10
```

Observer

"Define uma dependência um-para-muitos entre objetos de modo que quando um objeto muda o estado, todos seus dependentes são notificados e atualizados automaticamente. Permite que objetos interessados sejam avisados da mudança de estado ou outros eventos ocorrendo num outro objeto."
GAMMA, Erich et al.


Servidor de Cotação
register()
notify()
notify()
notify()
Banco
Casa de Câmbio
...
Investidor

```
public interface Subject {
    void registerObserver( Observer observer );
}


 public interface Observer {
     void notify( Cotacao lance );
 }
```

```java
public class Banco implements Observer {

    @Override
    public void notify( Cotacao cotacao ) {
        //some cool stuff here
        System.out.println( "Banco: " + cotacao );
    }

}


public class Investidor implements Observer {

    @Override
    public void notify( Cotacao cotacao ) {
        //some cool stuff here
        System.out.println( "Investidor: " + cotacao );
    }

}
```

```java
public class ServidorCotacao implements Subject {

    private List<Observer> observers = new ArrayList<>();

    public void novaCotacao( Cotacao cotacao ) {
        notifyObservers( cotacao );
    }

    @Override
    public void registerObserver( Observer observer ) {
        observers.add( observer );
    }

    private void notifyObservers( Cotacao lanceAtual ) {
        observers.forEach( o -> o.notify( lanceAtual ) );
    }

}
```

```
public class Main {

    public static void main( String[] args ) {

        Banco banco = new Banco();
        Investidor investidor = new Investidor();

        ServidorCotacao servidorCotacao = new ServidorCotacao();


        servidorCotacao.registerObserver( banco );
        servidorCotacao.registerObserver( investidor );

        servidorCotacao.novaCotacao( new Cotacao( "USD", 4 ) );

    }
}
```

```
Banco:  Cotacao{moeda='USD', valor=4}
Investidor:  Cotacao{moeda='USD', valor=4}
```

```
    @Override
    public void registerObserver( Observer observer ) {
        observers.add( observer );
    }
```

```
public class Banco implements Observer {

    @Override
    public void notify( Cotacao cotacao ) {
        //some cool stuff here
        System.out.println( "Banco: " + cotacao );
    }

}
```

```java
public class Main {

    public static void main( String[] args ) {


        ServidorCotacao servidorCotacao = new ServidorCotacao();


        servidorCotacao.registerObserver(
                cotacao -> System.out.println( "Banco:  " + cotacao ) );

        servidorCotacao.registerObserver(
                cotacao -> {
                    //some cool stuff here
                    System.out.println( "Investidor:  " + cotacao )
                } );

        servidorCotacao.novaCotacao( new Cotacao( "BRL", 1 ) );
    }
}
```

```
Banco:  Cotacao{moeda='BRL', valor=1}
Investidor:  Cotacao{moeda='BRL', valor=1}
```

Design Patterns +
Java +
Programação Funcional
<3

Obrigado

Eder Ignatowicz
@ederign

redhat.