O catálogo de refatorações é composto de 72 operações agrupadas em 5 categorias:

- Composição de métodos
- Movendo responsabilidades entre métodos
- Organizando dados
- Simplificando expressões condicionais
- Transformando a chamada de métodos mais simples
- Lidando com generalização
- Grandes refatorações

Formato das refatorações:

- Nome: utilizados como vocabulário do catálogo de refatorações.
- Resumo: uma breve descrição da situação na qual a refatoração é necessária e seu resultado.
- Motivação: descreve por que a refatoração deve ser realizada e descreve situações em que ela não deve ser realizada.
- Mecânica: uma descrição passo a passo de como a conduzir a refatoração em seu projeto.
- Exemplos: uma breve descrição para ilustrar como a refatoração ocorre.

Composição de métodos

Grande parte das refatorações ocorrem para resolver problemas de codificação inadequada de métodos:

- Métodos longos demais,
- Acoplamento indevido entre métodos e outros objetos, métodos e variáveis temporárias,
- Etc...
-

Extrair método: transforma um pedaço do método em um novo método.

Introduzir método: o oposto de extrair método. Você substitui uma chamada de método pelo corpo do método.

Substituir temporário por query: variáveis temporárias são problemáticas, principalmente quando usa-se a Refatoração *Extrair método*. Nestes casos substitua o temporário por uma consulta.

Dividir variável temporária: quando um temporário é utilizado de diversas maneiras, divide uma temporário em vários para ficar mais fácil de trocá-lo.

Trocar método por objeto método: quando variáveis temporárias estão tão emaranhadas com o código, substitua o método por um objeto método.

Remover atribuições a parâmetros: parâmetros são menos problemáticos do que temporários, desde que não haja atribuição a eles. Caso ocorra, remova-as com esta refatoração.

Substituir algoritmo: para introduzir um algoritmo mais fácil e mais claro de entender.

Refatorações do grupo Composição de Métodos:

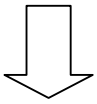
Extrair método

(Automatizado no Eclipse)

Situação: Você tem um fragmento de código que pode ser agrupado em um método.

Solução: Transforme o fragmento em um método cujo nome explica o propósito do método.

```
void printOwing() {
               printBanner();

               //print details
               System.out.println ("name:     " + _name);
               System.out.println ("amount    " + getOutstanding());
       }
```



```
void printOwing() {
               printBanner();
               printDetails(getOutstanding());
       }

       void printDetails (double outstanding) {
               System.out.println ("name:     " + _name);
               System.out.println ("amount    " + outstanding);
       }
```

Motivação: esta é uma refatoração amplamente aplicada. Métodos que são longos demais ou necessitam de comentários para explicar, são sujeitos a essa refatoração.

Transformar um fragmento de um método em outro método pode a) aumentar a chance de reutilização daquele método e b) aumentar o nível de entendimento do código, à medida em que a nomenclatura se torna adequada.

Mecânica da refatoração:

1) Criar um novo método e nomeá-lo com a intenção do método (nomeie-o pelo que ele faz, e não como ele faz).
2) Copie o código extraído do método de origem para o novo método.
3) Procure no método extraído por referências para variáveis locais no escopo do método de origem. Elas serão variáveis locais e parâmetros para o método.

4) Verifique se as variáveis temporárias são usadas apenas dentro do código extraído. Se sim, declare-as no novo método como variáveis temporárias.
5) Procure no código extraído variáveis locais que são modificadas no código. Se uma variável local é modificada, verifique se você pode tratar o código como uma query e atribuir o resultado à variável em questão.
    a. Se isto for estranho ou há mais de uma utilização da variável temporária, você não poderá extrair o método como ele está. Deve-se usar refatoração Dividir variável temporária.
6) Passe para o método-alvo como parâmetros, variáveis de escopo local que são lidas do código extraído.
7) Compile quando você tratou todas as variáveis de escopo local.
8) Substitua o código extraído no método fonte por uma chamada ao método alvo.
    a. Se você moveu quaisquer variáveis temporárias para o método novo, verifique se elas foram declaradas fora do método novo. Se sim, você pode remover a declaração.
9) Compile e teste.

Exemplo 1) Sem variáveis locais

Antes da refatoração

```
void printOwing() {

    Enumeration e = _orders.elements();
    double outstanding = 0.0;

    // print banner
    System.out.println ("**************************");
    System.out.println ("***** Customer Owes ******");
    System.out.println ("**************************");

    // calculate outstanding
    while (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        outstanding += each.getAmount();
    }

    //print details
    System.out.println ("name:" + _name);
    System.out.println ("amount" + outstanding);
}
```

Depois da refatoração

```
void printOwing() {

    Enumeration e = _orders.elements();
    double outstanding = 0.0;

    printBanner();

    // calculate outstanding
    while (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
```

```
            outstanding += each.getAmount();
        }

        //print details
        System.out.println ("name:" + _name);
        System.out.println ("amount" + outstanding);
    }

    void printBanner() {
        // print banner
        System.out.println ("**************************");
        System.out.println ("***** Customer Owes ******");
        System.out.println ("**************************");
    }
```

Exemplo 2) usando variáveis locais

Parâmetros passados para o método e variáveis temporárias são problemáticas nessa refatoração.

Caso mais fácil, quando uma variável é apenas lida no método. Solução: passe-a como parâmetro.

```
    void printOwing() {

        Enumeration e = _orders.elements();
        double outstanding = 0.0;

        printBanner();

        // calculate outstanding
        while (e.hasMoreElements()) {
            Order each = (Order) e.nextElement();
            outstanding += each.getAmount();
        }

        //print details
        System.out.println ("name:" + _name);
        System.out.println ("amount" + outstanding);
    }
```

Extração do método de impressão de detalhes, com um parâmetro

```
    void printOwing() {

        Enumeration e = _orders.elements();
        double outstanding = 0.0;

        printBanner();

        // calculate outstanding
        while (e.hasMoreElements()) {
            Order each = (Order) e.nextElement();
            outstanding += each.getAmount();
        }

        printDetails(outstanding);
    }
```

```
void printDetails (double outstanding) {
    System.out.println ("name:" + _name);
    System.out.println ("amount" + outstanding);
}
```

Exemplo 3) atribuindo valor a uma variável local.

Atribuição de valores a variáveis locais é problemático.

Se estiver atribuindo valor a um parâmetro do método, utilize a refatoração *Remover atribuições a parâmetros*.

Para variáveis temporárias podem ocorrer duas situações:

- Se a variável temporária é utilizada apenas no código extraído, mova-a para o código extraído.
- Se a variável temporária é utilizada depois do código extraído (e fora dele), o método extraído deve retornar o valor alterado para aquela variável.

```
void printOwing() {

    Enumeration e = _orders.elements();
    double outstanding = 0.0;

    printBanner();

    // calculate outstanding
    while (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        outstanding += each.getAmount();
    }

    printDetails(outstanding);
}
```

Enumeração é usada apenas dentro da iteração, portanto pode ser levado para dentro do método extraído.

Outstanding é utilizada depois de ser calculada, portanto o método deve retornar seu valor alterado.

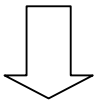
```
void printOwing() {
    printBanner();
    double outstanding = getOutstanding();
    printDetails(outstanding);
}

double getOutstanding() {
    Enumeration e = _orders.elements();
    double outstanding = 0.0;
    while (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        outstanding += each.getAmount();
    }
    return outstanding;
}
```

Introduzir método

Situação: O corpo do método é tão claro quanto seu nome.

Solução: Introduza o corpo do método no corpo daqueles que o chama e remova o método.

```
int getRating() {
      return (moreThanFiveLateDeliveries()) ? 2 : 1;
  }
  boolean moreThanFiveLateDeliveries() {
      return _numberOfLateDeliveries > 5;
  }
```



```
  int getRating() {
      return (_numberOfLateDeliveries > 5) ? 2 : 1;
  }
```

Motivação: Apesar de métodos com nomes claros serem bons, às vezes seu corpo já é claro o suficiente e não é necessário torná-lo um método. Portanto você pode excluir um nível de indireção ao excluir este método.

Outra situação possível é quando um grupo de métodos parecem estar mal refatorados. Agrupe-os todos em um método usando esta operação e posteriormente extraia novamente os métodos.

Utilizar Trocar método por objeto-método antes de extrair os métodos parece ser uma boa alternativa. Você move todo o método com o comportamento desejado para um único objeto, ao invés de mover o método e todos os seus métodos chamados.

Outra situação sugerida é quando muita indireção está sendo utilizada, em que métodos realizam uma simples delegação para outros métodos. Aplicar *Introduzir método* auxilia e definir quais indireções são de fato necessárias e eliminar as demais.

Mecânica:

- Verifique se o método não é polimórfico
  (!) Não introduza se subclasses sobrescrevem o método; elas não poderão subscrever o método se o método não está lá.
- Encontre todas as chamadas para o método.
- Substitua cada chamada com o corpo do método
- Compile e teste.
- Remova a definição do método.

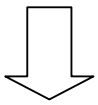
Problemas em sua utilização: em casos de recursão, vários pontos de retorno, introduzir o método em outros objetos os quais você não tem acesso. Se encontrar essas situações, não deve ser aplicado.

Introduzir temporário

Situação: Quando uma variável temporária recebe um valor uma única vez com uma expressão, e a variável temporária atrapalha outras operações de refatoração.

Solução: Trocar todas referências para aquela variável temporária pela expressão.

```
double basePrice = anOrder.basePrice();
return (basePrice > 1000)
```



```
return (anOrder.basePrice() > 1000)
```

Motivação: Deve ser utilizado como parte da refatoração *Trocar temporário por Query*.

Deve ser utilizado também em situações em que uma variável temporária recebe o valor de retorno de uma chamada de método.

Sempre que a variável temporária estiver atrapalhando outras operações de refatoração, tal como *Introduzir método*.

Mecânica:

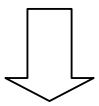
- Declare a variável temporária como **final** (se ela já não for) e compile.
  (!) Isto verifica se um valor é atribuído a variável temporária apenas uma vez.
- Procure todas referências para a variável temporária e substitua-as pela expressão de atribuição.
- Compile e teste depois de cada mudança.
- Remova a declaração e atribuição da variável temporária.
- Compile e teste.

Trocar variável temporária por consulta (Query)

Situação: Quando se utiliza uma variável temporária para armazenar o resultado de uma expressão.

Solução: Extraia a expressão para um método. Troque todas as referências para a variável temporária pelo novo método. O novo método pode então ser utilizado em outros métodos.

```
double basePrice = _quantity * _itemPrice;
if (basePrice > 1000)
    return basePrice * 0.95;
else
    return basePrice * 0.98;
```



```
     if (basePrice() > 1000)
         return basePrice() * 0.95;
     else
         return basePrice() * 0.98;
...
   double basePrice() {
       return _quantity * _itemPrice;
   }
```

Motivação: O problema com variáveis temporárias é que elas são locais, só podem ser vistas dentro do contexto do método no qual são utilizadas. Trocar a variável temporária por um método de consulta permite que qualquer método da classe obtenha a informação, o que auxilia a obter um código mais limpo para a classe.

Substituir uma variável temporária geralmente é fundamental antes de aplicar *Extrair Método*. Variáveis locais tornam a extração difícil, portanto troque quantas variáveis forem necessárias por consultas.

Mecânica:

- Procure por uma variável temporária que recebe valor uma única vez.
  (!) Se a variável temporária receber valores mais de uma vez, considere a operação de refatoração Dividir Variável Temporária.
- Declare a variável temporária como final.
- Compile.
  (!) Isso vai garantir que a variável temporária recebe valor apenas uma vez.
- Extraia a expressão de atribuição para um método.
  (!) Inicialmente defina o método como privado. Se necessário, posteriormente flexibilize o nível de acesso.
  (!) Certifique-se que método extraído está livre de efeitos colaterais, ou seja, não modifica nenhum outro objeto. Se ele não está livre de efeitos colaterais, use Separar Consulta de Modificador.
- Compile e teste.
- Aplique Introduzir Temporário na variável temporária.

Temporários geralmente são utilizados para armazenar informações em loops. Todo o loop pode ser extraído para um método. Caso haja mais de uma variável temporária, duplique o código do loop de modo que possa substituir cada temporário por uma consulta. Como o loop tende a ser muito simples, não há problemas em duplicação de código.

Exemplos:

Aplicando refatoração em dois temporários, uma de cada vez.

```
    double getPrice() {
        int basePrice = _quantity * _itemPrice;
        double discountFactor;
        if (basePrice > 1000) discountFactor = 0.95;
        else discountFactor = 0.98;
        return basePrice * discountFactor;
```

```
}
```

Adicionando o modificador **final** às variáveis temporárias e compilando para certificar que não há outro lugar que escreva nessas variáveis.

```
double getPrice() {
    final int basePrice = _quantity * _itemPrice;
    final double discountFactor;
    if (basePrice > 1000) discountFactor = 0.95;
    else discountFactor = 0.98;
    return basePrice * discountFactor;
}
```

Não havendo problemas, extraia a expressão de atribuição para um método (uma variável por vez).

```
double getPrice() {
    final int basePrice = basePrice();
    final double discountFactor;
    if (basePrice > 1000) discountFactor = 0.95;
    else discountFactor = 0.98;
    return basePrice * discountFactor;
}

private int basePrice() {
return _quantity * _itemPrice;
}
```

Substitui-se então a utilização da variável temporária pela consulta.

```
double getPrice() {
    final double discountFactor;
    if (basePrice() > 1000) discountFactor = 0.95;
    else discountFactor = 0.98;
    return basePrice() * discountFactor;
}
```

Aplicando a mesma refatoração na outra variável temporária:

```
double getPrice() {
    final double discountFactor = discountFactor();
    return basePrice() * discountFactor;
}

private double discountFactor() {
    if (basePrice() > 1000) return 0.95;
    else return 0.98;
}
```

Substituindo a temporária por uma consulta leva-nos a eliminar a variável temporária no método getPrice():

```
double getPrice() {
```

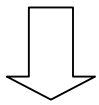
```
        return basePrice() * discountFactor();
    }
```

Introduzir variável explicativa

Situação: Você possui uma expressão complicada.

Solução: Coloque o resultado da expressão (ou partes da expressão) em uma variável temporária com um nome que explique o propósito.

```
    if ( (platform.toUpperCase().indexOf("MAC") > -1) &&
         (browser.toUpperCase().indexOf("IE") > -1) &&
         wasInitialized() && resize > 0 ) {
        // do something
    }
```



```
    final boolean isMacOs     = platform.toUpperCase().indexOf("MAC")
> -1;
    final boolean isIEBrowser = browser.toUpperCase().indexOf("IE")  >
-1;
    final boolean wasResized  = resize > 0;

    if (isMacOs && isIEBrowser && wasInitialized() && wasResized) {
        // do something
    }
```

Motivação:

Expressãoes podem se tornar muito complexas e difíceis de entender. Nesses casos variáveis podem auxiliar a quebrar as expressões em partes menores e mais fáceis de entender.

Particularmente interessante com lógica condicional na qual é útil representar e explicar cada cláusula com uma variável temporária bem nomeada.

Algoritmos longos também sujeitos a essa refatoração, em que cada passo do cálculo pode ser explicado por uma variável temporária.

Mecânica:

- Declare uma variável temporária com o modificador **final** e atribua a ela o resultado de uma parte da expressão complexa.
- Substitua parte da expressão pelo valor da variável temporária.
  (!) Se a parte a ser substituída na expressão é repetida, você pode trocar as repetições uma de cada vez.
- Compile e teste.
- Repita para outras partes da expressão.

Exemplos:

Um método para cálculo de preços.

```
    double price() {
    // price is base price - quantity discount + shipping
    return _quantity * _itemPrice -
        Math.max(0, _quantity - 500) * _itemPrice * 0.05 +
        Math.min(_quantity * _itemPrice * 0.1, 100.0);
    }
```

Preço básico = quantidade * preço unitário.

```
    double price() {
        // price is base price - quantity discount + shipping
        final double basePrice = _quantity * _itemPrice;
        return basePrice -
            Math.max(0, _quantity - 500) * _itemPrice * 0.05 +
            Math.min(_quantity * _itemPrice * 0.1, 100.0);
     }
```

Quantidade * preço unitário é usado outras vezes, portanto pode-se substituir a expressão pela temporária.

```
    double price() {
        // price is base price - quantity discount + shipping
        final double basePrice = _quantity * _itemPrice;
        return basePrice -
            Math.max(0, _quantity - 500) * _itemPrice * 0.05 +
            Math.min(basePrice * 0.1, 100.0);
    }
```

Refatorando a expressão de desconto:

```
   double price() {
      // price is base price - quantity discount + shipping
      final double basePrice = _quantity * _itemPrice;
      final double quantityDiscount = Math.max(0, _quantity - 500) *
_itemPrice * 0.05;
      return basePrice - quantityDiscount +
         Math.min(basePrice * 0.1, 100.0);
   }
```

Refatorando a expressão de frete:

```
   double price() {
      final double basePrice = _quantity * _itemPrice;
      final double quantityDiscount = Math.max(0, _quantity - 500) *
_itemPrice * 0.05;
      final double shipping = Math.min(basePrice * 0.1, 100.0);
      return basePrice - quantityDiscount + shipping;
   }
```

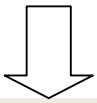
(!) Aplicar a refatoração *Extrair Método* neste exemplo levaria a um projeto similar, mas mais modulado. Aplique *Extrair Método* em cada parte da expressão.

Dividir variável temporária

Situação: Você possui uma variável temporária que recebe valor mais de uma vez mas não é um loop ou uma variável acumulativa (somatório, produtório, contador, etc...).

Solução: Crie uma variável temporária para cada atribuição.

```
double temp = 2 * (_height + _width);
System.out.println (temp);
temp = _height * _width;
System.out.println (temp);
```



```
final double perimeter = 2 * (_height + _width);
System.out.println (perimeter);
final double area = _height * _width;
System.out.println (area);
```

Motivação:

Temporários são usados para armazenar o resultado de um pedaço de código para uso posterior. Esse tipo de variável deve ter seu valor alterado apenas uma vez. Se a variável receber valor mais de uma vez é um sinal de que ela possui mais de uma responsabilidade dentro do método e deve ser então substituída por uma variável temporária para cada responsabilidade. Usar uma variável temporária para mais de uma responsabilidade diferente é muito confuso.

Mecânica:

- Mude o nome de uma temporária em sua declaração e em sua primeira atribuição.

  (!) Se as atribuições posteriores forem da forma i = i + some expression, é sinal que é uma variável acumuladora e por isso não deve ser dividida. O operador para uma temporária acumuladora usualmente é adição, concatenação de string, escrevendo em um arquivo ou adicionando algo para uma coleção.

- Declare a nova temporária com o modificador **final**.
- Mude todas as referências para a temporária à partir da segunda atribuição.
- Declare a temporária em sua segunda atribuição.
- Compile e teste.
- Repita em passos, em que cada passo deve-se renomear a declaração e mudar as referências até a próxima atribuição.

Exemplos:

```
  double getDistanceTravelled (int time) {
      double result;
      double acc = _primaryForce / _mass;
      int primaryTime = Math.min(time, _delay);
      result = 0.5 * acc * primaryTime * primaryTime;
      int secondaryTime = time - _delay;
      if (secondaryTime > 0) {
          double primaryVel = acc * _delay;
          acc = (_primaryForce + _secondaryForce) / _mass;
          result +=  primaryVel * secondaryTime + 0.5 * acc *
secondaryTime * secondaryTime;
      }
      return result;
  }
```

Mudando o nome da primeira variável e sua utilização até a segunda atribuição. Compile e teste.

```
    double getDistanceTravelled (int time) {
        double result;
        final   double primaryAcc  = _primaryForce / _mass;
        int primaryTime = Math.min(time, _delay);
        result = 0.5 * primaryAcc * primaryTime * primaryTime;
        int secondaryTime = time - _delay;
        if (secondaryTime > 0) {
            double primaryVel = primaryAcc * _delay;
            double acc = (_primaryForce + _secondaryForce) / _mass;
            result +=  primaryVel * secondaryTime + 0.5 * acc *
secondaryTime * secondaryTime;
        }
        return result;
    }
```

Trocando o nome da segunda utilização da variável temporária.

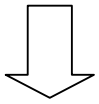
```
   double getDistanceTravelled (int time) {
       double result;
       final double primaryAcc = _primaryForce / _mass;
       int primaryTime = Math.min(time, _delay);
       result = 0.5 * primaryAcc * primaryTime * primaryTime;
       int secondaryTime = time - _delay;
       if (secondaryTime > 0) {
           double primaryVel = primaryAcc * _delay;
           final double secondaryAcc = (_primaryForce +
_secondaryForce) / _mass;
           result +=  primaryVel * secondaryTime + 0.5 *
                      secondaryAcc * secondaryTime * secondaryTime;
       }
       return result;
   }
```

Remover atribuições a parâmetros:

Situação: O código atribui valor a um parâmetro que foi passado.

Solução: Use uma variável temporária ao invés do parâmetro.

```
int discount (int inputVal, int quantity, int yearToDate) {
    if (inputVal > 50) inputVal -= 2;
```



```
int discount (int inputVal, int quantity, int yearToDate) {
    int result = inputVal;
    if (inputVal > 50) result -= 2;
```

Motivação:

O problema aparece devido à passagem por valor e à passagem por referência do parâmetro.

Com a passagem por valor qualquer alteração ao parâmetro não é refletida no método chamado. Para quem utilizar passagem por referência já não ocorre o mesmo, o que pode gerar confusões.

Outro motivo é a utilização consistente de parâmetros. O código fica muito mais claro se parâmetros não forem alterados, mas apenas representarem o que foi passado a eles.

Mecânica:

- Crie uma variável temporária para o parâmetro.
- Substitua todas as referências para o parâmetro pela variável temporária, encontradas depois da expressão de atribuição.
- Mude a atribuição para atribuir à variável temporária.
- Compile e teste.

  ? Se as chamadas são por referência, verifique no método chamado se há outras utilizações para o parâmetro. Veja também quantas chamadas cujos parâmetros por referência recebem valores e usados posteriormente neste método. Tente passar um único valor como retorno. Se há há mais de um valor de retorno, verifique se você pode agrupar os dados de retorno em um objeto ou crie métodos separados.

Exemplos:

```
int discount (int inputVal, int quantity, int yearToDate) {
    if (inputVal > 50) inputVal -= 2;
    if (quantity > 100) inputVal -= 1;
    if (yearToDate > 10000) inputVal -= 4;
    return inputVal;
}
```

Substituindo por uma variável temporária:

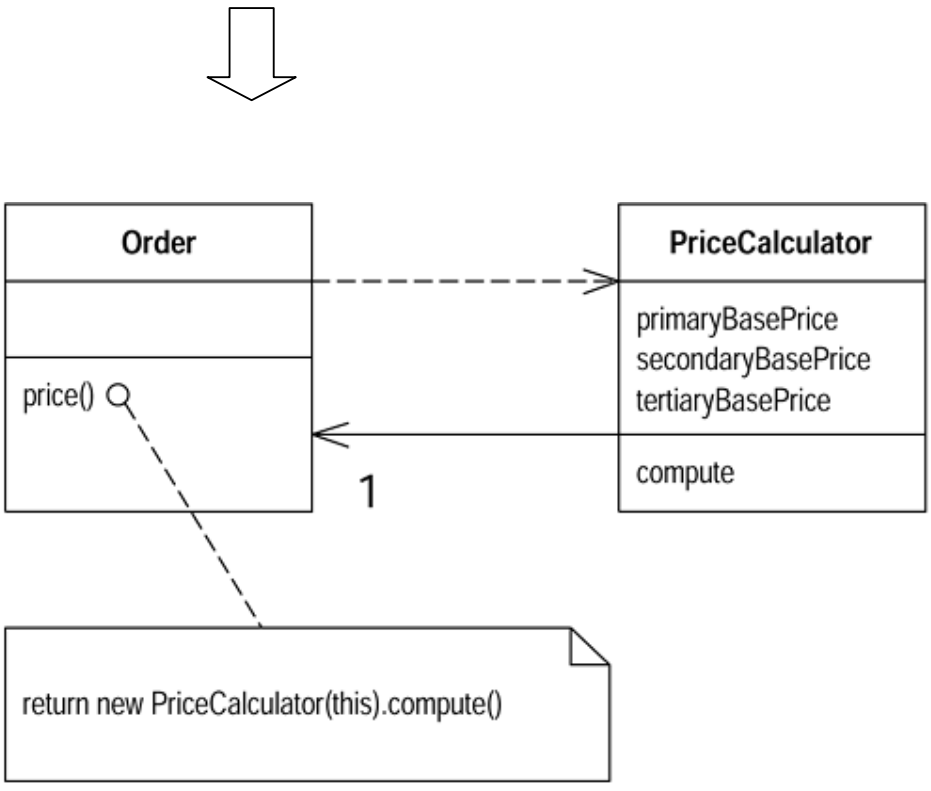
```
    int discount (final int inputVal, final int quantity, final int
yearToDate) {
        int result = inputVal;
        if (inputVal > 50) result -= 2;
        if (quantity > 100) result -= 1;
        if (yearToDate > 10000) result -= 4;
        return result;
    }
```

Substituir método com objeto-método:

Situação: Você possui um método longo que usa variáveis locais que não te permitem aplicar *Extrair Método.*

Solução: Transforme o método em seu próprio objeto, em que todas as variáveis locais se tornam campos naquele objeto. Você pode então decompor o método em outros métodos para o mesmo objeto.

```
class Order...
  double price() {
  double primaryBasePrice;
      double secondaryBasePrice;
      double tertiaryBasePrice;
      // long computation;
      ...
  }
```



Motivação:

Extrair pedaços de um método longo torna as coisas muito mais compreensíveis. A dificuldade em decompor métodos está nas variáveis locais: elas não permitem que isto seja realizado tão naturalmente. Utilizar *Substituir Temporário por Consulta* auxilia a reduzir esta dificuldade, mas nem sempre é possível quebrar um método que precisa ser quebrado.

Aplicar *Substituir Método por Método-Objeto* transforma todas as variáveis temporárias em campos do novo objeto-método. Você pode então utilizar *Extrair Método* neste novo objeto para criar métodos adicionais que quebrarão o método original.

Mecânica:

- Crie uma nova classe e nomeie-a depois do método.
- Declare na nova classe uma referência para o objeto que possuía o método original (o objeto fonte) e um campo para cada variável temporária e cada parâmetro do método.
- Declare na nova classe um construtor que receba o objeto fonte e cada parâmetro.
- Declare na nova classe um método chamado “Computar”.
- Copie o corpo do método original no método “Computar. Use o objeto apontado pela referência para quaisquer chamadas realizadas no objeto original.
- Compile.
- Troque o método antigo por um que cria o novo objeto e chame o método “Computar”.

Ao final, devido ao fato de todas variáveis locais serem campos, você pode decompor o método livremente sem passar parâmetros.

Exemplos:

Um método complicado de uma classe:

```
Class Account
  int gamma (int inputVal, int quantity, int yearToDate) {
      int importantValue1 = (inputVal * quantity) + delta();
      int importantValue2 = (inputVal * yearToDate) + 100;
      if ((yearToDate - importantValue1) > 100)
          importantValue2 -= 20;
      int importantValue3 = importantValue2 * 7;
      // and so on.
      return importantValue3 - 2 * importantValue1;
  }
```

Criando uma classe para representar o método (por isso ela recebeu o mesmo nome do método). Nesta classe todas as variáveis temporárias se tornam atributos, além de uma referência para o objeto de origem (que recebe o modificador final).

```
class Gamma...
  private final Account _account;
  private int inputVal;
  private int quantity;
```

```
    private int yearToDate;
    private int importantValue1;
    private int importantValue2;
    private int importantValue3;
```

Adicionando o construtor do objeto-método.

```
    Gamma (Account source, int inputValArg, int quantityArg, int
yearToDateArg) {
        _account = source;
        inputVal = inputValArg;
        quantity = quantityArg;
        yearToDate = yearToDateArg;
    }
```

Movendo o método original para o novo método *computar()*: Note que o método chamado no antigo objeto usa agora a referência do objeto de origem, localizado na própria classe.

```
    int compute () {
        importantValue1 = (inputVal * quantity) + _account.delta();
        importantValue2 = (inputVal * yearToDate) + 100;
        if ((yearToDate - importantValue1) > 100)
            importantValue2 -= 20;
        int importantValue3 = importantValue2 * 7;
        // and so on.
        return importantValue3 - 2 * importantValue1;
    }
```

Por fim, alterar o método original para delegar a execução ao objeto-método:

```
   int compute () {
       importantValue1 = (inputVal * quantity) + _account.delta();
       importantValue2 = (inputVal * yearToDate) + 100;
       importantThing();
       int importantValue3 = importantValue2 * 7;
       // and so on.
       return importantValue3 - 2 * importantValue1;
   }

   void importantThing() {
       if ((yearToDate - importantValue1) > 100)
           importantValue2 -= 20;
   }
```

Neste ponto poderia ser aplicado *Extrair Método* para continuar a dividir o método original.

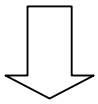
Substituir Algoritmo

Situação: Você quer substituir um algoritmo por um que seja mais claro.

Solução: Substitua o corpo do método pelo novo algoritmo.

```
    String foundPerson(String[] people){
```

```
        for (int i = 0; i < people.length; i++) {
            if (people[i].equals ("Don")){
                return "Don";
            }
            if (people[i].equals ("John")){
                return "John";
            }
            if (people[i].equals ("Kent")){
                return "Kent";
            }
        }
        return "";
    }
```



```
    String foundPerson(String[] people){
        List candidates = Arrays.asList(new String[] {"Don", "John",
"Kent"});
        for (int i=0; i<people.length; i++)
            if (candidates.contains(people[i]))
                return people[i];
        return "";
    }
```

Motivação:

Se você descobrir um jeito mais fácil de fazer algo, você deve substituir o jeito complicado pelo jeito mais fácil. Isso geralmente ocorre quando:

- À medida que você aprende mais sobre o problema e descobre que existe um jeito mais fácil de realizá-lo,
- Quando você utiliza bibliotecas que provê características que duplicam o seu código.

Essa refatoração deve ser realizada com método de granularidade mais fina (bem decompostos). Substituir um método grande, com algoritmo complexo é muito mais difícil.

Mecânica:

- Prepare seu algoritmo alternativo e faça-o compilar.
- Execute seu novo algoritmo com seus testes. Se os resultados são os mesmos, a refatoração está pronta.
- Se os resultados não são os mesmos, use o algoritmo velho para comparação de teste e debugging.

? Execute cada caso de teste com o velho e o novo algoritmo e verifique o resultado de ambos. Isto vai ajudá-lo a ver quais casos de teste estão causando problemas e como.