Movendo características entre objetos

Uma das decisões mais fundamentais, senão a mais fundamental, em projetos de objetos é decidir onde colocar as responsabilidades (de fazer e de conhecer).

Refatoração é uma forma de retirar esse compromisso das costas do projetista, pois as alterações subseqüentes no projeto podem ser realizadas de modo mais controlado e fácil.

Refatorações desse grupo ajudarão a definir o melhor local para atribuir as responsabilidades dos objetos:

- Mover método: mover comportamento entre objetos
- Mover campo: mover atributos entre objetos
- Extrair classes: quando uma classe está inchada com responsabilidades demais.
- Introduzir classes: quando uma classe está com pouquíssimas responsabilidades.
- Ocultar delegação: quando há a necessidade de esconder uma classe que está sendo utilizada.
- Remover homem do meio: quando há a necessidade de retirar uma indireção.
- Introduzir método estrangeiro: Quando não é possível acessar o código fonte de uma classe mas deseja-se mover responsabilidades para essa classe.
- Introduzir extensão local: idem à anterior.

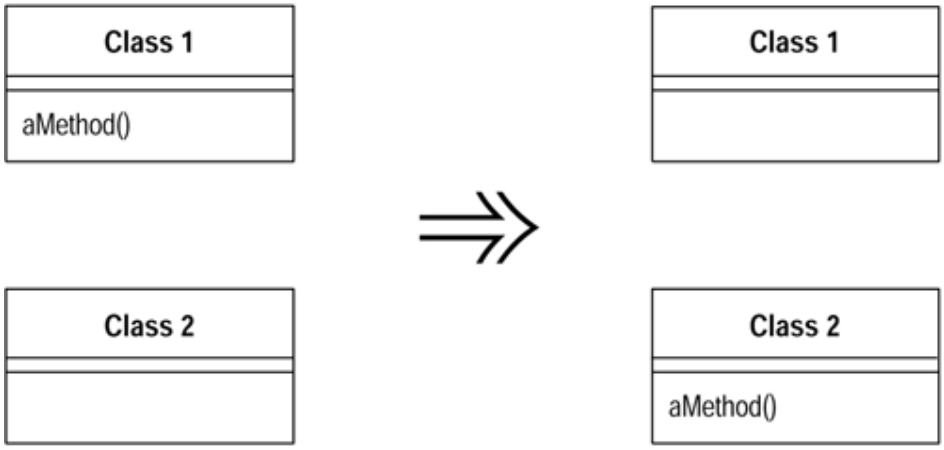
Refatorações do grupo Composição de Métodos:

Mover método

(Automatizado no Eclipse)

Situação: Um método está sendo (ou será) utilizado por diversos elementos de uma outra classe diferente daquela na qual ele é definido.

Solução: Criar um novo *método com uma implementação similar na classe que ele usa com freqüência. Transformar o método antigo em uma simples delegação ou removê-lo.*



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

Deve-se mover métodos quando classes possuem muito comportamento definido ou quando classes estão colaborando muito entre si e por isso estão altamente acopladas.

Deve-se procurar por métodos em uma classe que aparentam referenciar outro objeto mais do que o objeto em que ele se encontra. Um bom momento de realizar essa refatoração é depois de mover alguns campos.

Mecânica da refatoração:

1. Examine todos os elementos (atributos ou métodos) usados pelo método que estão definidos na classe do próprio método. Verifique se eles também devem ser movidos.

   Se um elemento é usado apenas pelo método que você deseja mover, você deve movê-lo também. Se o elemento é usado por outros métodos, verifique se eles devem ser movidos também. Geralmente é mais fácil de mover um conjunto de elementos do que movê-los um de cada vez.

2. Verifique nas sub e superclasses da classe em questão se há outras declarações (definições) para o método.

   Se houver quaisquer outras declarações, você não deve realizar a movimentação, há menos que o polimorfismo possa ser expresso na outra classe.

3. Declare o método na classe que receberá o método.

   Você pode escolher um outro nome na nova classe, que faça mais sentido na nova classe.

4. Copie o código original para o novo método definido na classe que o receberá. Ajuste-o para que ele funcione em sua nova casa.

Se o método utilizar elementos de sua classe original, você precisa determinar como referenciar o objeto original à partir do método movido. Se não houver mecanismo na classe alvo, passe uma referência do objeto para o novo método como um parâmetro.

Se o método possuir manipuladores de exceção, decida qual classe deverá tratar a exceção. Se a classe fonte do método deve ser responsável, deixe os manipuladores nela.

5. Compile a classe alvo (aquela que recebeu o método).
6. Determine como referenciar corretamente o objeto alvo (aquele que recebeu o método) à partir do objeto do qual ele foi extraído.

   No objeto alvo pode haver um campo ou método que armazene a referência para o objeto alvo. Caso não verifique se é possível criar facilmente um método que o fará. Caso não seja possível será necessário criar um novo campo no objeto fonte que armazene o objeto alvo. Esta referência pode se tornar permanente, até o momento em que futuras refatorações levem à sua remoção.

7. Transforme o método fonte em um método que realize uma delegação (indireção).
8. Compile e teste.
9. Decida se deve remover o método fonte ou mantê-lo como um método de delegação.

   Deixar um método no objeto fonte como um método de delegação facilita se você possuir muitas referências que o utilizam.

10. Se você remover o método no objeto fonte, você deve atualizar todas as referências de modo que elas apontem para o novo método (alvo).

    Você deve compilar e testar depois de mudar cada referência.

11. Compile e teste.

Exemplo

Imagine uma classe conta que possui um método **overdraftCharge()**. Vários tipos de conta serão definidos, de modo que cada tipo de conta tenha o seu próprio método **overdraftCharge()**. Portanto deseja-se mover o método para a classe **AccountType**.

```
class Account...
   double overdraftCharge() {
        if (_type.isPremium()) {
            double result = 10;
            if (_daysOverdrawn > 7) result += (_daysOverdrawn - 7) *
0.85;
            return result;
        }
```

```
        else return _daysOverdrawn * 1.75;
    }

    double bankCharge() {
        double result = 4.5;
        if (_daysOverdrawn > 0) result += overdraftCharge();
        return result;
    }
    private AccountType _type;
    private int _daysOverdrawn;
```

Verificar quais características (atributos e métodos) que o método utiliza e verificar se vale a pena movê-los. Neste caso, é utilizado o atributo **_daysOverdrawn**. Portanto move-se o corpo do método para a classe **AccountType**.

```
class AccountType...
    double overdraftCharge(int daysOverdrawn) {
        if (isPremium()) {
            double result = 10;
            if (daysOverdrawn > 7) result += (daysOverdrawn - 7) *
0.85;
            return result;
        }
        else return daysOverdrawn * 1.75;
  }
```

Nota: quando é necessário utilizar recursos da classe de onde o método foi extraído, pode-se fazê-lo de 4 maneiras:

1. Mover o elemento para a classe alvo também,
2. Criar ou usar uma referência na classe alvo para a classe fonte,
3. Passar o objeto fonte como parâmetro para o método,
4. Se o recurso for uma variável, passe-a como parâmetro.

Neste caso, a variável foi passada como parâmetro. Com o método em seu novo lugar e a classe sendo compilada, altera-se o método original de modo que sua implementação realize uma delegação ao novo método.

```
class Account...
    double overdraftCharge() {
        return _type.overdraftCharge(_daysOverdrawn);
    }
```

Compile e teste.

Neste ponto a refatoração está feita, mas pode-se excluir o antigo método da classe. Para isso é necessário descobrir todas as chamadas ao método antigo e redirecioná-las ao novo método.

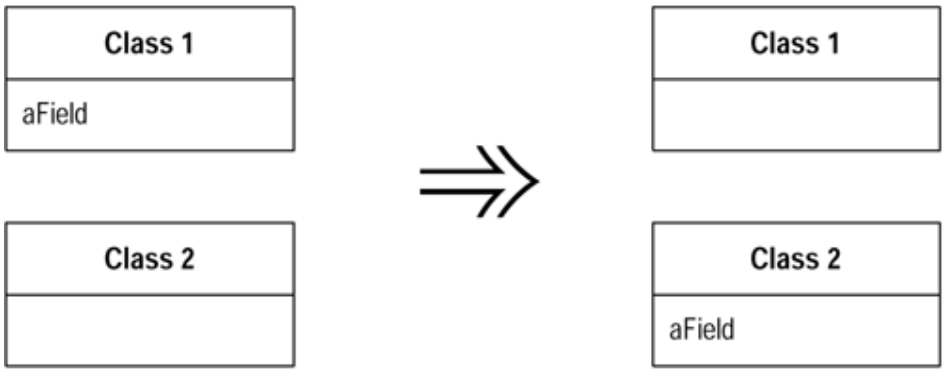
```
class Account...
    double bankCharge() {
        double result = 4.5;
        if (_daysOverdrawn > 0) result +=
_type.overdraftCharge(_daysOverdrawn);
        return result;
    }
```

A cada remoção deve-se compilar e testar, para verificar se não há erros nos projeto. Se o método for privado é uma operação relativamente simples de ser executada. Caso contrário é necessário descobrir todas as classes que usam aquele método.

Mover campo

Situação: Um campo é (ou será) mais utilizado por uma outra classe do que naquela em que ele está definido

Solução: Crie um novo campo na classe alvo e modifique todos seus utilizadores.



Motivação: Se vários métodos de uma classe utilizam um campo mais do que os métodos da própria classe em que ele está definido, mova-o para a classe que o utiliza mais. Essa utilização pode ser indireta através de métodos get e set.

Outro motivo para mover campos é quando aplica-se **Extrair Classe**. Nesta operação de refatoração inicialmente são movidos os campos e posteriormente os métodos.

Mecânica:

- Se o campo for público, use **Encapsular Campo.**

  *Se é provável que você vá mover os métodos que acessam o campo frequentemente ou se há um monte de métodos o acessam, pode ser útil usar a refatoração "**Auto-encapsular campo**".*

- Compile e teste.
- Crie um campo na classe-alvo (aquela que receberá o campo) com métodos **set** e **get**.
- Compile a classe-alvo.
- Determine como referenciar o objeto-alvo à partir de sua fonte.

  Um campo existente ou método pode lhe oferecer o alvo. *Se não, veja se você consegue facilmente criar um método que o fará para você. Não sendo possível você deverá criar um novo campo na classe-fonte que armazene o objeto-alvo.*

Esta pode ser uma mudança permanente, mas você poderá torná-la temporária desde que você refatore o suficiente para removê-la.

- Remova o campo da classe-fonte.
- Troque todas as referências para o campo da classe-fonte por referência para os métodos apropriados da classe-alvo.

  *Para acessar a variável, troque a referência por uma chamada a um método **get** do objeto-alvo; para atribuições, troque a referência por uma chamada ao método **set** correspondente.*

  Se o campo não é privado, procure em todas as subclasses da classe-fonte por referências a ele.

- Compile e teste.

Exemplo: Classe Account, deseja-se mover o campo **_interestRate** para a classe **AccountType**.

```
class Account...
   private AccountType _type;
   private double _interestRate;

   double interestForAmount_days (double amount, int days) {
       return _interestRate * amount * days / 365;
   }
```

Inicialmente cria-se o campo na classe **AccountType** como privado e define seus métodos de acesso.

```
   private double _interestRate;

   double interestForAmount_days (double amount, int days) {
       return _type.getInterestRate() * amount * days / 365;
   }


class AccountType...
   private double _interestRate;


   void setInterestRate (double arg) {
       _interestRate = arg;
   }

   double getInterestRate () {
       return _interestRate;
   }
```

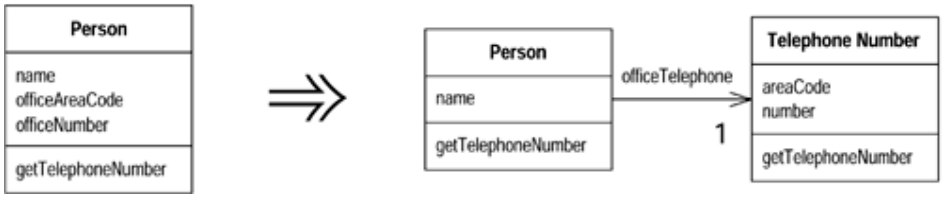
Compile a nova classe nesse ponto.

Redirecione os métodos da classe **Account** para usar o atributo em **AccountType** e remova o campo **_interestRate** da classe **Account.** (Isso vai garantir que o redirecionamento está funcionando).

Extrair Classe

Situação: Você possui uma classe fazendo trabalho que deveria ser feito por duas.

Solução: Crie uma nova classe e mova os campos e métodos relevantes da antiga classe para a nova classe.



Motivação: uma classe deve ser uma abstração pura e lidar com algumas poucas responsabilidades claras. Você adiciona uma responsabilidade a uma classe achando que não vale a pena defini-la em uma classe separada, mas a medida em que a responsabilidade cresce a classe se torna muito complicada.

Uma classe que é muito grande torna-se difícil de entender. Você deve decidir onde ela deve ser dividida e então separá-la. Um bom sinal para essa divisão é um subconjunto de atributos e um subconjunto de métodos que devem ser movidos juntos. Outro bom sinal são subconjuntos de dados que normalmente são alterados juntos ou são apresentam algum tipo de dependência entre si.

Mecânica:

- Decidir como partir as responsabilidade da classe.
- Criar uma nova classe para receber as responsabilidades.

  Se as responsabilidades da classe antiga não correspondem mais ao seu nome, defina um novo nome para a classe antiga.

- Crie uma associação da classe antiga para a nova classe.

  Pode ser necessária uma associação bidirecional. Entretanto não crie a associação Nova Classe → Classe Antiga até você descobrir que de fato você necessita dela.

- Use **Mover Campo** em cada campo que você deseja mover.

- Compile e teste depois de cada movimentação de campo.
- Use **Mover Método** para mover métodos da classe antiga para a nova classe. Comece pelos métodos de nível mais baixo (aqueles que são chamados, ao invés dos que chamam) e aos poucos vá para métodos de nível mais alto.
- Compile e teste depois de cada movimentação.
- Revise e reduza a interface de cada classe.

  Se você possui uma associação bi-direcional, verifique se ela não pode ser realizada unidirecionalmente.

- Decida se deve expor a nova classe. Se for o caso de expor a classe, decida se deve expor como uma referência para objeto ou como um objeto de valor imutável.

Exemplo:

Uma classe **Pessoa**:

```
class Person...
  public String getName() {
      return _name;
  }
  public String getTelephoneNumber() {
      return ("(" + _officeAreaCode + ") " + _officeNumber);
  }
  String getOfficeAreaCode() {
      return _officeAreaCode;
  }
  void setOfficeAreaCode(String arg) {
      _officeAreaCode = arg;
  }
  String getOfficeNumber() {
      return _officeNumber;
  }
  void setOfficeNumber(String arg) {
      _officeNumber = arg;
  }

  private String _name;
  private String _officeAreaCode;
  private String _officeNumber;
```

Para separar o comportamento relativo a número de telefone da classe Pessoa, inicialmente define-se uma classe **TelephoneNumber**.

```
class TelephoneNumber {
}
```

Agora utiliza-se **Mover Campo** em cada um dos campos.

```
class TelephoneNumber {
  String getAreaCode() {
      return _areaCode;
  }
  void setAreaCode(String arg) {
```

```
        _areaCode = arg;
    }
    private String _areaCode;
}

class Person...
    public String getTelephoneNumber() {
        return ("(" + getOfficeAreaCode() + ") " + _officeNumber);
    }
    String getOfficeAreaCode() {
        return _officeTelephone.getAreaCode();
    }
    void setOfficeAreaCode(String arg) {
        _officeTelephone.setAreaCode(arg);
    }
```

Aplicar **Mover campo** no outro campo e **Mover método** no método **getTelephoneNumber()**.

```
class Person...
    public String getName() {
        return _name;
    }
    public String getTelephoneNumber(){
        return _officeTelephone.getTelephoneNumber();
    }
    TelephoneNumber getOfficeTelephone() {
        return _officeTelephone;
    }

    private String _name;
    private TelephoneNumber _officeTelephone = new TelephoneNumber();



class TelephoneNumber...
    public String getTelephoneNumber() {
        return ("(" + _areaCode + ") " + _number);
    }
    String getAreaCode() {
        return _areaCode;
    }
    void setAreaCode(String arg) {
        _areaCode = arg;
    }
    String getNumber() {
        return _number;
    }
    void setNumber(String arg) {
        _number = arg;
    }
    private String _number;
    private String _areaCode;
```

A decisão então é quanto deve ser exposto da nova classe para os meus clientes. Pode-se ocultá-la completamente através de métodos de delegação ou pode expor os elementos da nova classe, ambos por meio de interfaces da classe. Neste caso o telefone foi exposto pelo método **getOfficeTelephone()**.

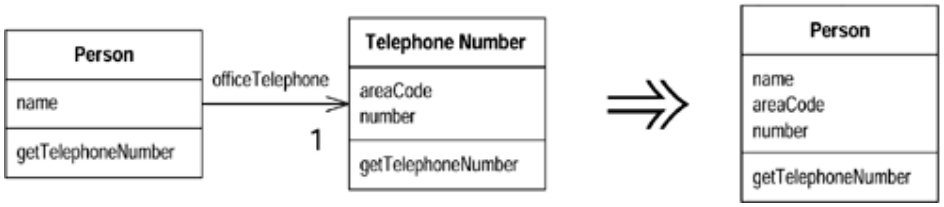
Ao expor o objeto três situações são possíveis e deve-se considerar uma delas:

1) Aceitar que qualquer objeto pode mudar o estado do objeto exposto.
2) Não aceitar nenhuma mudança no objeto exposto, há não ser pelo objeto de onde ele foi extraído.
3) Clonar o estado do objeto, antes dele ser passado aos seus clientes.

Introduzir Classe

Situação: Uma classe não está fazendo muita coisa.

Solução: Mover todas suas características para uma outra classe e excluí-la.



Motivação: Introduzir Classe é o contrário de **Extrair** Classe. Utilize **Introduzir** Classe se uma classe não justifica mais sua existência.

Geralmente ocorre como resultado de refatorações que movem outras responsabilidades da classe para fora dela, restando pouca responsabilidade em seu interior. Então deseja-se embutir o restante desta classe em uma outra classe, escolhendo aquela classe que parece utilizá-la ao máximo.

Mecânica:

- Declare o protocolo public da class fonte na classe que vai absorvê-la. Delegue todos esses métodos para a classe fonte.
  *Se faz sentido ter uma interface separada para os métodos da classe-fonte, use **Extrair Interface** antes de **Introduzir Classe**.*
- Mude todas as referências da classe-fonte para a classe que está recebendo os métodos.
  *Declare a classe-fonte como **private** para remover referências fora do pacote. Também altere o nome da class-fonte de modo que o compilador capture quaisquer referências quebradas para a classe fonte.*
- Compile e teste.
- Use **Mover Método** e **Mover Campo** para mover elementos da classe-fonte para a classe que os absorverá até que não falte nada mais.

Exemplo:

Fazendo o caminho de volta da classe **TelephoneNumber** para a classe **Pessoa**.

```
class Person...
  public String getName() {
      return _name;
  }
  public String getTelephoneNumber(){
      return _officeTelephone.getTelephoneNumber();
  }
  TelephoneNumber getOfficeTelephone() {
      return _officeTelephone;
  }

  private String _name;
  private TelephoneNumber _officeTelephone = new TelephoneNumber();


class TelephoneNumber...
  public String getTelephoneNumber() {
      return ("(" + _areaCode + ") " + _number);
  }
  String getAreaCode() {
      return _areaCode;
  }
  void setAreaCode(String arg) {
      _areaCode = arg;
  }
  String getNumber() {
      return _number;
  }
  void setNumber(String arg) {
      _number = arg;
  }
  private String _number;
  private String _areaCode;
```

Inicialmente declara-se todos os métodos visíveis da classe **TelephoneNumber** na classe **Pessoa**.

```
 class Person...
  String getAreaCode() {
      return _officeTelephone.getAreaCode();
  }
  void setAreaCode(String arg) {
      _officeTelephone.setAreaCode(arg);
  }
  String getNumber() {
      return _officeTelephone.getNumber();
  }
  void setNumber(String arg) {
      _officeTelephone.setNumber(arg);
  }
```

Procura-se por clientes de **TelephoneNumber** e atualize-os para utilizar a interface de Pessoa.O que era:

```
      Person martin = new Person();
      martin.getOfficeTelephone().setAreaCode ("781");
```

se torna

```
      Person martin = new Person();
```

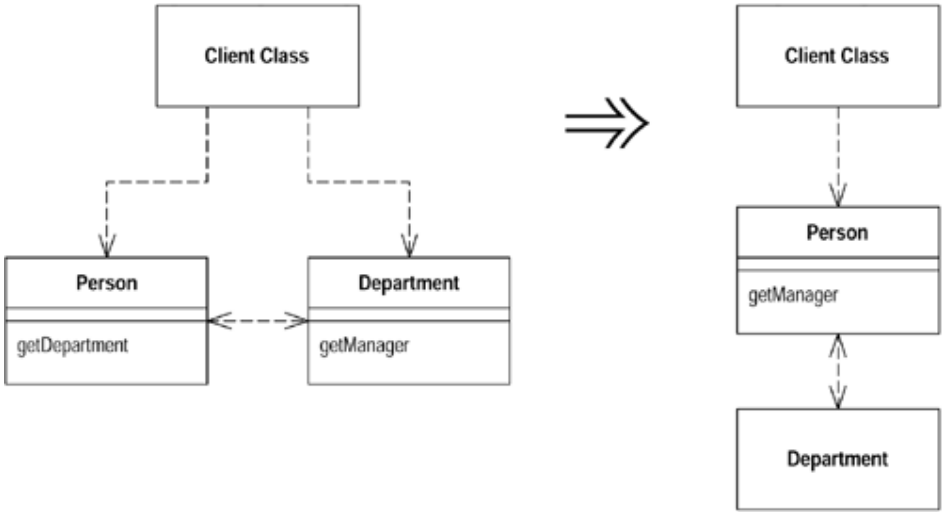
```
martin.setAreaCode ("781");
```

Aplique as refatorações **Mover método** e **Mover campo** até esvaziar a classe **telephone**.

Ocultar delegação

Situação: Um cliente realiza uma chamada a uma classe delegada de um objeto.

Solução: Crie métodos no servidor para ocultar a classe delegada.



Motivação:

Encapsulamento significa que objetos necessitam saber pouco de outras partes do sistema. Portanto quando as coisas mudam poucos objetos precisam ser notificados sobre a mudança – o que torna a mudança mais fácil de ser realizada.

Contra-exemplo: Se um cliente chama um método definido em um dos campos de um objeto "servidor", o cliente precisa saber sobre este objeto delegado. Se a delegação for alterada, o cliente também precisa ser alterado.

Pode-se remover essa dependência ao definir um método de delegação simples no objeto "servidor", o qual oculta o objeto delegado. Mudanças se tornam limitadas ao servidor e não propagam para o cliente.

Mecânica:

- Para cada método do objeto delegado, crie um método de delegação simples no objeto servidor.
- Altere o cliente para chamar o servidor.

Se o cliente não está no mesmo pacote do servidor, considere alterar o acesso considere alterar o acesso ao método do objeto delegado para visibilidade de pacote.

- Compile e teste após ajustar cada método.
- Se nenhum cliente necessita acessar o objeto delegado mais, retire do objeto-servidor o método de acesso ao objeto delegado.
- Compile e teste.

Exemplos:

Considere uma classe **Pessoa** e outra **Departamento.**

```
class Person {
  Department _department;

  public Department getDepartment() {
      return _department;
  }
  public void setDepartment(Department arg) {
      _department = arg;
  }
}


class Department {
  private String _chargeCode;
  private Person _manager;

  public Department (Person manager) {
      _manager = manager;
  }

  public Person getManager() {
      return _manager;
  }
```

Para obter saber o gerente de uma pessoa, é necessário inicialmente obter o departamento.

```
manager = john.getDepartment().getManager();
```

Isso revela como o departamento funciona. Para reduzir o acoplamento deve-se ocultar a classe departamento do cliente, através de um método de delegação na classe cliente.

```
  public Person getManager() {
      return _department.getManager();
  }
```

À partir de então todos os clientes devem ser atualizados para usar o novo método:

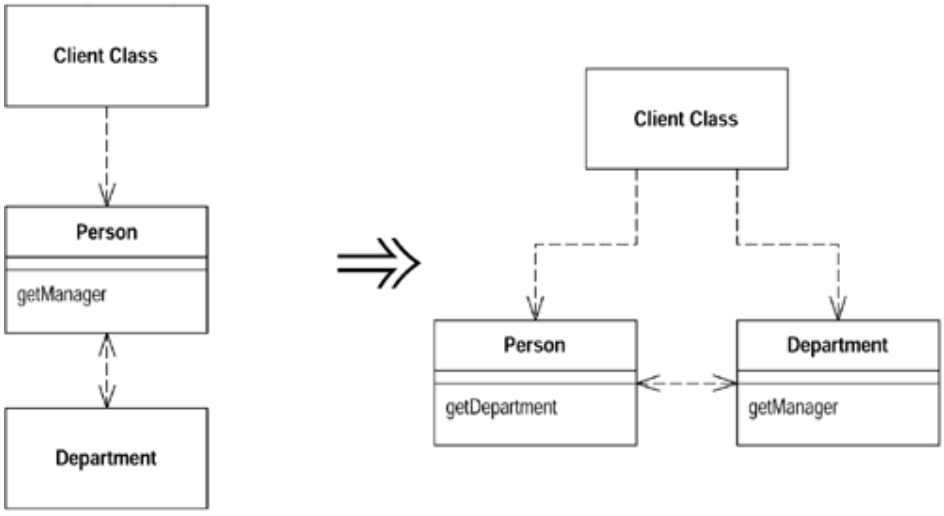
```
  manager = john.getManager();
```

Quando todos os métodos de departamento e todos os clientes de Pessoa tiverem sido atualizados, deve-se remover o método **getDepartament()** da classe Pessoa.

Remover homem do meio

Situação: A classe está fazendo muitas delegações simples.

Solução: faça o cliente chamar o objeto delegado diretamente.



Motivação:

Vantagens do encapsulamento de um objeto delegado: toda vez que o cliente quiser usar uma nova característica do objeto delegado, você tem que adicionar um método de delegação ao objeto "servidor".

Contudo depois de adicionar características por um tempo ele se torna complicado. A classe servidore é apenas um "homem do meio" (em outras palavras, uma indireção), e talvez seja hora para o cliente chamar o objeto delegado diretamente. É difícil de definir a quantidade ideal de encapsulamento.

Felizmente com **Ocultar Delegação** e **Remover Homem do Meio** isto não importa mais. É possível ajustar o projeto com o tempo por meio dessas refatorações.

Mecânica:

- Crie um método de acesso ao elemento delegado.
- Para cada uso (por parte do objeto-cliente) de um método delegado, remova o método do objeto "servidor" e substitua a chamada no cliente pelo método chamado no objeto delegado.
- Compile e teste depois de cada método.

Exemplos:

Uma classe **Pessoa** que oculta o **Departamento**.

```
class Person...
  Department _department;
  public Person getManager() {
      return _department.getManager();


class Department...
  private Person _manager;
  public Department (Person manager) {
      _manager = manager;
}
```

To find a person's manager, clients ask:

```
manager = john.getManager();
```

Isto é simples demais e encapsula o departamento. Para remover o nível de indireção, primeiro crie o método de acesso ao elemento que recebeu a delegação.

```
class Person...
  public Department getDepartment() {
      return _department;
   }
```

Daí em diante procura-se por métodos que usavam o método definido em **Pessoa** e altera-os para primeiro obter o objeto ao qual foi delegado. Para isso:

```
manager = john.getDepartment().getManager();
```

A partir de então pode-se excluir o método **getManager** de **Pessoa.**

Introduzir Método Estrangeiro

Situação: Uma classe servidora que você utiliza precisa de um método adicional, mas você não pode modificar a classe.

Solução: Crie um método na classe cliente com uma instância da classe servidora como um primeiro parâmetro.

```
  Date newStart = new Date (previousEnd.getYear(),
                   previousEnd.getMonth(), previousEnd.getDate() +
1);
```

```
    Date newStart = nextDay(previousEnd);

    private static Date nextDay(Date arg) {
```

```
        return new Date (arg.getYear(),arg.getMonth(), arg.getDate() +
1);
    }
```

Motivação:

Ao utilizar os métodos de uma classe é muito comum que ela não te ofereça todos os recursos dos quais necessita. Se você pode alterar o código da classe, você pode adicionar o método desejado (e que ela ainda não possui). Se você não pode modificar o código, você deve suprir a ausência deste método de alguma outra forma.

Se você utilizar este método várias vezes é provável que terá que repetir este código em diversos pontos. Este código repetitivo deve ser fatorado em um único método e reutilizado. O novo método deve então ser marcado como um método "estrangeiro", indicando claramente que ele deveria estar na classe original mas por razões de implementação não foi possível alocá-lo adequadamente.

Se você perceber que está criando muitos métodos estrangeiros para uma classe servidora ou muitas de suas classes utilizam o mesmo método estrangeiro, você deve usar a operação "Introduzir Extensão Local".

Mecânica:

- Crie um método na classe cliente que realiza a tarefa que você necessita.
  O método não deve acessar nenhum recurso da classe cliente. Se ele necessita de um valor, passe-o por parâmetro.
- Passe uma instância da classe servidora como o primeiro do método construtor.
- Comente o método como "Método estrangeiro; deveria estar no servidor".
  Deste modo você pode usar encontrar métodos estrangeiros posteriormente, caso tenha a chance de mover o método.

Exemplos:

Seja o seguinte código que é executado durante um período contábil.

```
Date newStart = new Date (previousEnd.getYear(),
   previousEnd.getMonth(), previousEnd.getDate() + 1);
```

O código do lado direito da atribuição pode ser extraído como um método estrangeiro da classe Date.

```
  Date newStart = nextDay(previousEnd);

  private static Date nextDay(Date arg) {
  // foreign method, should be on date
      return new Date (arg.getYear(),arg.getMonth(), arg.getDate() +
1);
  }
```

Introduzir Extensão Local

Situação: Uma classe servidora que você utiliza precisa de vários métodos adicionais, mas você não pode modificar a classe.

Solução: Crie uma nova classe que contenha esses métodos extras. Trasnforme-a em uma subclasse ou "pacote" da classe original.

Motivação:

Para poucos métodos (um ou dois) que você não consegue alocá-los na classe de origem, pode-se usar "Introduzir Método Estrangeiro". Para mais métodos, convém agrupar esses métodos em um lugar apropriado. Então deve-se usar as técnicas de Orientação por Objetos como uma subclasse ou alguma classe que atue como uma espécie de "invólucro" ou "pacote". Desse modo a classe servidora utiliza a subclasse ou a classe-pacote como sendo uma extensão local.

Uma extensão local é uma classe separada, sendo ela um subtipo da classe a qual ela está estendendo. Isso significa que possui responsabilidades adicionais às responsabilidades definidas na classe-base. Portanto ao invés de usar a classe original, deve-se instanciar uma extensão local e utilizá-la.

Problemas na utilização desta refatoração podem acontecer depois que ela for aplicada. Se utilizar herança, ao criar um objeto de uma subclasse cria-se também um objeto da classe-base. Se outros objetos referenciarem tem-se dois objetos com os mesmos dados. Se o objeto original é imutável, não há problema. Mas se o objeto original pode mudar, há um problema, pois mudanças em um objeto não devem alterar o outro objeto e, portanto, deve-se usar uma espécie de invólucro. Daquele modo, as mudanças feitas em uma extensão local afetam o objeto original e vice versa.

Mecânica:

- Crie uma classe de extensão, seja uma subclasse ou um "empacotamento (ou invólucro)" da classe original.
- Adicione métodos construtores de conversão para a extensão.

Um construtor recebe o original como parâmetro. No caso da subclasse, deve-se chamar um construtor apropriado da superclasse. Caso seja utilizada uma classe-invólucro, deve-se definir ao campo da classe o parâmetro que foi passado.

- Adicione novas responsabilidades à extensão.
- Troque a classe original pela extensão onde for necessário.
- Mova quaisquer métodos estrangeiros definidos para a classe-original para a extensão.

Exemplos:

A primeira coisa a decidir é se deve utilizar subclasse ou uma classe-invólucro. Subclasse é o modo mais óbvio.

```
Class mfDate extends Date {
  public nextDay()...
  public dayOfYear()...
```

Uma classe invólucro utiliza delegação.

```
class mfDate {
  private Date _original;
  ...
}
```

Usando Subclasses:

Inicialmente cria-se a nova classe como sendo uma subclasse da classe original:

```
class MfDateSub extends Date
```

Posteriormente deve-se lidar com diferenças entre a classe base e a extensão, com relação aos métodos construtores. O construtor da classe original deve ser repetido com uma delegação simples:

```
  public MfDateSub (String dateString) {
        super (dateString);
  };
```

Agora adiciona-se um construtor de conversão, que recebe um objeto-original como um argumento.

```
public MfDateSub (Date arg) {
    super (arg.getTime());
}
```

Agora pode-se adicionar novas responsabilidades à extensão e usar “Mover Método” para mover quaisquer métodos estrangeiros para a classe de extensão:

```
client class...
  private static Date nextDay(Date arg) {
```

```
    // foreign method, should be on date
        return new Date (arg.getYear(),arg.getMonth(), arg.getDate() +
1);
    }
```

becomes

```
  class MfDate...
    Date nextDay() {
        return new Date (getYear(),getMonth(), getDate() + 1);
  }
```

Usando classes-invólucro:

Inicie declarando a classe invólucro (com a referência para a classe original.

```
  class mfDate {
    private Date _original;
  }
```

Posteriormente defina os métodos construtores (de modo diferente à abordagem de subclasses). O construtor original é implementado com uma delegação simples.

```
   public MfDateWrap (String dateString) {
       _original = new Date(dateString);
   };
```

O construtor que realiza a conversão entre tipos, nessa abordagem simplesmente define a variável de instância.

```
   public MfDateWrap (Date arg) {
       _original = arg;
   }
```

Para cada método definido na classe original, deve-se defini-lo como uma delegação na classe de extensão.

```
   public int getYear() {
       return _original.getYear();
   }

   public boolean equals (MfDateWrap arg) {
       return (toDate().equals(arg.toDate()));
   }

        (... ... ...)
```

Uma vez que todos os métodos foram definidos, utilize “Mover Método” para adicionar responsabilidades específicas para a nova classe:

```
  client class...
    private static Date nextDay(Date arg) {
    // foreign method, should be on date
        return new Date (arg.getYear(),arg.getMonth(), arg.getDate() +
1);
    }
```

Se torna

```
  class MfDate...
    Date nextDay() {
        return new Date (getYear(),getMonth(), getDate() + 1);
  }
```

Um problema em classes-invólucros é como lidar com métodos que recebem um objeto da classe original como argumento, tal como:

```
  public boolean after (Date arg)
```

Por não ser possível alterar o objeto original, pode-se executar o método em apenas uma direção:

```
  aWrapper.after(aDate)                          // can be made to work
  aWrapper.after(anotherWrapper)                 // can be made to work
  aDate.after(aWrapper)                          // will not work
```

Usar o nome de métodos já definidos na classe original traz a vantagem de ocultar do cliente a utilização de um objeto-invólucro. De fato o cliente não está interessado em saber se está utilizando um objeto "original" ou um objeto "invólucro", mas sim em utilizar os métodos de ambos objetos de forma igual. Contudo sobrescrever alguns métodos pode não ser indicado. Considere o método:

```
  public boolean equals (Date arg)     // causes problems
```

Apesar de ser possível sobrescrevê-lo de modo que ele funcione, outras partes do sistema Java assumem que a operação equals é simétrica, isto é: se **a.equals(b)** então **b.equals(a)**. Se esta regra for violada, *bugs* estranhos podem ocorrer. A única forma de solucionar este problema é modificar a classe original Date, a qual não se tem acesso, e caso tivesse não seria necessário descrever essa refatoração. Não há alternativas: deve-se expor a utilização do "objeto-invólucro". Para testes de igualdade, defina e nomeie um novo método.

```
  public boolean equalsDate (Date arg)
```

Para testar diferentes tipos de objetos, devese prover diferentes versões deste método, uma para cada tipo de objeto (original e invólucro).

```
public boolean equalsDate (MfDateWrap arg)
```