Organizando Dados

Esse grupo contém várias operações de refatoração voltadas para lidar com dados das classes, ou seja, responsabilidade *de conhecer*.

Refatorações desse grupo ajudarão a definir a melhor forma para acessar os elementos de dados, organizar estruturas de dados em objetos, substituir a forma de representação de dados de estruturas para objetos, dentre tantas outras.

Brevemente pode-se descrever o propósito de cada operação como sendo:

- Auto-encapsular campo: discute se um elemento de um objeto deve ser acessado diretamente ou através de um método de acesso.
- Substituir valor de dado por objeto: permite transformar um dado "inerte" em objetos com operações.
- Mudar valor por referência: transformar em objetos que são utilizados em várias partes de um software.
- Substituir array por objeto: transformar um array que age como uma estrutura de dados em uma estrutura mais clara.
- Substituir número mágico por constante simbólica: substituir números mágicos por constantes que expressem mais diretamente seu propósito.
- Substituir associação unidirecional por bidirecional: quando necessita-se adicionar uma nova função.
- Substituir associação bidirecional por unidirecional: quando necessita-se remover complexidade desnecessária.
- Duplicar dados observáveis: dividir a alocação de dados entre camada de domínio e de interface com usuário.
- Encapsular campo: realizar o acesso a um campo por meio de métodos de acesso, mesmo para elementos da própria classe.
- Encapsular coleção: realizar o acesso a uma coleção por meio de métodos de acesso, mesmo para elementos da própria classe.
- Substituir registro com classe de dados: substituir a representação de um registro por uma classe.
- Substituir tipo de dados Código por classe: considerando código como sendo um valor especial que indica algo particular para um tipo de instância. Utilizar classes de dados para representá-lo permite uma melhor verificação de tipo e facilidades para mover comportamento posteriormente.
- Substituir tipo de dados Código por subclasse: idem à anterior, mas o comportamento é variável conforme o código.
- Substituir tipo de dados Código por State/Strategy: idem à anterior, mas solução mais refinada e complexa.

Obs.: As refatorações cujos nomes estão sublinhados não são apresentadas neste material. Para maiores informações consulte o livro adotado pela disciplina ou o site www.refactoring.com.

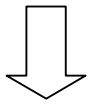
Refatorações do grupo Composição de Métodos:

Auto-encapsular campo

Situação: Você está acessando um campo diretamente, mas o acoplamento ao campo começa a ser incômodo.

Solução: Criar métodos **get** e **set** para o campo e usar somente eles para acessar o campo.

```
private int _low, _high;
boolean includes (int arg) {
    return arg >= _low && arg <= _high;
}
```



```
private int _low, _high;
boolean includes (int arg) {
    return arg >= getLow() && arg <= getHigh();
}
int getLow() {return _low;}
int getHigh() {return _high;}
```

Motivação: Ao tratar de acesso a campos, duas abordagens são consideradas pela comunidade. Uma defende que dentro da classe em que um campo é definido, pode-se acessá-la livre e diretamente, A outra abordagem defende que mesmo dentro da própria classe, deve-se acessar os atributos apenas através de métodos de acesso, ou seja, indiretamente.

Vantagens do acesso indireto são que ele permite que uma subclasse sobrescreva-o definindo uma forma alternativa para obter aquele dado e maior flexibilidade em gerenciar os dados.

A vantagem de acesso direto à variável é que o código fica mais fácil de ser lido e compreendido.

Com refatoração essa discussão pode ser deixada de lado: lança-se mão das operações de refatoração sempre que necessário encapsular os dados de uma classe.

Mecânica da refatoração:

- Crie um método **get** e **set** para o atributo.
- Localize todas as referência para o atributo e substitua-as pelo método **set** e get.

  Altere acessos ao campo com uma chamada ao método get, substitua atribuições com uma chamada ao método set.

- Transforme o campo em **privado**.
- Confira novamente se você trocou todas as referências.
- Compile e teste.

Exemplo

Um exemplo bem simples de uma classe que descreve um intervalo numérico:

```
class IntRange {

  private int _low, _high;

  boolean includes (int arg) {
      return arg >= _low && arg <= _high;
  }

  void grow(int factor) {
      _high = _high * factor;
  }
  IntRange (int low, int high) {
      _low = low;
      _high = high;
  }
```

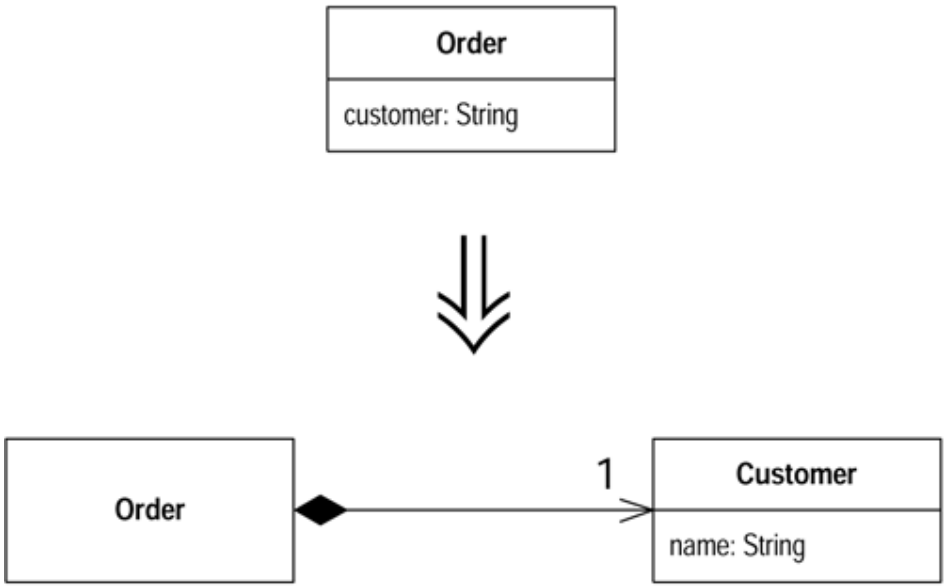
Para auto-encapsular defina inicialmente os métodos get e set (caso eles não existam) e utilize-os:

```
class IntRange {

  boolean includes (int arg) {
      return arg >= getLow() && arg <= getHigh();
  }

  void grow(int factor) {
      setHigh (getHigh() * factor);
  }

  private int _low, _high;

  int getLow() {
      return _low;
  }

  int getHigh() {
      return _high;
  }

  void setLow(int arg) {
      _low = arg;
  }

  void setHigh(int arg) {
      _high = arg;
  }
```

Substituir valor de dado por objeto

Situação: Quando possui um item de dado que necessita itens de dados adicionais ou comportamento.

Solução: Transforme o item de dado em um objeto.



Motivação: geralmente em estágios iniciais de desenvolvimento algumas decisões são tomadas em representar alguns fatos simples como atributos de uma classe. À medida que o desenvolvimento ocorre, percebe-se que esses item não são tão simples quanto se pensava.

Para um ou dois itens pode-se definir os métodos no objeto, o que pode levar a "cheiros" de duplicação ou inveja de recursos. Quando esses cheiros começarem a aparecer, transforme os valores de dados em objetos.

Mecânica da refatoração:

- Crie a classe para o valor. Defina nela um campo final de mesmo tipo daquele na classe fonte. Adicione um método **get** e um construtor que receba o campo como um parâmetro.
- Compile.
- Mude o tipo do campo na classe-fonte para a nova classe (referência para a nova classe).
- Mude o método **get** da classe fonte para chamar o método **get** da nova classe.
- Se o campo for utilizando no construtor da classe-fonte, atribua o campo utilizando o construtor da nova classe.
- Altere o método **get** para criar uma nova instância da nova classe.
- Compile e teste.
- Você pode agora precisar usar a operação "Mudar Valor para Referência" no novo objeto.

Exemplo:

Seja uma classe **Order** que armazena o cliente como uma string. Deseja-se transformar **Customer** em um objeto a parte.

```
class Order...
  public Order (String customer) {
      _customer = customer;
  }
  public String getCustomer() {
      return _customer;
  }
  public void setCustomer(String arg) {
      _customer = arg;
  }
  private String _customer;
```

Incialmente cria-se a classe **Customer.** Nela é definido um campo **final** do tipo String. Define-se ainda um método **get** e um construtor que utilize o atributo:

```
class Customer {
  public Customer (String name) {
      _name = name;
  }
  public String getName() {
      return _name;
  }
  private final String _name;
 }
```

Agora pode alterar o tipo de dado do campo **Customer** (definido na classe **ORDER**) e os métodos que o referenciam. Tais métodos deverão usar as referências apropriadas da classe **Customer,** sendo que para o método set atribui-se um novo **Customer.**

```
class Order...
  public Order (String customer) {
      _customer = new Customer(customer);
  }
  public String getCustomer() {
      return _customer.getName();
  }

  private Customer _customer;

  public void setCustomer(String arg) {
      _customer = new Customer(customer);
  }
```

Por fim verifica-se os métodos de **Order** que usam **Customer** e realiza-se algumas mudanças para torná-los mais claros. O método get foi renomeado para tornar mais claro que o nome do cliente é retornado e não o objeto em si.

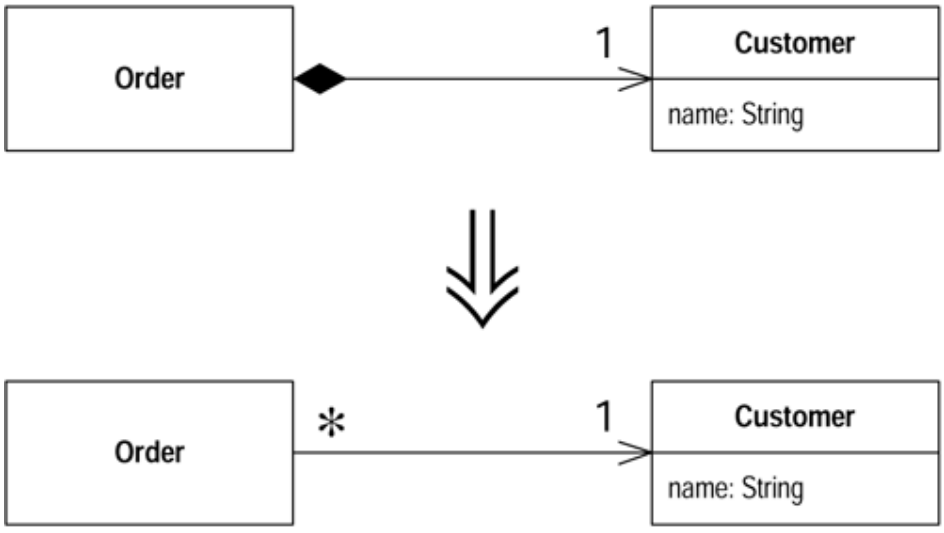
```
  public String getCustomerName() {
      return _customer.getName();
  }

  public Order (String customerName) {
```

```
        _customer = new Customer(customerName);
    }

    public void setCustomer(String customerName) {
        _customer = new Customer(customerName);
    }
```

Mudar valor por referência

Situação: Você possui uma classe com muitas instâncias iguais que você deseja substituir por apenas um objeto.

Solução: Transforme o objeto em um objeto-referência.



Motivação: Objetos de referências são coisas como cliente ou conta. Cada objeto representa um objeto do mundo real e você usa a identidade do objeto para testar se eles são iguais. Objetos de valor são coisas como data ou dinheiro. Eles são definidos completamente pelos seus valores de dados. Não importa quantas cópias existem; se você precisa avaliar se dois objetos são iguais então você precisa sobrescrever os métodos **equals** e **hashcode** (no caso de Java).

A decisão entre valor e referência nem sempre é clara. Algumas vezes você inicia algum objeto com um conjunto de valores imutáveis. Daí você precisar dar a esse objeto algum valor alterável de modo que as mudanças atinjam a todos que referem-se aquele objeto. Neste ponto você necessita transformá-lo em um objeto referência.

Mecânica da refatoração:

- Use “Substituir Construtor com método Fábrica”.
- Compile e teste.

- Decida qual objeto é responsável por prover acesso aos objetos.

  Isso pode ser um dicionário estático ou um objeto de registro.

  Você pode ter mais de um objeto que atua como ponto de acesso para o novo objeto.

- Decida se os objetos são pré-criados ou criados “on the fly”.

  Se os objetos são pré-criados e você está recuperando-os da memória, você necessita garantir que eles estejam carregados antes de serem utilizados.

- Altere o método fábrica para retornar a referência para o objeto.

  Se os objetos são pré-criados você deve decidir sobre quem tratará os erros caso alguém solicite um objeto que não exista.

  Você pode usar “Renomear Método” no método fábrica para ajustá-lo ao objeto que é retornado.

- Compile e teste.

Exemplo:

Continuando o exemplo da refatoração anterior, tem-se uma classe *Customer*:

```
class Customer {
  public Customer (String name) {
      _name = name;
  }
  public String getName() {
      return _name;
  }
  private final String _name;
}
```

Ela é utilizada por uma classe **Order.**

```
class Order...
  public Order (String customerName) {
      _customer = new Customer(customerName);
  }
  public void setCustomer(String customerName) {
      _customer = new Customer(customerName);
  }
  public String getCustomerName() {
      return _customer.getName();
  }
  private Customer _customer;
```

E algum método de uma classe cliente seria do tipo:

```
   private static int numberOfOrdersFor(Collection orders, String
customer) {
       int result = 0;
       Iterator iter = orders.iterator();
       while (iter.hasNext()) {
           Order each = (Order) iter.next();
           if (each.getCustomerName().equals(customer)) result++;
       }
       return result;
   }
```

Neste ponto é um objeto valor. Cada pedido tem seu próprio objeto **Customer,** ainda que eles representem o mesmo cliente (**customer)**. Deseja-se mudar isso de modo que vários pedidos para um mesmo cliente, compartilhem o mesmo objeto. Começa-se alterando o construtor, adaptando-o para o padrão Fábrica (operação refatoração "Substituir Construtor por método fábrica"). Isso permite ter controle do processo de criação de objetos.

```
class Customer {
  public static Customer create (String name) {
      return new Customer(name);
  }
```

Substitui-se então a chamada ao construtor por chamadas à Fábrica:

```
class Order {
public Order (String customer) {
    _customer = Customer.create(customer);
}
```

Transforme o construtor em **Privado:**

```
class Customer {
  private Customer (String name) {
     _name = name;
  }
```

Agora deve-se decidir como acessar os clientes. Neste exemplo foi utilizado um outro objeto, no caso um registro para ser o ponto de acesso. Por questões de simplicidade ele, foi definido como um campo estático em **Cliente,** transformando-o em um ponto de acesso global.

```
private static Dictionary _instances = new Hashtable();
```

Neste exemplo decidiu-se ainda criar os clientes quando for necessário ou antecipadamente.

```
class Customer...
  static void loadCustomers() {
      new Customer ("Lemon Car Hire").store();
      new Customer ("Associated Coffee Machines").store();
      new Customer ("Bilston Gasworks").store();
  }
  private void store() {
```

```
        _instances.put(this.getName(), this);
    }
```

Neste ponto pode-se alterar o método fábrica de modo a retornar os clientes pré-criados:

```
    public static Customer create (String name) {
        return (Customer) _instances.get(name);
    }
```

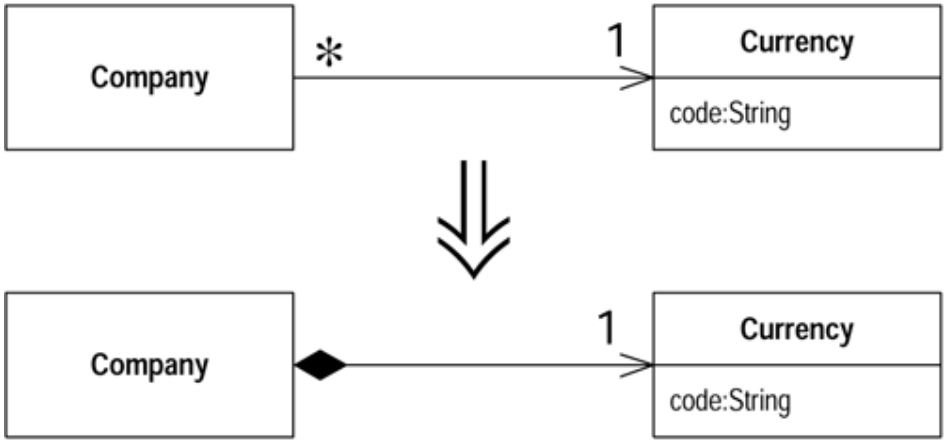
Por fim, renomeia-se o método que retorna um cliente já existente, de modo a tornar mais claro o seu entendimento (utilizando a operação “Renomear método”):

```
class Customer...
    public static Customer getNamed (String name) {
        return (Customer) _instances.get(name);
    }
```

Mudar referência por valor

Situação: Você possui um objeto referência que é pequeno, imutável e difícil de ser gerenciado.

Solução: *Transforme-o em um objeto-valor.*



Motivação: O gatilho para ir de um objeto referência para um valor é quando trabalhar com um objeto referência se torna difícil. Objetos referência têm que ser controlados de alguma maneira. Você sempre precisa pedir ao controlar pelo objeto apropriado. A ligação na memória também pode ser difícil de ser realizada (em algumas linguagens é comum). Objetos valores são particularmente úteis em sistemas concorrentes e distribuídos.

Uma propriedade importe de objetos valores é o fato de que eles podem ser **imutáveis**. Sempre que você solicitar um objeto, você sempre vai obter o mesmo resultado. Se isso for verdade, não há problemas em ter vários objetos representando a mesma coisa. Se o valor é **mutável,** você tem que garantir que a mudança em qualquer objeto também deverá ser realizada em todos os outros objetos que representam a mesma coisa.

Mecânica da refatoração:

- Verifique se o objeto candidato é **imutável** ou pode se tornar **imutável.**

  Se o objeto não for atualmente imutável, utilize "Remover Método Setting" até ele se tornar.

  Se o objeto candidato não pode se tornar imutável, você deve abandonar essa refatoração.

- Crie um método **equals** e **hash** (no caso de Java. Outras linguagens devem ser estudadas).
- Compile e teste.
- Considerer remover quaisquer métodos fabrica e transformar o construtor em público.

Exemplo

Seja uma classe **Currency**:

```
class Currency...
  private String _code;

  public String getCode() {
      return _code;
  }
  private Currency (String code) {
      _code = code;
  }
```

Tudo o que essa classe faz é armazenar e retornar um código. É um objeto referência, de modo que para obter o objeto precisa-se usar:

```
      Currency usd = Currency.get("USD");
```

A classe **Currency** mantém uma lista de instâncias. Não é possível usar um construtor (por ele ser privado).

```
new Currency("USD").equals(new Currency("USD")) // returns false
```

Iniciando a refatoração: verificar se o objeto é imutável. Se for é possível continuar com a operação de refatoração. O próximo passo é definir o método **equals**.

```
  public boolean equals(Object arg) {
      if (! (arg instanceof Currency)) return false;
      Currency other = (Currency) arg;
      return (_code.equals(other._code));
  }
```

Se o método **equals** foi definido, também é necessário definir o método **hashCode.** O modo fácil de fazer isso écalcuar o hash code de todos os campos usados no método **equals** e realizar um OU EXCLUSIVO (^) neles. Neste caso é fácil pois há apenas um campo:

```
public int hashCode() {
    return _code.hashCode();
}
```

Com ambos métodos implementados, pode-se compilar e testar. É necessário definir ambos os métodos.

Neste ponto pode-se criar quantas instancias de **Currency** forem necessárias.

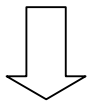
```
new Currency("USD").equals(new Currency("USD")) // now returns true
```

Substituir array por objeto

Situação: Você possui um array em que vários elementos significam coisas diferentes.

Solução: *Substitua o array por um objeto que possui um campo para cada elemento.*

```
String[] row = new String[3];
row [0] = "Liverpool";
row [1] = "15";
```



```
Performance row = new Performance();
row.setName("Liverpool");
row.setWins("15");
```

Motivação: Arrays são estruturas comuns para organização de dados. Entretanto elas devem ser usadas para conter apenas uma coleção de objetos similares em alguma ordem.

Contudo é comum encontrar situações em que um array contém um número de coisas de diferentes. Com um objeto você pode usar nomes de campos e métodos convenientes à cada parte do vetor, de modo que você não precise lembrar ou contar com comentários explicativos realizados no código.

Mecânica da refatoração:

- Cire uma nova classe para representar a informação do array. Dê a ela um campo público para representar o array.

- Altere todos os usuários do array para utilizar a nova classe.
- Compile e teste.
- Um por um, adicione métodos **get** e **set** para cada elemento do array. Nomeie os métodos de acesso de acordo com o seu propósito no array. Altere os clientes para usar os métodos de acesso. Compile e teste depois de cada alteração.
- Quando todos os acessos aos elementos do array estiverem definidos pelos métodos, altere o array para privado.
- Compile.
- Para cada elemento do array, crie um campo na classe e mude os métodos de acesso para utilizá-lo.
- Compile e teste depois de cada elemento ser modificado.
- Quando todos os elementos tiverem sido substituídos pelos campos, delete o array.

Exemplo

Declaração de um vetor e uma possível utilização:

```
String[] row = new String[3];


row [0] = "Liverpool";

row [1] = "15";


String name = row[0];

int wins = Integer.parseInt(row[1]);
```

Para transformá-lo em um objeto inicialmente cria-se uma classe, com uma estrutura de dados pública:

```
 class Performance {

    public String[] _data = new String[3];
}
```

Posteriormente procura-se pelos lugares em que o array foi criado e acessado:

```
Performance row = new Performance();
(...)
row._data [0] = "Liverpool";
row._data  [1] = "15";

String name = row._data[0];
int wins = Integer.parseInt(row._data[1]);
```

Um por um, cria-se métodos de acesso aos elementos do vetor:

```
class Performance...
  public String getName() {
```

```
        return _data[0];
    }
    public void setName(String arg) {
        _data[0] = arg;
    }
```

Alterando os elementos que acessavam diretamente o vetor, fazendo-os usar os métodos de acesso:

```
    row.setName("Liverpool");
    row._data  [1] = "15";

    String name = row.getName();
    int wins = Integer.parseInt(row._data[1]);
```

Uma vez feito isso para todos os elementos do array, modifique seu acesso para privado:

```
     private String[] _data = new String[3];
```

Neste ponto a interface da classe está pronta. Cada posição de acesso pode então se tornar um campo em separado, o que implica em alterar seus métodos de acesso.

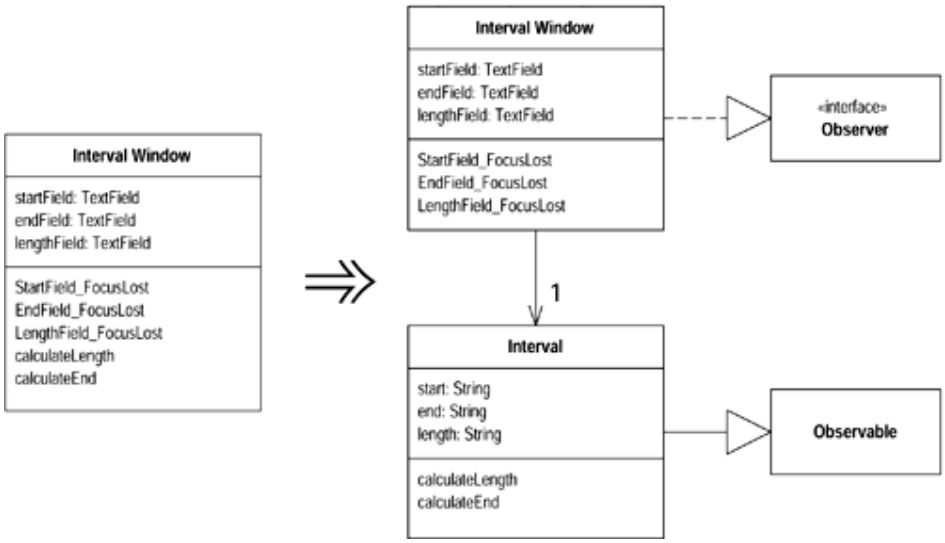
```
  class Performance...
    public String getName() {
        return _name;
    }
    public void setName(String arg) {
        _name = arg;
    }
    private String _name;
```

Feito isso, pode-se excluir o vetor.

Duplicar dado observado

Situação: Você possui um dado disponível apenas em um objeto de interface com usuário e métodos de domínio necessitam acessá-lo.

Solução: *Copie o dado para um objeto de domínio. Defina um observador para sincronizar os dois pedaços de dados.*



Motivação:

Um sistema bem particionado separa código de interface de usuário, dos códigos que representam a lógica do negócio, por alguns motivos:

- Você pode ter várias interfaces para regras de negócio similares;
- A interface de usuário se torna complicada demais se ela faz as duas coisas;
- É mais fácil de manter e evoluir objetos de domínio separadamente da interface gráfica;
- Você pode ter diferentes desenvolvedore trabalhando em diferentes partes.

Apesar do comportamento poder ser separado facilmente com os dados não ocorre o mesmo, é bem mais complicado. Dados que precisam ser apresentados na interface possuem o mesmo significado de dados que estão representados no modelo de domínio.

Caso você encontre código com regras de negócio embutidas na camada de apresentação, você deve separá-los por meio desta operação de refatoração. Contudo, para os dados você não pode simplesmente movê-los, você deve duplicá-los e prover algum mecanismo de sincronismo.

Mecânica da refatoração:

- Transforme a classe de apresentação em uma observadora do objeto de domínio. [Gang of Four].

  Se não há uma classe de domínio ainda, crie uma.

Se não há associação entre classe de apresentação e classe de domínio, coloque a classe de domínio em um campo da classe de apresentação.

- Use “auto encapsular campo” no dado de domínio, dentro da classe de apresentação.
- Compile e teste.
- Adicione uma chamada ao método **set** do manipulador de eventos para atualizar o componente gráfico com seu valor atual, utilizando acesso direto.

  Defina um método no manipulador de eventos que atualize o valor no componente gráfico com base em seu valor atual. Isto parece ser descenssário, colocar um manipulador de eventos que atualize o valor do componente com base em seu valor atual, mas usar este método permite definir qualquer comportamento a ser executado.

  Quando você realizar essa mudança, não utilize método get para o componente gráfico; use acesso direto para o componente. Posteriormente o método get vai obter o valor do domínio, o qual não altera até que o método set seja executado.

  Certifique-se que o mecanismo de manipulação de evento é disparado pelo código de teste.

- Compile e teste.
- Defina o dado e os métodos de acesso na classe de domínio.

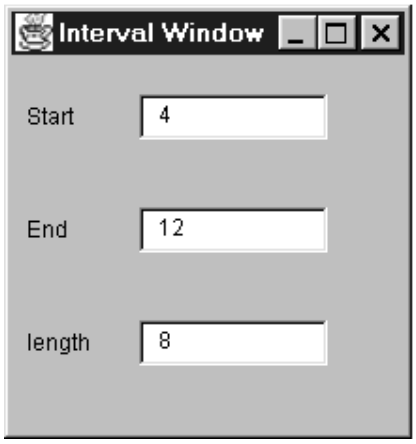
  Certifique que o método set no domínio dispara o mecanismo de notificação do padrão observador.

  Use o mesmo tipo de dado no domínio e na apresentação (tipicamente uma String). Converta o tipo de dado posteriormente em uma operação de refatoração.

- Redirecione os métodos de acesso para escrever no campo do domínio.
- Modifique o método **update** do observador para copiar o dado do campo no objeto de domínio para o campo do objeto de interface.
- Compile e teste.

Exemplo

Seja a seguinte aplicação, em que os campos são automaticamente calculados com base nos valores de entrada: se os valores de início e fim são informados calcula-se o intervalo, se o valor de início e o intervalo forem informados, calcula-se o valor final.




```
public class IntervalWindow extends Frame...
  java.awt.TextField _startField;
  java.awt.TextField _endField;
  java.awt.TextField _lengthField;

  class SymFocus extends java.awt.event.FocusAdapter
  {
      public void focusLost(java.awt.event.FocusEvent event)
      {
          Object object = event.getSource();
          if (object == _startField)
              StartField_FocusLost(event);
          else if (object == _endField)
              EndField_FocusLost(event);
          else if (object == _lengthField)
              LengthField_FocusLost(event);
      }
  }
```

Os listeners de eventos:

```
  void StartField_FocusLost(java.awt.event.FocusEvent event) {
      if (isNotInteger(_startField.getText()))
          _startField.setText("0");
      calculateLength();
  }

  void EndField_FocusLost(java.awt.event.FocusEvent event) {
      if (isNotInteger(_endField.getText()))
          _endField.setText("0");
      calculateLength();
  }

  void LengthField_FocusLost(java.awt.event.FocusEvent event) {
      if (isNotInteger(_lengthField.getText()))
          _lengthField.setText("0");
      calculateEnd();
  }
```

Os métodos de cálculo definidos na classe GUI:

```
  void calculateLength(){
```

```
      try {
        int start = Integer.parseInt(_startField.getText());
        int end = Integer.parseInt(_endField.getText());
        int length = end - start;
        _lengthField.setText(String.valueOf(length));
      } catch (NumberFormatException e) {
        throw new RuntimeException ("Unexpected Number Format Error");
      }
}
void calculateEnd() {
    try {
      int start = Integer.parseInt(_startField.getText());
      int length = Integer.parseInt(_lengthField.getText());
      int end = start + length;
      _endField.setText(String.valueOf(end));
    } catch (NumberFormatException e) {
      throw new RuntimeException ("Unexpected Number Format Error");
    }
}
```

O problema aqui é separar a camada de visão da lógica da aplicação!!! Realizar os cálculos independentemente da classe de visão:

Considerando que inicialmente não existe uma classe de domínio, deve-se criá-la:

```
class Interval extends Observable {}
```

A janela de intervalo necessita de uma associação com essa classe de domínio:

```
    private Interval _subject;
```

Próximo passo é inicializar este objeto e transformar a interface em um observador de Intervalo. Pode-se fazer isso no construtor da janela gráfica:

```
    _subject = new Interval();
    _subject.addObserver(this);
    update(_subject, null);
```

*Essa chamada ao método **update** garante que por ter duplicado o dado na classe de domínio, o objeto de interface é inicializado a partir da classe de domínio.*

Para implementar o observador é necessário criar um método **update,** que nesse momento pode ser vazio:

```
    public void update(Observable observed, Object arg) {
    }
```

Pode-se compilar e testar neste ponto para garantir que nenhum erro foi gerado.

Agora pode-se lidar com os campos do objeto, um por vez. Deve-se auto-encapsular os campos na classe de interface, criando métodos **set** e **get**, que serão utilizados pelos campos de texto para acessá-los. I

```
    String getEnd() {
        return _endField.getText();
    }

    void setEnd (String arg) {
```

```
        _endField.setText(arg);
    }
```

Todo lugar que utilizar uma referência direta para o campo deve ser substituída pelo método de acesso:

```
    void calculateLength(){
      try {
        int start = Integer.parseInt(_startField.getText());
        int end = Integer.parseInt(getEnd());
        int length = end - start;
        _lengthField.setText(String.valueOf(length));
      } catch (NumberFormatException e) {
        throw new RuntimeException ("Unexpected Number Format Error");
      }
    }

    void calculateEnd() {
      try {
        int start = Integer.parseInt(_startField.getText());
        int length = Integer.parseInt(_lengthField.getText());
        int end = start + length;
        setEnd(String.valueOf(end));
      } catch (NumberFormatException e) {
        throw new RuntimeException ("Unexpected Number Format Error");
      }
  }

    void EndField_FocusLost(java.awt.event.FocusEvent event) {
        if (isNotInteger(getEnd()))
            setEnd("0");
        calculateLength();
     }
```

Na interface gráfica o usuário pode mudar o valor do campo diretamente sem chamar o método **setEnd.** Portanto é necessário colocar uma chamada ao método **setEnd** no manipulador de eventos, de modo que essa chamada mude o valor do campo final pelo próprio valor do campo final.

```
    void EndField_FocusLost(java.awt.event.FocusEvent event) {
        setEnd(_endField.getText());
        if (isNotInteger(getEnd()))
            setEnd("0");
        calculateLength();
    }
```

Agora pode-se definir os atributos na classe de domínio e inicializá-los com os valores da GUI por meio de métodos **set** e **get**.

```
  class Interval { ...
    private String _end = "0";

    String getEnd() {
        return _end;
    }
    void setEnd (String arg) {
        _end = arg;
        setChanged();
        notifyObservers();
    }
```

```
    ...
}
```

Por utilizar o padrão **Observer** deve-se adicionar o código de notificação no método set. Note que os atributos da classe e os parâmetros dos métodos set e get são String: isso ocorre para diminuir o quanto possível as diferenças entre objetos de interface e de domínio. À partir do momento em que os dados estiverem duplicados, pode-se alterar seus tipos de dados para inteiros, reais, etc...

Com os dados duplicados, é hora de fazer a classe de interface utilizar os métodos de acesso da classe de domínio.

```
class IntervalWindow...
  String getEnd() {
      return _subject.getEnd();
  }
  void setEnd (String arg) {
      _subject.setEnd(arg);
  }
```

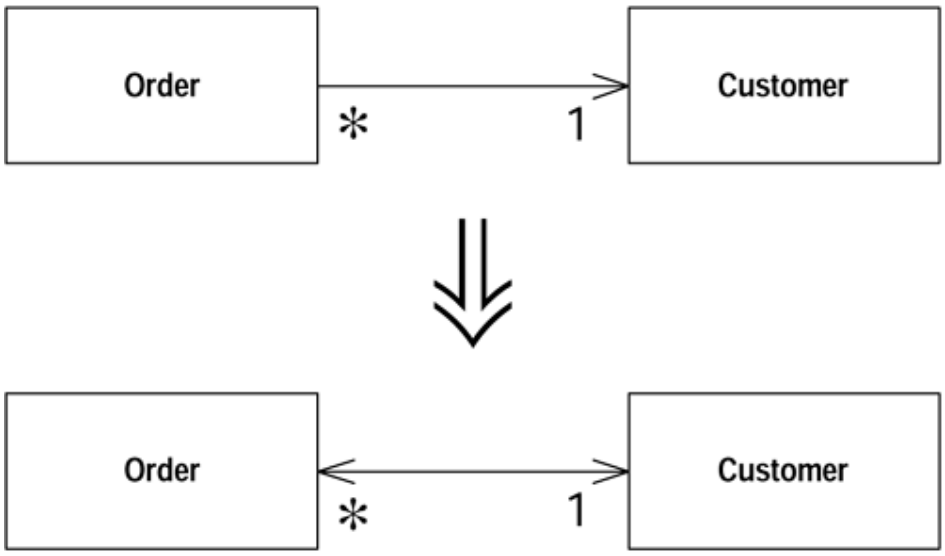
É necessário ainda que alterar o método **update** para garantir que a GUI reaja à notificação.

```
class IntervalWindow...
  public void update(Observable observed, Object arg) {
    _endField.setText(_subject.getEnd());
  }
```

Modificar associação unidirecional para bidirecional

Situação: Você possui duas classes em que uma necessita usar os recursos da outra, mas há apenas uma ligação direcional entre elas.

Solução: Adicione ponteiros de retorno (no sentido que ainda não existe), e altere os métodos de modificação para atualizar ambos os conjuntos.



Motivação: Você inicialmente definiu duas classes de modo que apenas uma classe referencie a outra. Com o tempo você percebe que um cliente de uma classes precisa obter os objetos que o referenciam. Desse modo é necessário definir uma referência bidirecional entre elas, muitas vezes chamado de ponteiro de retorno.

Mecânica da refatoração:

- Adicione um campo para o ponteiro de retorno.
- Decida qual classe vai controlar a associação.
- Crie um método de auxílio no lado da associação que não realiza o controle. Nomeie esse método para que ele claramente indique sua utilização restrita.
- Se o método que modifica as referências está no lado que controla a associação, modifique-o para atualizar os ponteiros de retorno.
- Se o método que modifica as referências está no lado controlado da associação, crie um método de controle no lado que controla a associação e chame-o à partir do método de modificação já existente.

Exemplo:

Considerando o diagrama de classes mostrado acima, sendo que um pedido está associado a um cliente:

```
class Order...
  Customer getCustomer() {
      return _customer;
  }
  void setCustomer (Customer arg) {
      _customer = arg;
  }
  Customer _customer;
```

A classe **customer** não possui referências para a classe **order**.

A refatoração inicia-se com a inclusão de referência na classe Customer, como um cliente possui vários pedidos este campo é implementado através de uma coleção, que não permite pedidos duplicados:

```
class Customer  {
  private Set _orders = new HashSet();
```

Agora deve-se decidir qual classe vai ser responsável por manter a associação. Algumas soluções comuns:

- Se a associação é um-para-muitos, então o objeto que possui a referência única é quem deve controlar as associações.
- Se um objeto é um componente do outro (relação todo-parte), o objeto composto (que representa o todo) é quem deve controlar a associação.
- Se a associação é muitos-para-muitos, não importa qual objeto é responsável por manter as associações.

Como a classe **Order** fica sendo a responsável por manter as associações, adiciona-se um método que permita a classe **Customer** acesso à coleção de **Orders**. Este método deve ser usado apenas em casos especiais e pela classe **Customer**.

```
class Customer...
  Set friendOrders() {
  /** should only be used by Order when modifying the association */
      return _orders;
  }
```

Modifica-se então o método de acesso para atualizar as referências de volta para a classe **Customer**:

```
class Order...
  void setCustomer (Customer arg) ...
      if (_customer != null) _customer.friendOrders().remove(this);
      _customer = arg;
      if (_customer != null) _customer.friendOrders().add(this);
  }
```

Este método pode sofrer variações conforme os tipos de associações utilizadas (um para um, um para muitos, muitos para muitos). Contudo o padrão de sua implementação é: inicialmente informar ao outro objeto para remover ser ponteiro para você, apontar seu ponteiro para o novo objeto e, por fim, informar ao novo objeto para adicionar um ponteiro para você.

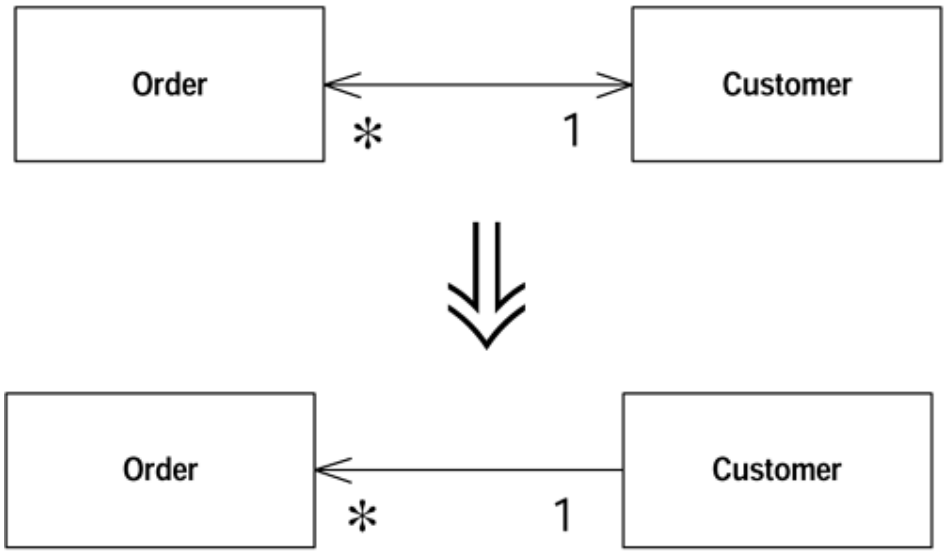
Caso queira modificar a associação pela classe **customer,** faça-o chamar o método de controle:

```
class Customer...
  void addOrder(Order arg) {
      arg.setCustomer(this);
  }
```

Modificar associação bidirecional para unidirecional

Situação: Você possui uma associação bidirecional, mas uma classe não precisa mais dos recursos da outra.

Solução: Elimine a associação desnecessária.



Motivação: Associações bidirecionais são úteis mas paga-se um preço por mantê-las no projeto. Este preço é a complexidade de manter associações bidirecionais e garantir que objetos são devidamente criados e removidos. Associações bidirecionais não são tão naturais para muitos desenvolvedores.

Associações bidirecionais forçam uma interdependência entre duas classes. Qualquer mudança em uma classe pode causar uma mudança em outra classe. Muitas interdependência levam a um sistema fortemente acoplado, no qual uma pequena alteração leva a uma série de ramificações imprevisíveis.

Portanto, sempre que você perceber uma associação bidirecional que não se justifica mais, elimine a parte desnecessária da associação.

Mecânica da refatoração:

- Examine todas as leituras realizadas no campo que representa o ponteiro que você gostaria de remover, para verificar se é possível realizar a remoção.

  Procure por "leitores diretos" (que o acessam diretamente) e ainda por métodos que chamam métodos de acesso à referência (acesso indireto).

  Avalie se é possível obter o outro objeto sem a utilização de um ponteiro. Se isso for possível você poderá utilizar a refatoração "Substituir Algoritmo" no método ***get*** *para permitir clientes usá-lo, ainda que não haja referência direta.*

  Avalie a possibilidade de passar o objeto como um parâmetro para todos os métodos que utilizam o campo.

- Se clientes precisam usar o método **get**, utilize a refatoração "Auto-encapsular campo", depois utilize "Substituir Algoritmo" no método **get**, compile e teste.
- Se os clientes não precisam do método **get**, altere cada cliente do campo de modo que eles obtenham o objeto de alguma outra maneira. Compile e teste depois de cada mudança.
- Quando não houver mais nenhum leitor do campo, remova todas as atualizações do campo e remova o campo.

  *Se há vários lugares que realizam atribuições no campo, utilize "Auto-encapsular campo" de modo que todos eles utilizem um único método **set**. Compile e teste. Altere o método **set** para que ele tenha um corpo vazio. Compile e teste. Se funcionar, remova o campo, o método **set** e todas as chamadas ao método **set.***

- Compile e teste.

Exemplo

Utilizando o projeto final da refatoração anterior e revertendo-a para associação unidirecional. Nesse projeto, **customer** e **order** são objetos que referenciam um ao outro.

```
class Order...
  Customer getCustomer() {
      return _customer;
  }
  void setCustomer (Customer arg) {
      if (_customer != null) _customer.friendOrders().remove(this);
      _customer = arg;
      if (_customer != null) _customer.friendOrders().add(this);
  }
  private Customer _customer;
```

```
class Customer...
  void addOrder(Order arg) {
      arg.setCustomer(this);
  }
  private Set _orders = new HashSet();
  Set friendOrders() {
      /** should only be used by Order */
      return _orders;
  }
```

Não há pedido (**order**) sem cliente (**customer**), portanto a ligação order → customer não é necessária, mas o inverso sim.

A primeira coisa a se fazer é avaliar todos os clientes da associação e os métodos que utilizam esses clientes. Como prover acesso ao objeto **customer** aqueles que precisam dele? A maneira mais comum disso acontecer é passando o objeto como um parâmetro. Exemplo:

```
class Order...
      double getDiscountedPrice() {
```

```
        return getGrossPrice() * (1 - _customer.getDiscount());
    }
```

Se torna

```
  class Order...
    double getDiscountedPrice(Customer customer) {
        return getGrossPrice() * (1 - customer.getDiscount());
    }
```

Uma alternativa para obter o objeto é não utilizar o campo de referência. Nesse caso, obtém-se um objeto **costumer** sem usar a referência definida em **order**. Se isso for possível, altere o código do método **get** utilizando a refatoração "Substituir Algoritmo". O método **get** ficaria sendo:

```
    Customer getCustomer() {
        Iterator iter = Customer.getInstances().iterator();
        while (iter.hasNext()) {
            Customer each = (Customer)iter.next();
            if (each.containsOrder(this)) return each;
        }
        return null;
    }
```

Neste ponto duas coisas podem acontecer:

- Se o método de acesso for mantido, a associação ainda é bidirecional nas interfaces, mas é unidirecional na implementação. A referência de customer → order foi removida, mas as interdependências entre as classes ainda estão mantidas.
- Se substituir o método **get**, deve-se deixar as alterações para depois. Caso contrário altera-se cada um dos métodos clientes do método **get,** de modo que eles obtenham um objeto **customer** de alguma maneira alternativa. Deve-se compilar e testar após cada mudança.

Uma vez que todos os leitores do campo foram eliminados, deve-se eliminar as associações com os objetos que escrevem na referência. Isso se dá simplesmente através da remoção de atribuições ao campo e do próprio campo. Justificativa: já que ninguém lê o campo, pode-se excluí-lo tranquilamente.

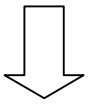
Substituir Número Mágico por Constante Simbólica

Situação: Você possui um número literal com um significado particular.

Solução: Crie uma constante, nomeie-a conforme seu significado e substitua o número por ela.

```
    double potentialEnergy(double mass, double height) {
```

```
    return mass * 9.81 * height;
}
```



```
double potentialEnergy(double mass, double height) {
    return mass * GRAVITATIONAL_CONSTANT * height;
}
static final double GRAVITATIONAL_CONSTANT = 9.81;
```

Motivação: Números mágicos são números com valores especiais que geralmente não são óbvios. Números mágicos são difíceis quando você precisa referencias o mesmo número lógico em mais de um lugar. Se os números mágicos variarem, realizar a mudança torna-se ainda mais complicado. Ainda que você não tenha que fazer a mudança, você pode ter dificuldades em entender o que significa o número em um trecho de código.

Várias linguagens permitem que você declare tais valores como constantes. Não há perda em performance.

Contra-indicações: Quando o número mágico representa um código (nesse caso utilize a refatoração "Substituir tipo-código por classe"). Quando o número mágico representa o comprimento de um array, utilize **array.length** ao percorrer o vetor.

Mecânica da refatoração:

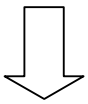
- Declare uma constante e atribua à ela o valor do número mágico.
- Procure por todas ocorrências do número mágico.
- Verifique se o numero mágico representa a utilização da constante; se sim, substitua o numero mágico pelo uso da constante.
- Compile.
- Quando todos os números mágicos forem substituídos, compile e teste. Neste ponto tudo deve funcionar como se nada tivesse sido alterado.

Encapsular campo

Situação: Quando existe um campo público.

Solução: *Torne-o privado e forneça métodos de acesso.*

```
public String _name
```



```
private String _name;
public String getName() {return _name;}
public void setName(String arg) {_name = arg;}
```

Motivação:

Um dos princípios da orientação por objeto é o **encapsulamento** (ou ocultação de informação). Tal princípio afirma que os dados de um objeto nunca devem ser públicos. Quando os dados são públicos, outros objetos podem alterar e acessar valores sem o objeto (que mantém os dados) ficar sabendo. Desvantagem: reduz a modularidade do programa. Quando dados e comportamentos (métodos) que usam tais dados são agrupados, é mais fácil de modificar o código pois ele está todo centralizado, ao invés de espalhado por diversos pontos do programa.

Encapsular campo começa o processo ao ocultar os dados e oferecer métodos de acesso. Mas isso é apenas o primeiro passo. Uma vez executado "Encapsular Campo" deve-se procurar métodos que utilizem esses novos métodos (de acesso ao campo encapsulado) e verificar se deverão ser movidos para o novo objeto através da operação "Mover método".

Mecânica da refatoração:

- Crie métodos **set** e **get** para o campo.
- Procure por todos os clientes fora da classe que referenciam o campo. Se o cliente usa o valor, substitua a referência por uma chamada ao método **get,** se o cliente modifica o valor, substitua a referência por uma chamada ao método **set.**

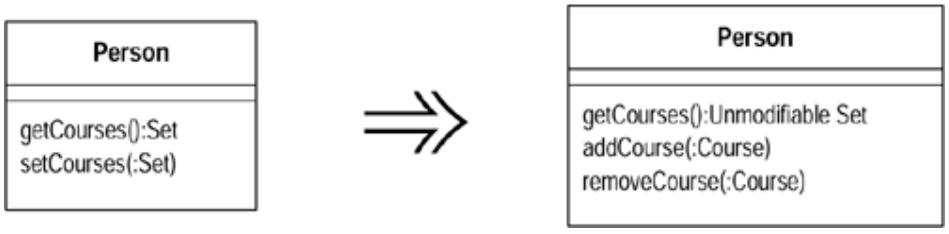
  *Se o campo é um objeto e o cliente chama um modificado do objeto, isso é uma utilização. Utilize o método **set** apenas para substituir operações de atribuição.*

- Compile e teste depois de cada alteração.
- Uma vez que todos os clientes foram modificados, declare o campo como sendo privado.
- Compile e teste.

Encapsular Coleção

Situação: Um método retorna uma coleção.

Solução: Faça-o retornar uma visão que permita apenas leitura e disponibilize métodos de adição e remoção de objetos.



Motivação: Geralmente uma classe contém uma coleção de instâncias (um vetor, lista, conjunto ou matriz). Tais classes geralmente oferecem os métodos usuais **get** e **set** para a coleção.

Entretanto coleções devem usar um protocolo um pouco diferente: o método **get** não deve retonar a coleção de objetos, pois permite que clientes possam manipular o conteúdo da coleção sem a calsse "proprietária" fica sabendo. Esse método também revela muito aos seus clientes sobre as estruturas de dados internas do objeto. Um método **get** para um atributo multivalorado dever retornar algo que previna a manipulação da coleção pelo cliente e ocultar detalhes desnecessários sobre sua estrutura.

Não deve haver um método **set** para uma coleção: ao invés deve haver operações para adicionar e remover elementos. Isso dá ao objeto que mantém a coleção a responsabilidade de adicionar e remover elementos da coleção.

Com este protocolo a coleção está devidamente encapsulada, o que reduz o acoplamento da classe que a possui com os seus clientes.

Mecânica da refatoração:

- Adicione um método "adicionar" e "remover" para a coleção.
- Inicialize o campo com uma coleção vazia.
- Compile.
- Procure por clientes do método **set.** Ou modifique o método **set** para utilizar as operações "adicionar" e "remover" ou faça os clientes chamar essas operações.

  Set's *são usados em dois casos: quando a coleção está vazia e quando quem chama o método* ***set*** *está substituindo uma coleção inteira e não-vazia.*

  Você pode querer usar "Renomear método" para renomear o método set. Altere-o para algo como inicializar ou substituir.

- Compile e teste.
- Procure por todos os usuários do método **get** que modificam a coleção. Altere-os para utilizar os métodos "adicionar" e "remover". Compile e teste após cada mudança.
- Quando todas as utilizações do método get que realizam modificações tiverem sido substituídas, modifique o método **get** para que ele retorne uma visão da coleção de apenas leitura.

  Em Java 2, isso pode ser feita através de uma visão **unmodifiable.**

  Em Java 1.1, você deve retornar uma cópia da coleção.

- Compile e teste.
- Procure pelo usuários do método **get.** Procure por código que deveria estar alocado no objeto que hospeda a coleção. Utilize as operações "extrair método" e "mover método" para mover o código para o objeto em questão.

- Para Java 2, a refatoração encerra aqui. For Java 1.1, clientes podem preferir usar uma enumeração. Para prover a enumeração:
- Altere o nome do método **get** atual e adicione um novo método **get** para retornar uma enumeração. Procure por usuários do antigo método **get** e altere-os para usar um dos dois novos métodos.

  Se esse passo for muito grande, utilize "renomear método" no antigo método ***get,*** *crie um novo método que retorna uma enumeração, e altere os clientes para utilizar o novo método.*

- Compile e teste.

Exemplo

(Java 2)

Considere que uma pessoa realiza vários cursos. Cada curso é muito simples:

```
class Course...
  public Course (String name, boolean isAdvanced) {...};
  public boolean isAdvanced() {...};
```

A classe que representa cada pessoa é definida por (note a coleção de referências para cursos):

```
class Person...
  public Set getCourses() {
      return _courses;
  }
  public void setCourses(Set arg) {
      _courses = arg;
  }
  private Set _courses;
```

Com essa interface, cursos são adicionados com código similar a esse:

```
            Person kent = new Person();
            Set s = new HashSet();
            s.add(new Course ("Smalltalk Programming", false));
            s.add(new Course ("Appreciating Single Malts", true));
            kent.setCourses(s);
            Assert.equals (2, kent.getCourses().size());
            Course refact = new Course ("Refactoring", true);
            kent.getCourses().add(refact);
            kent.getCourses().add(new Course ("Brutal Sarcasm",
false));
            Assert.equals (4, kent.getCourses().size());
            kent.getCourses().remove(refact);
            Assert.equals (3, kent.getCourses().size());
```

Um cliente que queria saber sobre cursos avançados, pode fazê-lo da seguinte maneira:

```
            Iterator iter = person.getCourses().iterator();
            int count = 0;
```

```
            while (iter.hasNext()) {
               Course each = (Course) iter.next();
               if (each.isAdvanced()) count ++;
            }
```

Primeiro passo consiste em criar os modificadores para a coleção e compilar o código (destaque para a inicialização do conjunto):

```
  class Person
    public void addCourse (Course arg) {
        _courses.add(arg);
    }
    public void removeCourse (Course arg) {
        _courses.remove(arg);
    }

    private Set _courses = new HashSet();
```

Posteriormente deve-se atualizar o método que escreve no conjunto. Isso se faz olhando para os clientes do método **set** e verificando o quanto eles são utilizados. Há dois casos: o caso mais simples o cliente usa o método **set** para inicializar os valores zerados do conjunto. Neste caso altera-se o código do método **set** para que ele utilize o método **add.** Posteriormente deve-se renomear o método adequadamente para mostrar o seu propósito.

```
  class Person...
    public void initializeCourses(Set arg) {  //anteriormente era
setCourses
        Assert.isTrue(_courses.isEmpty());
        Iterator iter = arg.iterator();
        while (iter.hasNext()) {
            addCourse((Course) iter.next());
        }
     }
```

Não se pode simplesmente atribuir o conjunto, ainda que o conjunto anterior esteja vazio. Se o cliente simplesmente criar um conjunto e usar o método **set,** pode ocorrer de utilizar o método **add** para modificar o conjunto depois de tê-lo passado como parâmetro, o que viola o encapsulamento. Deve ser feita então uma cópia. Se o cliente simplesmente criar o conjunto e utilizar o método **set** (que inicializa o conjunto), é possível chamar os métodos **add** e **remove** diretamente, eliminando a necessidade do método **set.** Deste modo um código que antes fazia uso do método **set:**

```
            Person kent = new Person();
            Set s = new HashSet();
            s.add(new Course ("Smalltalk Programming", false));
            s.add(new Course ("Appreciating Single Malts", true));
            kent.initializeCourses(s);
```

se torna

```
            Person kent = new Person();
            kent.addCourse(new Course ("Smalltalk Programming",
false));
            kent.addCourse(new Course ("Appreciating Single Malts",
true));
```

Agora é hora de olhar para os clientes do método **get**. Os primeiros que devem ser pesquisados são os clientes do método que modificam a coleção de objetos, como por exemplo:

```
kent.getCourses().add(new Course ("Brutal Sarcasm", false));
```

Deve-se alterar a chamada para que utilize o novo método de acesso e adição no conjunto:

```
kent.addCourse(new Course ("Brutal Sarcasm", false));
```

Uma vez que essa alteração foi feita para todos, é prudente que ninguém consiga realizar alterações no conjunto através do método **get**, o qual deve retornar um conjunto imodificável.

```
public Set getCourses() {
    return Collections.unmodifiableSet(_courses);

}
```

Neste ponto a coleção está encapsulada e ninguém consegue alterar seus objetos, há não ser os elementos da classe **person.**