REFATORAMENTOS

MAC 413 - Tópicos de Programação Orientada a Objetos

Rodrigo Mendes Leme
BCC 99 - IME - USP

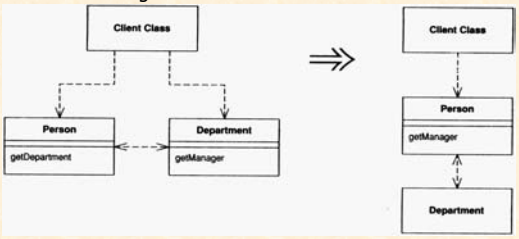
2 de outubro de 2002

Refatoramentos a serem abordados

- ***Hide Delegate (157)***
- ***Introduce Parameter Object (295)***
- ***Extract Superclass (336)***
- ***Form Template Method (345)***

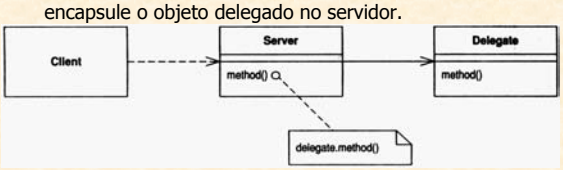
Hide Delegate (157)

- **Resumo:** um cliente está chamando uma classe delegada de um objeto. Crie métodos no servidor para ocultar o delegado.



Hide Delegate (157)

- **Motivação:** uma das chaves de POO é encapsulamento, pois facilita a implementação de mudanças. Se um cliente chama um método de um campo do servidor, o cliente precisa saber a respeito desse objeto delegado, diminuindo o encapsulamento. Elimine essa dependência criando um método que encapsule o objeto delegado no servidor.



Hide Delegate (157)

- **Mecânica:**
  - Para cada método no objeto delegado, crie um método no servidor.
  - Ajuste o cliente para chamar o servidor.
  - Compile e teste após ajustar cada método.
  - Se nenhum cliente precisa acessar mais o objeto delegado, remova o método que acessa o delegado no servidor.
  - Compile e teste.

Hide Delegate (157)

- **Exemplo:** começamos com uma pessoa e um departamento:

```
class Person {
  Departmento _department;

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

Hide Delegate (157)

Agora a classe *Departament*:

```
class Department {
  private String _chargeCode;
  private Person _manager;

  public Department(Person manager) {
    _manager = manager;
  }
  public Person getManager() {
    return _manager;
  }
...
```


Hide Delegate (157)

Para um cliente obter o gerente de uma pessoa, precisa do departamento primeiro:

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

Isso revela para o cliente a estrutura da classe *Departament*. Reduzimos esse acoplamento criando um método em *Person*:

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


Hide Delegate (157)

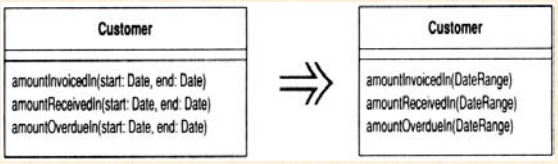
Agora mudamos todos os clientes de *Person* para usar esse novo método:

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

Finalmente, removemos o método *getDepartment* na classe *Person*.

Introduce Parameter Object (295)

- **Resumo:** você tem um grupo de parâmetros que naturalmente ficam juntos. Substitua-os por um objeto-parâmetro.



Introduce Parameter Object (295)

- **Motivação:** frequentemente um mesmo conjunto de parâmetros é utilizado por vários métodos. Substitua esses parâmetros por um objeto que carregue todos os seus dados (objeto-parâmetro).
- **Benefícios:**
  - Redução do tamanho das listas de parâmetros dos métodos.
  - Código fica mais consistente (devido à criação de métodos de acesso aos dados).
  - Comportamento que manipula os dados pode ser movido para a nova classe.

Introduce Parameter Object (295)

- **Mecânica:**
  - Crie uma nova classe, imutável, para representar o grupo de parâmetros a ser substituído.
  - Compile.
  - Use *Add Parameter (275)* para a nova classe. Passe *null* em todas as chamadas de métodos que a usam.
  - Para cada parâmetro do grupo de parâmetros, remova-o. Modifique os chamadas e o corpo do método para usar o objeto-parâmetro para aquele valor.

Introduce Parameter Object (295)

- **Mecânica (continuação):**
  - Compile e teste após remover cada parâmetro.
  - Depois de ter removido todos os parâmetros, procure comportamento que possa ser movido para o objeto-parâmetro, usando *Move Method (142)*.

Introduce Parameter Object (295)

- **Exemplo:** uma conta bancária e entradas:

```
class Entry...
  Entry(double value, Date chargeDate) {
    _value = value;
    _chargeDate = chargeDate;
  }
  Date getDate() { return _chargeDate; }
  Date getValue() { return _value; }

  private Date _chargeDate;
  private double _value;
```


Introduce Parameter Object (295)

Agora, a conta:

```
class Account...
  private Vector _entries = new Vector();
  double getFlowBetween(Date start, Date end) {
    double result = 0;
    Enumeration e = _entries.elements();
    while (e.hasMoreElements()) {
      Entry each = (Entry) e.nextElement();
      if (each.getDate.equals(start) ||
   each.getDate.equals(end) || (each.getDate().after(start)
   && each.getDate().before(end)))
        result += each.getValue();
    }
    return result;
  }
```

Cliente:
```
double flow =
anAccount.getFlowBetween(startDate,
endDate);
```


Introduce Parameter Object (295)

Criamos uma classe, imutável, para as duas datas:

```
class DateRange {
  DateRange(Date start, Date end) {
    _start = start;
    _end = end;
  }
  Date getStart() { return _start; }
  Date getEnd() { return _end; }

  private final Date _start;
  private final Date _end;
```


Introduce Parameter Object (295)

Adicionamos a nova classe na lista de parâmetros:

```
class Account...
  double getFlowBetween(Date start, Date end, DateRange
   range) {
    double result = 0;
    Enumeration e = _entries.elements();
    while (e.hasMoreElements()) {
      Entry each = (Entry) e.nextElement();
      if (each.getDate.equals(start) ||
   each.getDate.equals(end) || (each.getDate().after(start)
   && each.getDate().before(end)))
        result += each.getValue();
    }
    return result;
  }
```

Cliente:
```
double flow =
anAccount.getFlowBetween(startDate,
endDate, null);
```


Introduce Parameter Object (295)

Removemos o parâmetro *start* e usamos o novo objeto:

```
class Account...
  double getFlowBetween(Date end, DateRange range) {
    double result = 0;
    Enumeration e = _entries.elements();
    while (e.hasMoreElements()) {
      Entry each = (Entry) e.nextElement();
      if (each.getDate.equals(range.getStart()) ||
   each.getDate.equals(end) ||
   (each.getDate().after(range.getStart()) &&
   each.getDate().before(end)))
        result += each.getValue();
    }
    return result;
  }
```

Cliente:
```
double flow =
anAccount.getFlowBetween(endDate, new
DateRange(startDate, null));
```

Introduce Parameter Object (295)

Agora removemos o parâmetro *end*:

```
class Account...
  double getFlowBetween(DateRange range) {
    double result = 0;
    Enumeration e = _entries.elements();
    while (e.hasMoreElements()) {
      Entry each = (Entry) e.nextElement();
      if (each.getDate.equals(range.getStart()) ||
   each.getDate.equals(range.GetEnd()) ||
   (each.getDate().after(range.getStart()) &&
   each.getDate().before(range.getEnd())))
        result += each.getValue();
    }
    return result;
  }
```

Cliente:
```
double flow =
anAccount.getFlowBetween(new
DateRange(startDate, endDate));
```


Introduce Parameter Object (295)

Por fim, movemos comportamento para *DateRange*:

```
class Account...
  double getFlowBetween(DateRange range) {
    double result = 0;
    Enumeration e = _entries.elements();
    while (e.hasMoreElements()) {
      Entry each = (Entry) e.nextElement();
      if (range.includes(each.getDate()))
        result += each.getValue();
    }
    return result;
  }
```

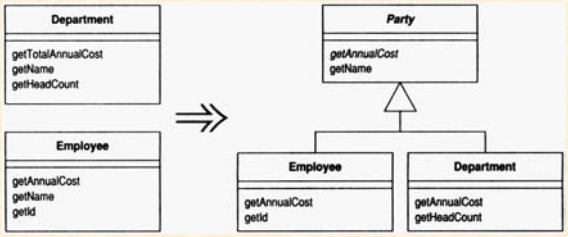

Introduce Parameter Object (295)

Em *DateRange*:

```
class DateRange...
  boolean includes(Date arg) {
    return (arg.equals(_start) ||
            arg.equals(_end) ||
            (arg.after(_start) && arg.before(_end)));
  }
```


Extract Superclass (336)

- **Resumo:** você tem duas classes com características similares. Crie uma superclasse e move as características comuns para essa superclasse.



Extract Superclass (336)

- **Motivação:** código duplicado é um dos principais problemas em sistemas, pois dificulta modificações. Uma forma de duplicação é ter duas classes que fazem coisas similares (da mesma maneira ou não), pois possuem interface e comportamento semelhante. Elimine o código duplicado através de herança.

Extract Superclass (336)

- **Mecânica:**
  - Crie uma superclasse abstrata em branco; torne as classes originais subclasses dessa superclasse.
  - Um por um, use *Pull Up Field (320)*, *Pull Up Method (322)* e *Pull Up Constructor Body (325)* para mover elementos comuns para a superclasse.
  - Compile e teste após cada passo.
  - Examine os métodos restantes nas subclasses, e tente aplicar *Extract Method (110)* seguido de *Pull Up Method (322)* ou *Form Template Method (345)*.

Extract Superclass (336)

- **Mecânica (continuação):**
  - Após mover os elementos em comum, cheque cada cliente das subclasses; se eles usam apenas a interface comum, mude o tipo requerido para a superclasse.

Extract Superclass (336)

- **Exemplo:** um empregado e um departamento:

```
class Employee...
  public Employee(String name, String id, int annualCost){
    _name = name;
    _id = id;
    _annualCost = annualCost;
  }
  public int getAnnualCost() { return _annualCost; }
  public String getId() { return _id; }
  public String getName() { return _name; }

  private String _name;
  private int _annualCost;
  prinvate String _id;
```


Extract Superclass (336)

Agora o departamento:

```
class Department ...
  private String _name;
  private Vector _staff = new Vector();

  public Department(String name) { _name = name; }
  public int getHeadCount() { return _staff.size(); }
  public Enumeration getStaff(){ return _staff.elements; }
  public void addStaff(Employee arg) {
    _staff.addElement(arg);
  }
  public String getName() { return _name; }
```


Extract Superclass (336)

Ainda o departamento:

```
public int getTotalAnnualCost() {
  Enumeration e = getStaff();
  int result = 0;
  while (e.hasMoreElements()) {
    Employee each = (Employee) e.nextElement();
    result += each.getAnnualCost();
  }
  return result;
}
```


Extract Superclass (336)

Criamos uma superclasse e estendemos as já existentes:

```
abstract class Party{}
class Employee extends Party...
class Department extends Party...
```

Aplicamos *Pull Up Field (320)*:

```
class Party...
  protected String _name;
```

Pull Up Method (320):

```
class Party {
  public String getName() { return _name; }
}
```


Extract Superclass (336)

Pull Up Constructor Body (325):

```
class Party...
  protected Party(String name) { _name = name; }
  private String _name;

class Employee...
  public Employee(String name, String id, int annualCost){
    super(name);
    _id = id;
    _annualCost = annualCost;
  }

class Department...
  Public Department(String name) { super(name); }
```

Extract Superclass (336)

Rename Method (325):

```
class Department extends Party {
  public int getAnnualCost() {
    Enumeration e = getStaff();
    int result = 0;
    while (e.hasMoreElements()) {
      Employee each = (Employee) e.nextElement();
      result += each.getAnnualCost();
    }
    return result;
  }
```


Extract Superclass (336)

Declaramos *getAnnualCost* abstrato na superclasse:

```
abstract public int getAnnualCost();
```

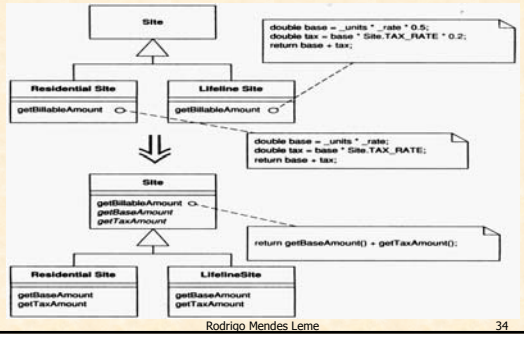
Por fim, checamos os clientes das duas classes em busca de referências à interface comum:

```
class Department...
public int getAnnualCost() {
    Enumeration e = getStaff();
    int result = 0;
    while (e.hasMoreElements()) {
      Party each = (Party) e.nextElement();
      result += each.getAnnualCost();
    }
    return result;
  }
```


Form Template Method (345)

- **Resumo:** dois métodos em subclasses executam passos similares na mesma ordem, mas mesmo assim os passos são diferentes. Coloque os passos em métodos com a mesma assinatura, de forma que os métodos originais tornem-se iguais. Então mande-os para a superclasse.

Form Template Method (345)



Form Template Method (345)

- **Motivação:** herança é uma ferramenta poderosa para eliminar comportamento duplicado. Quando dois métodos similares numa subclasse não são exatamente iguais, eliminamos a duplicação pegando a sequência de passos dos métodos, movendo-a para a superclasse e deixando polimorfismo assegurar que os passos diferentes executem código diferente. Esse tipo de método é chamado *Template Method* [GoF].

Form Template Method (345)

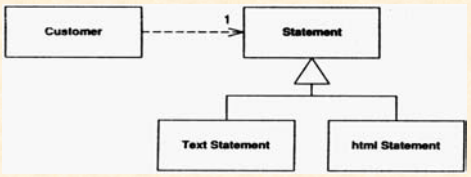
- **Mecânica:**
  - Decomponha os métodos de modo que todos os métodos extraídos sejam ou completamente idênticos ou completamente diferentes.
  - Use *Pull Up Method (322)* para mover os métodos idênticos na superclasse.
  - Para os métodos diferentes use *Rename Method (273)* de forma que as assinaturas de todos os métodos em cada passo sejam as mesmas.
  - Compile e teste após cada mudança de assinatura.

Form Template Method (345)

- **Mecânica (continuação):**
  - Use *Pull Up Method (322)* num dos métodos originais. Defina os métodos diferentes como abstratos na superclasse.
  - Compile e teste.
  - Remova os outros métodos, compile e teste após cada remoção.

Form Template Method (345)

- **Exemplo:** impressão de registros em ASCII e em HTML:



A classe *Statement* começa vazia:

```
class Statement {}
```


Form Template Method (345)

Classe *TextStatement*:

```
class TextStatement extends Statement {
  public String value(Customer aCustomer) {
    Enumeration rentals = aCustomer.getRentals();
    String result = "Rental Record for " +
                    "aCustomer.getName" + "\n";
    while (rentals.hasMoreElements()) {
      Rental each = (Rental) rentals.nextElement();
      result += "\t" + each.getMovie.getTitle() + "\t" +
              String.valueOf(each.getCharge()) + "\n";
    }
    result += "Amount owed is " + String.valueOf(
              aCustomer.getTotalCharge()) + "\n";
    return result;
  }
```


Form Template Method (345)

Classe *HTMLStatement*:

```
class HTMLStatement extends Statement {
  public String value(Customer aCustomer) {
    Enumeration rentals = aCustomer.getRentals();
    String result = "<H1>Rentals for <EM>" +
                    "aCustomer.getName" + "</EM></H1><P>";
    while (rentals.hasMoreElements()) {
      Rental each = (Rental) rentals.nextElement();
      result += "\t" + each.getMovie.getTitle() + ": " +
              String.valueOf(each.getCharge()) + "<BR>\n";
    }
    result += "<P>You owe <EM>" + String.valueOf(
              aCustomer.getTotalCharge()) + "</EM><P>\n";
    return result;
  }
```


Form Template Method (345)

Usamos *Extract Method (110)* no cabeçalho:

```
class TextStatement extends Statement {
  public String value(Customer aCustomer) {
    Enumeration rentals = aCustomer.getRentals();
    String result = headerString(aCustomer);
    while (rentals.hasMoreElements()) {
      Rental each = (Rental) rentals.nextElement();
      result += "\t" + each.getMovie.getTitle() + "\t" +
              String.valueOf(each.getCharge()) + "\n";
    }
    result += "Amount owed is " + String.valueOf(
              aCustomer.getTotalCharge()) + "\n";
    return result;
  }
```


Form Template Method (345)

Em *HTMLStatement* também:

```
class HTMLStatement extends Statement {
  public String value(Customer aCustomer) {
    Enumeration rentals = aCustomer.getRentals();
    String result = headerString(aCustomer);
    while (rentals.hasMoreElements()) {
      Rental each = (Rental) rentals.nextElement();
      result += "\t" + each.getMovie.getTitle() + ": " +
              String.valueOf(each.getCharge()) + "<BR>\n";
    }
    result += "<P>You owe <EM>" + String.valueOf(
              aCustomer.getTotalCharge()) + "</EM><P>\n";
    return result;
  }
```

Form Template Method (345)

Método *headerString* nas duas classes:

```
class TextStatement extends Statement...
  String headerString(Customer aCustomer) {
    return "Rental Record for " + "aCustomer.getName" +
           "\n";
  }

class HTMLStatement extends Statement...
  String headerString(Customer aCustomer) {
    return "<H1>Rentals for <EM>" + "aCustomer.getName" +
           "</EM></H1><P>";
  }
```


Form Template Method (345)

Usamos *Extract Method (110)* no rodapé:

```
class TextStatement extends Statement {
  public String value(Customer aCustomer) {
    Enumeration rentals = aCustomer.getRentals();
    String result = headerString(aCustomer);
    while (rentals.hasMoreElements()) {
      Rental each = (Rental) rentals.nextElement();
      result += "\t" + each.getMovie.getTitle() + "\t" +
              String.valueOf(each.getCharge()) + "\n";
    }
    result += footerString(aCustomer);
    return result;
  }
```


Form Template Method (345)

Em *HTMLStatement* também:

```
class HTMLStatement extends Statement {
  public String value(Customer aCustomer) {
    Enumeration rentals = aCustomer.getRentals();
    String result = headerString(aCustomer);
    while (rentals.hasMoreElements()) {
      Rental each = (Rental) rentals.nextElement();
      result += "\t" + each.getMovie.getTitle() + ": " +
              String.valueOf(each.getCharge()) + "<BR>\n";
    }
    result += footerString(aCustomer);
    return result;
  }
```


Form Template Method (345)

Método *footerString* nas duas classes:

```
class TextStatement extends Statement...
  String footerString(Customer aCustomer) {
    return "Amount owed is " + String.valueOf(
           aCustomer.getTotalCharge()) + "\n";
  }

class HTMLStatement extends Statement...
  String footerString(Customer aCustomer) {
    return "<P>You owe <EM>" + String.valueOf(
           aCustomer.getTotalCharge()) + "</EM><P>\n";
  }
```


Form Template Method (345)

Agora *Extract Method (110)* no corpo do método:

```
class TextStatement extends Statement {
  public String value(Customer aCustomer) {
    Enumeration rentals = aCustomer.getRentals();
    String result = headerString(aCustomer);
    while (rentals.hasMoreElements()) {
      Rental each = (Rental) rentals.nextElement();
      result += eachRentalString(each);
    }
    result += footerString(aCustomer);
    return result;
  }
```


Form Template Method (345)

Em *HTMLStatement* também:

```
class HTMLStatement extends Statement {
  public String value(Customer aCustomer) {
    Enumeration rentals = aCustomer.getRentals();
    String result = headerString(aCustomer);
    while (rentals.hasMoreElements()) {
      Rental each = (Rental) rentals.nextElement();
      result += eachRentalString(each);
    }
    result += footerString(aCustomer);
    return result;
  }
```

Form Template Method (345)

Método *eachRentalString* nas duas classes:

```
class TextStatement extends Statement...
  String footerString(Customer aCustomer) {
    return "\t" + each.getMovie.getTitle() + "\t" +
           String.valueOf(each.getCharge()) + "\n";
  }

class HTMLStatement extends Statement...
  String footerString(Customer aCustomer) {
    return "\t" + each.getMovie.getTitle() + ": " +
           String.valueOf(each.getCharge()) + "<BR>\n";
  }
```


Form Template Method (345)

Os dois métodos *value* ficaram iguais. Usamos *Pull Up Method (322)* num deles, o de *TextStatement*:

```
class Statement...
  public String value(Customer aCustomer) {
    Enumeration rentals = aCustomer.getRentals();
    String result = headerString(aCustomer);
    while (rentals.hasMoreElements()) {
      Rental each = (Rental) rentals.nextElement();
      result += eachRentalString(each);
    }
    result += footerString(aCustomer);
    return result;
  }
```


Form Template Method (345)

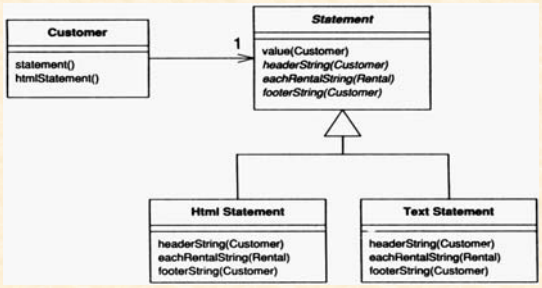
Declaramos os três métodos auxiliares das subclasses como abstratos:

```
class Statement...
  abstract String headerString(Customer aCostumer);
  abstract String footerString(Customer aCostumer);
  abstract String eachRentalString(Rental aRental);
```

Por fim, removemos *value* de *TextStatement* e *HTMLStatement*.

Form Template Method (345)

Resultado final:



Dúvidas?

- **Fonte:** *Refactoring: Improving the Design of Existing Code* - Martin Fowler, et al. - Addison-Wesley.

E-mail: leme@linux.ime.usp.br