Orientação a Objetos (Parte 2)
Enviado por Quarta, agosto 14 @ 14:43:22 EST por **Erro! A referência de hyperlink não é válida.**

http://www.portaljava.com/home/modules.php?name=Search&query=&topic=2Orientação a Objetos (Parte 2)

5.2.2 Usando gerentes de disposição da apresentação

Uma das características da orientação a objetos é a postergação de processamento para o momento da execução. Essa característica permite uma maior flexibilidade às aplicações orientadas a objeto.

O nosso applet exemplo deve ser capaz de se adaptar as várias interfaces gráficas de diferentes plataformas. Por exemplo, os botões da tela são apresentados de maneira diferente em diferentes ambientes. O ambiente Open Look da Sun apresenta botões arredondados em forma de elipse, o Microsoft Windows apresenta botões quase retangulares enquanto que o iMac da Apple apresenta os menores botões.

Em um mesmo ambiente, as cores, tamanhos e fontes dos componentes de tela podem ser alteradas, algumas vezes pelo próprio usuário. Por exemplo, o Microsoft Windows oferece opções de acessibilidade que alteram radicalmente cores e tamanhos dos componentes de tela. Dessa forma, não é possível determinar as coordenadas e dimensões físicas para cada componente da tela, a não ser durante a execução do applet.

```
package com.intexnet.phone;

import java.awt.*;
import java.applet.*;
import java.net.*;
import java.io.*;

public class PhoneApplet extends Applet {
  PhoneModel phonemodel;
  public void ClearData(){…}
  public void init(){…}

  java.awt.TextField t1;
  java.awt.Choice c1;

  java.awt.TextField t2;
  java.awt.TextField c2;
  java.awt.TextField tx;
  java.awt.TextArea t4;
  java.awt.Button button1, button2, button3, button4, button5;
  java.awt.Label label1;
  java.awt.Label label2;
  java.awt.Label label3;
  java.awt.Label label4;
  java.awt.Label label6;
  java.awt.List list1;
```

```
  public boolean handleEvent(Event event) {…}
  void button1_ActionPerformed(java.awt.Event event) {…}
  void button2_ActionPerformed(java.awt.Event event) {…}
  void button3_ActionPerformed(java.awt.Event event) {…}
  void button4_ActionPerformed(java.awt.Event event) {…}
  void button5_ActionPerformed(java.awt.Event event) {…}
}

class PhoneModel {
  PhoneNet phonenet;
  public int validate(){…}

  public void clearPhones() {...}
  public void removePhone(String area, String phone) {...}
  public void addPhone(String area, String phone) throws PhoneException {
    ...
    throws PhoneTooManyPhonesException();
    ...
    throws PhoneAlreadyInCallListException();
    ...
  }
  public void send(...) {...}
}

class PhoneNet {
  public Socket soquete;
  public DataInputStream in;
  public DataOutputStream out;
  int   portnumber;
  String host;
  String linex;
  public boolean ListenServer() {…}
  public void CloseCom(){…}
}
```

Figura 5.2 – Novas classes definidas após o primeiro passo da reengenharia.

Para solucionar esse problema, Java oferece os gerentes de disposição (layout managers)[KOS 98]. Os gerenciadores permitem a organização dos componentes para a sua exibição através de atributos lógicos ao invés de coordenadas fixas. Existem em Java quatro tipos de gerenciadores de Layout:

- FlowLayout
- BorderLayout
- GridLayout
- GridBagLayout

Determinadas classes de Java possuem uma apresentação visual e agrupam outros elementos visuais. Essas classes são chamadas de contentores (*containers*). Cada contentor é controlado por um gerente. Os contentores usam o *flowlayout* como gerente *default*.

O FlowLayout, gerenciador mais simples, garante que os componentes serão exibidos na ordem em que foram inseridos. O FlowLayout coloca um componente ao lado do outro enquanto houver espaço na tela, e então coloca o próximo componente logo abaixo e prossegue preenchendo a nova linha.

O BorderLayout já é um pouco mais elaborado. A área de exibição é dividida em quadrantes baseados em coordenadas geográficas ("North", "South", "East", "West", ou "Center"). No momento de inserção dos componentes é possível dizer em que quadrante ele será inserido. Por exemplo, o método abaixo cria um Panel com dois componentes que serão apresentados na tela um sobre o outro (Figura 5.3).

```
private Panel doMakeMessagePanel() {
  Panel p = new Panel();

  p.setLayout(new BorderLayout());
  p.add("North", new Label("Norte"));
  p.add("Center", new Label("Centro"));
  p.add("South", new Label("Sul"));

  return p;
}
```

Figura 5.3 – Exemplo do uso do BorderLayout.

O GridLayout permite a definição de um esquema de divisão da área de exibição em uma grade de células de mesmo tamanho. Os componentes são então colocados e redimensionados em relação a essas células.

O GridBagLayout é o mais sofisticado dos quatro tipos de gerenciadores. Apesar de extremamente poderoso, permitindo construir praticamente qualquer interface, devido a sua complexidade o seu uso não é trivial.

Ao invés do uso do GridBagLayout, recomenda-se o uso de composição de gerentes. Por exemplo, deseja-se obter uma área de texto sob um conjunto de botões dispostos em linhas. Para isso podemos utilizar o GridBagLayout ou a composição de gerentes mais simples (Figura 5.4, Figura 5.5).

```
import java.awt.*;
import java.applet.*;

public class Gerentes extends Applet {
  public void init() {
    Panel menu = new Panel();

    menu.setLayout(new FlowLayout());
    menu.add(new Button("Novo"));
    menu.add(new Button("Abrir"));
    menu.add(new Button("Salvar"));

    setLayout(new BorderLayout());
    add("North", menu);
    add("Center", new TextArea());
  }
}
```

Figura 5.4 – Usando composição de gerentes.

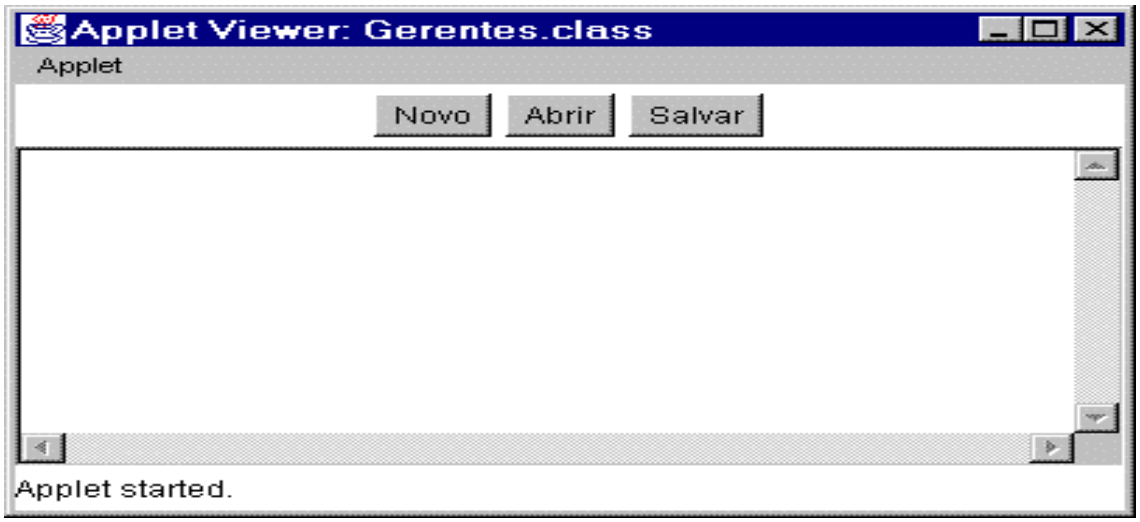

Figura 5.5 – Resultado da composição de gerentes.

Os gerentes oferecem um ótimo exemplo da implementação de mais um padrão clássico descrito por Gamma: o padrão estratégia.

5.2.3 Gerentes de disposição e o padrão estratégia

O padrão clássico “Estratégia” [GAM 95] é utilizado quando se deseja encapsular uma seqüência de passos em uma classe. Esse padrão dá origem a implementação dos gerentes de disposição de elementos de interface da linguagem Java (layout managers).

Elementos envolvidos no padrão Gerente de disposição:

- **Contentor** – um objeto que necessita dispor visualmente seus subcomponentes
- **Gerente de disposição** – um objeto que representa o algoritmo de disposição de elementos
- **Componente** – um objeto cuja posição e dimensões são calculadas durante a disposição do contentor.

A função do gerente de disposição (LayoutManager) é negociar entre o contentor (Container) e o componente de tela (Component) o espaço disponível e a disposição dos elementos. Essa negociação é feira através de métodos disponíveis em cada classe. Note que as classes Panel e Applet são exemplos de contentores e as classes Button, TextField e TextArea são exemplos de componentes.

A diferença entre os gerentes de disposição é apenas a forma como o algoritmo de disposição usa essas informações para decidir o local e o tamanho de cada componente na tela.

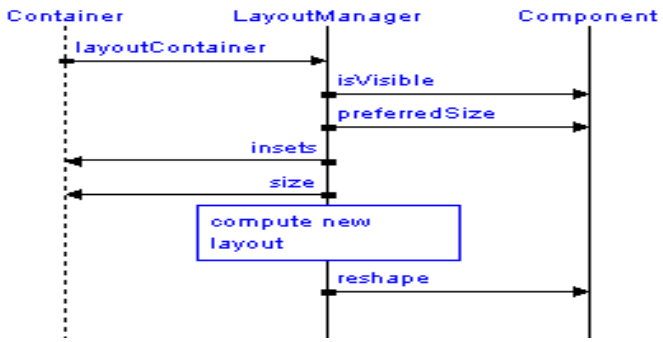

Figura 5.6 - Cenário (role model) do padrão Estratégia.

O padrão estratégia possui muitas aplicações. Por exemplo, poderíamos implementar diferentes algoritmos de ordenação de vetores como diferentes estratégias de ordenação.

5.2.4 Criando novos componentes

Uma das estratégias para tornar o código mais legível e facilitar o reuso é a identificação de códigos repetidos. Isolando esses trechos de códigos é possível criar novos componentes que podem inclusive ser reutilizados em outras aplicações.

O código abaixo utilizado para limitar o comprimento de um campo de texto se repete para cada componente, apenas com a variação de alguns parâmetros. No trecho abaixo, a variável **t1**, colocada em negrito, é o campo de texto sendo limitado. A expressão sublinhada é o teste de validação. A variável *t2* é o próximo componente que irá receber o foco quando o usuário pressionar a tecla enter.

```
// limita o comprimento máximo do textfield t1 em 14 caracteres
if (event.target == t1 && event.id == Event.KEY_PRESS) {
  if ( (t1.getText().length() > 14) || (Validate() >= 150) )
     return (event.key != '_');
  else
     if (event.key==' ')
       t2.requestFocus();
     else
       return false;
}
```

Figura 5.7 – Teste de consistência de cada campo de texto.

Poderíamos transformar esse trecho em um método da classe PhoneApplet e usar chamadas apropriadas a esse método para obter o mesmo resultado que a implementação atual. Entretanto, essa nossa solução ficaria limitada ao código dessa classe.

Poderíamos criar uma classe derivada de Applet, digamos BoundApplet, que oferecesse esse método. Então usaríamos essa nova classe no lugar da classe java.awt.Applet sempre que desejássemos um applet com restrições sobre

campos de texto. Contudo, essa solução também não é satisfatória, pois um campo de texto pode aparecer em outros contentores além do Applet. A melhor alternativa é criar uma classe nova que ficará encarregada de controlar o comprimento de um campo de texto (Figura 5.8). A classe BoundTextFieldListener possui três atributos. *textField* é o componente sendo controlado. *nextComponent* é o próximo componente que será acionado quando o usuário pressionar a tecla Enter e *length* é a largura máxima permitida para o valor do campo de texto.

```
class BoundTextFieldListener {
  TextField textField;
  Component nextComponent;
  int length;
  public BoundTextFieldListener(TextField tf, Component nc, int l) {
    textField = tf;
    nextComponent = nc;
    length = l;
  }
  public boolean validate(char key, boolean flag) {
    if (textField.getText().length() > length || flag) {
      return (key != '_');
    }
    if (key == ' ') {
      nextComponent.requestFocus();
      return true;
    }
    return false;
  }
}
```

Figura 5.8 – As classes BoundTextFieldListener e BoundNunericFieldListener.

De forma análoga, para controlar os campos do tipo número de telefone foi criada uma nova classe que inclui a restrição de que o caracter pressionado seja um dígito.

```
private BoundTextFieldListener  t1Bound = new
BoundTextFieldListener(tName, tTargetPhone, 14);
...
boolean validInput = (phone.Validate() > 150);
...
       if (target == tName) {
         processed = t1Bound.validate(key, validInput);
...
}
```

Figura 5.9 – Controlando o comprimento de campos de texto.

5.3 MODELO

Esta classe fica encarregada de armazenar os dados da mensagem, por este motivo, ela apresenta o maior número de atributos diretamente relacionados com o problema.
Os métodos desta classe apenas inserem e consultam valores dos atributos. Um método especial efetua o teste da validade do estado interno da classe. Um último método adicional inicia a transmissão da mensagem.

```
class PhoneModel {

public int validate(){…}
public void clearPhones() {...}

public void removePhone(String area, String phone) {...}

public void addPhone(String area, String phone) throws PhoneException
{
   ...
   throws PhoneTooManyPhonesException();
   ...
   throws PhoneAlreadyInCallListException();
   ...
 }

 public void send(...) {...}

}
```

Figura 5.10 – A classe PhoneModel.

Note que os métodos correspondem as operações possíveis da aplicação. A classe PhoneApplet somente necessita usar chamadas adequadas a esses métodos para executar a montagem e o envio de mensagem. Observe o aparecimento de exceções para sinalizar a ocorrência de situações anormais durante a execução.

```
try {
   if (event.id == Event.ACTION_EVENT) {
      if (target == btnSend) {
         phonemodel.send(...);
      } else if (target == btnClear) {
         phonemodel.clear();

      ...

      } else if (target == button5) {
         phonemodel.addPhone(...);
      ...
```

```
} catch (PhoneAlreadyInCallListException e1) {...}

  catch (PhoneTooManyPhonesException e2) {...}
}
```

Figura 5.11 – Fragmento da classe PhoneApplet acionando métodos do nível da aplicação.

5.4 INTEGRAÇÃO DE SISTEMAS

A programação da conexão sockets pode ser particionada em diversos métodos dividindo o processamento de uma operação. Por exemplo, a operação de envio de uma mensagem do applet para o servidor pode ser dividida em métodos que executam cada etapa do envio como estabelecimento de conexão, montagem da mensagem, envio da mensagem e encerramento da conexão. Note que o único método da classe público é *send()* os demais são de uso exclusivo da própria classes. Essa técnica mantém a interface simples e esconde a complexidade envolvida na implementação da classe.

```
class PhoneNet {

   private Socket soquete;
   private DataInputStream in;
   private DataOutputStream out;
   private int portnumber;

   private String host;
   private String linex;

   public void send (String name, String sourceAreaPhone,
String targetAreaPhone, String message) throws PhoneException {...}
   private void open() throws PhoneException {...}
   private void validate() throws PhoneException {...}
   private void send(String m) throws Exception {...}
   private void send() throws PhoneException {...}
   private void listen() throws PhoneException {...}

   private void close() throws PhoneException {
      try {
         soquete.close();
         out.close();
         in.close();
      } catch (Exception e) {
         throw new PhoneCloseException();
      } finally {
         clear();
      }
   }
}
```

Figura 5.12 – A classe PhoneNet.

A conexão socket é sujeita a falhas. Por essa razão é necessário tratar das condições de erro através de exceções. Quando um método encontra uma condição anormal (condição de exceção) e não sabe como tratá-la, o método pode gerar uma exceção (*throw an exception*) [VEN 98b]. A geração de uma exceção indica que há um problema que não pode ser tratado no método em que ele foi detectado. Exceções devem ser tratadas (*catch an exception*) por tratadores (*handlers*). Em Java a forma de realizar esse tratamento é através da definição de um bloco *try-catch*. No momento em que é gerada uma exceção dentro do bloco *try* (através de um comando *throw*), essa exceção cria uma instância de um objeto derivado da classe Throwable (veja Figura 5.13), o objeto que indica a exceção pode ser analisado dentro do bloco *catch* e as medidas de tratamento de exceção podem ser realizadas.

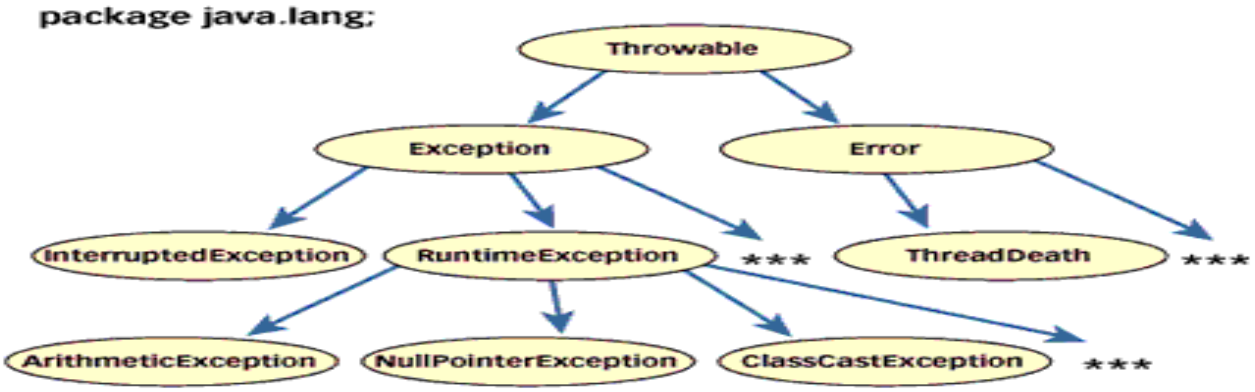

Figura 5.13 – Algumas subclasses de Throwable.

É importante notar que além de gerar exceções com as subclasses pré-definidas de Throwable, é possível definir classes específicas para o tratamento de exceções em um dado programa.

O uso de exceções tem como principal vantagem a separação do tratamento de erro do código de execução normal. Deve-se usar exceções para tratar de situações que devem ocorrer com pouca freqüência. [VEN 98c] estabelece como uma regra que deve-se evitar o uso de exceções para indicar condições que podem ser esperadas como parte do funcionamento típico de um método.

No entanto, em alguns casos o uso de exceções pode ser feito para facilitar a legibilidade do código. No nosso applet as exceções de Java são substituídas por exceções da própria aplicação. A implementação do método *close()* demonstra como isso pode ser feito. A classe Phone que aciona o método *send()* é que deve se preocupar com a interpretação da exceção.

```
try {
  phonenet.send(...);
  showStatus("Sua mensagem foi recebida e está sendo processada. Obrigado.");
} catch (PhoneInvalidDataException e){
  showStatus("Dados inválidos...");
} catch (PhoneException e){
  showStatus("Não foi possível enviar sua mensagem...tente novamente");
}
```

Figura 5.14 – Fragmento da classe Phone e o tratamento de exceções de PhoneNet.

6 VISÃO GERAL

Após a reengenharia, foram obtidas novas classes que trabalham em cooperação para fornecer as funcionalidades da aplicação.

As novas classes podem ser utilizadas na programação de outras aplicações. Por exemplo, as duas classes BoundTextFieldListener e BoundNumericFieldListener podem ser associadas com campos de texto de outras aplicações. Essas duas classes formam um pacote de classes de aplicação genérica. Por exemplo, a maioria das aplicações de entrada de dados necessitam de rotinas de validação. Essas classes podem ser facilmente estendidas para atender a outros tipos de entrada como datas, idades, valores monetários etc.

A divisão em classes específicas facilita também a alteração da aplicação. Por exemplo, poderíamos reescrever a classe PhoneNet para utilizar o mecanismo RMI, no lugar de sockets, sem causar alterações nas demais classes. De forma semelhante, poderíamos criar um interface (PhoneApplet) com apresentação totalmente diferente, para atender um novo público ou para ser inserida em outra página da companhia.

Por fim, observe que as classes específicas da aplicação ficaram reduzidas ao mínimo. Por outro lado, as demais classes criadas operam em conjunto complementado-se, de forma semelhante ao que foi observado nas classes da biblioteca Java.

A nova biblioteca de classes permitirá ao desenvolvedor encontrar um número maior de funcionalidade já disponíveis em sua caixa de ferramentas, prontas para o reuso.

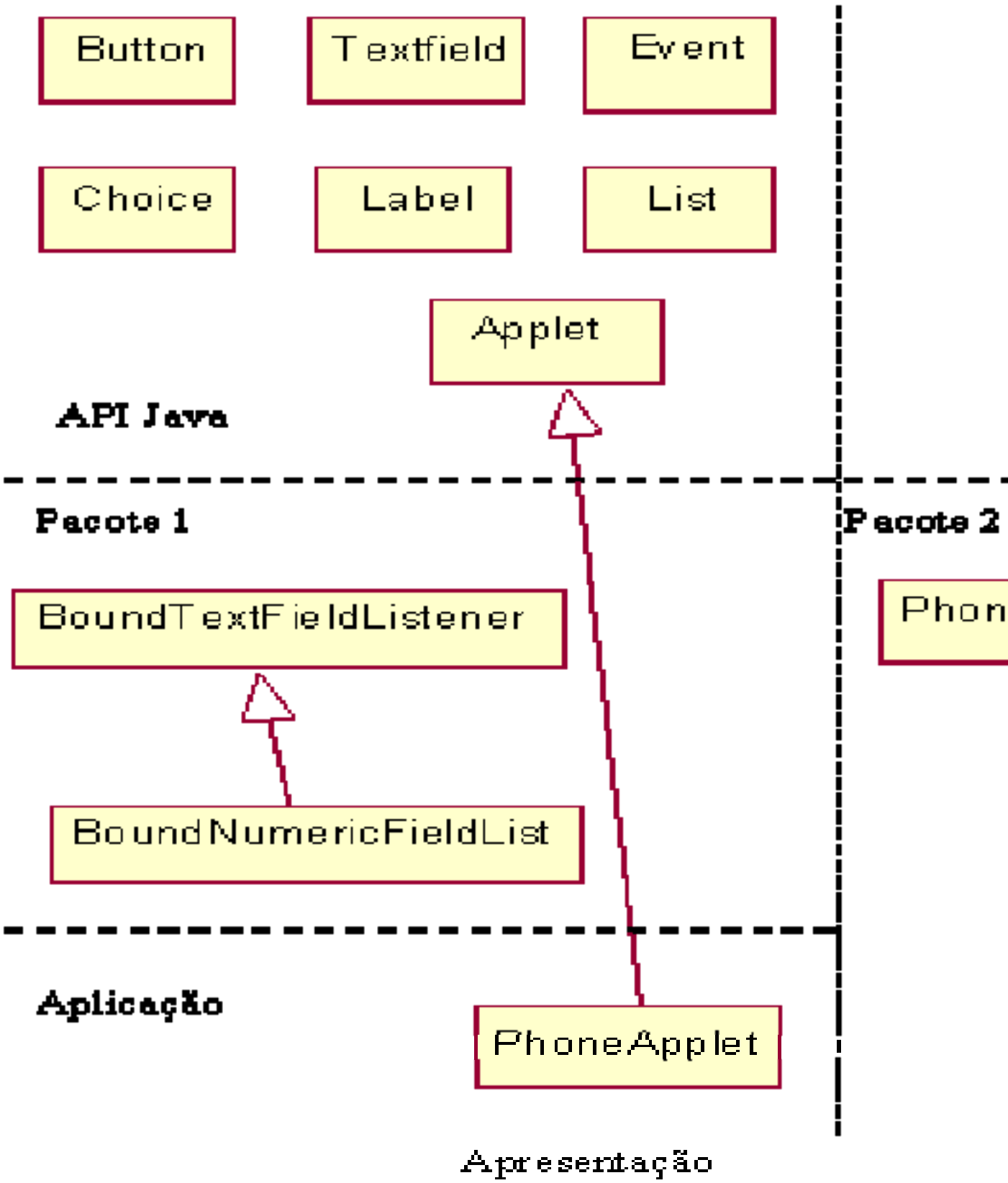
Button
Textfield
Event
Choice
Label
List
Applet
API Java
Pacote 1
Pacote 2
BoundTextFieldListener
Phon
BoundNumericFieldList
Aplicação
PhoneApplet
Apresentação

Figura 6.1 – Classes reusáveis criadas após a reengenharia da aplicação Phone.

7 CONCLUSÃO

O processo de desenvolvimento apresentado neste tutorial é totalmente empírico. As fases apresentadas foram registradas a partir da prática no decorrer do desenvolvimento da aplicação.

O processo de desenvolvimento da aplicação da telefônica foi dividido em três etapas (Figura 7.1). Na etapa de prototipação, é desenvolvido um programa limitado em suas funcionalidades que é usado para convencer o cliente da viabilidade da aplicação e obter a liberação de recursos necessários para a conclusão do sistema.

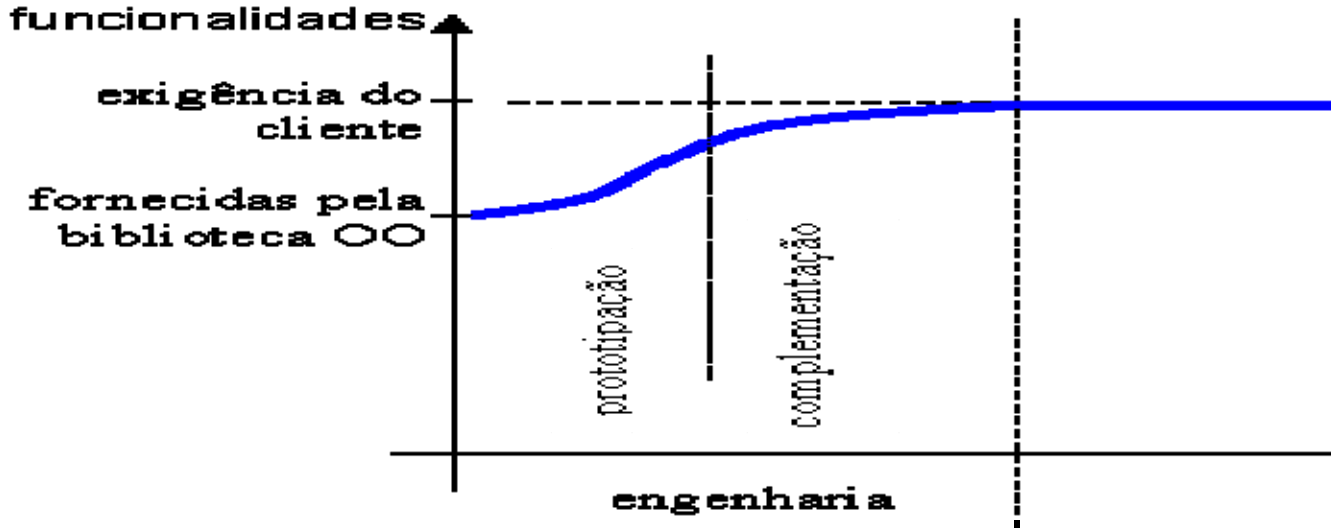

Figura 7.1 – As três etapas do processo de desenvolvimento.

Na segunda etapa, o protótipo passa por um processo de refinamentos sucessivos até que atinja um conjunto de funcionalidades que atenda a exigência definida pelo cliente. O papel da orientação a objetos nessa primeira fase é oferecer um ponto de partida adiantado em que boa parte das funcionalidades já se encontram disponíveis através de classes reusáveis. Na terceira etapa, a aplicação que já atende a exigência do cliente, é reorganizada de forma a proporcionar um novo conjunto de classes reusáveis que permitirão um novo ponto de partida adiantado para uma nova aplicação.

Outras considerações importantes podem ser extraídas da experiência dessa aplicação exemplo. Durante a fase de engenharia, ou seja, antes de se atingir as funcionalidades exigidas pelo cliente, os parâmetros internos de qualidade de produto de software variam de forma descontrolada. Por exemplo, a coesão dos métodos aumenta e diminui a medida que novas funcionalidades são inseridas e concluídas. Na etapa de reengenharia, a coesão é aumentada em refinamentos sucessivos. Esse aumento de coesão é acompanhado pelo aumento do número de métodos nas classes da aplicação. Em contrapartida, o

tamanho dos métodos diminui. Esses três indicadores aumentam a qualidade do produto de software e facilitam a sua manutenção (Figura 7.2).

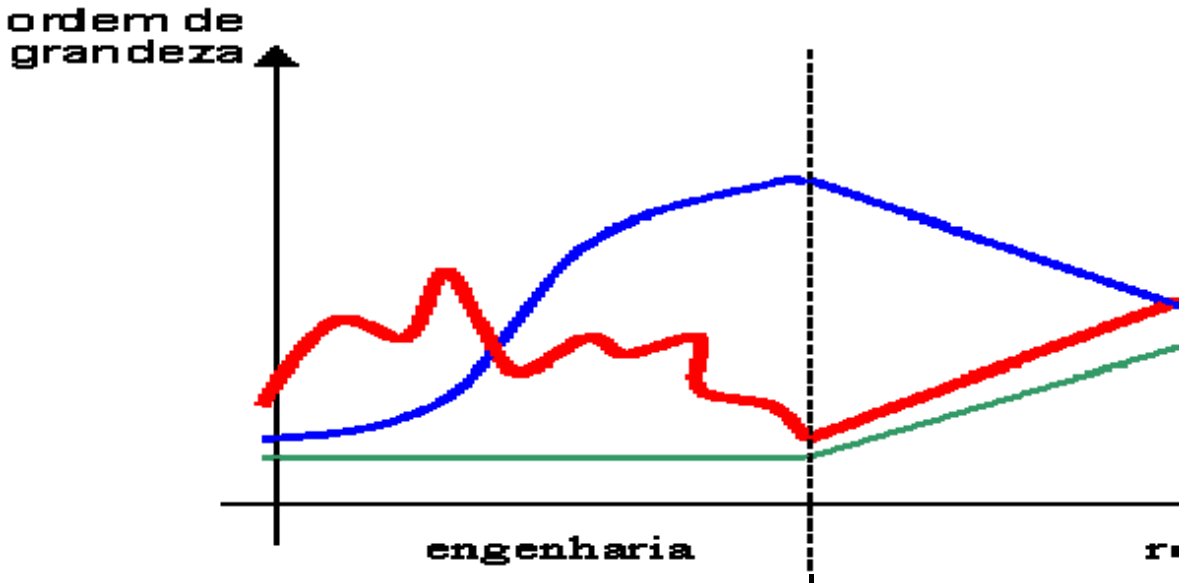

Figura 7.2 – Aumento da coesão.

Durante as duas etapas da engenharia, outra medida da qualidade de software, o acoplamento do sistema (Figura 7.3), varia durante a engenharia influenciado também pela inclusão de novas funcionalidades e refinamentos.

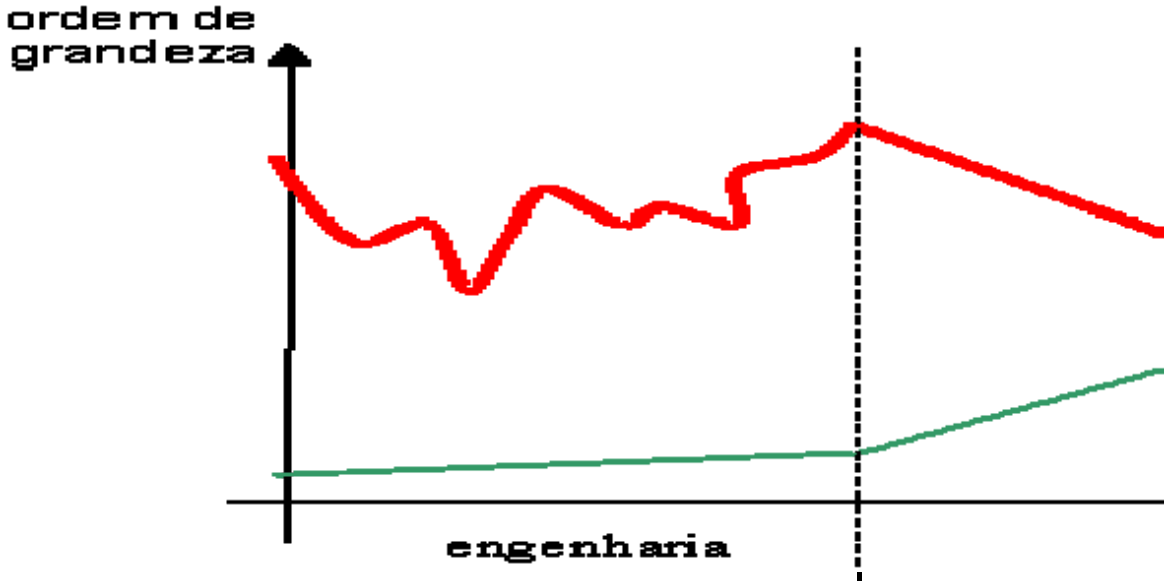

Figura 7.3 – Diminuição do acoplamento.

Ainda nesta fase, observa-se um número constante de classes que vão recebendo um número cada vez maior de métodos e campos. Poucas são as interfaces e eventos criados nestas duas primeiras etapas. Na fase de reengenharia, o acoplamento do sistema se reduz através da criação de novas

classes, interfaces e eventos. O reduzido acoplamento entre os componentes da aplicação facilita o reuso das classes Dessa forma, a reengenharia atende seus dois principais objetivos: possibilitar o reuso e facilitar a manutenção.

Neste tutorial foram apresentados diversos aspectos do desenvolvimento orientado a objetos, a partir de uma experiência prática de desenvolvimento de software comercial, com o uso da linguagem Java.

Uma linguagem de programação encontra-se inserida em um contexto econômico. A linguagem é uma ferramenta que auxilia o desenvolvedor na tarefa de construir aplicações. Essas aplicações são utilizadas por um cliente para obter uma vantagem sobre seus competidores ou para facilitar o seu próprio trabalho.

Essas aplicações devem atender da melhor forma possível às necessidades do cliente. Entretanto, existem vários fatores que limitam o desenvolvimento de software, entre eles, o tempo disponível para a entrega do sistema, disponibilidade de recursos humanos capacitados, domínio da tecnologia e equipamentos.

A vantagem oferecida pela orientação a objetos é a possibilidade de reutilizar soluções. As bibliotecas de classes das linguagens orientadas a objeto e as ferramentas visuais permitem ao desenvolvedor criar aplicações de alta qualidade, em um tempo reduzido. Mas isso é apenas uma parte das vantagens da orientação a objetos.

Linguagens orientadas a objeto associadas com novas técnicas de desenvolvimento possibilitam a criação novas bibliotecas de classes a partir da experiência prática do desenvolvedor.

Este tutorial utilizou a linguagem de programação Java, associada com o uso de padrões e técnicas de programação orientadas a objeto para demonstrar a possibilidade da criação de um pequeno pacote de classes reutilizáveis. Existem muitas outras linguagens orientadas a objeto, bem como diversas outras técnicas que podem ser utilizadas para o desenvolvimento de aplicações. Mesmo os padrões comentados são apenas uma pequena amostra da diversidade de padrões disponíveis na literatura.

A principal contribuição da linguagem Java foi de levar ao grande público um conjunto de técnicas e mecanismos que já vinham sendo usados por diversas outras linguagens de programação.

Atualmente, as aplicações atingiram um nível de complexidade nunca antes alcançado. O desenvolvedor tem a responsabilidade de desenvolver sistemas que atendem a milhares de usuários. Além disso, as chances de que um programa seja reutilzado em outro contexto aumenta ainda mais a responsabilidade no desenvolvimento de aplicações com alta qualidade.

Dentre as diversas alternativas para o desenvolvimento de aplicações, a orientação a objetos é uma das mais promissoras. O maior desafio é encontrar é encontrar usos criativos para o imenso potencial desse paradigma.

8 REFERÊNCIAS BIBLIOGRÁFICAS

[ALE 77] ALEXANDER, Christopher; ISHIKAWA, Sara; SILVERSTEIN, Murray. **A Pattern Language:** Towns, Buildings, Construction. Oxford University Press, 1977. 1171p.

[BUR 92] BURBECK, Steve. **Applications Programming in Smalltalk-80: How to use Model-View-Controller (MVC)**. Available at http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html.

[COA 93] COAD, Peter; YOURDON, Edward. **Projeto Baseado em Objetos.** Campus, 1993. 195p.

[COA 96] COAD, Peter; MAYFIELD, Mark; NORTH, David. **Object Models:** Stategies, Patterns and Applications. Yourdon Press, 1996. 515p.

[DIK 97] DIKEL, David; HERMANSEN, Christy; KANE, David; MALVEAUX, Raphael. Organizational Patterns for *Software* Architecture. **Proceedings...** PLoP'97 Pattern Languages of Programming Conference. Available at http://st-www.cs.uiuc.edu/users/hanmer/PloP-97/Workshops.html .

[ECK 98] ECKEL, Bruce. **Thinking in Java.** Prentice Hall, 1998. 1098p.

[ECR 98] ECKSTEIN, Robert; LOY, Marc; WOOD, Dave. **Java Swing.** O'Reilly, 1998. 1232p.

[FLA 97] FLANAGAN, David. **Java in a Nutshell.** O'Reilly & Associates, 1997. 620p.

[FOW 96] FOWLER, Martin. **Analysis Patterns:** Reusable Object Models. Addisson-Wesley, 1996..

[FOW 97] FOWLER, Martin; SCOTT, Kendall; JACOBSON, Ivar. **UML Distilled:** Applying the Standard Object Modelling Language. Addison-Wesley, 1997. 179p.

[BUS 96] BUSCHMANN, Frank; MEUNIER, Regine; ROHNERT, Hans; SOMMERLAND, Peter; STAL, Michael. **Pattern-Oriented *Software* Architecture: A System of Patterns.** John Wiley & Sons, 1996. 457p.

[GAM 95] GAMA, Erich; HELM, Richard; JOHNSON, Ralph; VLISSIDES, John. **Design Patterns**: Elements of Reusable Object-Oriented *Software*. Addison-Wesley, 1995.

[GOL 95] GOLDBERG, Adele; RUBIN, Kenneth S. **Succeeding with Objects**. Addison-Wesley Publishing Company, 1995. 542p.

[GOO GOODMAN, Danny. **JavaScript Bible**. IDG Books Worldwide, 1998.

98]

[KOS 98] KOSKI-LAMMI, Samuli; TEEMAA, Teno; KARTAU, Olev. **Design Patterns in Java AWT**. Paper written to Design Pattern Course Tik-76.278, 1998. Available at http://mordor.cs.hut.fi/tik-76.278/group6/awtpat.html#layoutmanager

[MAN 96] MANSFIELD, Richard. **The Comprehensive Guide to VbScript**. Ventana Communications Group Inc, 1996.

[MEY 97] MEYERS, Scott. **Effective C++:** 50 Specific Ways to Improve your Programs and Designs. Addison-Wesley, 1997. 256p.

[OAK 99] OAKES, Scott; WONG, Henry. **Java Threads.** O'Reilly, 1999. 315p.

[POT 96] POTEL, Mike. **MVP: Model-View-Presenter The Taligent Programming Model for C++ and Java**, 1996. Available at http://www.ibm.com/java/education/mvp.html

[SRI 97] SRINIVASAN, Skiram. **Advanced Perl Programming.** O'Reilly, 1997. 434p.

[SUN 99] SUN. **Java Tutorial**. Available at http://java.sun.com/docs/books/tutorial/index.html.

[TOL 99] TOLKSDORF, Robert. Programming Languages for the Java Virtual Machine. Available at http://grunge.cs.tu-berlin.de/~tolk/vmlanguages.html

[VEN 98b] VENNERS, Bill. **Exceptions in Java**. JavaWorld, July 1998. Available at http://www.javaworld.com/jw-07-1998/jw-07- exceptions.html

[VEN 98c] VENNERS, Bill. **Designing with Exceptions**. JavaWorld, July 1998. Available at http://www.javaworld.com/jw-07-1998/jw-07-techniques.html

[WEI 94] WEINAND, André, GAMMA, Erich. **ET++ - a Portable, Homogenous Class Library and Application Framework**. BISCHOFBERGER, W.R.; FREI, H.P. (eds.): Proceedings of the UBILAB '94 Conference. Zurich, September 1994, pp.66-92. Available at http://www.ubs.com/e/index/about/ubilab/ext/publications/e_wei94.htm