Unidade 6: Arquitetura de Componentes

Prof. Daniel Caetano

Objetivo: Produzir e compreender documentações de componentes.

Bibliografia: DEITEL, 2005; HOFF, 1996

INTRODUÇÃO

Como já foi visto, componentes são blocos autônomos que compõem um sistema. O funcionamento de um componente pode ser simples ou complexo, envolvendo uma ou até centenas ou milhares de classes; o uso de um componente, entretanto, deve ser mantido o mais simplificadamente possível.

Na aula passada foram apresentados os primeiros detalhes que permitem a especificação da arquitetura de componentes de uma aplicação:

a) As interfaces
b) O conjunto de componentes da aplicação
c) Suas relações atruturais
d) As dependências entre eles

Nesta aula veremos com um pouco mais de detalhe como deve ser especificado um componente.

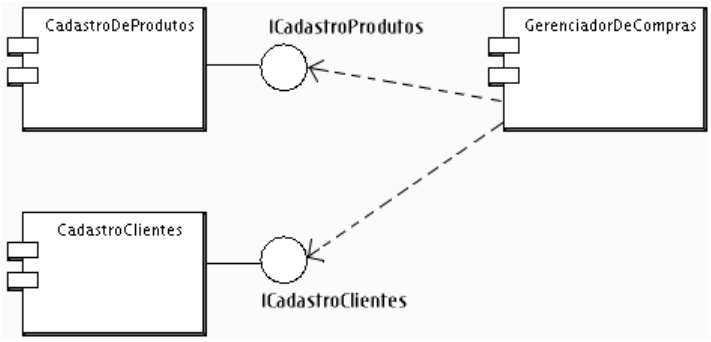
1. PROJETO DE ARQUITETURA

Um projeto de arquitetura de componentes de um sistema deve responder basicamente a três perguntas:

a) Quais são as partes do sistema?
b) Como combiná-las?
c) Qual o impacto das mudanças que possam ocorrer?

Em grande parte, os diagramas de classes vistos já apresentam estes elementos, em especial os diagramas que envolviam as interfaces, como Ilivro e IArquivo, que pode ser consideradas as interface de componentes muito simples.

Muitas vezes, entretanto, é preciso representar a interação entre diversos componentes. Esta representação de interação é feita conforme o diagrama a seguir, chamado de Diagrama de Componentes.



Neste diagrama temos uma representação simplificada dos componentes, na notação UML: uma caixa com o nome do componente. O componente que é utilizado por outros componentes usualmente possui uma interface, indicada pela bolinha com o nome da interface.

Um componente que use outros componentes é ligado à interface daquele componente por uma seta de dependência, indicando que mudanças na interface do componente usado pode obrigar mudanças no componente que o utiliza.

Este tipo de especificação indica que o elemento principal de um componente é, em geral, uma classe que implemente a interface de comunicação deste cliente, o que é chamado de **realização.**

Quando se define uma arquitetura de componentes, define-se também - **obrigatoriamente** - as interfaces destes componentes, para que um objeto ou outro componente possa interagir com esses componentes. A interface de um componente é altamente influenciada pelos **usos** que serão dados ao componente, ou seja, é bastante importante a especificação dos **casos de uso**.

A definição da interface define dois importantes contratos:

A) O contrato de uso
B) O contrato de realização

1.1. Contrato de Uso

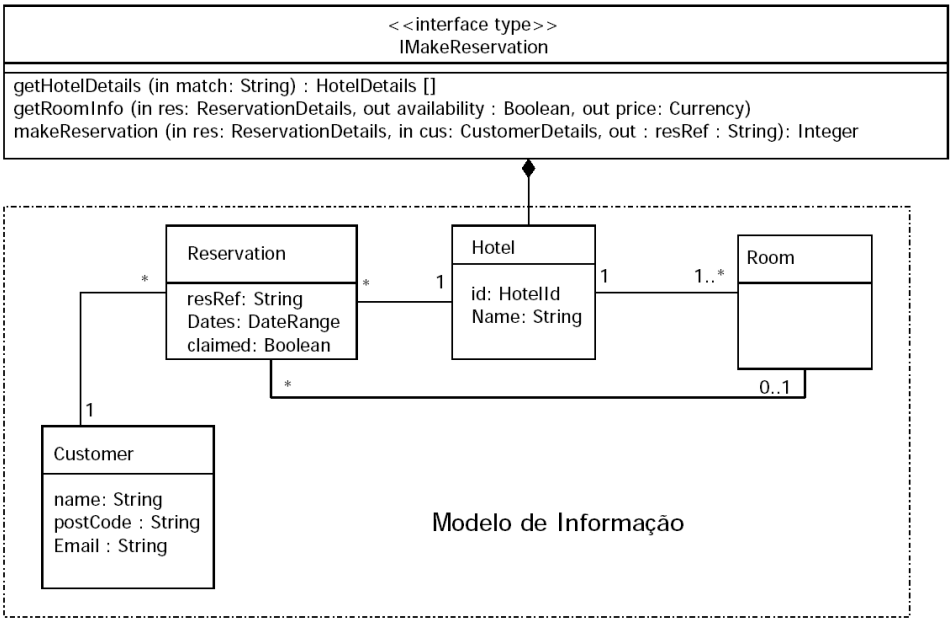
O Contrato de Uso é o que define como um componente é usado, isto é, o que ele faz e como ele pode ser usado para obter os resultados desejados.

Existem dois tipos de componentes: aqueles que o componente sempre reage de maneira igual, isto é, um componente "sem memória", que chamamos de "Stateless Components". Existem, porém, componentes mais complexos, que o resultado de uma operação depende do que aconteceu anteriormente, isto é, são componentes "com memória", que são chamaods "Statefull Components".

1.1.1. Stateless Components

Em alguns casos, os mais desejáveis, o uso do componente é simples: ele não guarda estado e, portanto, seu comportamento é sempre previsível. Neste caso, basta definir as **assinaturas das operações** e o **modelo de informação** (diagrama de classes de entidade).

As assinaturas são simplesmente os métodos públicos da interface: já vimos isso, é fácil. O modelo de informação é um pouco mais complexo, porque ele é composto de objetos de entidade, como especificado na figura a seguir (fonte: UML Components: Arquitetura de Componentes, de Patrícia Machado, UFCG, 2003).



Os objetos de entidade representados sob o retângulo "modelo de informação" indicam como os dados estão armazenados, isto é, quais são as informacões do tipo HotelDetails (Hotel), CustomerDetails (Customer) e ReservationDetails (Reservation), especificados na interface, e como elas se relacionam.

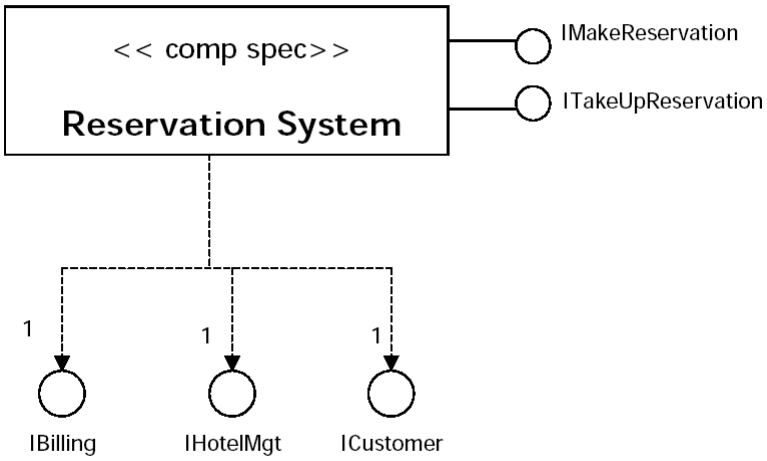
1.1.2. Statefull Components

Quando os componentes possuem estados, isto é, o resultado de uma operação pode variar drasticamente a depender das operações anteriores - como, por exemplo, a execução de uma instrução em um componente que emule um processador de computador - é preciso também especificar as **pré-condições** e as **pós-condições**.

As pré-condições e pós-condições explicitam o que é necessário fazer para que o componente se comporte de uma maneira esperada e o que é necessário fazer após algumas operações, para que ele continue se comportando de maneira esperada. A falha em seguir estas pré e pós-condições pode implicar em resultados imprevisíveis.

1.2. Contrato de Realização

O contrato de realização é basicamente a indicação de quais interfaces são oferecidas pelo componente e quais outras interfaces este componente usa, isto é, com que outros componentes ele interagem. A figura a seguir mostra este tipo de especificação, que deve ser seguida à risca pelo programador.



O componente, no caso, fornece as interfaces IMakeReservation e ITakeUpReservation, e usa outros componentes, através das interfaces IBilling, IHotelMgt e ICustomer.

2. GERANDO DOCUMENTAÇÃO USANDO JAVA

Uma vez que a função do código é explicar para o computador quais são as tarefas que ele deve cumprir, é natural que o código nem sempre seja facilmente compreendido pelos seres humanos, em especial aqueles que não estão totalmente "por dentro" do funcionamento do programa.

Tendo isto em mente, documentar um código é a técnica/arte de inserir comentários no código de maneira que ele seja mais facilmente compreendido por quem quer que o venha a ler. Trata-se de técnica porque existem algumas regras básicas a se seguir; trata-se também de uma arte, pois não há regras para definir tudo que precisa ser comentado, sendo essa uma tarefa deixada para o bom senso do desenvolvedor.

Os comentários, em java, podem ser especificados de três formas:

1) Comentários de uma linha: usa-se duas barras no início da linha:

```
// Este é um comentário de uma linha
```

2) Comentários de várias linhas: usa-se /* para iniciar e */ para finalizá-lo:

```
/* Esta é a primeira linha
   de um comentário de múltiplas
   linhas. Simples não?
 */
```

Por questões estéticas, é comum que os programadores coloquem alguns asteriscos a mais:

```
/* Esta é a primeira linha
 * de um comentário de múltiplas
 * linhas. Simples não?
 */
```

3) Comentários do tipo JavaDoc: usa-se /** para iniciar e */ para finalizá-lo:

```
/** Esta é a primeira linha
 *  de um comentário de múltiplas
 *  linhas em formato JavaDoc. Simples não?
 */
```

NOTA: A razão para se usar comentários do tipo JavaDoc será vista mais adiante.

Agora que já foram apresentadas as formas de se inserir comentários em um programa Java, precisamos entender quais são os tipos de comentários que precisaremos fazer.

Existem, basicamente, dois tipos de comentário em um código:

A) Comentários que descrevem O QUE o código faz (sempre necessários)
B) Comentários que descrevem COMO o código faz (nem sempre necessários).

Examinemos cada um deles com maior profundidade.

1.1 Comentários do tipo "O QUÊ"

Os comentários do tipo "o que" são aqueles que explicam em linhas gerais o que um trecho de código faz. É comum ter comentários deste tipo para as classes e para cada um de seus métodos, descrevendo em detalhes para que essa classe/método serve e como devem ser utilizado em um programa, como mostra o exemplo a seguir.

Main.java

```
/*  Esta classe Main é a principal do programa.
 *  Ela é responsável por inicializar o programa
 *  E executar suas tarefas mais básicas.
 *  Esta classe não depende de nenhuma outra.
 *
 *  Criada em: 10/04/2010
 *  Autor: Daniel Caetano (daniel@caetano.eng.br)
 */
public class Main {

   /*  Este método imprime um texto em uma janela.
    *  Este método depende do parâmetro "texto", do tipo String,
    *  que é o texto que será impresso na janela.
    */
   public static void imprime(String texto) {
      JOptionPane.showMessageDialog(null,texto,"Mensagem!", JOptionPane.PLAIN_MESSAGE);
      }

   /*  Este método abre uma janela e pede que o usuário digite um número.
    *  Este método depende do parâmetro "texto", do tipo String,
    *  que é a pergunta que o usuário deverá responder no campo.
    *  Este método retorna o número digitado pelo usuário, como um valor
    *  do tipo "double".
    */
   public static double entrada(String texto) {
      String valor = JOptionPane.showInputDialog(texto);
      return (Double.parseDouble(valor));
      }

   }
```

Estes comentários são obrigatórios e são os mais importantes para que nosso código possa ser USADO por alguém que não o conhece perfeitamente ou não se lembra como se usava o código. Faça com cuidado, o usuário da documentação pode ser você mesmo, no futuro, e os minutos gastos documentando o código poderão lhe poupar horas de aborrecimento futuro.

1.2 Comentários do tipo "COMO"

Os comentários do tipo "como" são aqueles que explicam em detalhe o que alguns trechos do código mais complexos fazem. Como não existe uma definição clara de o que é um "trecho complexo", a criação desse tipo de comentário acaba sendo um pouco de "arte", já deve-se evitar comentar demais - o que polui o código, assim como se deve evitar comentários de menos - o que não ajuda ninguém. Este tipo de comentário é mostrado no exemplo a seguir.

```
   public static boolean processa(double av1, double av2, double av3, double freq) {

      double media; // usada como variável auxliar para o cálculo da média

      // Alunos com AV2 menor que 4 e frequencia menor que 75% são reprovados
      if (av2 < 4.0 || freq < 75.0) return false;

      // Para o aluno ser aprovado, pelo menos a AV1 ou AV3 precisa ser >= 4
      if (av1 < 4.0 && av3 < 4.0) return false;
```

```
        // Se AV1 é a maior, ela que entra na média com AV2
        if (av1 >= av3) media = (av1 + av2) / 2;

        // Caso contrário, usa-se AV3 para compor a média com AV2
        else media = (av3 + av2) / 2;

        // Finalmente... se a média for menor que 6, aluno reprovado!
        if (media < 6.0) return false;

        // Se todos os critérios forem atendidos, aluno aprovado!
        return true;
        }
```

3. COMENTÁRIOS JAVADOC

Como é muito complicado manter as duas documentações - a do código e a em papel - ao mesmo tempo, seria interessante se pudéssemos fazer uma documentação única e, dela, extrair a outra. A Sun Microsystems pensou nisso e criou a aplicação JavaDoc, que usa os comentários do tipo "o que" para gerar a documentação externa.

O JavaDoc é um programa que lê o código das classes que escrevemos e gera um arquivo HTML par cada uma delas, resumindo todas as informações importantes que colocamos nos comentários de nosso código. O JavaDoc é uma ferramente muito versátil e permite gerar diferentes tipos de documentação "externa" com base em nosso código:

Só Públicos: com o parâmetro **-public** , o JavaDoc documenta apenas as classes, métodos e atributos públicos de um programa. Basicamente serve para documentar **componentes** que serão distribuídos a outros programadores que irão utilizar este componente.

Públicos e Protegidos: com o parâmetro **-protected** , o JavaDoc documenta as classes, métodos e atributos públicos e protegidos de um programa. Este é o comportamento padrão e serve para documentar classes e componentes que serão extendidos por outros programadores.

Públicos, Protegidos e Pacotes: com o parâmetro **-package** , o JavaDoc documenta as classes, métodos e atributos públicos e protegidos de um programa, além de especificar os pacotes. É uma versão mais completa de Públicos e Protegidos.

Tudo: com o parâmetro **-private** , o JavaDoc documenta as classes integralmente, incluindo métodos e atributos públicos, protegidos e privados de um programa, além de especificar os pacotes. Serve para documentar os componentes e classes para que possa ser feita sua manutenção futura.

Mas, como devemos especificar os comentários para que o aplicativo JavaDoc os compreenda e possa gerar a documentação externa para nós?

3.1. Sintaxe JavaDoc

Como já foi visto, os comentários JavaDoc tem uma especificação levemente diferente dos comentários de múltiplas linhas, começando com o sinal /** e terminando com o sinal */. Este comentário só será reconhecido pelo JavaDoc se vier imediatamente ANTES da classe, interface, construtor, método ou campo/atributo (daqui em diante chamados apenas de **entidades**).

A primeira linha de um comentário JavaDoc deve ser sempre uma descrição clara e concisa do que a entidade faz, pois esta linha será usada como referência. Um ponto final ou um "tab" indica o "fim" dessa linha para o JavaDoc. Observe o exemplo:

Main.java

```
/**  Classe principal, responsável pela inicialização e gerenciamento.
 *  Esta classe é responsável por inicializar o programa
 *  E executar suas tarefas mais básicas.
 *  Esta classe não depende de nenhuma outra.
 */
public class Main {

   /** Este método imprime um texto em uma janela.
    */
   public static void imprime(String texto) {
      JOptionPane.showMessageDialog(null,texto,"Mensagem!", JOptionPane.PLAIN_MESSAGE);
      }

   /** Este método abre uma janela e pede que o usuário digite um número.
    */
   public static double entrada(String texto) {
      String valor = JOptionPane.showInputDialog(texto);
      return (Double.parseDouble(valor));
      }
   }
```

Dentro destes comentários pode-se usar tags HTML se considerado interessante (<B>, <EM>, <I>...). Evite usar tags estruturadores, como <P>, <H1>, <HR> e outros.

Depois da primeira linha, pode-se fazer uma explicação mais extensa sobre a entidade sendo documentada. Observe o exemplo:

Main.java

```
/**  Classe principal, responsável pela inicialização e gerenciamento.
 */
public class Main {

   /** Este método imprime um texto em uma janela.
    */
   public static void imprime(String texto) {
      JOptionPane.showMessageDialog(null,texto,"Mensagem!", JOptionPane.PLAIN_MESSAGE);
      }
```

```
    /** Este método abre uma janela e pede que o usuário digite um número.
     */
    public static double entrada(String texto) {
       String valor = JOptionPane.showInputDialog(texto);
       return (Double.parseDouble(valor));
       }
    }
```

Existem, ainda, diversos indicadores que podemos e alguns que devemos usar dentro dos comentários. Estes indicadores são feitos com o uso de *tags* especiais, que devem vir no início da linha do comentário. Eles estão descritos a seguir:

Tag	**Significado Resumido**
@author	Indica o autor
@deprecated	Indica que não deve ser usado
@exception	Indica exceções
{@link}	Link pra outro documento
@param	Indica atributos de entrada
@return	Indica resultados de retorno
@see	Indicação para outros docs.
@serial / @serialData / @serialField	Possibilidade de serialização
@since	Em que versão foi criado
@throws	Indica lançamento de exceções
@version	Versão atual

É interessante que todos os *tags* de um mesmo tipo venham agrupados, pois isso facilita o trabalho do aplicativo JavaDoc. Por exemplo, se um método ou classe tem mais de um autor, cada um deles deve ser especificado em uma linha iniciando com @author, mas todas essas linhas devem ser agrupadas.

Foge ao escopo deste curso estudar em profundidade todas as tags do JavaDoc, mas você pode encontrar informações sobre elas na Internet. Aqui iremos falar das mais comuns e importantes neste ponto do curso: @author, @deprecated, @param, @return, @version.

@author serve para identificar o autor de um trecho de código. Usa-se assim:

@author Nome do Autor

@deprecated serve para identificar uma classe/método que existe por compatibilidade (e, portanto, não deve ser usada em nada novo). Usa-se assim:

@deprecated Evite usar esta classe. Use a classe XXXXX no lugar desta.

@param serve para identificar para que serve um dos parâmetros de um método. Usa-se assim:

```
/** Este método imprime um texto em uma janela.
 *  @param texto Indica o texto que deve ser impresso na janela de mensagem.
 */
public static void imprime(String texto) {
   JOptionPane.showMessageDialog(null,texto,"Mensagem!", JOptionPane.PLAIN_MESSAGE);
   }
```

Observe que o "nome" depois do @param deve ser o mesmo que aparece na declaração do parâmetro do método!

@return serve para indicar o que um método retorna. Usa-se assim:

```
/** Este método abre uma janela e pede que o usuário digite um número.
 *  @return O número digitado pelo usuário.
 */
public static double entrada(String texto) {
   String valor = JOptionPane.showInputDialog(texto);
   return (Double.parseDouble(valor));
   }
```

@version serve para indicar a versão de uma classe, pacote ou método. Usa-se assim:

@version 1.10.17

4. ATIVIDADE: GERANDO DOCUMENTAÇÃO USANDO O JAVADOC

A) Comente o código abaixo, usando a sintaxe vista para o JavaDoc:

```
import javax.swing.*;

public class ContraCheque {
    public static void imprime(String texto) {
        JOptionPane.showMessageDialog(null,texto,"Atenção!",JOptionPane.PLAIN_MESSAGE);
    }

    public static double entrada(String texto) {
       return (Double.parseDouble(JOptionPane.showInputDialog(texto)));
    }

    public static double calculaImpostoRetido(double salario) {
        if (salario <= 1499.15) return 0;
        else if (salario <= 2246.75) return arDinheiro((salario-112.43)*0.075);
        else if (salario <= 2995.70) return arDinheiro((salario-280.94)*0.150);
        else if (salario <= 3743.19) return arDinheiro((salario-505.62)*0.225);
        else return arDinheiro((salario-692.78)*0.275);
    }

    public static double calculaInssRetido(double salario) {
        if (salario <= 1024.197) return arDinheiro(0.08*salario);
        else if (salario <= 1708.27) return arDinheiro(0.09*salario);
        else if (salario <= 3416.54) return arDinheiro(0.11*salario);
        else return 375.82;
    }

    public static double arDinheiro(double valor) {
        return (Math.rint(valor*100)/100);
    }

    public static void processa() {
        double bruto, inss, irrf;
        String saida;
        bruto = entrada("Digite o Salário Bruto");
        inss = calculaInssRetido(bruto);
        irrf = calculaImpostoRetido(bruto-inss);
        saida  = "Salário Bruto:   R$ " + bruto +"\n";
        saida += "Desconto INSS:   R$ " + inss +"\n";
        saida += "Salário-IRRF:   R$ " + arDinheiro(bruto-inss) +"\n";
        saida += "Desconto IRRF:   R$ " + irrf +"\n";
        saida += "Salário Líquido:   R$ " + arDinheiro(bruto-inss-irrf);
        imprime(saida);
    }

    public static void main(String[] args) {
        processa();
        System.exit(0);
    }
}
```

B) Usando o NetBeans, no menu Executar selecione a opção Gerar JavaDoc e observe o resultado.

BIBLIOGRAFIA

DEITEL, H.M; DEITEL, P.J. **Java: como programar** - Sexta edição. São Paulo: Pearson-Prentice Hall, 2005.
HOFF, A; SHAIO, S; STARBUCK, O. **Ligado em Java**. São Paulo: Makron Books, 1996.
SUN MICROSYSTEMS: http://java.sun.com/j2se/5.0/docs/api/index.html