

# Unidade IV

Agora que já vimos como fazer a passagem do modelo de análise para o modelo de projeto e que já entramos efetivamente na fase de projetos, nas atividades de projeto de dados e classes e também nas atividades de projeto arquitetural, sabemos quais os objetivos de cada atividade e quais modelos precisamos produzir, com o auxílio da UML.

Veremos, nesta unidade, como produzir esses modelos, representados pelos diagramas da UML, quais são os seus elementos e como podemos utilizar cada um da melhor forma.

## 7 REFINANDO A MODELAGEM DE ASPECTOS DINÂMICOS DO SOFTWARE

Anteriormente, vimos que na fase de projeto arquitetural é necessária a representação dos aspectos dinâmicos da arquitetura e, basicamente, utilizamos o diagrama de sequência da UML para dar corpo a essa representação.

O objetivo deste tópico é apresentar modelos e diagramas da UML que complementem a visão dada pelo diagrama de sequência quando da modelagem dos aspectos dinâmicos do *software*.

Complementar não significa substituir, portanto nenhum diagrama que veremos a seguir tem como objetivo substituir a visão dada pelo diagrama de sequência, mas auxiliar no refinamento da visão arquitetural dinâmica dada por este diagrama.

### 7.1 Diagrama de comunicação

O diagrama de comunicação passou a ter esse nome a partir da versão 2.0 da UML; nas versões anteriores, esse diagrama é chamado de diagrama de colaboração (UML-DIAGRAMS, 2009-2014b).

A característica marcante do diagrama de comunicação é a forte semelhança com o diagrama de sequência. As informações modeladas em ambos são, no geral, as mesmas, todavia a representação em cada um dos modelos possui ênfases diferentes.

O diagrama de comunicação é um tipo de diagrama comportamental da UML que representa as interações de dois objetos e suas partes utilizando para isso uma sequência de mensagens representadas de forma livre de formatação (UML-DIAGRAMS, 2009-2014b).



#### Lembrete

Embora os diagramas de comunicação e de sequência possuam grande semelhança, o diagrama de comunicação é um complemento do diagrama de sequência.

Enquanto o diagrama de sequência dá ênfase à troca de mensagens em uma linha de tempo, o diagrama de comunicação dá ênfase a como os objetos estão interligados e quais mensagens são trocadas entre eles para realizar uma determinada tarefa.

No diagrama de comunicação as mensagens possuem uma numeração, é como se elas fossem etiquetadas com uma numeração em ordem crescente, e é essa sequência numérica que representa a sequência em que as mensagens são trocadas entre os objetos.

O diagrama de sequência obedece a uma ordem natural de leitura, ou seja, sabemos que devemos ler a sequência das mensagens da esquerda para a direita. Enquanto isso, no diagrama de comunicação, começamos a ler a sequência das mensagens procurando inicialmente a mensagem identificada como 1.0, seguindo essa mensagem do objeto que envia até o objeto que a recebe, e assim sucessivamente.

Podemos dizer que os diagramas de sequência e de comunicação são isomórficos, ou seja, um pode ser transformado no outro. Isso porque os objetos, atores e mensagens que são representados nesses diagramas são originados nos diagramas de caso de uso e nos diagramas de classes de análise e projeto, e o que é representado no diagrama de sequência; também é representado no diagrama de comunicação (LEE e TAPFENHART, 2001).



### Observação

Existem muitas ferramentas que fazem a conversão automática do diagrama de sequência para o diagrama de comunicação e vice-versa. Todavia, sempre tenha muito cuidado e faça uma revisão no diagrama gerado a partir destes tipos de ferramentas.

Quando escolher um e quando escolher o outro?

Se o seu objetivo for representar a interação dos objetos no decorrer de uma linha do tempo, então você deverá usar o diagrama de sequência, mas se você desejar dar ênfase à interação desses objetos no contexto do sistema, então o melhor que você terá a fazer é dar prioridade ao diagrama de comunicação (MEDEIROS, 2004).

De qualquer forma, é importante que tenhamos em mente o principal: os diagramas são complementares. No momento do desenvolvimento de um diagrama de comunicação a partir de um de sequência, podemos identificar pontos de melhoria ou até mesmo erros no modelo original, daí uma ótima oportunidade para revisar, alterar e melhorar o modelo.

Lembre-se sempre de que o objetivo dos diagramas não é produzir informação em larga quantidade, mas produzir informação de qualidade.

Produzir um diagrama de sequência e um diagrama de comunicação com qualidade é importante fator para a qualidade na comunicação entre os envolvidos no projeto e na qualidade da construção do produto final.

O diagrama de comunicação é baseado em alguns componentes:

- **Objetos:** têm a mesma conotação do objeto que representamos no diagrama de sequência, com uma diferença fundamental, pois no diagrama de comunicação não damos ênfase à linha de tempo de vida do objeto.
- **Vínculos:** identificam uma ligação entre dois objetos, que é feita a partir da troca de mensagens entre esses objetos.
- **Mensagens:** análogas às mensagens do diagrama de sequência, possuem apenas duas diferenças básicas: as mensagens são identificadas numericamente e não existe uma pré-formatação para a representação dessas mensagens, como temos no diagrama de sequência. Na representação de mensagens, podemos representar todos os tipos de mensagem – síncronas, assíncronas e automensagem.
- **Atores:** nesse diagrama, bem como no diagrama de sequência, podemos representar o autor. Com o detalhe importante de que esse autor deve necessariamente estar representado também no diagrama de caso de uso.

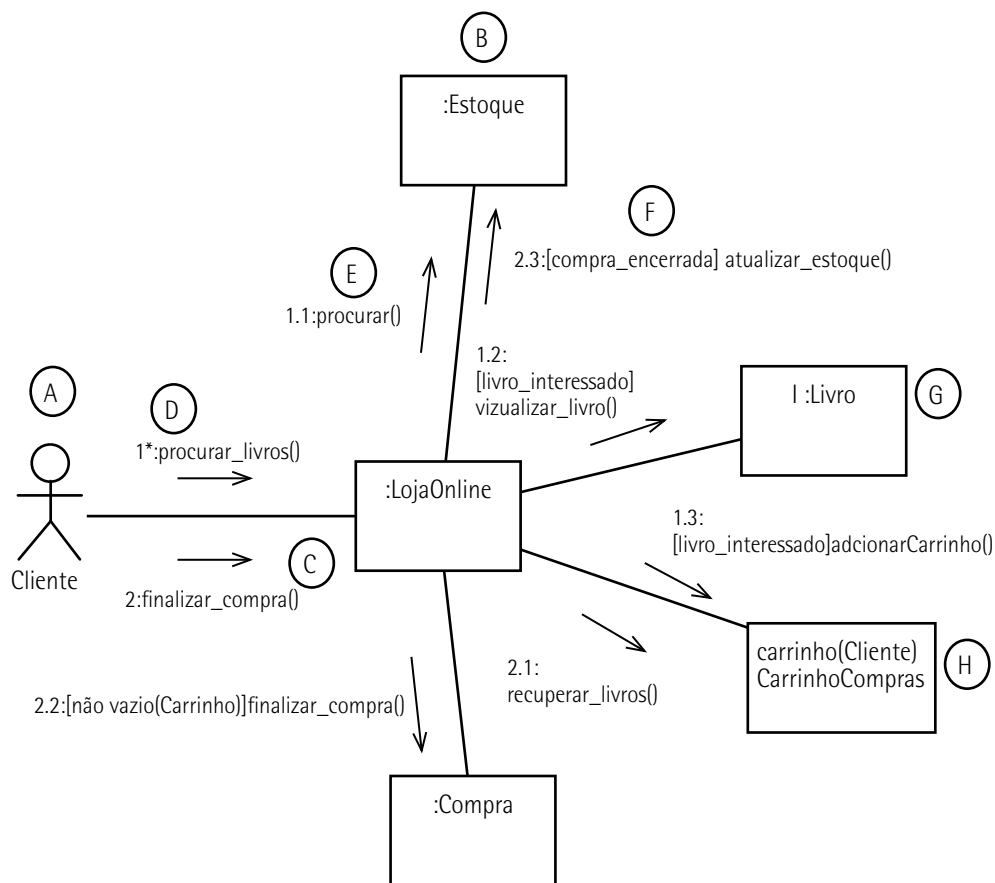


Figura 41 – Elementos do diagrama de comunicação UML

**Quadro 12 – Elementos do diagrama de comunicação UML**

Letra	Descrição
A	Notação de ator – com o estereótipo idêntico ao visto no diagrama de caso de uso.
B	<p>Notação de objeto – representamos um objeto com um retângulo, a exemplo de como representamos um objeto no diagrama de sequência.</p> <p>Podemos representar um objeto apenas como um tipo de, como no exemplo da letra B, em que “:Estoque” lê-se um objeto anônimo do tipo Estoque</p> <p>OU</p> <p>Podemos representar um objeto nominal como no exemplo da letra G, onde “!Livro” lê-se o objeto l do tipo Livro.</p>
C	<p>Representação de vínculo – a linha contínua que liga dois objetos indica que existe um vínculo, ou uma dependência, entre esses objetos.</p> <p>Quando há um vínculo entre objetos, presume-se que haverá troca de mensagens entre esses objetos.</p>
D	<p>Indicador de sequência da mensagem – no caso da figura, temos a representação 1.*, onde podemos ler que 1 é a sequência da mensagem e * é o número da iteração.</p> <p>Ainda no exemplo da figura, podemos ver que a mensagem inicial é a mensagem “procurar_livros()”</p>
E	<p>Notação de mensagem – nota-se que a representação da mensagem é idêntica à representação de mensagem no diagrama de sequência, ou seja: nome do método + parâmetros de entrada + tipo de retorno.</p> <p>Observação: se a mensagem for do tipo retorno, será representada com uma seta tracejada:</p> <p>-----➔</p>
F	Indicação de parâmetro da mensagem – no caso da figura, estamos passando como parâmetro para a mensagem “atualizar_estoque” os livros comprados que constam na ordem de compra.
G	Declaração de objeto com nome e tipo – vide comentários descritos na letra B.
H	<p>Descrição de objeto com seletor e tipo – no caso da figura, o objeto carrinho não pode ser qualquer carrinho, mas deve ser um carrinho selecionado para um determinado cliente.</p> <p>Observação: na notação de objetos, também podemos utilizar os estereótipos de responsabilidade – entidade, fronteira e controle –, exatamente da mesma forma que representamos no diagrama de sequência.</p>

Como ficaria o processo que representamos na figura anterior, utilizando o diagrama de comunicação, se optássemos por utilizar o diagrama de sequência?

A figura a seguir mostra um diagrama de sequência análogo ao diagrama de comunicação:

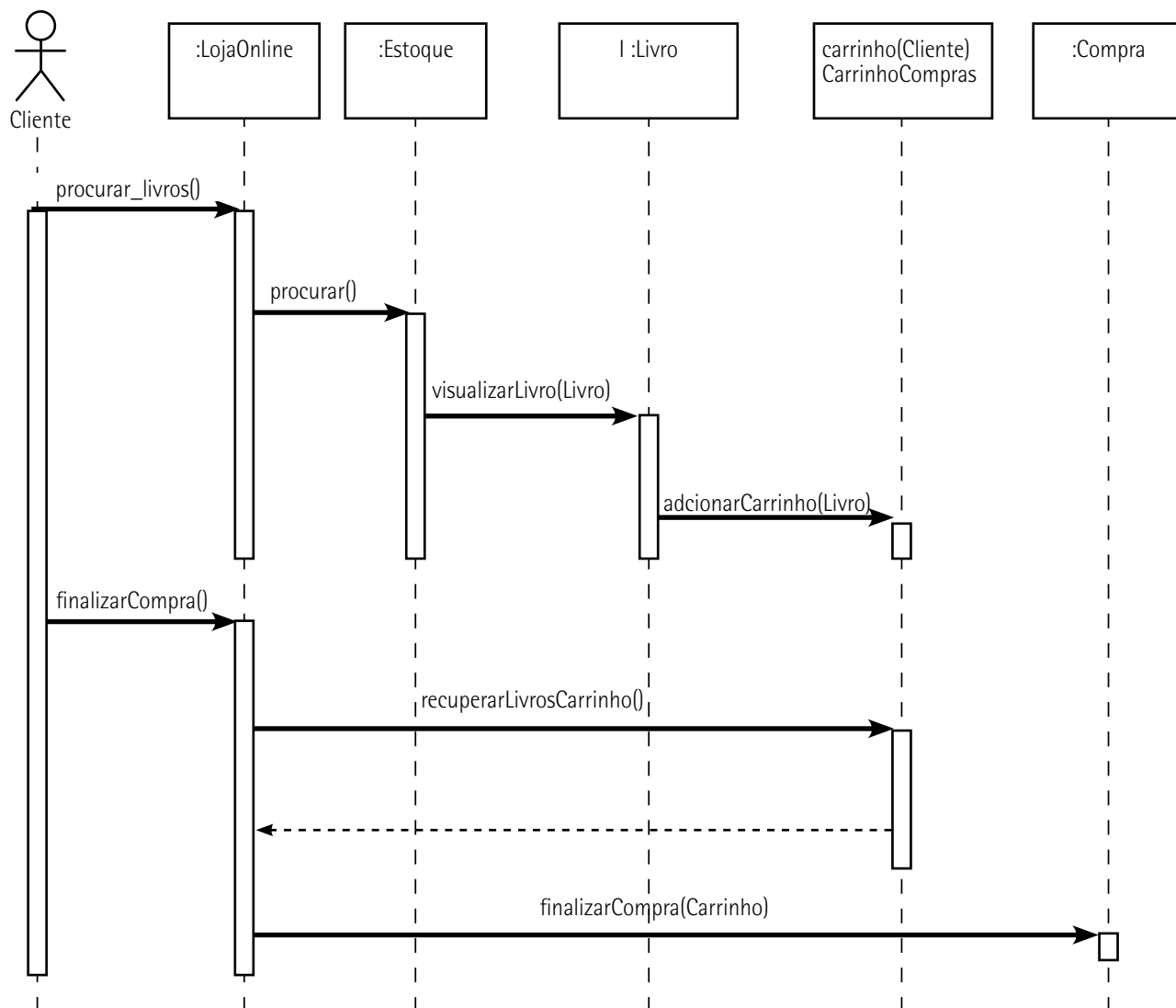


Figura 42 – Exemplo de diagrama de sequência análogo ao diagrama de comunicação

A partir das figuras apresentadas podemos ter a clara noção das diferenças e semelhanças entre os diagramas e de por que dizemos que eles são complementares.

Podemos notar também que a representação desse cenário, especificamente, utilizando o diagrama de comunicação, projeta um diagrama que, em linhas gerais, se apresenta mais fácil de ser entendido. Isso porque, como você pode encontrar em algumas referências, o diagrama de comunicação pode ser utilizado para a modelagem de cenários mais simples, o que não pode ser encarado como mentira, mas também não é uma verdade absoluta.

## 7.2 Diagrama de máquina de estado

O diagrama de máquina de estado, ou simplesmente diagrama de estado, tem como objetivo representar o comportamento de um determinado elemento a partir de um conjunto finito de estados.

Esse elemento a ser representado no diagrama de estado é, na maioria das vezes, uma instância de uma classe, um objeto, uma vez que um objeto, pelos princípios da orientação a objetos, possui um estado (LARMAN, 2007).

Note que mencionamos que na maioria das vezes utilizamos um diagrama de máquina de estados para representar uma instância de uma classe. Isso porque podemos representar, também, a mudança de estados de um caso de uso.

### Observação

O estado de um objeto, ou a identidade desse objeto, é denominado pelo conjunto de valores associados aos seus atributos em um determinado período dentro do universo sistêmico.

A figura a seguir mostra a diferença entre classe, objeto e estado do objeto. Podemos notar que a classe Cliente possui alguns atributos: agência, conta-corrente, CPF e saldo de conta-corrente.

A partir do momento em que essa classe se torna um objeto no sistema, os atributos passam a ter um determinado valor associado e esse objeto passa a ter uma identidade única dentro do sistema, passa a estar apto a receber e enviar mensagens, ou seja, ele passa a ser concretamente um cliente.

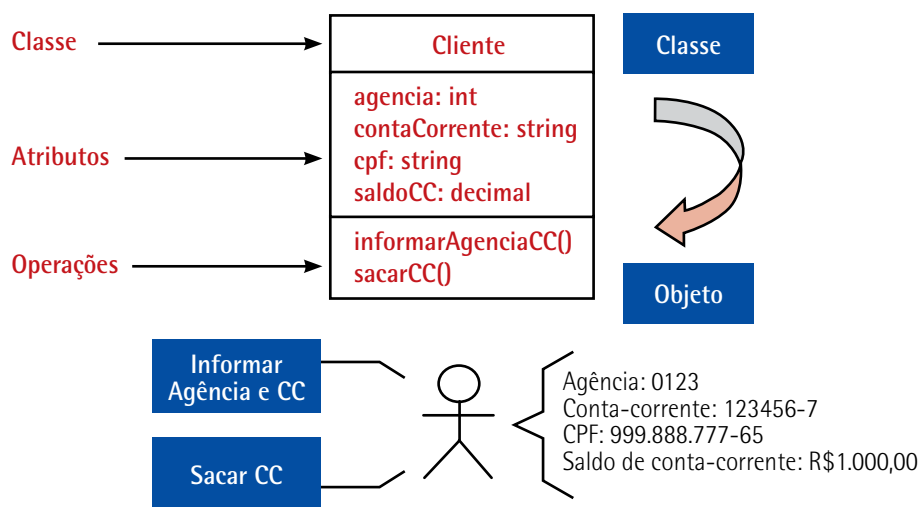


Figura 43 – Classe, objeto e estado de um objeto

Já verificamos que o objeto possui um estado, por exemplo, no momento da "fotografia", nota-se que o cliente possui R\$ 1.000,00 em sua conta-corrente. Podemos dizer então que, nesse momento, ele possui este estado.

Em contrapartida, se este mesmo cliente, pouco tempo depois dessa "fotografia", efetuar um saque em conta-corrente no valor de R\$ 300,00, seu estado será alterado, pois o valor do seu atributo "saldoCC" será alterado, ou seja, haverá uma mudança de estado.

É exatamente isto que o diagrama de máquina de estado se dispõe a representar: os estados dos objetos no decorrer do tempo e o que motivou uma possível mudança de estado. O "motivador" de uma mudança de estado é chamado de evento.

Segundo Larman (2007), um evento é uma ocorrência digna de nota que promove a mudança de estado de algum objeto. Como o exemplo do próprio autor: um telefone está ocioso até que é tirado do gancho; a partir desse evento ele passa a estar ativo.

Outro exemplo simples (e aqui resumimos bem o processo, apenas para mostrar uma aplicação para o diagrama), como mostra a figura a seguir, é o de uma máquina de autoatendimento bancário, que está inativa até que um cartão seja inserido.

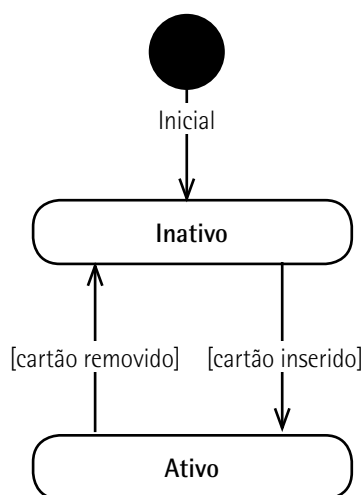




Figura 44 – Diagrama de estado para um ATM

Aqui, podemos notar os elementos básicos da notação de um diagrama de estados, que estão representados no quadro a seguir:

### Quadro 13 – Elementos básicos do diagrama de estados

Símbolo	Significado
<div style="border: 1px solid black; border-radius: 10px; padding: 5px; display: inline-block;">Estado</div>	<p>Estado – representa uma determinada situação de um elemento em um determinado momento.</p> <p>Um estado pode possuir os seguintes significados:</p> <ol style="list-style-type: none"> <li>1. A espera pela ocorrência de um evento.</li> <li>2. A reação a um estímulo.</li> <li>3. A execução de alguma atividade.</li> <li>4. A satisfação de alguma condição.</li> </ol> <p>(LARMAN, 2007)</p>
	<p>Transição – a seta representa um evento que gera uma transição de estado no sentido apontado.</p>

	<p>Estado inicial – representa o marco inicial, o ponto de partida da máquina de estados.</p> <p>Neste momento os objetos ainda não possuem um estado inicial, eles passam a ter um no momento em que se inicia a máquina, ou seja, em que a máquina sai do estado inicial.</p>
	<p>Estado final – indica o final dos estados modelados.</p>

O estado de um objeto pode ser classificado como:

- Simples: quando é autossuficiente, ou seja, não possui a necessidade da composição ou da divisão em estados menores.
- Composto: quando um estado contém internamente dois ou mais estados, chamados subestados, como mostra o exemplo da figura seguinte:

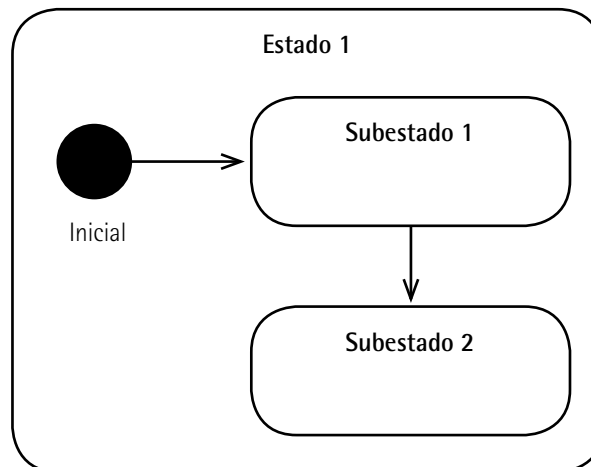


Figura 45 – Exemplo de estado composto

Quando um objeto entra em um determinado estado, ele executa determinadas atividades, que são:

- Atividade de entrada (*entry*): é a primeira atividade a ser executada quando um objeto assume um determinado estado.
- Atividades de fazer (*do*): são atividades executadas quando um objeto se encontra em um determinado estado.
- Atividades de saída (*exit*): são atividades executadas no momento em que um objeto sai de um determinado estado.

Existem alguns tipos de transições que não produzem alteração no estado de um objeto, ou seja, ocorrem durante o estado do objeto sem que esse estado seja modificado.



Essas transições são chamadas de:

- Transições internas: são transições que acontecem enquanto um objeto se encontra em um determinado estado e que não promovem modificação neste estado do objeto. Essas transições executam atividades de fazer (*do*).
- Autotransições: são transições que executam atividades do tipo sair (*exit*) de um objeto, executam uma determinada ação e voltam para o estado de que saíram sem que haja alteração desse estado, como mostra o exemplo a seguir:

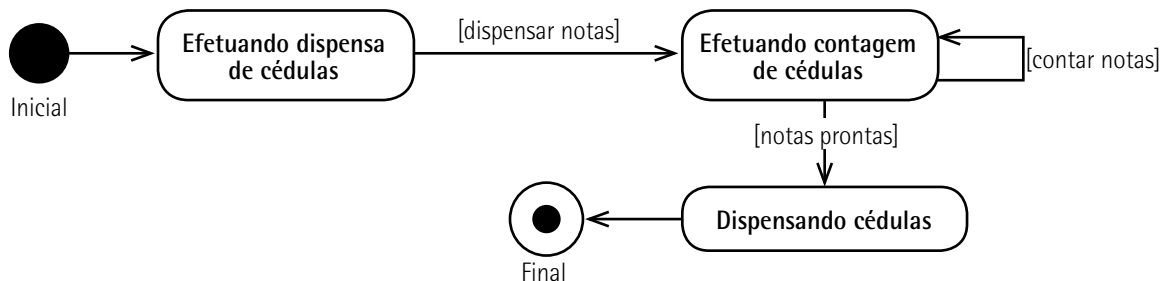


Figura 46 – Exemplo de autotransição

Podemos notar que, enquanto as notas não estiverem prontas na contagem, não haverá a transição de estado entre efetuar a dispensa e efetivamente dispensar as cédulas (BEZERRA, 2006).

No diagrama de máquina de estados, podemos ainda representar processamentos paralelos, ou seja, quando em um determinado ponto do processo, existe uma divisão de processamentos, logo mudanças de estados de objetos que ocorrem em paralelo.

Para representar esse paralelismo, utilizamos alguns mecanismos, como a barra de bifurcação e a barra de união, como mostra o exemplo da figura a seguir:

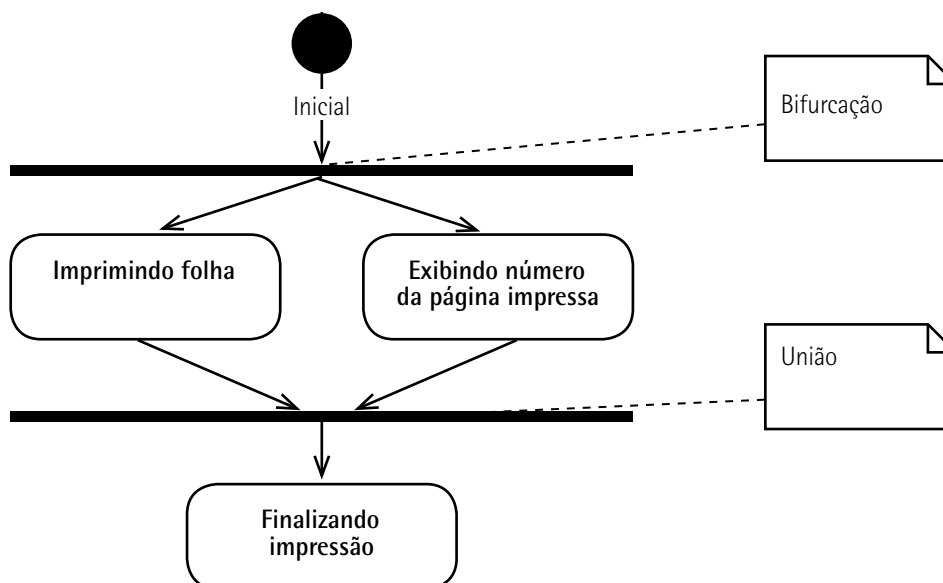


Figura 47 – Exemplo de barra de bifurcação e barra de união



### Lembrete

Sempre que utilizarmos uma barra de bifurcação, em algum momento do processo deveremos utilizar uma barra de união; não é possível, no desenvolvimento de sistemas, que não haja um ponto de conversão de processamentos paralelos.

Existe ainda a situação em que devemos representar um ponto de decisão, uma espécie de controle do tipo *if/else*, um ponto em que a máquina de estado se divida em função de uma decisão.

Para essas situações, utilizamos o pseudoestado de escolha, que representa um ponto na transição de dois estados em que deve ser tomada uma decisão, conforme mostra o exemplo da figura seguinte:

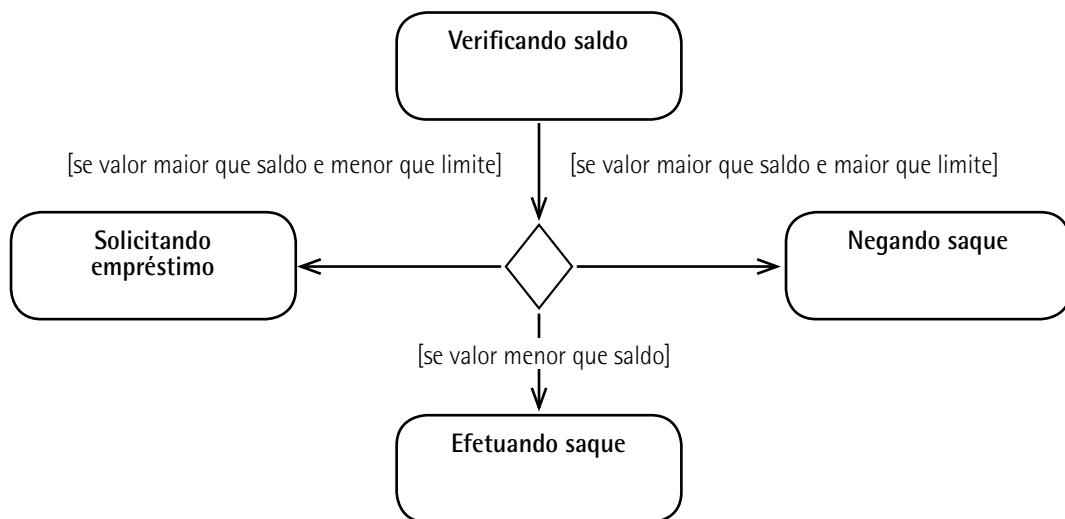


Figura 48 – Exemplo de pseudoestado de escolha

Além desses elementos considerados básicos, pois utilizamos com frequência em praticamente qualquer processo que desejamos modelar, temos também os elementos considerados elementos avançados de modelagem (GUEDES, 2009).

São eles:

- Pseudoestado de história: utilizado para representar o registro do último estado em que um objeto se encontrava quando, por alguma razão, o processo foi interrompido. Utilizamos esse estado para que possamos voltar o processo exatamente no ponto em que o estado se encontrava antes de uma interrupção. Utilizamos um H para representar esse estado.

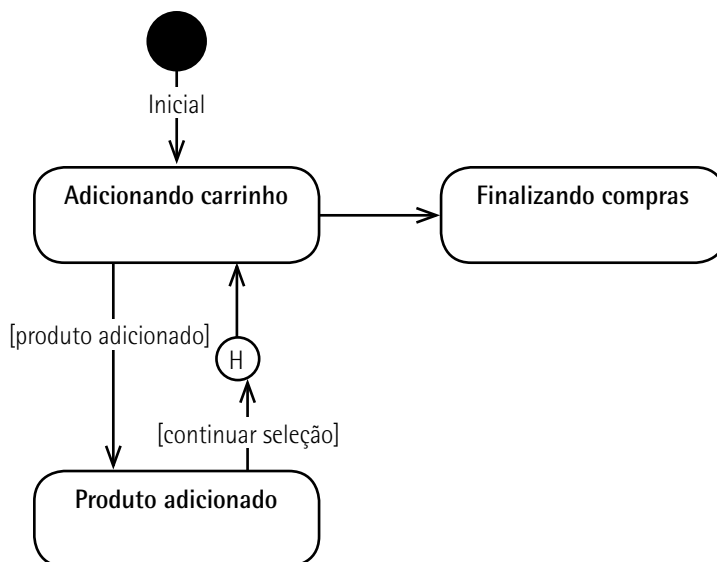


Figura 49 – Exemplo de pseudoestado de história

- Estado de submáquina: utilizamos um estado de submáquina quando precisamos representar um estado composto de alta complexidade e que, se utilizarmos apenas a representação de estado composto, dificultará muito o entendimento em apenas um diagrama.

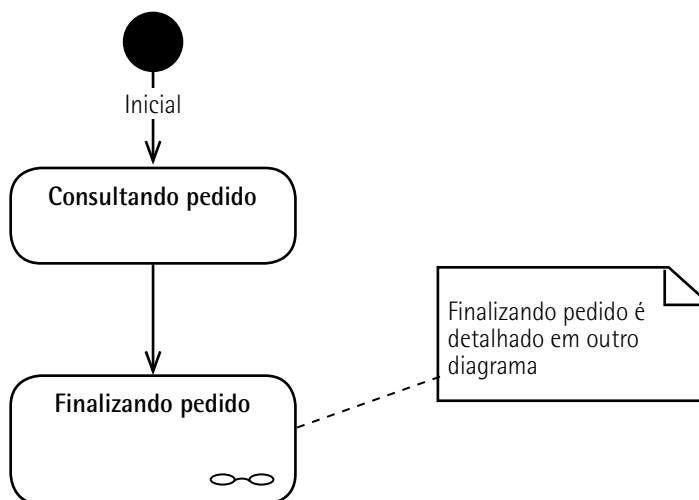


Figura 50 – Exemplo de estado de submáquina

- Pseudoestado de junção: utilizamos esse estado para representar a união de múltiplos fluxos em um único ponto.

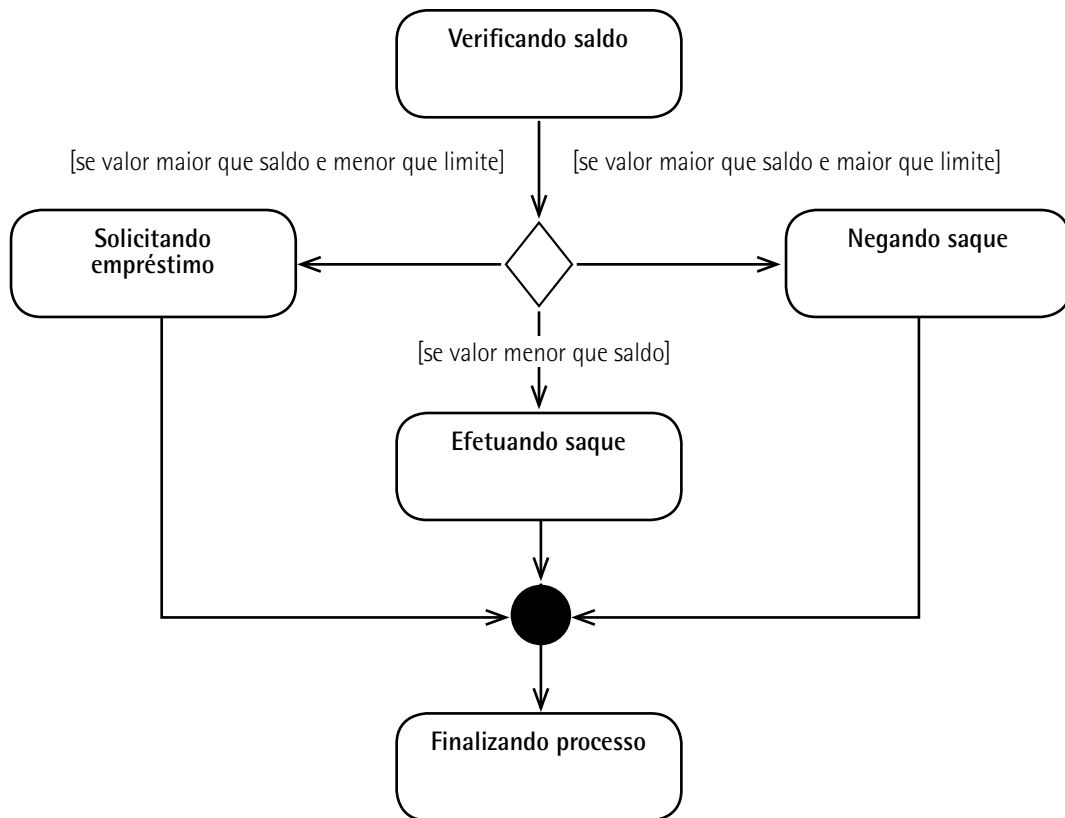


Figura 51 – Exemplo de pseudoestado de junção

Com o diagrama de máquina de estados e com o diagrama de comunicação, fechamos a parte de refinamento da representação dos aspectos dinâmicos da arquitetura, no projeto arquitetural.



### Saiba mais

Na UML existe ainda um diagrama classificado como diagrama comportamental, mas que é pouco utilizado atualmente, chamado diagrama de tempo, ou de temporização. Acesse em:

UML-DIAGRAMS. *Timing diagrams*, 2009-2014a. Disponível em: <<http://www.uml-diagrams.org/timing-diagrams.html>>. Acesso em: 28 abr. 2015.

## 8 PROJETO DE INTERFACES E PROJETO DE COMPONENTES

### 8.1 Projeto de interfaces

Pressman (2006, p. 222) faz uma analogia interessante sobre o que vêm a ser os componentes de um projeto de interfaces de um sistema de *software*: "o projeto de interfaces é análogo a um conjunto de desenhos detalhados de portas, janelas e ligações externas de uma casa [...] representam a maneira pela

qual as ligações de serviços públicos entram na casa e são distribuídos entre os cômodos representados na planta”.

Os elementos do projeto de interfaces, ainda segundo Pressman (2006), representam como as informações entram em um sistema de *software* e saem dele e como essas informações trafegam entre as estruturas desse sistema definidas no projeto arquitetural.

O projeto de interface pode ser dividido em três níveis:

- Interfaces internas entre os componentes do sistema.
- Interfaces externas entre o sistema e outros sistemas de *software*, dispositivos de *hardware* ou qualquer entidade que produza ou consuma informação relevante para o sistema de *software*.
- Interface com o usuário final.

Neste tópico daremos ênfase ao projeto de interfaces internas e externas e debateremos apenas aspectos introdutórios sobre o projeto de interface com o usuário.

### 8.1.1 Especificando as interfaces dos objetos

O primeiro passo para a definição das interfaces internas de um sistema, ou seja, as interfaces entre os componentes e objetos internos de um sistema, é definirmos claramente como esses objetos estão organizados.

Como já vimos em projeto arquitetural, agrupamos objetos que tenham alguma correlação, geralmente, agrupamos os objetos que tenham as mesmas responsabilidades, por exemplo, no estilo arquitetural em camadas.



#### Lembrete

A arquitetura em camadas pode ser entendida como “caixas”, ou agrupamentos que são estritamente conceituais, de objetos com funções semelhantes.

Na arquitetura em camadas, organizamos os objetos em blocos conceituais, e a comunicação entre essas camadas se dá através de um protocolo, definido por objetos de negócio que contêm as informações trafegadas entre essas camadas.

Podemos dizer que, ao definirmos essa camada de objetos de negócio e esse protocolo de comunicação entre as camadas de negócio, apresentação e integração, estamos definindo uma interface de comunicação entre esses objetos e essas camadas.

Obviamente, até então, estávamos pensando nessas camadas como blocos conceituais, ou "caixas", para entendermos o conceito da organização arquitetural interna de um *software*.

Agora, iremos partir, efetivamente, para a representação desse modelo em uma linguagem de modelagem de *software*. Utilizaremos, para isso, o diagrama de pacotes da UML.

### 8.1.2 Diagrama de pacotes

Como vimos, o processo de organização e de modelagem da solução de um sistema de *software* pode envolver uma grande quantidade de objetos, classes e diagramas que podem ficar tão complexos quanto se queira.

Ante esse cenário, é possível que você perceba a necessidade de se organizar esses componentes em um nível de abstração um pouco maior, de tal forma que facilite o entendimento de uma visão um pouco mais macro da solução.

Um pacote é utilizado com o propósito de agrupar logicamente objetos, entendendo-se um objeto como qualquer elemento passível de agrupamento no processo de modelagem de um sistema: classes, objetos, componentes e casos de uso (LARMAN, 2007).

A função principal do pacote é organizar esses objetos, esses elementos de modelagem, que possuem semelhantes características, em agrupamentos maiores que podem ser mais facilmente entendíveis.

Como mencionado por Bezerra (2006), quando modelamos a arquitetura de uma aplicação utilizando pacotes, podemos representar as diferentes visões da arquitetura desse sistema, além de definirmos e controlarmos a visibilidade dos elementos de um pacote.

A partir do momento em que definimos a visibilidade dos elementos de um pacote e a forma, ou o protocolo, de como estes pacotes se comunicam, estamos definindo, como pano de fundo, as interfaces dos objetos dentro do universo do sistema, também chamadas de interfaces internas.

Para representar um pacote, no diagrama de pacotes, utilizamos uma pasta com guias, como mostra o exemplo a seguir:

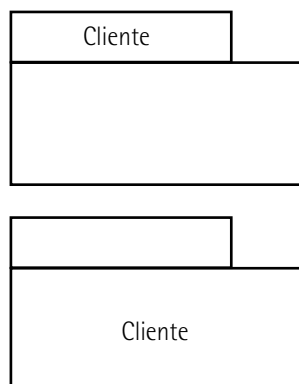


Figura 52 – Notação de pacotes na UML

Note ainda que temos duas formas de descrever o nome de um pacote: na aba do pacote, como na notação superior, ou dentro do pacote, como na notação inferior, sendo mais comum que utilizemos a descrição na guia da pasta.

Como um pacote é um agrupamento de elementos, podemos também utilizar um pacote como um agrupamento de outros pacotes; nesse caso, dizemos que um pacote está contido em outro:

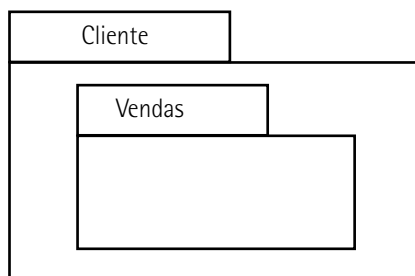


Figura 53 – Exemplo de notação de pacotes e subpacotes

No caso da figura anterior, utilizamos o nome qualificado da seguinte forma: "Cliente::Vendas", ou seja, no prefixo, indicamos o nome do pacote que contém o elemento.

Ainda no caso de subpacotes, podemos representar essa relação de composição, da mesma forma que representamos composição de classes.

No caso da figura a seguir, vemos que o pacote "Cliente" é composto pelo pacote "Vendas", ou podemos utilizar o nome qualificado.

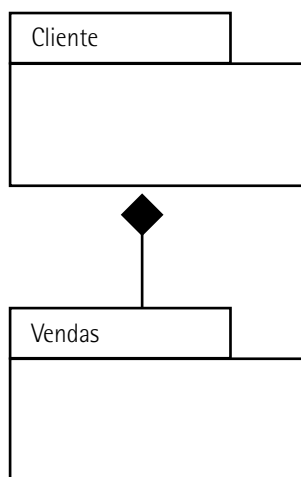


Figura 54 – Exemplo de composição de pacotes e subpacotes

Podemos utilizar pacotes para agrupar casos de uso que sejam de um contexto comum, como mostra a figura a seguir, em que temos dois pacotes de casos de uso separados pelos seus canais de execução.

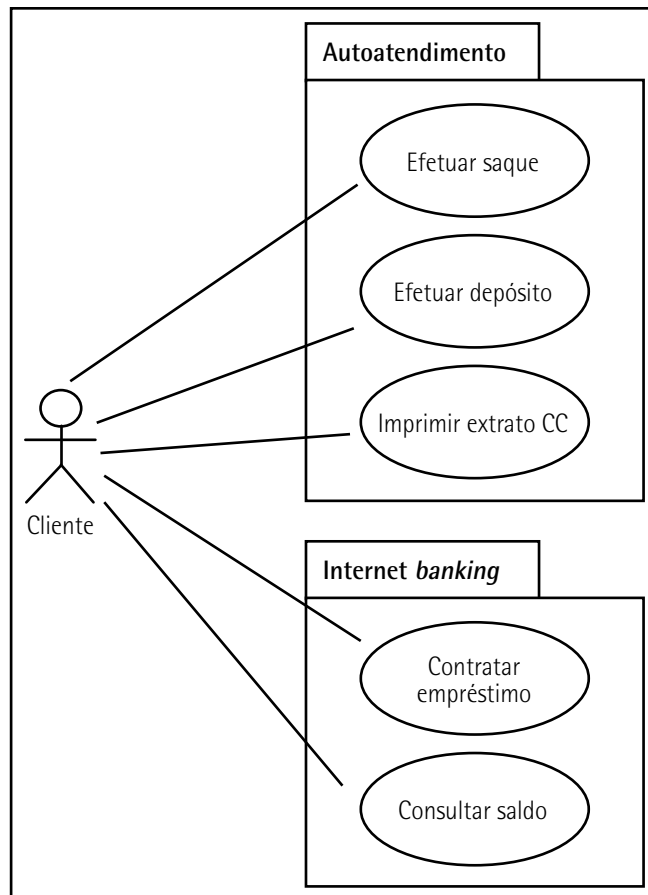


Figura 55 – Exemplo de pacotes de caso de uso

Para representar um pacote como um agrupamento de classes, incluímos essas classes no pacote desejado, como mostra o exemplo da figura a seguir, e nesse exemplo, também podemos utilizar a notação de nome qualificado: "Vendas::Cliente" e "Vendas::Pedido".

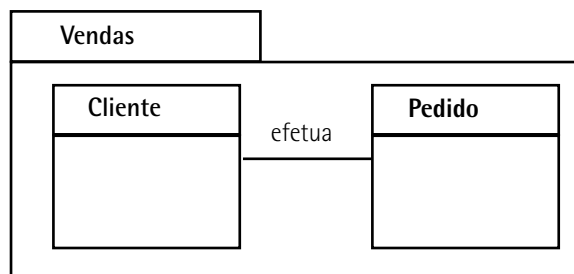


Figura 56 – Exemplo de pacote de classes

Vamos nos aprofundar um pouco mais na parte mais importante do diagrama de pacotes, que é modelar a arquitetura estática de alto nível de um sistema de *software*.





### Observação

Não confunda agrupamento lógico, que é o promovido pelo diagrama de pacotes, com agrupamento físico de classes ou componentes. Veremos como representar agrupamento físico adiante.

Como dissemos anteriormente, podemos definir a visibilidade de elementos dentro de um pacote.

A visibilidade dos elementos segue o mesmo conceito básico de visibilidade de atributos e métodos que vimos nos capítulos anteriores. Temos três níveis de visibilidade: público, representado pelo sinal de positivo (+); privado, representado pelo sinal de negativo (-); e protegido, representado pelo sinal sustenido (#), com algumas diferenças exploradas no contexto de pacotes, como se explica a seguir:

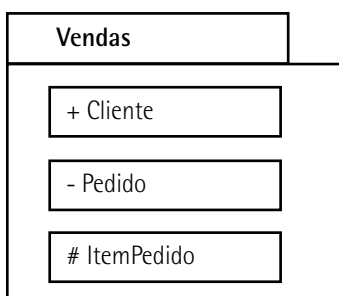


Figura 57 – Exemplo de visibilidade de elementos de um pacote

Na figura anterior, temos um pacote com três classes: "Cliente", "Pedido" e "ItemPedido"; seus respectivos níveis de visibilidade estão descritos na representação da própria classe.

Nesse caso, fazemos as seguintes leituras:

- Um elemento marcado como público (+) é visível por todos os outros pacotes do sistema. As partes públicas desse pacote constituem a interface desse pacote.
- Um elemento marcado como privado (-) é visível somente por outros elementos do mesmo pacote. Elementos privados não podem ser vistos por outros pacotes.
- Um elemento marcado como protegido (#) é visível somente em pacotes-filhos do pacote ao qual pertence. Um pacote-filho é um pacote que possui relação de herança com outro pacote.

Já que tocamos no assunto herança de pacotes, vejamos quais são os tipos de relacionamento possíveis entre pacotes e como podemos representá-los no diagrama.

Podemos dizer que um pacote X dependerá de um pacote Y se um elemento, ou um objeto qualquer de X, depender de outro elemento ou objeto qualquer de Y (BEZERRA, 2006).

É importante que se diga que, a partir do momento em que criamos dependência entre pacotes, uma alteração no pacote de destino afeta diretamente o pacote dependente. Observe alguns tipos de dependências (MEDEIROS, 2004):

**Dependência simples** é quando um elemento de um pacote possui alguma relação com um elemento de outro pacote. Como mostra o exemplo da figura a seguir, que é a representação, em pacotes, da arquitetura em camadas que vimos anteriormente. Neste caso, podemos afirmar que classes do pacote de interface do usuário possuem ligação com classes do pacote de negócios, e assim sucessivamente.

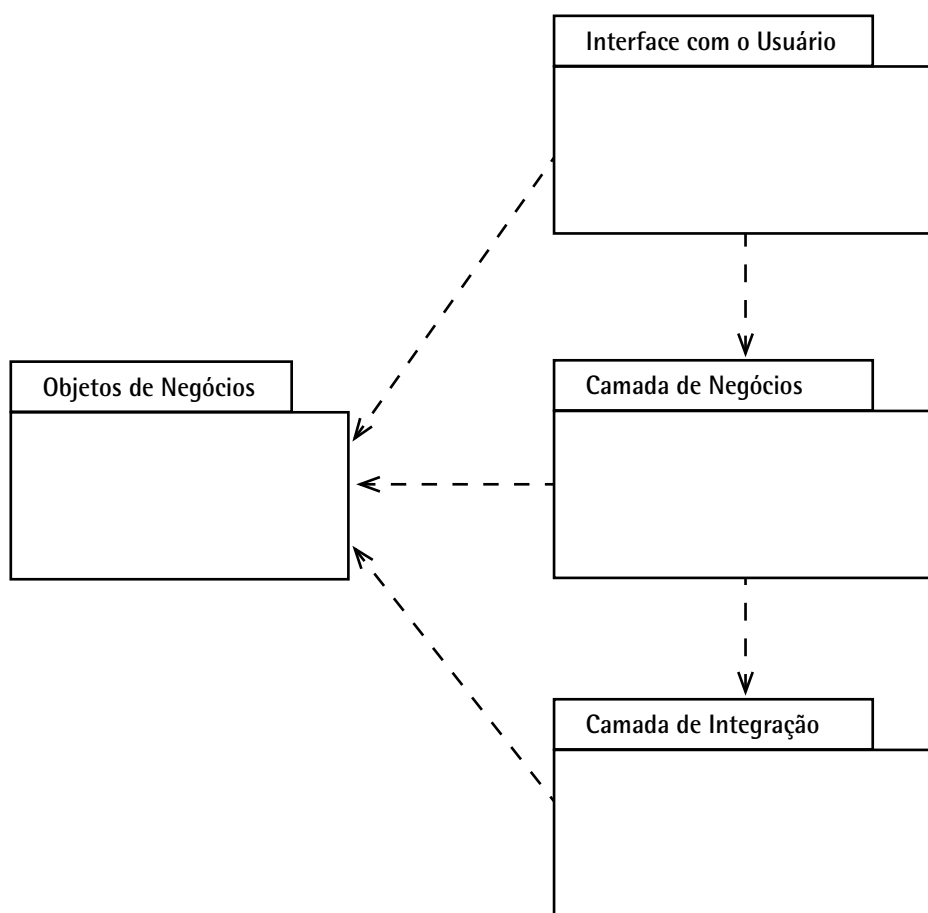


Figura 58 – Exemplo de dependência simples de pacotes

**Dependência com estereótipo de acesso** é quando o pacote dependente é incorporado aos elementos públicos do pacote de destino.

**Dependência com estereótipo de importação** é quando os elementos públicos do pacote de destino são adicionados ao pacote dependente, como mostra a figura a seguir:

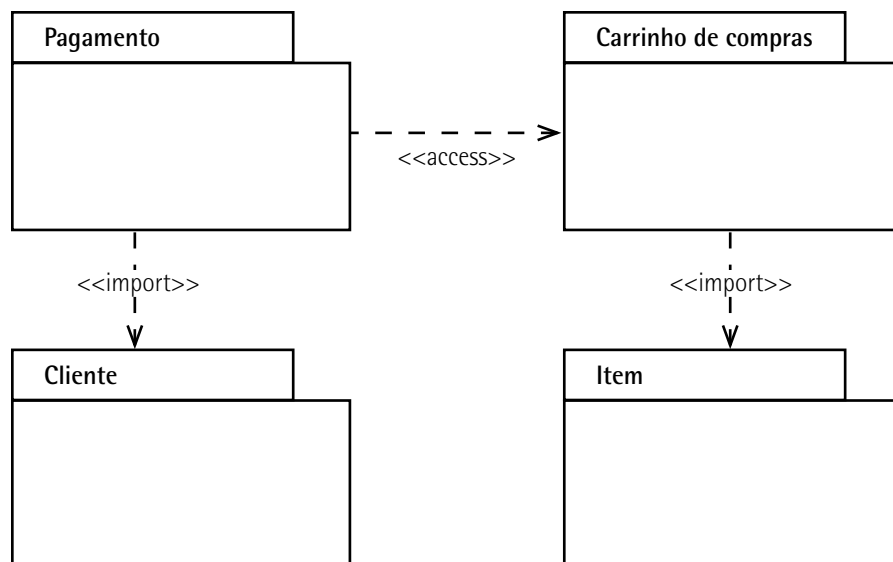


Figura 59 – Exemplo de dependência estereotipada de pacotes

Herança aplicada a pacotes possui o mesmo conceito utilizado em herança do paradigma da orientação a objetos. Utilizamos herança de pacotes quando desejamos representar generalização ou especialização de famílias de pacotes. O exemplo, representado pela figura a seguir, mostra pacotes com classes específicas para desenho de formulários de sistemas *desktop*, ou baseado em janelas.

Nesse caso, temos num pacote-mãe, todos os objetos responsáveis pelo desenho de janelas genéricas, e, nos pacotes-filhos, temos as especializações, para desenho de janelas em sistemas operacionais dos tipos Windows, Linux e Mac.

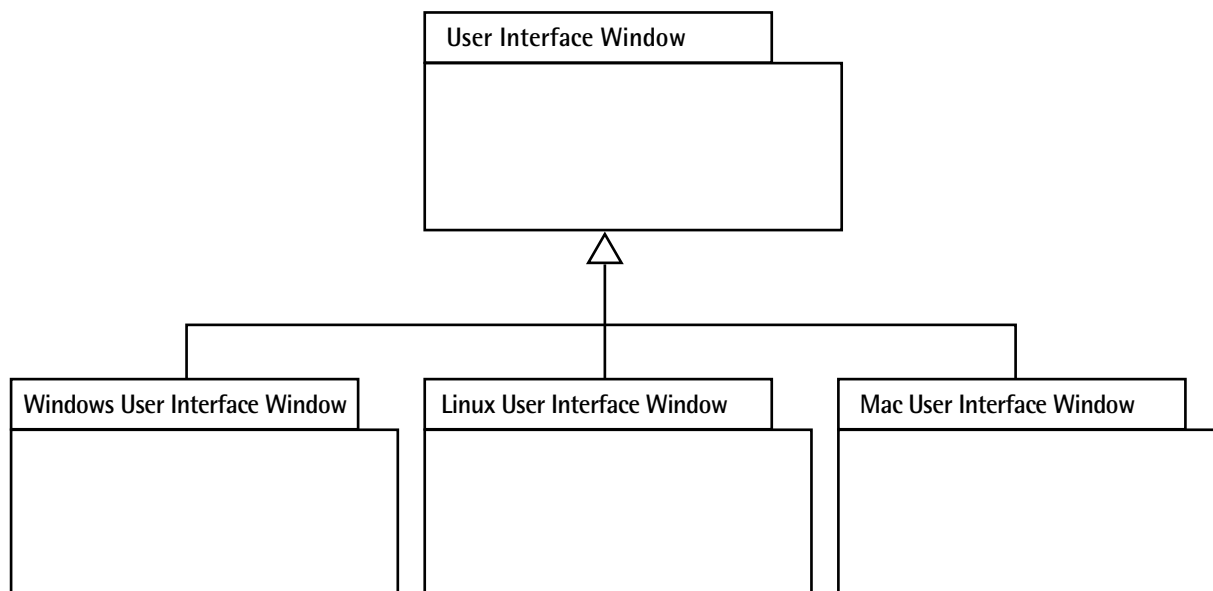


Figura 60 – Exemplo de herança de pacotes



### Saiba mais

A implementação de "pacotes" na linguagem da plataforma Microsoft .Net corresponde ao conceito de *namespace*. Para aprofundar seus conhecimentos sobre o tema, leia:

CAMARA, F. *Orientação a objeto com .NET*. 2. ed. Santa Catarina: Visual Books, 2006.

Assim como "pacotes" corresponde ao conceito de *package* da linguagem Java, como se vê em:

SIERRA, K; BATES, B. *Use a Cabeça! Java*. 2. ed. Rio de Janeiro: Alta Books, 2007.

### 8.1.3 Introdução ao projeto de interfaces com o usuário

O projeto de interfaces com o usuário está ligado diretamente ao fator usabilidade. Abordaremos aspectos introdutórios e que são muito importantes para o sucesso do projeto de *software*.

Dentro de um modelo de atributos de qualidade de *software*, de acordo com a ISO 25010 (2011a), a usabilidade encaixa-se entre os seis principais que delimitam a qualidade de um produto, que são: confiabilidade, funcionalidade, eficiência, manutenibilidade, portabilidade e usabilidade.

Usabilidade está ligada ao projeto, à avaliação e à implementação de sistemas computacionais, contemplando *hardware* e *software*, de tal forma que ajude os usuários em suas tarefas, em um contexto específico, de maneira produtiva (ISO, 2011a; ROCHA; BARANAUSKAS, 2003).

Engenharia de usabilidade é o nome dado ao conjunto de atividades distribuídas na forma estruturada de projeto de sistemas computacionais cujo objetivo é a usabilidade (ROCHA; BARANAUSKAS, 2003).

Mayhew (1999), outra referência quando falamos sobre usabilidade, também define engenharia da usabilidade como uma disciplina que visa fornecer técnicas, processos e ferramentas que possam servir de suporte a um projeto de um sistema computacional com ênfase no usuário.

O principal objetivo da engenharia da usabilidade, qualquer que seja o modelo adotado, é a diminuição do custo no desenvolvimento de um projeto centrado no usuário através da redução nos tempos de desenvolvimento, treinamento e retrabalho (ROCHA; BARANAUSKAS, 2003; NIELSEN, 1993).

É importante mencionar que engenharia de usabilidade não está ligada apenas a atividades, processos e técnicas, mas também à filosofia de considerar o usuário como centro do processo, levando

em consideração o ambiente e aspectos cognitivos e psicológicos na execução de sua rotina (AVELINO, 2005; MAYHEW, 1999; NIELSEN, 1993).

Como podemos notar, o projeto de interface do usuário está ligado à engenharia da usabilidade, que deve ser tratada com muito mais aprofundamento do que simplesmente desenharmos interfaces do usuário.

Shneiderman (1998), por exemplo, aponta como um dos fatores fundamentais de sucesso para um sistema computacional interativo, a preocupação com aspectos inerentes às atividades do usuário, como sua capacidade perceptiva e sua habilidade cognitiva na execução de suas tarefas diárias.

A usabilidade não tem recebido a atenção necessária na fase de elaboração da arquitetura de um sistema computacional, fato este que pode levar ao retrabalho tardio, bem como tornar o sistema menos usável do que poderia ser (GOLDEN; JOHN; BASS, 2005).



### Saiba mais

Para aprofundar seus conhecimentos sobre a preocupação com a usabilidade e os aspectos que influenciam o projeto de um sistema de *software*, leia:

GOLDEN, E.; JOHN, B. E.; BASS, L. The value of a usability-supporting architectural pattern in software architecture design: a controlled experiment. *In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING – ICSE, 2005, St. Louis. Proceedings...* St. Louis: ICSE, 2005.

## 8.2 Projeto de componentes

Antes de pensarmos em como projetar componentes, precisamos ter muito claro o que é um componente, bem como quais os objetivos e os benefícios em se componentizar um *software*.

### 8.2.1 Introdução à componentização e ao reúso de *software*

A componentização, ou o desenvolvimento componentizado de *software*, é um paradigma, tal qual o paradigma da orientação a objetos, que não se limita apenas ao desenvolvimento de componentes, assim como a orientação não se limita a classes a esmo.

Analogamente ao paradigma da orientação a objetos, que define um guia, uma forma de se desenvolver *software*, cujo fundamento é o conceito de objetos, a componentização tem como base o conceito de componentes.

Os cinco princípios básicos de um componente, segundo Cheesman e Daniels (2000), uma das melhores referências no assunto, são:

- Um componente obrigatoriamente deve possuir uma especificação.
- Um componente obrigatoriamente deve possuir uma implementação.
- Um componente obrigatoriamente deve seguir uma padronização.
- Um componente obrigatoriamente deve ter a capacidade de ser empacotado em módulos.
- Um componente obrigatoriamente deve ter a capacidade de ser distribuído.

Um componente de *software* é como um componente de um sistema eletrônico, uma unidade projetada e construída para possuir seus próprios processamento, regras e informações.

Essa unidade pode ser substituída, por qualquer motivo (erro ou evolução), por outra unidade com as mesmas características de relação para com o restante do sistema, sem que haja dano ou modificação no funcionamento do todo.

Nesta disciplina, não conseguiremos abordar todo o paradigma da componentização, mas veremos os aspectos introdutórios fundamentais que fazem a conexão entre o paradigma da orientação a objetos e o da componentização.

Um componente de *software* se dispõe a prestar um serviço para o restante do sistema; por princípio, ele deve ser consistente e conciso com relação às suas responsabilidades, de tal forma que atinja baixo acoplamento e alta coesão.



### Lembrete

Os conceitos de acoplamento e coesão de componentes são os mesmos que aplicamos quando debatemos acoplamento e coesão de classes anteriormente.

Esse serviço prestado pelo componente deve ser visível a outros componentes (clientes) que, porventura, desejem acessar esse serviço. Analogamente ao paradigma da orientação a objetos, é como se cada componente possuísse um método público que pudesse ser chamado por outros componentes.

Podemos dizer que entre o componente que disponibiliza o serviço e o componente que consome esse serviço existe um contrato, uma regra que não pode ser quebrada por nenhuma das partes.

Para o componente que consome o serviço, é indiferente como o componente implementa ou qual a lógica interna que é utilizada para resolver um determinado problema; o importante é que o protocolo de comunicação seja mantido.

A interface de um componente é análoga à que temos no paradigma da orientação a objetos, ou seja, é um contrato que determina **o que** deve ser feito, mas que não estipula **como** deve ser feito.

Essa padronização do protocolo de comunicação entre os componentes e entre os objetos nos garante uma alta capacidade de reúso do componente que oferece um determinado serviço; se um componente quiser utilizar um determinado serviço, bastará adaptar-se a esse protocolo, sem que haja necessidade de alteração no componente fornecedor.

Os paradigmas da componentização e da orientação a objetos se cruzam em um ponto, na medida em que podemos considerar que um componente é uma estrutura de *software* composta por objetos, que, quando compilados, geram estruturas físicas e concisas que podem ser encaradas como um sistema computacional, por exemplo, bibliotecas (dlls), executáveis (exes), tabelas, documentos etc.



### Observação

Componentes e pacotes são semelhantes quanto à função de agrupamento, todavia diferem com relação ao final desse agrupamento: componentes são agrupamentos físicos de objetos, enquanto pacotes são meramente agrupamentos lógicos.

Se temos serviços definidos por interfaces e implementados por classes, podemos dizer que essas interfaces podem ser externas ou exportadoras de serviços desses sistemas de *software*.



### Saiba mais

Para aprofundar seus conhecimentos sobre o assunto, leia:

CHEESMAN, J.; DANIELS, J. *UML components: a simple process for specifying component-based software*. Addison-Wesley, 2000.

GIMENES, I. M. S.; HUZITA, E. H. M. *Desenvolvimento baseado em componentes: conceitos e técnicas*. Rio de Janeiro: Ciência Moderna, 2005.

### 8.2.2 Definindo as interfaces externas

As interfaces externas ao componente, ou ao sistema de *software*, são determinadas pelas suas interfaces, que definem o protocolo, ou o contrato de comunicação entre objetos consumidores e provedores de serviço.

Essa relação de consumo de serviços acaba por criar uma dependência entre esses componentes ou entre objetos. As dependências podem ser classificadas como simples ou estereotipadas.

## Observação

A relação de dependência entre componentes e objetos, nesse contexto, é considerada fraca, uma vez que o consumidor depende apenas da interface, e não da implementação do fornecedor do serviço.

Dependências simples são dependências entre componentes delimitadas pela interface, por exemplo, uma dependência entre um executável e uma DLL, enquanto dependências estereotipadas são dependências particulares de um determinado cenário, por exemplo, dependências entre páginas de um *web site*, que podem ser marcadas como `<<hiperlink>>`.

Para representação e modelagem de componentes e de suas interfaces, utilizaremos o diagrama de componentes da UML.

### 8.2.3 Diagrama de componentes

No diagrama de componentes da UML utilizamos, para representação de componentes, uma caixa com *tabs*, como mostra o exemplo da figura a seguir. Ainda na figura, podemos notar a identificação do nome do componente.

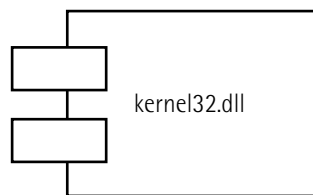


Figura 61 – Exemplo de notação de componentes

Podemos ainda utilizar a estereotipação para identificar o tipo do componente, como mostra o exemplo da figura a seguir:

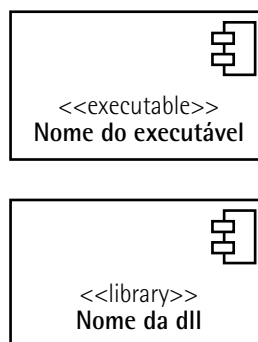


Figura 62 – Exemplo de representação de componentes com estereotipação



O quadro seguinte mostra os possíveis tipos de estereotipação para componentes e seus respectivos significados.

**Quadro 14 – Tipos de componentes e estereotipação**

Estereótipo	Tipo de componente
<<executable>>	Componente que pode ser executado.
<<library>>	Biblioteca, que pode ser estática ou dinâmica.
<<database>>	Banco de dados.
<<table>>	Tabela de um banco de dados.
<<document>>	Documento.
<<file>>	Arquivo, que pode ser desde um arquivo que possui apenas informações a um arquivo com código-fonte.

A representação de interface de componentes, no diagrama da UML, é feita como mostra a figura a seguir. Nessa figura, podemos ler que o componente DLL possui uma interface que é consumida pelo executável; o conector convexo indica a interface, e o conector côncavo indica o consumidor.

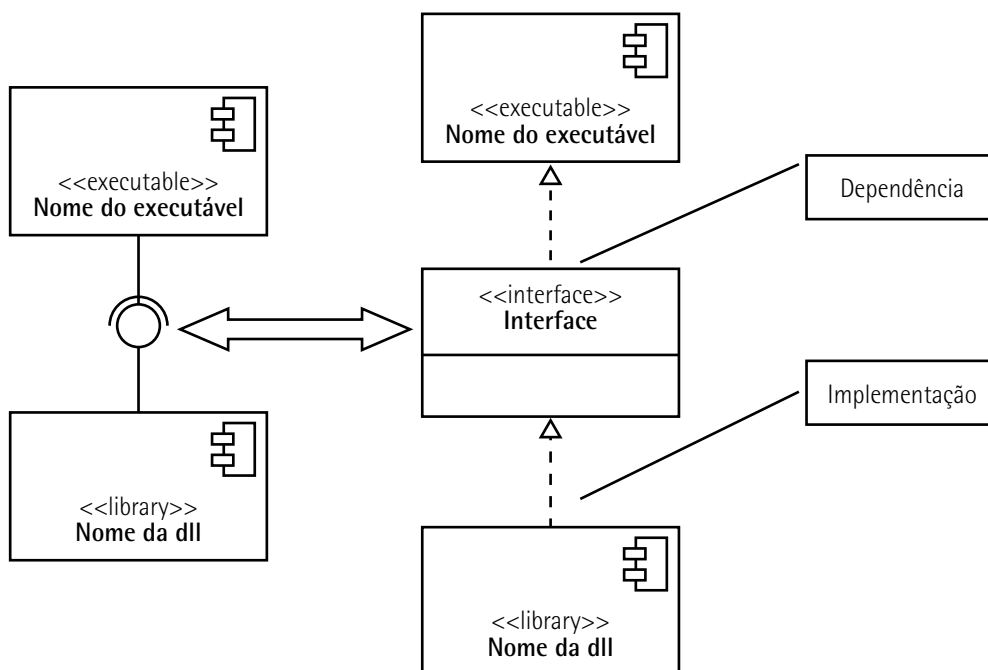


Figura 63 – Exemplo de conexão e de dependência entre componentes

Essa é a representação de dependências simples entre componentes, ou seja, quando há a conexão feita a partir de interfaces. A figura a seguir mostra um exemplo de dependência estereotipada.

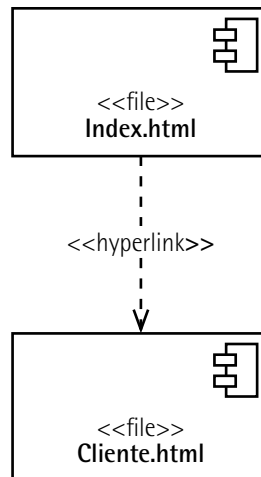


Figura 64 – Exemplo de representação de dependência estereotipada



### Lembrete

Pacotes são agrupamentos lógicos de elementos, que podem ser classes ou casos de uso, como vimos anteriormente; ou podemos utilizar pacotes para agrupar logicamente componentes que possuam funções semelhantes.

O diagrama de componentes representa uma visão mais física do sistema de *software*; as partes desse sistema são apresentadas sob uma perspectiva mais concreta, ou seja, podemos empacotar esses componentes e distribuí-los.

Contudo, esse diagrama não nos dá a visão da organização desses componentes sob o enfoque da organização da estrutura física sobre a qual o *software* será implantado e executado.

Pensando em estrutura física, podemos imaginar que iremos distribuir esses componentes em diversas plataformas, ou que, eventualmente, parte do nosso sistema de *software* será executada em uma plataforma, um sistema operacional, enquanto outra parte poderá ser executada em outra plataforma, ou em outro tipo de sistema operacional.

Como representar, modelar e especificar essas questões importantes e que nem sempre têm a atenção devida? O diagrama de distribuição ou de implantação auxilia-nos nesse primeiro momento.

### 8.2.4 Diagrama de distribuição

O diagrama de distribuição, ou de implantação, mostra como os componentes são configurados para execução, em "nós" de processamento (LARMAN, 2007).

Um "nó" de processamento é um recurso computacional que permite a execução de um sistema de *software*, ou de parte dele, como um componente. Esse "nó" pode ser um computador, um dispositivo móvel ou até mesmo uma estrutura de memória ou um dispositivo periférico.

No diagrama de implantação, utilizamos um cubo para representar um "nó", que "nó" deve possuir um nome e um tipo, como mostra a figura a seguir:

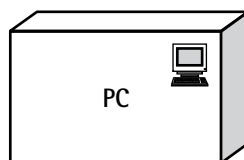


Figura 65 – Exemplo de representação de nó do tipo computador do usuário

Com a visão que nos é fornecida pelo diagrama de implantação, podemos visualizar as dependências entre os "nós" e como se dá a comunicação entre esses "nós".

Por exemplo, em um sistema cliente-servidor, teremos dois "nós": o cliente e o servidor.

Com o diagrama, poderemos deixar clara a dependência entre esses dois "nós", além de especificar o protocolo de comunicação entre esses "nós", por exemplo, se será uma comunicação via HTTP ou HTTPS. Esses detalhes são fundamentais para o dimensionamento da infraestrutura necessária ao sistema de *software*.

Neste diagrama podemos representar os componentes contidos em cada "nó", como mostra o exemplo da figura a seguir:

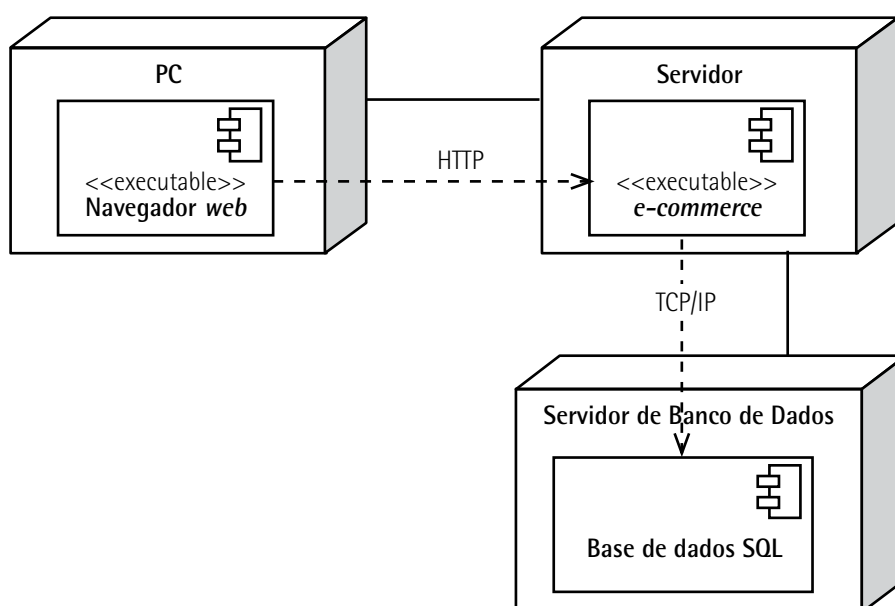


Figura 66 – Exemplo de distribuição de componentes em nós



### Resumo

Nesta unidade demos ênfase a como produzir os modelos referentes às fases de projeto arquitetural, projeto de interfaces e projeto de componentes, uma vez que, na unidade anterior, vimos onde, quais e quando esses modelos seriam utilizados.

Começamos o debate desta unidade explanando que refinar os aspectos dinâmicos da arquitetura de um sistema de *software* significa complementar a visão dada pelo diagrama de sequência, e não substituí-lo.

O objetivo desse refinamento se dá pela necessidade de prover uma melhor qualidade da informação que é passada pela equipe de construção e pelos demais envolvidos no projeto, pois quantidade de informação, como vimos, não necessariamente quer dizer melhor qualidade.

Para complementar a visão do diagrama de sequência, vimos, inicialmente, o diagrama de comunicação, que, nas versões anteriores da UML, era chamado de diagrama de colaboração.

Este diagrama, assim como o de sequência, tem característica comportamental, ou seja, visa à modelagem das interações dos objetos, mas, diferentemente do diagrama de sequência, não dá ênfase à troca de mensagens em uma linha de tempo, mas à interligação e à dependência dos objetos.

Temos, no diagrama de colaboração, três elementos principais de modelagem: o objeto, os vínculos entre eles e as mensagens trocadas, que são etiquetadas em ordem numérica e crescente para que possamos visualizar a sequência em que elas são executadas.

Vimos que os diagramas de sequência e colaboração são análogos, ou seja, as informações contidas em um, obrigatoriamente, aparecem no outro; a diferença fica apenas no enfoque de cada diagrama. No entanto, deve-se ficar atento para ferramentas que promovem a conversão automática dos diagramas.

O último diagrama a complementar a visão comportamental da arquitetura é o diagrama de máquina de estado, ou diagrama de estado, cujo objetivo é representar as mudanças de estado de um objeto no decorrer do tempo.

O estado de um objeto é dado pelo conjunto de valores de seus atributos, que pode ser modificado por um evento do sistema. Esse evento promove uma transição de estado do objeto, e esse processo é objeto de representação desse diagrama a partir de uma máquina finita de estados.

Além dos elementos básicos de modelagem: estado/transição e todos os tipos de mensagens presentes no diagrama de sequência, o diagrama de estado provê mecanismos avançados de representação de fluxos alternativos, paralelos, além de mecanismos para facilitação da representação de estados mais complexos, por exemplo, o pseudoestado de história, o estado composto e o estado de submáquina.

Avançando para o projeto de interfaces, vimos que essa fase pode ser dividida em três níveis: projeto de interfaces internas, externas e interface com o usuário final.

O projeto de interfaces internas dá ênfase a como os objetos internos do sistema de *software* se comunicam, e para representar a organização lógica desses componentes utilizamos o diagrama de pacotes, pois o primeiro passo para a definição do protocolo de comunicação é estabelecer quais objetos possuem interfaces com quais objetos.

O pacote é um mecanismo de organização lógica de elementos de modelagem; esses elementos podem ser: os casos de uso, as classes e os componentes.

O projeto de interface com o usuário envolve a engenharia da usabilidade, que é a área de estudo preocupada em fazer o usuário executar as tarefas do dia a dia de forma produtiva.

O projeto de componentes é um assunto profundo e ligado intimamente ao paradigma da componentização, que cruza com o paradigma da orientação a objetos.

Um componente obrigatoriamente deve possuir: especificação, implantação, padronização e capacidade de ser empacotado e distribuído.

Um componente de *software* é uma unidade autônoma e suficiente, que objetiva baixo acoplamento e alta coesão e promove o reúso. Esse componente é formado por objetos.

Cada componente é um provedor de serviços, e esses serviços são definidos por interfaces, que são contratos que determinam quais

serviços são fornecidos e qual é o protocolo de comunicação entre o cliente e o fornecedor, sem se preocupar com o modo pelo qual esses serviços serão implementados.

Para modelar os componentes de um sistema, utilizamos o diagrama de componentes da UML, sempre lembrando que o mais importante é a definição clara das dependências entre os componentes, dadas a partir das interfaces.

No diagrama de distribuição representamos em que lugar, dentro de uma infraestrutura, esses componentes serão distribuídos. A importância desse diagrama se dá pela visão que temos para mensurar a infraestrutura necessária para o sucesso do projeto.



### Exercícios

**Questão 1.** Um componente de *software* é como um componente de um sistema eletrônico, uma unidade projetada e construída para possuir seu próprio processamento, regras e informações.

Com base nos cinco princípios básicos de um componente, segundo Cheesman e Daniels (2001), analise as afirmativas a seguir:

- I – Um componente obrigatoriamente deve possuir uma especificação em UML.
- II – Um componente obrigatoriamente deve possuir uma implementação.
- III – Um componente opcionalmente deve seguir uma padronização.
- IV – Um componente obrigatoriamente deve ter a capacidade de ser empacotado em módulos.
- V – Um componente obrigatoriamente deve ter a capacidade de ser distribuído.

É correto apenas o que se afirma em:

- A) I e II.
- B) II e IV.
- C) I, II, III e IV.
- D) II, IV e V.
- E) I, II, e IV.

Resposta correta: alternativa D.

### Análise das afirmativas

I – Afirmativa incorreta.

Justificativa: **não há menção da** obrigatoriedade da linguagem UML para a especificação do componente na descrição dos cinco princípios básicos de um componente, segundo Cheesman e Daniels (2001).

II – Afirmativa correta.

Justificativa: um componente obrigatoriamente deve possuir uma implementação.

III – Afirmativa incorreta.

Justificativa: um componente deve **obrigatoriamente**, e não opcionalmente, seguir uma padronização.

IV – Afirmativa correta.

Justificativa: um componente obrigatoriamente deve ter a capacidade de ser empacotado em módulos.

V – Afirmativa correta.

Justificativa: um componente obrigatoriamente deve ter a capacidade de ser distribuído.

**Questão 2.** O diagrama de comunicação é um tipo de diagrama de comportamental da UML, que representa as interações entre dois objetos e suas partes utilizando para isso uma sequência de mensagens representadas de forma livre de formatação (UML *Diagrams*, 2015).

De acordo com os conceitos do diagrama de comunicação, analise as duas afirmativas apresentadas a seguir:

O diagrama de comunicação dá ênfase em como os objetos estão interligados e quais mensagens são trocadas entre eles para realizar uma determinada tarefa.

### Porque

No diagrama de comunicação, as mensagens possuem uma numeração, é como se elas fossem "etiquetadas" com uma numeração, em ordem crescente, e é essa sequência numérica que representa a sequência nas quais as mensagens são trocadas entre os objetos.

Acerca dessas afirmativas, assinale a alternativa correta:

- A) As duas afirmativas são proposições verdadeiras e a segunda é uma justificativa correta da primeira.
- B) As duas afirmativas são proposições verdadeiras e a segunda não é uma justificativa correta da primeira.
- C) A primeira afirmativa é uma proposição verdadeira e a segunda é uma proposição falsa.
- D) A primeira afirmativa é uma proposição falsa e a segunda é uma proposição verdadeira.
- E) As duas afirmativas são proposições falsas.

**Resolução desta questão na plataforma.**



## FIGURAS E ILUSTRAÇÕES

### Figura 1

SOMMERVILLE, I. *Engenharia de software*. São Paulo: Pearson, 2010. p. 30. Adaptado.

### Figura 4

PRESSMAN, R. S. *Engenharia de software*. 6. ed. São Paulo: Mcgraw-hill, 2006. p. 208. Adaptado.

### Figura 8

MEDEIROS, L. F. *Banco de dados: princípios e prática*. Curitiba: InterSaberes, 2013. Adaptado.

### Figura 9

SAXENA, V.; PRATAP, A. Performance comparison between relational and object-oriented databases. *International Journal of Computer Applications*, p. 6-9, 2013. Adaptado.

### Figura 10

BEZERRA, E. *Princípios de análise e projeto de sistemas com UML: um guia prático para modelagem de sistemas orientados a objetos através da linguagem de modelagem unificada*. Rio de Janeiro: Campus, 2006. p. 168. Adaptado.

### Figura 11

BEZERRA, E. *Princípios de análise e projeto de sistemas com UML: um guia prático para modelagem de sistemas orientados a objetos através da linguagem de modelagem unificada*. Rio de Janeiro: Campus, 2006. p. 168. Adaptado.

### Figura 12

BEZERRA, E. *Princípios de análise e projeto de sistemas com UML: um guia prático para modelagem de sistemas orientados a objetos através da linguagem de modelagem unificada*. Rio de Janeiro: Campus, 2006. p. 168. Adaptado.

### Figura 17

BOOCH, G.; JACOBSON, I.; RUMBAUGH, J. *UML: guia do usuário*. 2. ed. Rio de Janeiro: Campus, 2006. p. 258. Adaptado.

### Figura 18

BOOCH, G.; JACOBSON, I.; RUMBAUGH, J. *UML: guia do usuário*. 2. ed. Rio de Janeiro: Campus, 2006. p. 258. Adaptado.

### Figura 19

ELMASRI, R.; NAVATHE, S. B. *Sistemas de banco de dados*. 6. ed. São Paulo: Addison-Wesley, 2011. p. 133. Adaptado.

### Figura 30

MERSON, P. *How to represent the architecture of your application using UML 2.0 and more*. Pittsburgh: Software Engineering Institute, 2006. Adaptado.

### Figura 39

TIAKO, P. F. *Software applications: concepts, methodologies, tools, and applications*. 1. ed. Langston: Information Science Reference, 2009. p. 597. Adaptado.

### Figura 41

UML-DIAGRAMS. *UML communication diagrams overview*, 2009-2014. Disponível em: <<http://www.uml-diagrams.org/communication-diagrams.html>>. Acesso em: 27 abr. 2015. Adaptado.

### Figura 52

LARMAN, C. *Utilizando UML e padrões: uma introdução à análise e ao projeto orientados a objetos e ao processo unificado*. 2. ed. Porto Alegre: Bookman, 2007. Adaptado.

### Figura 53

LARMAN, C. *Utilizando UML e padrões: uma introdução à análise e ao projeto orientados a objetos e ao processo unificado*. 2. ed. Porto Alegre: Bookman, 2007. Adaptado.

### Figura 56

LARMAN, C. *Utilizando UML e padrões: uma introdução à análise e ao projeto orientados a objetos e ao processo unificado*. 2. ed. Porto Alegre: Bookman, 2007. Adaptado.

### Figura 59

UML-DIAGRAMS. *UML package diagrams overview*, 2009-2014. Disponível em: <<http://www.uml-diagrams.org/timing-diagrams.html>>. Acesso em: 28 abr. 2015. Adaptado.

## Figura 66

LARMAN, C. *Utilizando UML e padrões: uma introdução à análise e ao projeto orientados a objetos e ao processo unificado*. 2. ed. Porto Alegre: Bookman, 2007. p. 585. Adaptado.

## REFERÊNCIAS

### Textuais

ACOPLAMENTO. *In: Michaelis: moderno dicionário da língua portuguesa*. São Paulo: Companhia Melhoramentos, 1998.

ALBIN, S. T. *The art of software architecture: design methods and techniques*. Indianapolis: John Wiley, 2003.

ARQUITETURA. *In: Michaelis: moderno dicionário da língua portuguesa*. São Paulo: Companhia Melhoramentos, 1998.

AVELINO, V. F. *Merusa: metodologia de especificação de requisitos de usabilidade e segurança orientada para arquitetura*. 2005. Tese (Doutorado em Engenharia) – Escola Politécnica da Universidade de São Paulo, São Paulo, 2005.

BASS, L.; CLEMENTS, P.; KAZMAN, R. *Software architecture in practice*. Boston: Addison-Wesley, 2010.

BEZERRA, E. *Princípios de análise e projeto de sistemas com UML: um guia prático para modelagem de sistemas orientados a objetos através da linguagem de modelagem unificada*. Rio de Janeiro: Campus, 2006.

BOOCH, G.; JACOBSON, I.; RUMBAUGH, J. *UML: guia do usuário*. 2. ed. Rio de Janeiro: Campus, 2006.

BROOKS, F. No silver bullet: essence and accidents of software engineering. *Computer*, v. 20, n. 4, p. 10-9, 1987.

BUSCHMANN, F. *et al. Pattern-oriented software architecture: a system of patterns*. New York: Wiley, 1996. v. 1.

BUSCHMANN, F.; HENNEY, K.; SCHIMIDT, D. *Pattern-oriented software architecture: a pattern language for distributed computing*. New York: Wiley, 2007. v. 4.

CAMARA, F. *Orientação a objeto com .NET*. 2. ed. Santa Catarina: Visual Books, 2006.

CATTELL, R. G. *et al. The object data management standard: ODMG 3.0*. California: Morgan Kaufmann, 2000.

CHEESMAN, J.; DANIELS, J. *UML components: a simple process for specifying component-based software*. Addison-Wesley, 2000.

CHIAVENATO, I. *Os novos paradigmas: como as mudanças estão mexendo com as empresas*. Barueri: Manole, 2008.

COAD, P.; YOURDON, E. *Projetos baseados em objetos*. Rio de Janeiro: Campus, 1993.

COESO. *In: Michaelis: moderno dicionário da língua portuguesa*. São Paulo: Companhia Melhoramentos, 1998.

CORTÊS, M. L.; CHIOSSI, T. C. S. *Modelos de qualidade de software*. Campinas: Unicamp, 2001.

COSTA, I. *et al. Qualidade em Tecnologia da Informação*. São Paulo: Atlas, 2013.

COUTAZ, J. PAC, an object oriented model for dialog design. *In: HUMAN-COMPUTER INTERACTION – INTERAC' 87, 1987, Stuttgart. Proceedings...* Stuttgart: Elsevier Science Publishers, 1987. p. 431-6. Disponível em: <<http://iihm.imag.fr/publs/1987/Interact87.PAC.pdf>>. Acesso em: 24 abr. 2015.

DOFACTORY. *.NET design pattern framework 4.5*, [s.d.]. Disponível em: <<http://www.dofactory.com/products/net-design-pattern-framework>>. Acesso em: 4 maio 2015.

DRIVER, M. Java and .NET: you can't pick a favorite child. *In: DEVELOPER SUMMIT, 2007, Palm Springs. Proceedings...* Palm Springs, 2007. Disponível em: <[http://proceedings.esri.com/library/userconf/devsummit07/papers/keynote\\_presentation-mark\\_driver\\_gartner.pdf](http://proceedings.esri.com/library/userconf/devsummit07/papers/keynote_presentation-mark_driver_gartner.pdf)>. Acesso em: 22 abr. 2015.

ELMASRI, R.; NAVATHE, S. B. *Sistemas de banco de dados*. 6. ed. São Paulo: Addison-Wesley, 2011.

FREEMAN, E *et al. Head first design patterns*. Sebastopol: O'Reilly, 2004.

GAMMA, E. *et al. Design patterns: elements of reusable object-oriented software*. Boston: Addison-Wesley, 1995.

GIMENES, I. M. S.; HUZITA, E. H. M. *Desenvolvimento baseado em componentes: conceitos e técnicas*. Rio de Janeiro: Ciência Moderna, 2005.

GOLDEN, E.; JOHN, B. E.; BASS, L. The value of a usability-supporting architectural pattern in software architecture design: a controlled experiment. *In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING – ICSE, 2005, St. Louis. Proceedings...* St. Louis: ICSE, 2005.

GUEDES, G. T. A. *UML 2: uma abordagem prática*. São Paulo: Novatec, 2009.

GUERRA, A. C.; COLOMBO, R. M. T. *Tecnologia da informação: qualidade de produto de software*. Brasília: PBQP, 2009.

HEINEMAN G. T.; COUNCILL, W. T. *Component-based software engineering: putting the pieces together*. Boston: Addison-Wesley Longman Publishing, 2001.

HUANG, R. Making active CASE tools: toward the next generation CASE tools. *Software Engineering Notes*, v. 23, p. 47-50, janeiro, 1998.

INTERNATIONAL ORGANIZATION FOR STANDARDIZATION (ISO). *ISO 10746-3: open distributed processing – Reference model*. Geneve: ISO, 2009.

\_\_\_\_. *ISO 25010: systems and software engineering – Systems and software quality requirements and evaluation (square) – system and software quality models*. Geneve: ISO, 2011a.

\_\_\_\_. *ISO 42010: systems and software engineering – Architecture description*. Geneve: ISO, 2011b.

KRUCHTEN, P. The 4+1 view model of architecture. *IEEE Software*, Washington, v. 12, n. 6, p. 42-50, nov. 1995.

LARMAN, C. *Utilizando UML e padrões: uma introdução à análise e ao projeto orientados a objetos e ao processo unificado*. 2. ed. Porto Alegre: Bookman, 2007.

LEE, R. C; TAPFENHART, W. M. *UML e C++: guia prático de desenvolvimento orientado a objeto*. São Paulo: Makron Books, 2001.

MAGDY, A. *et al. Flexible CASE Tools for Requirements Engineering: A benchmarking Survey*. Cairo, p. 20-6, 2008.

MAHDY, A. A. E-R.; AHMED, M. M. A. E-S.; ZAHRAN, S. M. Flexible CASE tools for requirements engineering: a benchmarking survey. In: INTERNATIONAL CONFERENCE OF INFORMATICS AND SYSTEMS, 2008. *Proceedings...* Cairo, 2008. Disponível em: <[http://infos2008.fci.cu.edu.eg/infos/se\\_03\\_p020-026.pdf](http://infos2008.fci.cu.edu.eg/infos/se_03_p020-026.pdf)>. Acesso em: 17 abr. 2015.

MARTINS, C. T. K.; RODRIGUES, M. *Algoritmos elementares C++*. São Paulo: LTC, 2006.

MAYHEW, D. J. *The usability engineering lifecycle: a practitioner's handbook for user interface design*. São Francisco: Morgan Kaufmann, 1999.

MCGLAUGHLIN, R. Some Notes on Program Design, *Software Engineering Notes*, v. 16, n. 4, p. 53-4, Oct. 1991.

MEDEIROS, L. F. *Banco de dados: princípios e prática*. Curitiba: InterSaberes, 2013.

MEDEIROS, S. E. *Desenvolvendo software com UML 2.0: definitivo*. São Paulo: Makron Books, 2004.

MERSON, P. *How to represent the architecture of your application using UML 2.0 and more*. Pittsburgh: Software Engineering Institute, 2006.

MICHAELIS. *Moderno dicionário da língua portuguesa*. São Paulo: Melhoramentos, 1998.

NIELSEN, J. *Usability engineering*. São Francisco: Morgan Kaufmann, 1993.

OBJECT MANAGEMENT ARCHITECTURE. *Object Management Group*, 2014. Disponível em: <<http://www.omg.org/oma/>>. Acesso em: 17 abr. 2015.

OBJECT MANAGEMENT GROUP (OMG). *Corba basics*, 2015. Disponível em: <<http://www.omg.org/gettingstarted/corbafaq.htm>>. Acesso em: 5 maio 2015.

OPENUP. *Introduction to OpenUP*. [s.d.]a. Disponível em: <[http://epf.eclipse.org/wikis/openup/publish.openup.base/guidances/supportingmaterials/introduction\\_to\\_openup\\_EFA29EF3.html?nodeId=e073de25](http://epf.eclipse.org/wikis/openup/publish.openup.base/guidances/supportingmaterials/introduction_to_openup_EFA29EF3.html?nodeId=e073de25)>. Acesso em: 4 maio 2015.

\_\_\_\_. *Role: analyst*, [s.d.]c. Disponível em: <[http://epf.eclipse.org/wikis/openup/core.default.role\\_def.base/roles/analyst\\_39D7C49B.html](http://epf.eclipse.org/wikis/openup/core.default.role_def.base/roles/analyst_39D7C49B.html)>. Acesso em: 4 maio 2015.

\_\_\_\_. *Role: architect*, [s.d.]b. Disponível em: <[http://epf.eclipse.org/wikis/openup/core.default.role\\_def.base/roles/architect\\_E7A12309.html](http://epf.eclipse.org/wikis/openup/core.default.role_def.base/roles/architect_E7A12309.html)>. Acesso em: 4 maio 2015.

\_\_\_\_. *Role: stakeholder*, [s.d.]d. Disponível em: <[http://epf.eclipse.org/wikis/openup/core.default.role\\_def.base/roles/stakeholder\\_9FFD4106.html](http://epf.eclipse.org/wikis/openup/core.default.role_def.base/roles/stakeholder_9FFD4106.html)>. Acesso em: 4 maio 2015.

PFLEEGER, S. L. *Engenharia de software: teoria e prática*. 2. ed. São Paulo: Prentice Hall, 2004.

PRESSMAN, R. S. *Engenharia de software*. 6. ed. São Paulo: McGraw-Hill, 2006.

ROCHA, H. V.; BARANAUSKAS, M. C. C. *Design e avaliação de interfaces humano-computador*. Campinas: Instituto de Computação da Universidade Estadual de Campinas, 2003.

SAXENA, V.; PRATAP, A. Performance comparison between relational and object-oriented databases. *International Journal of Computer Applications*, p. 6-9, 2013.

SHNEIDERMAN, B. *Designing the user interface: strategies for effective human-computer interaction*. 3. ed. Menlo Park: Addison-Wesley, 1998.

SIERRA, K; BATES, B. *Use a Cabeça! Java*. 2. ed. Rio de Janeiro: Alta Books, 2007.

SILBERSCHATZ, A.; KORTH, H. F.; SUDARSHAN, S. *Sistema de banco de dados*. 5. ed. Rio de Janeiro: Campus, 2006.

SOFTWARE ENGINEERING INSTITUTE (SEI). *Architecture tradeoff analysis method*, 2015a. Disponível em: <[www.sei.cmu.edu/architecture/tools/evaluate/atam.cfm](http://www.sei.cmu.edu/architecture/tools/evaluate/atam.cfm)>. Acesso em: 4 maio 2015.

\_\_\_\_. *Defining software architecture*, 2015b. Disponível em: <<http://www.sei.cmu.edu/architecture/>>. Acesso em: 4 maio 2015.

SOMMERVILLE, I. *Engenharia de software*. São Paulo: Pearson, 2010.

STADZISZ, P. C. *Projeto de software usando a UML*. Paraná: UTFPR, 2002.

SZYPERSKI, C. *Component software: beyond object-oriented programming*. 2. ed. Boston: Addison-Wesley Longman, 2002.

THE HISTORY OF SMALLTALK. *Smalltalk.org*, 2010. Disponível em: <<http://www.smalltalk.org/smalltalk/history.html>>. Acesso em: 17 abr. 2015.

TIMING DIAGRAMS. *UML Diagrams*, 2009-2014. Disponível em:<<http://www.uml-diagrams.org/timing-diagrams.html>>. Acesso em: 28 abr. 2015.

TIAKO, P. F. *Software applications: concepts, methodologies, tools, and applications*. Langston: Information Science Reference, 2009.

TSUDA, M. *et al.* Productivity analysis of software development with an integrated CASE tool. *In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING*, 14., 1992, Melbourne. *Proceedings...* Melbourne: ICSE, 1992.

UML-DIAGRAMS. *Timing diagrams*, 2009-2014a. Disponível em:<<http://www.uml-diagrams.org/timing-diagrams.html>>. Acesso em: 28 abr. 2015.

\_\_\_\_. *UML 2.5 Diagrams Overview*, 2009-2014b. Disponível em:<<http://www.uml-diagrams.org/uml-25-diagrams.html>>. Acesso em: 28 abr. 2015.

VERSOLATTO, F. R. *Uma abordagem arquitetural aplicada ao ciclo de vida da Engenharia da Usabilidade em prontuário eletrônico de pacientes*. 2012. Dissertação (Mestrado em Engenharia de Software) – Instituto de Pesquisas Tecnológicas de São Paulo (IPT), São Paulo, 2012.

## **Sites**

<<http://epf.eclipse.org/wikis/openup/>>.

<<http://layerguidance.codeplex.com/>>.

<<http://msdn.microsoft.com/en-us/vstudio/aa496123.aspx>>.

<<http://www.odbms.org/>>.

<<http://www.omg.org/spec/CORBA/>>.

<<http://www.oracle.com/technetwork/java/index.html>>.

<<http://www.smalltalk.org/>>.

<<http://staruml.io/>>.

<<http://uml-directory.omg.org/>>.

<<http://www.uml.org/>>.



Handwriting practice lines consisting of 30 horizontal rows. Each row is defined by three horizontal lines: a top line, a middle line, and a bottom line, providing a guide for letter height and placement.





Lined writing area with horizontal lines.



Handwriting practice lines consisting of 30 horizontal blue lines. Each line is preceded by a small blue dot on the left margin, serving as a starting point for letter formation.



Handwriting practice lines consisting of 30 horizontal blue lines. Each line is preceded by a small blue dot, serving as a guide for letter height and placement.





# Interativa

Informações:  
[www.sepi.unip.br](http://www.sepi.unip.br) ou 0800 010 9000