Capítulo 4

Linguagem unificada de modelagem (UML)

- Orientação a objetos
- As várias opções da UML
- O diagrama da UML
- Exemplo de desenvolvimento de projetos utilizando UML
- Considerações finais
- Referências bibliográficas

Nosso objetivo nesse capítulo é apresentar a UML (Linguagem Unificada de Modelagem), que possibilita visualização, especificação, construção e documentação de artefatos de um sistema complexo de software - o software orientado a objetos. Ou seja, vamos estudar a linguagem de definição desses softwares. Começaremos por um rápido histórico, mas só entraremos propriamente no estudo da UML após vermos os principais conceitos do paradigma orientado a objetos. Além dos conceitos mais relevantes do modelo, mostraremos, sempre que for possível, sua representação na UML, para que você vá se acostumando à linguagem.

Quando vários autores propunham metodologias para o desenvolvimento de software orientado a objetos, três estudiosos se juntaram e criaram uma linguagem unificada de modelagem, a UML. Estamos falando dos norte-americanos Grady Booch, e James Rumbaugh e do suíço Ivar Jacobson, que, em 1995, lançaram a UML 0.8 , unificando os respectivos métodos, os quais, na verdade, estavam confluindo naturalmente:

- Método de Booch, desenvolvido por Grady Booch, da Rational Software Corporation, expressivo principalmente nas fases de projeto e construção de sistemas;
- OOSE (Object-Oriented Software Engineering), de Ivar Jacobson, que fornecia excelente suporte para casos de usos como forma de controlar a captura de requisitos, a análise e o projeto de alto nível;
- OMT (Object Modeling Technique), de James Rumbaught, que era mais útil para análise e sistemas de informações com o uso intensivo de dados.

Mas podemos dizer que essa história começou bem antes, nos anos 1960, com Alan Curtis Kay, na Universidade de Utah, Estados Unidos. Considerado um dos pais da orientação a objetos com sua "analogia algébrico-biológica", Kay postulou que o computador ideal deveria funcionar como um organismo vivo, isto é, cada célula, mesmo autônoma, deveria se relacionar com outras a fim de alcançar um objetivo. As células poderiam também se reagrupar para resolver outros problemas ou desempenhar outras funções. Kay, que era pesquisador da Xerox quando surgiu a linguagem Smalltalk, no centro de pesquisas da empresa, em 1970, foi o primeiro a usar o termo orientação a objetos.

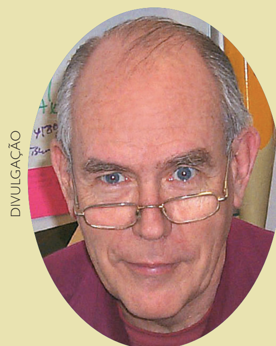
Por oferecer ferramentas que podem ser utilizadas em todas as fases do desenvolvimento de software, a UML acabou se tornando padrão de desenvolvimento de software orientado a objetos.

4.1. Orientação a objetos

A orientação a objetos é um projeto antigo na área de informática e traz consigo a idéia de aproximar o desenvolvimento de software do mundo real, criando elementos que tenham dados e funcionalidades em si mesmos. Mas sua implementação plena ainda está por vir, pois ainda hoje são várias as dificuldades no

Origem e evolução da UML

1962-1963


DIVULGAÇÃO

Ivan Sutherland

1962 - Krysten Nygaard e Ole-Johan Dahl, pesquisadores do Centro de Computação da Noruega, em Oslo, criam a linguagem Simula, derivada do Algol, introduzindo os primeiros conceitos de orientação a objetos.

1963 - Ivan Sutherland desenvolve, no MIT (Massachusets Institute of Tecnology), nos Estados Unidos, o Sketchpad, primeiro editor gráfico para desenhos orientado a objetos. O Sketchpad usava conceitos de instância e herança.

DIVULGAÇÃO

Krysten Nygaard

1969

AP PHOTO/IMAGEPLUS

Alan Curtis Kay

Alan Curtis Kay apresenta sua tese de doutorado intitulada *The Reative Engine* na Universidade de Utah, na qual propõe uma linguagem (Flex), que contém conceitos de orientação a objetos.

1970-1983

É lançada a linguagem de programação Smalltalk, desenvolvida no centro de pesquisas da Xerox, em Palo Alto, Estados Unidos. Essa foi por muito tempo a única linguagem 100% orientada a objetos.

1980 - Surge a linguagem de programação C++, que também pode ser orientada a objetos.

1983 - Jean Ichbiah cria a linguagem ADA a pedido do Departamento de Defesa dos Estados Unidos, também orientada a objetos.

caminho Há limitações de hardware, que se relacionam a problemas de acesso e armazenamento de dados, e de software, relativas a processos do sistema operacional e a falta de sistemas gerenciadores de banco de dados orientados a objetos.

4.1.1. Abstração

É característica essencial de uma entidade, que a diferencia de todos os outros tipos de entidades. Uma abstração define uma fronteira relativa à perspectiva do observador, segundo Booch, Rumbaugh e Jacobson (2005).

Como já dissemos anteriormente, abstração é a capacidade de fixar a atenção apenas nos detalhes relevantes para a construção da solução dentro de seu escopo. Isto é, quando penso em um cliente, por exemplo, não preciso pensar em todos os atributos de um cliente, mas apenas nos atributos que interessam para a solução do problema em questão. Como também vimos no capítulo 2, um cliente pode ser apenas um número.

4.1.2. Classe

De acordo com Booch, Rumbaugh e Jacobson (2005), classe é uma descrição de um conjunto de objetos que compartilham os mesmos atributos, operações, relacionamentos e semântica. A representação completa de uma classe tem quatro divisões, conforme conceituamos e mostramos na figura 85.

Nome da classe – Cada classe deve ter um nome que a diferencie das outras classes (BOOCH, RUMBAUGH e JACOBSON, 2005).
Atributo – É uma propriedade nomeada de uma classe, que descreve um intervalo de valores que as instâncias da propriedade podem apresentar (BOOCH, RUMBAUGH e JACOBSON, 2005).

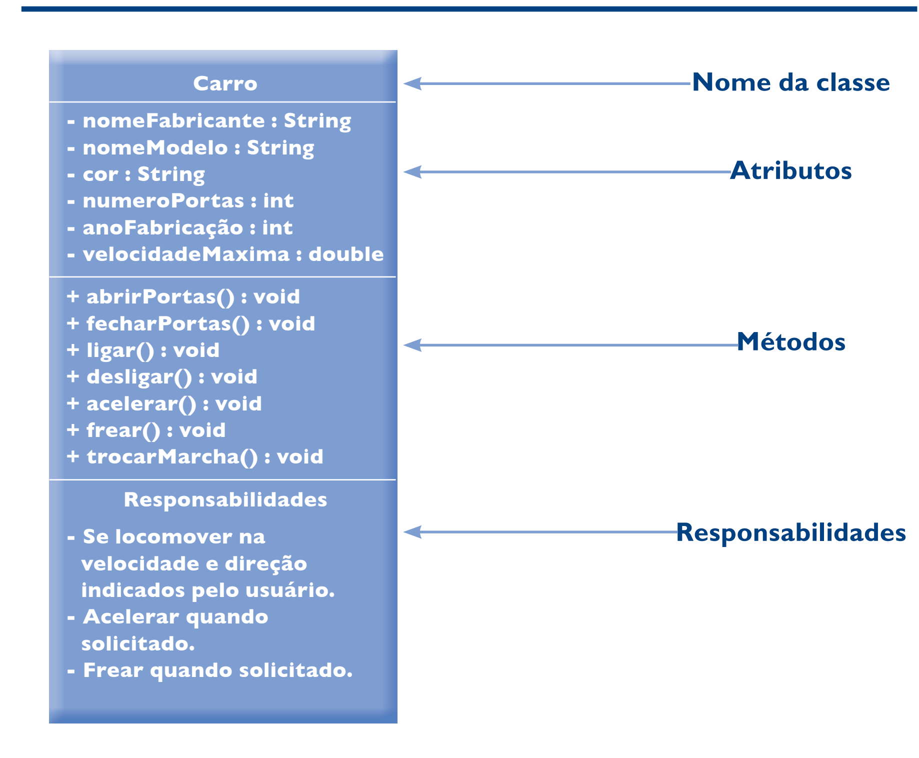


Figura 85
Definição da classe segundo UML 2.

4.1.2.1. Método

É a implementação de um serviço que pode ser solicitado por um objeto da classe para modificar o seu comportamento, algo que pode ser feito com um objeto e que é compartilhado por todos os objetos dessa classe (BOOCH, RUMBAUGH e JACOBSON, 2005). Existem alguns métodos especiais em praticamente todas as classes, os quais, geralmente, não representamos nos diagramas da UML por

1985-1989

DIVULGAÇÃO

Bertrand Meyer

Bertrand Meyer lança a linguagem Eifel, orientada a objetos.

1988 - Sally Shlaer e Stepen Mellor publicam o livro *Object-Oriented Systems Analysis: Modeling the World in Data*, que propõe uma técnica para análise e projetos orientados a objeto.

1989 - É criado o OMG (Object Management Group), que aprova padrões para aplicações orientadas a objeto.

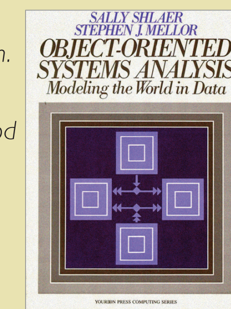
1990-1993

Peter Coad e Ed Yourdon lançam o livro *Object-Oriented Analysis*, também contendo técnicas para análise e projetos orientados a objeto.

Rebecca Wirfs-Brock, Brian Wilkerson e Lauren Wiener publicam o livro *Designing Object-Oriented Software*, tratando de técnicas de modelagem orientada a objetos.

1992 - Ivar Jacobson, Magnus Christerson, Patrik Jonsson e Gunnar Overgaard lançam o livro *Object-Oriented Software Engineering: A Use Case Driven Approach*, também sobre técnicas de orientação a objetos.
D. Embley, B. Kurtz, e S. Woodfield publicam o livro *Object-Oriented System Analysis. A Model-Driven Approach.*

1993 - Grady Booch lança seu *Booch Method* com técnicas para análise e projeto orientado a objetos.



DIVULGAÇÃO

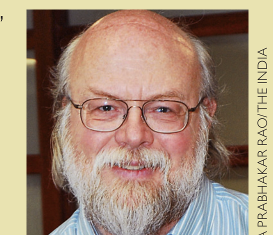
1994-1995

James Martin e Jim Odell lançam OOIE (Object-Oriented Information Engineering).

1995 - Grupo de desenvolvedores da Sun Microsystems, da Califórnia, Estados Unidos, chefiado por James Gosling, cria a linguagem Java. James Rumbaugh publica sua OMT (Object Modeling Technique).
Os metodologistas Grady Booch, James Rumbaugh e Ivar Jacobson lançam a UML 0.8 (em 1996, surgirá a UML 0.9; em 1997, a UML 1.0; em 2000, a UML 1.4; e em 2005, a UML 2.0).

DIVULGAÇÃO

James Martin


A PRABHAKAR RAO/THE INDIA TODAY GROUP/GETTY IMAGES

James Gosling

já terem se tornado senso comum entre os desenvolvedores. Mas, sempre que você achar necessário, poderá defini-los dentro da classe.

Os métodos especiais

- **Construtor:** é o método que constrói, isto é, reserva o espaço em memória onde serão armazenadas as informações daquele objeto da classe.
- **Get:** é o método que apresenta o valor armazenado em determinado atributo de um objeto.
- **Set:** dá um valor a um atributo.

Os métodos get e set são muito úteis para preservar os atributos e garantir que sua alteração seja feita unicamente por intermédio deles. Chamamos isso de encapsulamento dos atributos de uma classe, pois podemos deixar todos eles com visibilidade privada e só manipulá-los utilizando os métodos get (para retornar o valor que está no atributo) e set (para atribuir um valor a ele). Geralmente adotamos um método set e um método get para cada **atributo da classe**.

Você deve estudar mais sobre o funcionamento e a utilização de construtores, getters e setters no livro *Programação de Computadores*, desta coleção, e nos sites que abordem a linguagem Java e linguagem C#, além de sua representação UML.

- **Destrutor:** destrói o objeto criado da memória, liberando o espaço de memória alocado na sua criação. Não é necessário criá-lo em linguagens orientadas a objetos que possuam **garbage collector**, isto é, que excluam os objetos que já não tenham referência alguma na memória.

O garbage collector procura o lixo existente na memória, ou seja, os objetos que não são utilizados e, no entanto, ocupam espaço na memória. Essa tarefa pode ser executada automaticamente ou programada.

- **Assinatura:** é a primeira linha da definição de um método, no qual podemos observar sua visibilidade, seu nome, seus parâmetros de entrada e de retorno.

Exemplo: + soma(valor1:double,valor2:double):double

A assinatura do método mostrado no exemplo acima indica que:

- o símbolo + no início da assinatura demonstra que se trata de um método público, ou seja, que todos os objetos que tiverem acesso à classe a que pertencem também poderão acessá-lo;
- o nome do método é soma;
- o método soma recebe como parâmetros de entrada dois valores do tipo double, chamados valor1 e valor2, e devolve como resultado um número do tipo **double**.

Por meio da assinatura do método podemos obter várias informações sobre sua utilização e, em alguns casos, seu funcionamento. Portanto, para facilitar a compreensão sobre o que o método faz e como devemos utilizá-lo, precisamos ter muito cuidado na definição dessas assinaturas.

4.1.2.2. Responsabilidades

São contratos ou obrigações de determinada classe. Ao criarmos uma classe, estamos criando uma declaração de que todos os seus objetos têm o mesmo tipo de estado e o mesmo tipo de comportamento (BOOCH, RUMBAUGH e JACOBSON, 2005). Dependendo do nível de detalhe (abstração) que estamos analisando no diagrama, podemos também representar graficamente uma classe apenas com seu nome ou com nome dos principais atributos e principais métodos, conforme o que queremos analisar no momento em que estamos criando o diagrama. Não precisamos, então, escrever todos os componentes da classe (figura 86).

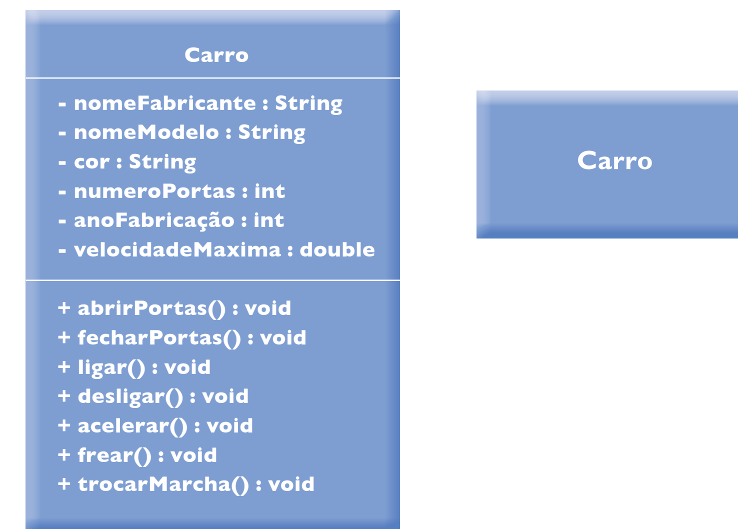


Figura 86
Outra forma de representar uma classe em UML 2.0.

4.1.2.3. Tipos de relacionamento entre classes

Existem basicamente três tipos de relacionamento entre classes: dependência, associação e herança.

1. Dependência: é um relacionamento de utilização, determinando que um objeto de uma classe use informações e serviços de um objeto de outra classe, mas não necessariamente o inverso. A dependência é representada graficamente por uma linha tracejada com uma seta indicando o sentido da dependência. Como você pode observar na figura 87, a classe Janela é dependente da classe Evento.

2. Associação: é um relacionamento estrutural que especifica objetos de uma classe conectados a objetos de outra classe. A partir de uma associação, conectando duas classes, você é capaz de navegar do objeto de uma classe até o objeto de outra classe e vice-versa (BOOCH, RUMBAUGH e JACOBSON, 2005). Representada por uma linha interligando as duas classes, uma associação pode definir papéis das classes relacionadas, assim como a multiplicidade de sua associação, além de ter um nome. Mas nenhum desses componentes é obrigatório em uma associação e só devem ser usados para deixar mais clara a sua definição.

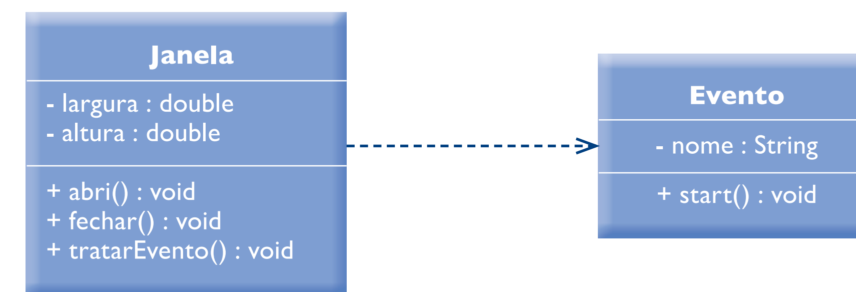


Figura 87
Representação de uma dependência em UML 2.0.

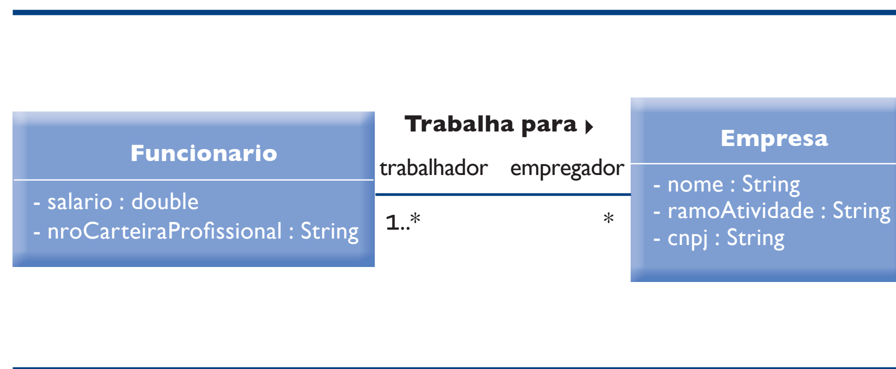


Figura 88
Exemplo de associação entre classes UML.

No diagrama da figura 88, identificamos uma associação, representada em UML 2.0 por uma linha interligando as duas classes, de nome Trabalha para, onde a classe Funcionario representa o papel de trabalhador e a classe Empresa representa o papel de empregador. Podemos observar pela representação da multiplicidade que cada objeto da classe Empresa tem, como trabalhador, 1 ou mais objetos da classe Funcionário e que um objeto da classe Funcionario tem, como empregador, no mínimo 0 e no máximo vários objetos empresa.

Já a agregação é um tipo de associação entre classes na qual é mostrada a relação todo/parte, nela uma classe fará o papel do todo e a outra, da parte. A agregação entre duas classes é representada em UML 2.0 como uma linha ligando duas classes com um losango na ponta da classe toda para diferenciar o todo da parte. Veja o exemplo na figura 89.

Observamos nesse diagrama da figura 89, que uma empresa é formada por um conjunto de departamentos, ou seja, tem a multiplicidade (leia o quadro abaixo). Vemos ainda que um departamento está associado a uma empresa.

A composição, por sua vez, é uma forma de agregação com propriedade bem definida e tempo de vida coincidente das partes pelo todo. As partes com multiplicidade não associada poderão ser criadas após a própria composição, mas, uma vez criadas, vivem e morrem com ela. Estas partes só podem ser removidas explicitamente antes da morte do elemento composto (BOOCH, RUMBAUGH e JACOBSON, 2005). Podemos dizer que numa relação de composição só faz sentido existir a parte se houver o todo. Observe o diagrama da figura 90.

Em relação à multiplicidade podemos ter:

0..* – multiplicidade de zero a muitos.
1..* – multiplicidade de 1 a muitos.
0..1 – multiplicidade de 0 a 1.
* – multiplicidade de zero a muitos; possui o mesmo significado de 0..*.
N – multiplicidade N, onde N deve ser trocado por um número que representará o número exato de objetos relacionados àquela classe.

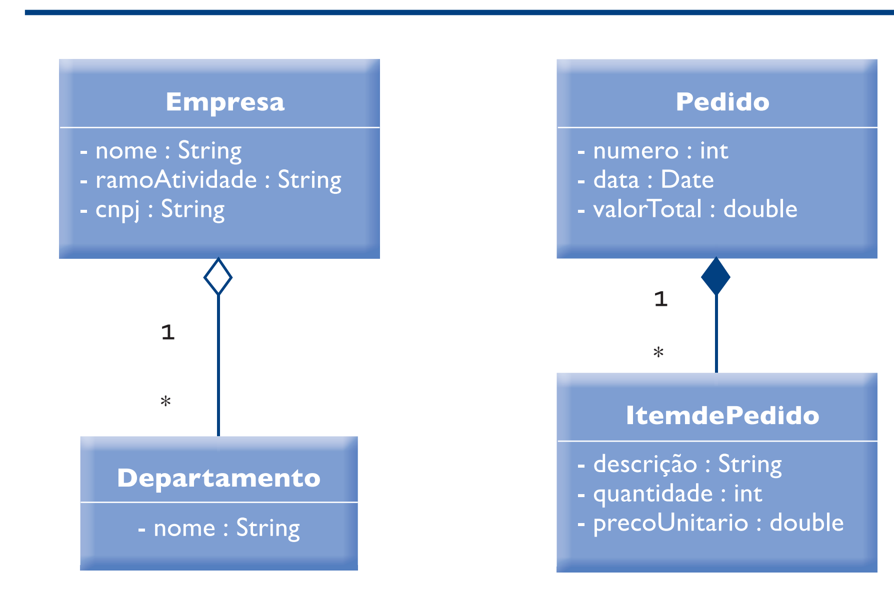


Figura 89
À esquerda representação da agregação entre classes UML 2.0.

Figura 90
Representação de uma composição UML 2.0.

Podemos analisar que só faz sentido existirem os itens de pedido (parte) se existir o pedido (todo).

3. Herança: refere-se ao mecanismo pelo qual classes mais específicas incorporam a estrutura e o comportamento de classes mais gerais (BOOCH, RUMBAUGH e JACOBSON, 2005).

Podemos verificar, na figura 91, que a classe Pessoa possui os atributos nome e cpf e pode executar os métodos de andar e falar. Já a classe Funcionario, por herdar os atributos e métodos da classe Pessoa, possui nome, cpf, salário e nro Carteira Profissional, podendo executar os métodos andar, falar e trabalhar. Dizemos que a classe Pessoa é a superclasse da classe Funcionario, que é sua subclasse, ou que Pessoa é a classe pai de Funcionario e que Funcionario é classe filha de Pessoa.

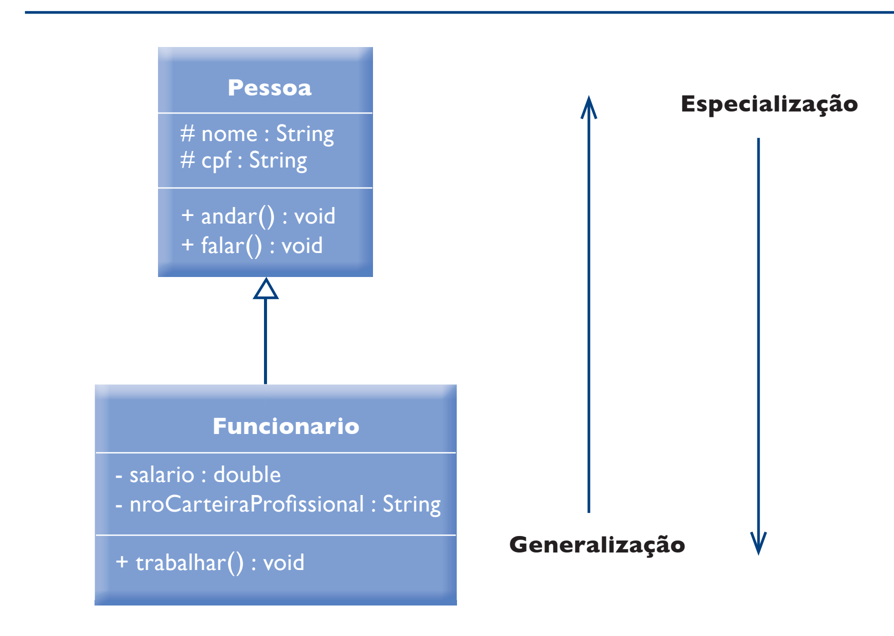
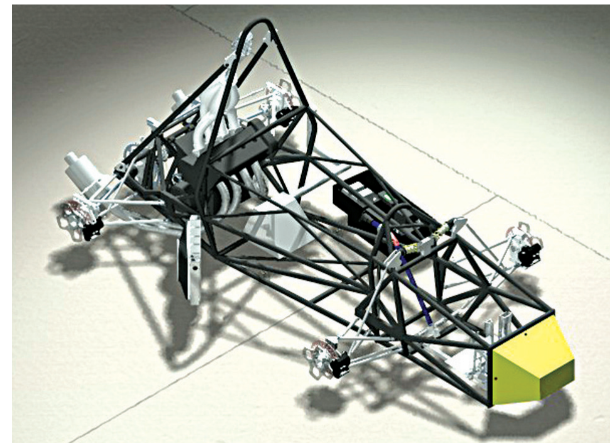


Figura 91
Representação de herança utilizando UML.

Figura 92
Classe e objeto. O “molde” que dará origem ao produto.



EQUIPE EESC-USP FÓRMULA SAE

4.1.3. Objeto

É uma manifestação concreta de uma abstração; uma entidade com uma fronteira bem definida e uma identidade que encapsula estado e comportamento; instância de uma classe (BOOCH, RUMBAUGH e JACOBSON, 2005). Podemos imaginar uma classe como sendo o molde e os objetos, os produtos criados a partir desse molde.

Um bom exemplo é pensar nos atributos do carro como sendo: modelo, número de portas, cor, ano de fabricação, tipo de combustível, velocidade máxima. Já os métodos do carro podemos definir como: andar, parar, acelerar, entre outros.

Pensando em responsabilidades, podemos dizer que o carro tem a responsabilidade de obedecer aos comandos de seu piloto, conduzindo-o na velocidade e pelo caminho que ele escolheu, acelerando e freando de acordo com o que foi sinalizado (figura 92).

4.1.3.1. Interação entre objetos

Agora que nós já definimos e diferenciamos classes e objetos, precisamos saber como os objetos interagem entre si. De acordo com o paradigma de orientação a objetos, isso ocorre por meio de trocas de mensagens. Para entendermos como essas trocas funcionam, estudemos mais alguns conceitos.

4.1.3.2. Mensagem

Segundo Booch, Rumbaugh e Jacobson (2005), mensagens são a especificação de uma comunicação entre objetos que contêm informações à espera da atividade que acontecerá, isto é, as informações trocadas entre os objetos para que executem as funções necessárias para o funcionamento do sistema. Para seguir adiante, precisamos compreender também certas características das classes. Vejamos:

- **Encapsulamento:** Propriedade de uma classe de restringir o acesso a seus atributos e métodos. A possibilidade de definir áreas públicas e privadas em sua implementação garante mais segurança ao código criado, já que podemos definir os parâmetros de entrada e saída de um método sem revelar como ele é implementado.

- **Visibilidade:** Existem quatro tipos de visibilidade de atributos ou métodos:

  Private (privada): representada por um sinal de menos (–), é a visibilidade mais restrita e permite o acesso ao atributo ou método apenas dentro da própria classe.

  Protected (protegida): representada por um sustenido (#), permite o acesso aos métodos e atributos dentro da própria classe ou de suas subclasses.

  Public (pública): representada por um sinal de mais (+), permite o acesso aos métodos e atributos a todas as classes que tiverem algum tipo de relação com a classe em que foram criados. É a menos restritiva das visibilidades.

  Package (pacote): representada por um til (~) permite o acesso aos métodos e atributos a todas as classes incluídas no mesmo pacote.

• Métodos públicos e privados

Se analisarmos a classe Pessoa, na figura 93, veremos que seus atributos são privados, mas como faremos para acessá-los? Utilizando os métodos get e set de cada atributo. É uma forma de garantir que os atributos somente serão alterados pelos métodos get e set, o que permite maior controle sobre seu uso.

Figura 93
Exemplo de visibilidade de atributos e métodos. UML 2.0.

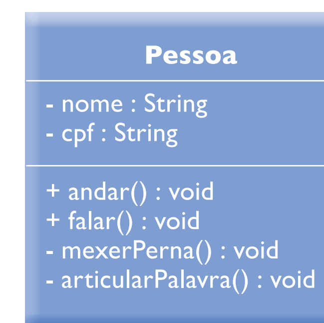
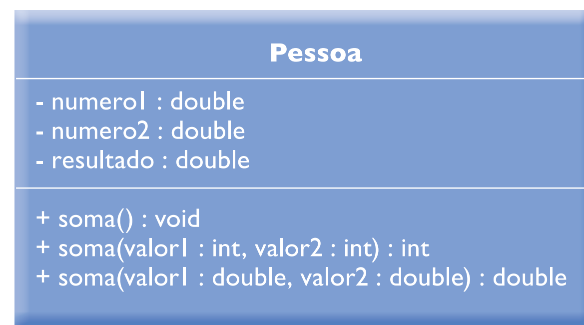


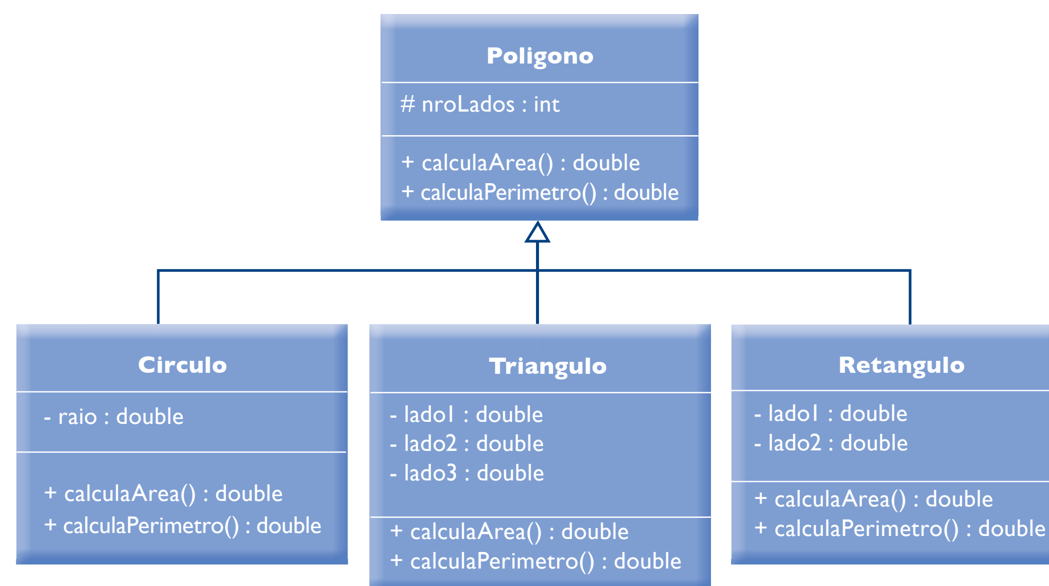
Figura 94
Exemplo de polimorfismo de método dito sobrecarga.



Veja também que os métodos andar e falar são públicos, isto é, qualquer objeto que tiver acesso à classe poderá utilizá-los. Já os métodos mexerPerna e articularPalavra são privados, isto é, só podem ser utilizados dentro da classe Pessoa. Mas qual a vantagem desse tipo de implementação? Claro que para andar, uma pessoa tem que mexer a perna. O segredo do andar está no modo como ela mexe a perna. O que fizemos, então, foi garantir que o segredo, que também chamamos de regra de negócio, não faz parte da interface pública da classe, isto é, da forma com que as outras classes vão utilizá-la. Com o método encapsulado na classe, seu código está mais protegido de eventual cópia ou reprodução de como foi implementado, pois uma classe externa nem sabe de sua existência. Logo, da forma como projetamos sua implementação nenhuma outra classe saberá que para andar é necessário mexer a perna e muito menos como isso é feito. Essa é a vantagem do encapsulamento no desenvolvimento de software orientado a objetos. O mesmo princípio foi usado ao encapsular o método articularPalavra.

• **Poliformismo**: Palavra de origem grega que significa várias formas. No paradigma de orientação a objetos polimorfismo aparece em três formas diferentes:

Figura 95
Exemplo de polimorfismo sobrescrita.



- **Sobrecarga**: na mesma classe podemos ter métodos com o mesmo nome, mas que recebem parâmetros diferentes (assinaturas diferentes) e têm, por isso mesmo, funcionalidades e/ou implementações diferentes, como mostra a figura 94.

Neste exemplo da figura 94, vemos que a classe Calculadora implementa três métodos soma diferentes: o primeiro efetua a soma com os valores dos atributos numero1 e numero2, colocando o resultado no atributo resultado. Já o segundo método soma recebe como parâmetros os números inteiros valor1 e valor2 e devolve como resultado a soma dos dois. O terceiro método recebe dois valores de tipo double valor1 e valor2 e retorna como resultado a sua soma. O interpretador de comandos escolherá, em tempo de execução, com base nos tipos e no número de parâmetros informados, qual dos métodos soma será executado.

- **Sobrescrita:** em classes associadas por uma hierarquia pode haver métodos com a mesma assinatura, mas que efetuam operações diferentes. Assim, se optará pela execução de um ou de outro, dependendo da classe que estiver instanciada no momento da execução (figura 95).

Vemos, no exemplo da figura 95, que há uma superclasse chamada Poligono, que implementa os cálculos de área e perímetro para os polígonos que não sejam círculo, triângulo e retângulo, para os quais foram criadas classes filhas, de acordo com suas fórmulas. Esse é um exemplo de sobrescrita dos métodos calculaArea() e calculaPerimetro(), que, dependendo da classe a que pertence, o objeto instanciado usará o método implementado por essa classe.

- **Polimorfismo de classe**: Uma subclasse pode, dentro da aplicação, fazer o papel da superclasse (classe pai) e da subclasse (classe filha). Sempre que uma subclasse é referenciada como a superclasse, também temos polimorfismo.

• Análise e projeto de software orientado a objetos

Como já vimos no capítulo 1, o desenvolvimento de software foi dividido em fases para facilitar o processo, o que permite reduzir as questões a serem respondidas em cada etapa, e o melhor acompanhamento do projeto.

Para desenvolver softwares orientados a objetos, seguimos as mesmas fases (análise, projeto, programação, testes, implantação e manutenção) e também podemos usar as técnicas de prototipagem. Mas é necessário definir os objetivos de cada fase, principalmente a de análise e projeto orientados a objeto, que têm características um pouco diferentes das vistas até agora.

• Análise orientada a objetos

Nessa fase, fazemos levantamento e análise de requisitos, conforme as técnicas que aprendemos no capítulo 1, e definimos o que precisamos criar para satisfazer os requisitos. Resumindo: precisamos identificar quais classes deverão ser implementadas e quais serão seus principais atributos e métodos para que os requisitos sejam satisfeitos.

Utilizamos para isso, basicamente, os diagramas de casos de uso e o diagrama

de classes, que em alguns livros são chamados de diagramas de classes de análise porque as definições das classes são ainda incompletas. Isso porque nessa fase queremos apenas definir as classes e seus principais atributos e métodos e não definir em detalhes sua implementação. Em alguns casos utiliza-se também o diagrama de objetos para se poder analisar como ficariam as estruturas das classes em determinado ponto do processamento do sistema.

Todos esses diagramas serão vistos ainda nesse capítulo, mas daremos por hora uma visão de como se desenvolvem softwares orientados a objetos.

Como podemos imaginar, o produto final dessa fase são as principais classes a serem desenvolvidas para que nosso software resolva todos os requisitos definidos.

• Projeto orientado a objetos

Agora que sabemos quais são as principais classes que comporão nossa solução de software para resolver o problema proposto, precisamos definir como os objetos criados dessas classes interagirão entre si para gerar o resultado final esperado.

Nessa fase devemos projetar todo o funcionamento do software, em detalhes. Para isso podemos utilizar os demais diagramas da UML, o que nos ajudará a definir, também em detalhes, como funcionará cada um dos itens da solução, até, por exemplo, em que estrutura de hardware e software será implementada e como estarão dispostos seus diversos componentes nos computadores da rede. Geralmente, nessa fase são criadas novas classes responsáveis pela interação do usuário com o sistema, assim como com outras classes/sistemas definidos e em funcionamento. Definimos também os procedimentos de interação entre usuário e sistema, além dos requisitos de segurança de acesso às informações utilizadas pelo sistema. A UML nos oferece ferramentas que permitem analisar em detalhes cada um dos componentes de nossa solução de software nos mais diversos aspectos de sua construção e funcionamento.

Em resumo, na fase de análise orientada a objetos devemos nos preocupar com o que o sistema deve fazer para responder a todos os requisitos, definindo suas principais classes e funcionalidades. Na fase de projeto temos de pensar em como as classes deverão se comportar para que o sistema funcione de forma a cumprir todos os seus requisitos, com o tempo de resposta definido e na estrutura de software e hardware proposta.

4.2. As várias opções da UML

Como afirmaram seus próprios criadores, a linguagem **UML** oferece opções para análise e definição em todas as fases do desenvolvimento de software – da concepção à arquitetura de hardware e software em que a solução será implementada, passando pelo detalhamento das funcionalidades, tanto estáticas quanto dinâmicas, e fornecendo apoio à definição da melhor solução para o software orientado a objetos a ser criado.

Ao utilizar a UML, precisamos de bom senso, para oferecer soluções adequadas e no prazo esperado pelo usuário, criando modelos apenas para as partes que realmente demandam definição mais aprofundada.

Deve ficar claro para nós o que é estático e o que é dinâmico, na visão da UML.

Estático é aquilo que não muda dentro do software, isto é, a estrutura, a definição das classes, a modularização, as camadas e a configuração do hardware. Já a parte dinâmica diz respeito às mudanças de estado que os itens podem sofrer no decorrer da execução do software, por exemplo, pelas alterações ocasionadas pelas trocas de mensagens entre os itens nesse momento. Utilizamos a UML para criar modelos em que os diversos aspectos relevantes ao estudo e à definição da solução de software podem ser observados, para que o programa tenha qualidade e implemente as funcionalidades necessárias, com a performace esperada pelo usuário.

Já discutimos em outros capítulos as vantagens de se criar modelos no desenvolvimento de software, e a UML nos permite criá-los para todas as fases desse processo, oferecendo ao desenvolvedor subsídios para chegar a uma solução de qualidade, com uma boa visão de cada etapa.

Os autores apontam cinco diferentes visões proporcionadas pela UML durante a construção de modelos de software. São elas:

Visão de casos de uso: permite melhor compreensão do problema a ser resolvido, ajudando na definição das fronteiras do sistema, seus principais usuários e as principais funcionalidades a serem implementadas.

Visão de projeto: auxilia na análise da estrutura e das funcionalidades esperadas da solução.

Visão de processo: também chamada de visão de interação, foca o fluxo de controle entre os diversos componentes da solução, permitindo também a análise de seu desempenho, a sincronização e a concorrência entre seus componentes, necessária para o perfeito funcionamento da solução.

Visão de implementação: ajuda a definir a estrutura da solução, isto é, os arquivos de instalação, seu controle de versões.

Visão de implantação: trata da estrutura de hardware e software, ou seja, do ambiente em que a solução será implementada (figura 96).

Figura 96
Visões de um projeto utilizando UML.

4.2.1. Conceitos da estrutura da UML

Basicamente, a UML é formada por três componentes: blocos de construção, regras que restringem como os blocos de construção podem ser associados entre si e mecanismos de uso geral, que dão mais clareza às definições criadas pelos blocos de construção. Estes, por sua vez, são de três tipos: itens, relacionamentos e diagramas.

• Itens

Os itens são a base da UML, as abstrações. Já os relacionamentos são as relações entre os itens, enquanto os diagramas agrupam itens e relações, permitindo a análise de um dos aspectos da solução a ser criada. Também há diferentes tipos de itens, que são divididos em quatro grupos: estruturais, comportamentais, de agrupamento e anotacionais. Vamos estudar, a seguir, cada um dos grupos.

• Itens estruturais

Itens estruturais nos permitem definir a estrutura da solução. São formados pelas classes, as interfaces, as colaborações, os casos de uso, os componentes, os artefatos e os nós. Para começar, vamos a uma breve definição de cada um desses elementos, que mais adiante serão aprofundados, para mostrar como funcionam e explicar, por meio de um diagrama, onde são utilizados. Empregaremos também as definições já descritas no tópico sobre orientação a objetos desse livro, que deve ser consultado, caso haja necessidade de uma revisão. Apresentaremos também a notação gráfica da UML de cada componente definido.

Classes: são os elementos básicos da orientação a objetos e, consequentemente, da UML. (A classe já foi definida no tópico que trata dos conceitos de orientação a objetos, no qual você encontra inclusive sua notação na UML.)

Interfaces: são as funcionalidades a serem implementadas por uma classe ou por um componente. Demonstram o comportamento visível de uma classe, mas nunca a implementação de tal comportamento, pois contêm apenas a assinatura dos métodos, e sua implementação é feita pelas classes que herdam seu comportamento. Servem para definir comportamentos padronizados para conjuntos de classes e itens. As interfaces são representadas por um círculo, quando se trata da interface de uma classe/item, ou por um arco, no caso da interface requerida

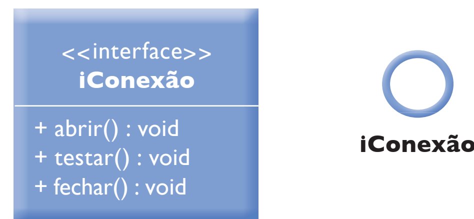


Figura 97
As duas formas de representar uma interface.

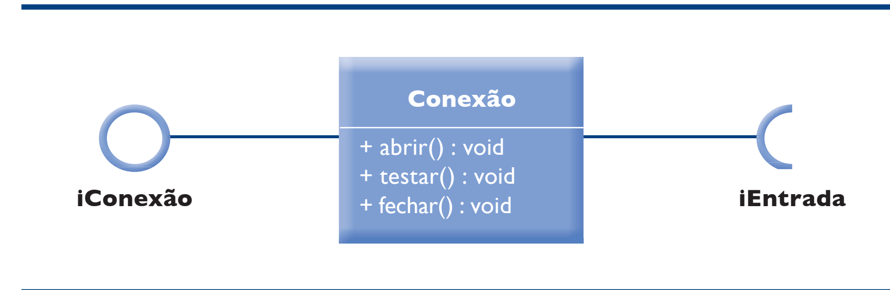


Figura 98
Classe implementando uma interface de entrada definida.

por uma classe/item. Ambas aparecem ligadas por uma linha à classe que as implementa (figura 97).

Existem, então, duas formas de representar uma interface em UML. A primeira utiliza o recurso do estereótipo <<interface>> para enfatizar que se trata de uma interface e mostra as assinaturas dos métodos que são definidos por ela. A segunda forma é a representação da interface, que não informa detalhes das funcionalidades que esta define.

Vemos, na figura 98, a representação de uma interface sendo implementada por uma classe que também tem uma definição de sua interface de entrada.

Colaboração: são agrupamentos de classes, relacionamentos e interfaces que constituem uma unidade do sistema (figura 99). Dizemos que essa unidade é maior que a soma das classes e relacionamentos implementados. São representados por uma elipse tracejada com o nome da colaboração ao centro. A colaboração serve também para nos dar uma visão em nível mais alto de abstração, quando não é necessário entrar em todos os detalhes de determinado item em análise.

Casos de uso: descrevem uma sequência de ações a serem executadas pelos componentes da solução. São ativados por um ator, servem de base para definir os comportamentos dos elementos da solução de software e são realizados por uma colaboração. São representados por uma elipse com o nome da operação que implementa no centro (figura 100).

Componentes: são estruturas que instituem uma funcionalidade de uma solução de software por meio da implementação de uma ou mais interfaces definidas. Podem ser substituídos dentro de uma solução por outro componente que implemente todas as suas interfaces, sem prejuízo ao sistema como um

Figura 99
Exemplo (à esquerda) de colaboração.

Figura 100
Exemplo de caso de uso.

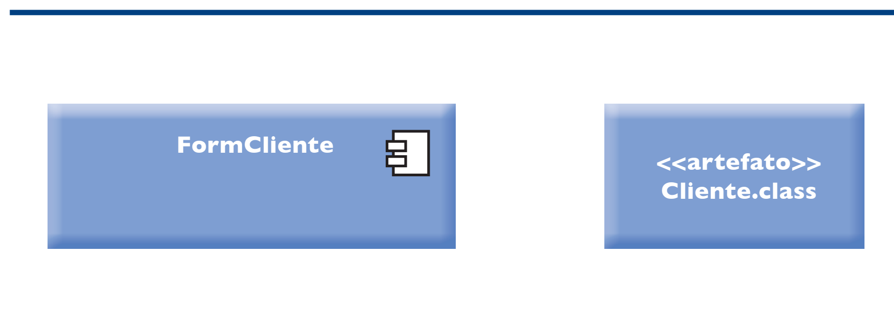


Figura 101 Exemplo de componente.

Figura 102 Exemplo de artefato.

todo. São representados por um retângulo com o símbolo da UML para componentes (figura 101).

Artefato: é um elemento físico do sistema, que pode ser um programa (fonte ou executável), um script do sistema operacional e tudo o mais que pode ser transformado em bits e bytes. É representado por um retângulo com o estereótipo <<artefato>> e o seu nome (figura 102).

Nó: representa um recurso de computação. Pode ser um servidor, um computador cliente, um switch, um hub etc. Qualquer elemento computacional que faça parte da arquitetura na qual será implementada a solução pode ser definido como um nó. Em UML é desenhado como um cubo com seu nome dentro (figura 103).

Figura 103 Exemplo de nó.

• Itens comportamentais

São os itens que definem as partes dinâmicas dos modelos UML. São também chamados de verbos do modelo. Constituem itens: interações, máquina de estados e atividades.

Interações: são os conjuntos de troca de mensagens entre objetos, também chamados de comportamento. Em UML as mensagens são representadas por uma seta traçada sob seu nome (figura 104).

Máquina de estados: especifica os diversos estados pelo qual pode passar um objeto ou uma interação em seu ciclo de vida. Sua definição inclui outros componentes como estados, transições, eventos e atividades. Em UML é representada por um retângulo com os vértices arredondados (figura 105).

Figura 104 Exemplo de mensagem.

Figura 105 Exemplo de máquina de estados.

Atividade: é um comportamento que especifica a sequência de etapas realizadas por um processo computacional. É representada em UML por um retângulo de vértices arredondados (figura 106).

Figura 106 Exemplo de atividade.

• Itens de agrupamento

Servem para agrupar os demais itens da UML, ordenando-os em blocos de modo a possibilitar melhor organização do projeto. É composto apenas pelo item pacote.

Pacote: permite a inclusão de itens em seu interior para organizar o projeto, tornando-o modular e mais organizado. É conceitual, não existindo em tempo de execução. É representado por uma pasta, que pode receber apenas seu nome ou a visualização dos itens que a compõem (figura 107).

• Item anotacional

É o componente que permite a inserção de comentários nos modelos, tornando-os mais claros e inteligíveis. É composto apenas pelo item nota.

Nota: tem como objetivo inserir comentários em um modelo para deixá-lo mais compeensível. É representado por um retângulo com a ponta superior direita dobrada para dentro. Em seu interior são inseridos os comentários pertinentes ao que se quer explicar melhor dentro do modelo (figura 108). Também pode ser utilizada uma linha tracejada para apontar exatamente a que ponto do modelo se destina a explicação da nota.

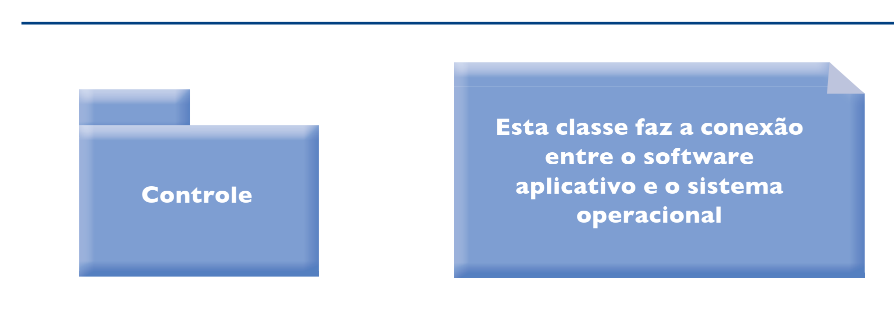


Figura 107 Exemplo de pacote.

Figura 108 Exemplo de nota.

4.2.2. Relacionamentos

Definidos os principais itens da UML, trataremos agora dos relacionamentos, que são outros blocos de construção que permitem a ligação entre os itens da UML definidos anteriormente. Existem três tipos de relacionamento em orientação a objetos: dependência, associação com seus tipos especiais (agregação e composição) e generalização (que permite a implementação do conceito de herança e realização). Todos foram apresentados quando falamos dos conceitos básicos de orientação a objetos. E reforçaremos as explicações sobre seu funcionamento quando os utilizarmos em diagramas, mais adiante.

Se você tiver alguma dúvida, volte a ler os tópicos relativos aos relacionamentos na parte de conceitos de orientação a objetos desse livro.

Assim, o próximo bloco de construção que iremos definir são os diagramas.

4.2.3. Diagramas

Existem 13 diagramas na UML 2.0, os quais são divididos em quatro grupos, de acordo com o tipo de análise que os modelos gerados por sua utilização possibilitam. São esses os grupos: diagramas estruturais, diagramas comportamentais, diagramas de interação e diagramas de implementação (figura 109).

Trataremos da construção e do uso dos diagramas implementados pela UML mais à frente, quando apresentaremos uma definição detalhada de cada um, mostrando ainda seu uso e suas principais funcionalidades. Para podermos combinar os blocos de construção da UML devemos observar as cinco regras que essa linguagem propõe, de forma que os modelos gerados contenham uma definição clara e precisa para a criação de boas soluções de software. As regras – nome, escopo, visibilidade, integridade, execução – sugerem que, ao inserir um item em um diagrama, você tem de se preocupar com cinco características que devem ficar claras à medida que cada um dos itens é inserido. As regras devem ser observadas para que possam ser criados modelos que os autores da UML chamam de bem formados, isto é, consistentes e harmônicos com todos os demais modelos que se relacionam com ele. Entenda as cinco regras:

Nome: sempre devemos lembrar que o nome de um item deve deixar claras sua formação, suas ações e responsabilidades. Não devemos nos esquecer também de que esse nome é único dentro de um modelo.

Escopo: todo item inserido em um modelo deve mostrar claramente quais são seus limites, o que implementa e quando pode ser utilizado.

Visibilidade: indica que é necessário também que fique claro quando um item estará disponível para ser utilizado e que ações estarão disponíveis por seu intermédio.

Integridade: também é importante levar em conta na criação de um item a definição clara de como este se relaciona e a consistência de tal relacionamento.

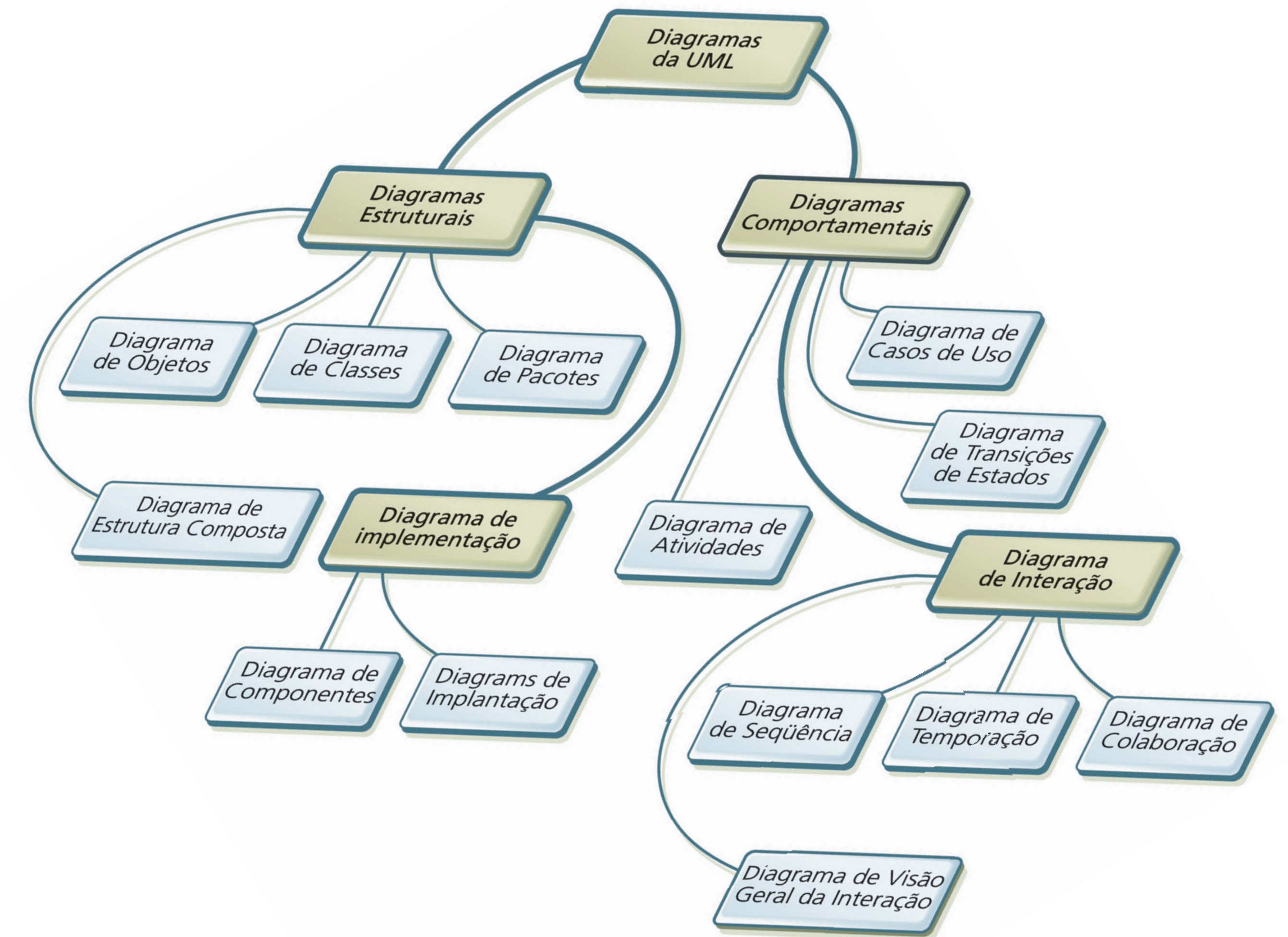


Figura 109
Diagramas definidos pela UML 2.0.

Execução: deve estar evidente ainda, o que o modelo representa e/ou simula. O que queremos observar com a criação desse modelo.

4.2.4. Adornos

Além dos três blocos de construção, a UML oferece componentes, denominados adornos, que podem ser utilizados tanto para melhorar o entendimento dos modelos criados quanto para estender o uso da UML em situações onde não existem componentes definidos. São eles:

Estereótipos: componentes de uso geral, servem para estender o significado de determinado item em um diagrama. Por serem de propósito geral, podem ser utilizados em qualquer item da UML onde for necessária uma definição mais clara de seu papel no modelo. A UML já traz uma série de estereótipos predefinidos como interface, ator, realização, mas permite que o projetista defina outros mais, sempre que surgir a necessidade. Sua representação pode ser apresentada de duas formas: uma palavra entre os símbolos de menor e maior ou um desenho do próprio estereótipo que representa. Veja o exemplo da figura 110.

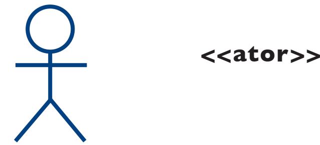


Figura 110 Exemplo de estereótipo.

Restrições: são utilizadas para definir regras em modelos, de forma a melhorar sua compreensão. As regras devem ser inseridas entre chaves ({}) e devem explicitar claramente a restrição. Por exemplo: {valor >=10}.

Podemos ainda criar novos compartimentos em itens da UML, tomando o cuidado de documentar claramente o que significa o novo compartimento.

A importância do profissional

A UML oferece diversos subsídios para a criação de modelos claros que nos auxiliem na construção de soluções de software de qualidade. Permite também a criação de modelos que simulam o comportamento do software em construção em diversos aspectos. Mas nunca se esqueça: sempre caberá ao desenvolvedor a responsabilidade de usar as informações de modo a obter soluções de qualidade, de acordo com as expectativas do usuário e capazes de produzir os melhores resultados possíveis.

4.3. Os diagramas da UML

Vamos agora a descrever os diagramas, seus principais componentes, a documentação envolvida. Também apresentaremos um exemplo, na medida do possível, recorreremos ao estudo de caso da padaria do senhor João (apresentada no capítulo 2 deste livro), que, você se lembra, usamos para mostrar como se cria e implementa um software em uma empresa. Por isso, recomendamos que você volte ao tema para rever as definições do problema e, assim, acompanhar com mais facilidade o próximo passo, a aplicação dos diagramas da UML.

- Todos os diagramas da UML são úteis para análise de aspectos importantes do desenvolvimento de software, mas não é necessário aplicar todos os diagramas em um mesmo projeto. Escolha apenas os quais o ajudarão a entender melhor o sistema que está desenvolvendo e a deixar mais claras as soluções que irá implementar. Nunca deixe que os diagramas fiquem grandes e complexos demais. Se perceber que isso está acontecendo, tente separá-los em mais de um, dividindo suas funcionalidades.

- Normalmente todos os diagramas são desenvolvidos com auxilio de um software. Há até mesmo softwares que ajudam a verificar o que podemos ou não fazer em cada diagrama. Há inúmeras ferramentas que nos auxiliam nessa tarefa, muitas com versões gratuitas. Procure a que mais se adapta às suas necessidades e mãos à obra. Sugerimos que você conheça o Jude (www.jude.change-vision.com), que tem uma versão gratuita em inglês, é fácil de utilizar e permite a construção de quase todos os diagramas da UML. Escolha a que identificar como a mais adequada a seu projeto.

4.3.1 Diagrama de casos de uso

É um diagrama que mostra um conjunto de casos de uso, atores e seus relacionamentos. Abrange a visão estática de caso de uso de um sistema, conforme descrição de BOOCH, RUMBAUGH e JACOBSON (2005), *UML – Guia do usuário*, livro de uma coleção bastante útil para quem quer se aprofundar no estudo de UML (veja quadro *UML – Guia do usuário: vale a pena consultar*).

O diagrama de casos de uso é geralmente o primeiro a que recorremos no início da análise de um projeto que utilize **UML**. Ele é criado após o levantamento dos requisitos da solução imaginada – cada caso de uso é um de seus requisitos funcionais.

Ao utilizar a UML, precisamos de bom senso, para oferecer soluções adequadas e no prazo esperado pelo usuário, criando modelos apenas para as partes que realmente demandam definição mais aprofundada.

O diagrama permite visualizar os limites do sistema, sua relação com os demais sistemas, com seus componentes internos e as funções que deve realizar.

Você pode criar diagramas de caso de uso para avaliar alguma situação não

UML – Guia do usuário, vale a pena consultar

O livro *UML – Guia do usuário* integra um conjunto de obras constituído ainda pelos títulos *The Unified Modeling Language Reference Manual Second Edition* (2005) e *The Unified Software Development Process* (1999). Escritos pelos autores da linguagem – Grady Booch, James Rumbaugh e Ivar Jacobson – e editados pela Addison-Wesley, esses três livros trazem as definições e aplicações de cada um dos elementos que compõem a UML.
Os dois primeiros, que já estão na segunda edição, lançada em 2005, abordam teoria e prática, com base na versão 2.0 da UML. Já o terceiro descreve de forma completa o processo de desenvolvimento de software utilizando a linguagem.
O livro *UML – Guia do usuário* tem versão em português e, além do embasamento teórico da UML e seu uso, descreve um processo de desenvolvimento de software por meio da linguagem, exatamente como propomos aqui. Adotamos os conceitos por acreditar que, entre as mais de 50 teorias existentes sobre desenvolvimento de software, as elaboradas pelos autores do livro são as mais interessantes.

muito clara identificada nas entrevistas ou para definir como será a relação dos diversos agentes de software no sistema, ou ainda para verificar que funcionalidades este deverá implementar. O que faremos é mapear os requisitos funcionais do sistema, sua análise e também as relações que tais requisitos terão com os demais componentes, internos ou externos ao sistema.

Todo diagrama de caso de uso deve ter um assunto, caso de uso, atores e relacionamentos.

Principais componentes: ator, caso de uso, relacionamentos.

Como esses componentes já foram definidos, reforçaremos apenas os conceitos de relacionamento. Se você tiver alguma dúvida sobre os demais itens, releia as definições formais já apresentadas e reveja os exemplos.

Um relacionamento representa os itens relacionados a um caso de uso e/ou um ator. Figura também que tipo de relação há entre dois itens. Sempre que tivermos um relacionamento entre dois casos de uso, estes devem ser obrigatoriamente um include, um extend ou uma generalização.

Vamos tratar agora dos relacionamentos include e extend.

O include é um relacionamento de dependência que, como o próprio nome diz, é de inclusão, isto é, indica que o caso de uso de onde parte o relacionamento sempre inclui/executa o comportamento do outro caso de uso, que é apontado pela seta.

O extend é um relacionamento de dependência que poderá ter o comportamento da classe apontada extendido pela classe que aponta, como você pode observar na figura 111.

Como podemos ver na figura 111, a implementação do caso de uso Validar login implica necessariamente na execução dos casos de uso Validar usuário digitado e Validar senha.

Figura 111
Exemplo da utilização de relacionamentos include/extend.

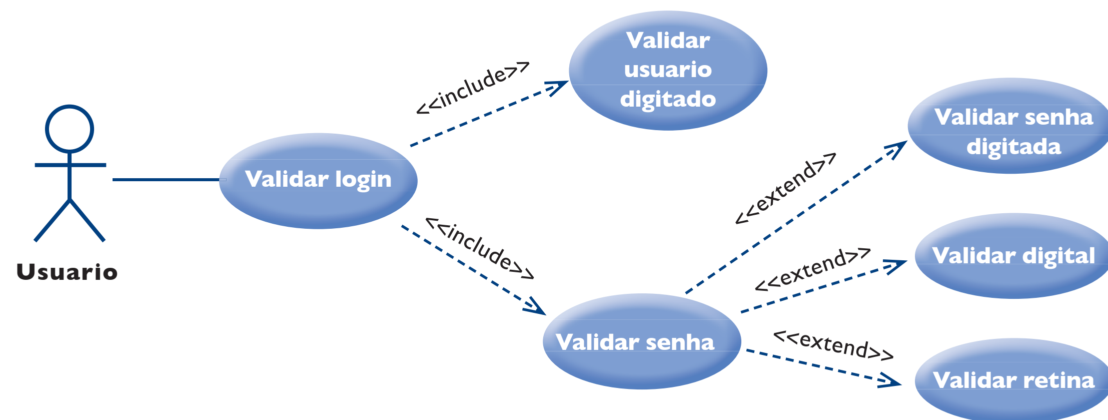
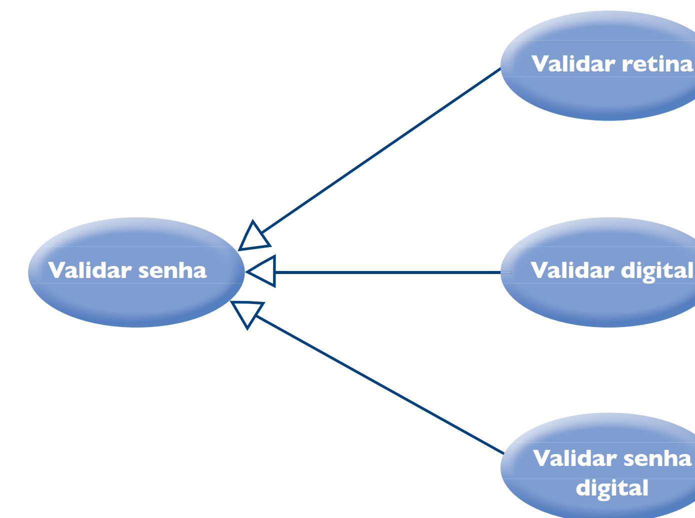


Figura 112
Relação de Generalização Especialização.



Já o caso de uso validar senha pode implicar em um dos três casos de uso. Dependendo do tipo de ação efetuada pelo usuário, será extendido o comportamento do caso de uso Validar senha digitada, validar digital ou validar retina.

Uma característica interessante que a UML nos oferece é ser extremamente flexível, possibilitando que utilizemos os blocos de construção de forma diferente, conforme a visão que queremos analisar no momento.

Voltemos ao exemplo da figura 111. O caso de uso validar senha, com o foco que descrevemos no diagrama, nos mostra uma relação extendida com os casos de uso Validar senha digitada, Validar digital ou Validar retina.

Também podemos escrever a relação entre esses casos de uso como se fosse uma relação de Generalização/Especialização. Assim, dependendo do enfoque que quisermos analisar, podemos combinar os elementos UML. Veja na figura 112 como ficaria com essa abordagem o fragmento do diagrama.

Podemos ter diagramas de caso de uso demonstrando diferentes níveis de abstração do sistema. Por exemplo, é possível ter um diagrama que represente o sistema como um todo, para poder analisar suas principais funcionalidades, como elas se agrupam, seus limites e a relação dos atores em cada um dos casos macro de uso.

Vamos retomar o estudo de caso da padaria do senhor João. Veja na figura 113 como ficaria um diagrama de caso de uso que representa as funcionalidades gerais a serem implementadas pelo sistema pedido.

Observe no diagrama que o retângulo no qual os casos de uso estão inseridos representa o sistema que estamos estudando. Seus limites são claros.

Os casos de uso representam as funções macro que devem ser implementadas pelo sistema, de acordo com as solicitações do senhor João.

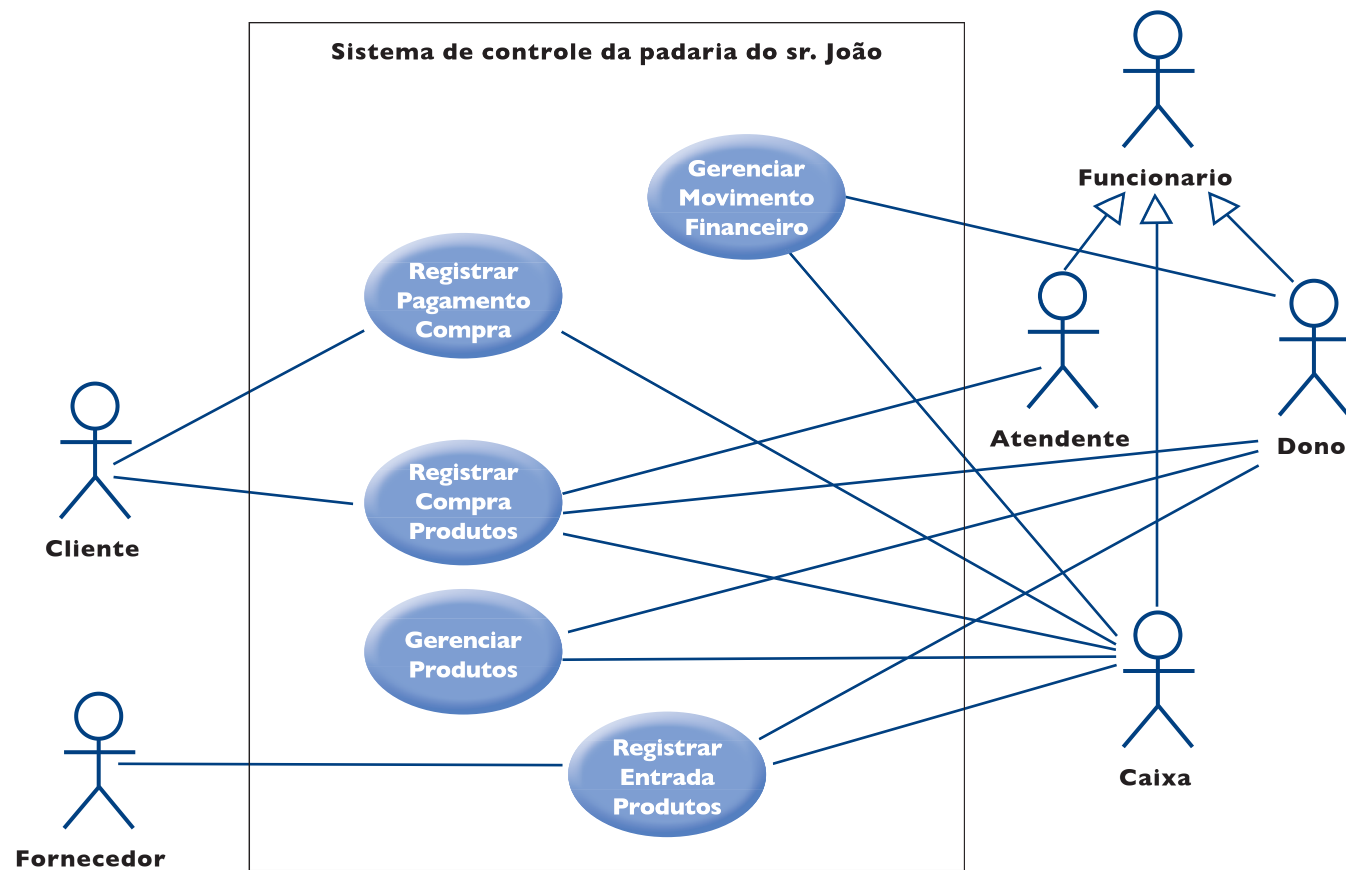


Figura 113
Diagrama de casos de uso do sistema de controle da padaria do senhor João.

Note que os atores podem ser representados isoladamente, como é o caso dos atores Cliente e Fornecedor, ou já indicando uma relação de generalização/especialização, como no caso da relação entre Funcionario, Caixa, Atendente e Dono.

Que outras informações podemos extrair desse diagrama?

Podemos ver que tanto o Caixa quanto o Atendente e o Dono (senhor João) podem registrar um produto vendido, mas apenas o Caixa e o Dono estão aptos a registrar entrada de produtos do Fornecedor.

Veja quantas informações importantes esse diagrama nos oferece. Mas ficou claro para você como cada um desses casos de uso devem ser implementados? Provavelmente não, porque ainda estamos com uma visão macro do problema e precisamos "descer" para outro nível de abstração.

É nesse momento que devemos ter bom senso para perceber até onde devemos ir na construção de níveis mais baixos de abstração, de modo que não empreguemos tempo demais criando diagramas desnecessários para situações que já estão bastante claras.

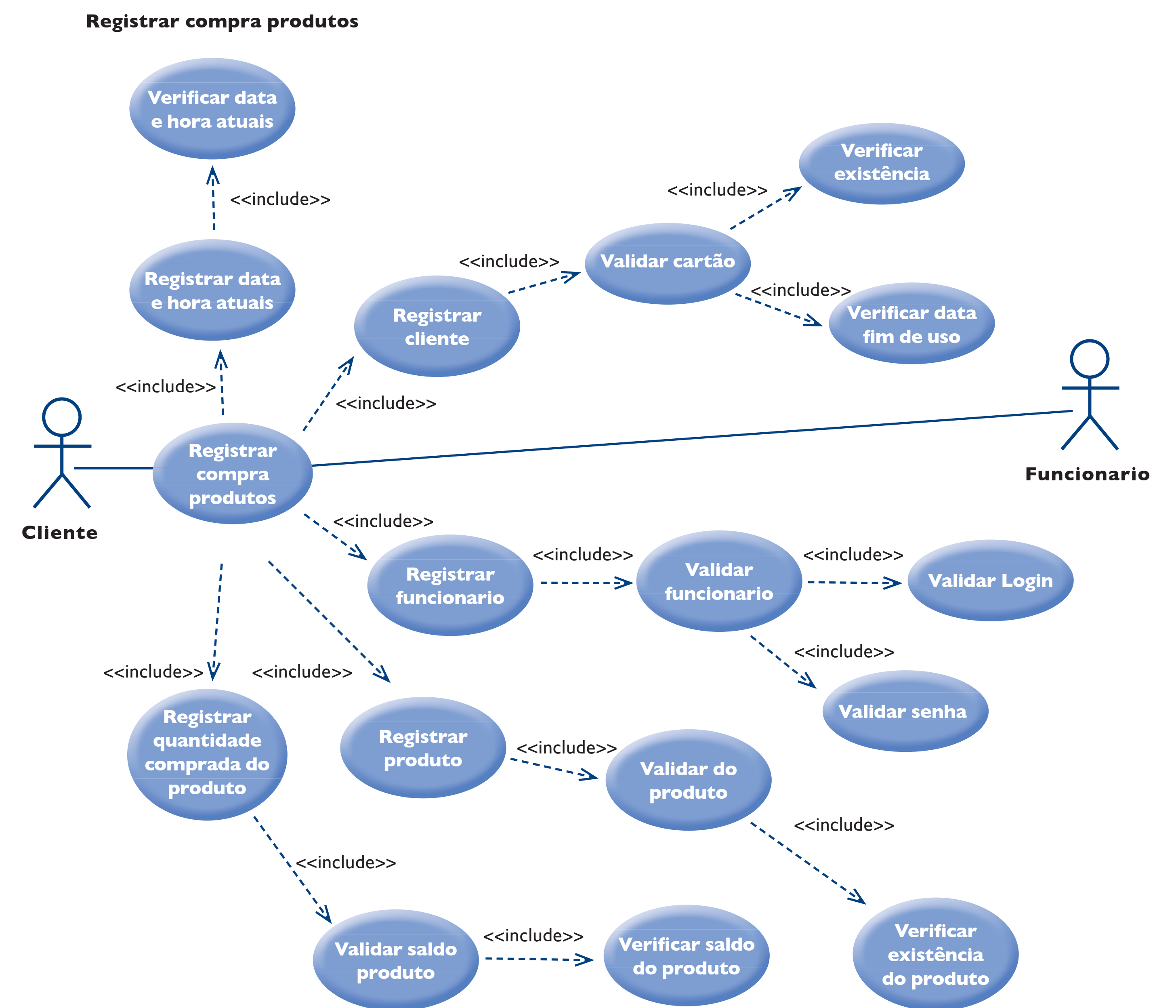
O próximo passo será criar um diagrama para cada caso de uso listado, mostrando mais detalhadamente seu funcionamento.

Vamos analisar agora o caso de uso Registrar compra produtos (figura 114).

Nesse diagrama foi feito um detalhamento do caso de uso Registrar compra produtos. Analisando o diagrama pode-se verificar que, para registrar uma compra, é preciso registrar data e hora atuais, registrar o cliente, por intermédio do número do cartão, registrar o funcionário, confirmando seu login e senha, registrar o produto, confirmando sua existência, e registrar a quantidade vendida, confirmando se há saldo suficiente para fazer a venda.

Observe que esse diagrama já possui muitas informações, talvez até mais do

Figura 114
Caso de uso Registrar compra produtos.



que deveria. Se em algum momento você achar que seu diagrama está muito complexo, diminua nele o número de casos de uso.

Neste exemplo poderíamos nos limitar aos casos de uso Registrar data e hora atuais, Registrar cliente, Registrar produto e Registrar funcionário e, então, em um novo nível, mostrar os casos de uso que implementam cada um deles. Mas isso não seria necessário, pois os casos de uso estão claros e o diagrama não está muito poluído, isto é, não possui excesso de componentes que possa atrapalhar e compreensão.

Como já vimos, cada um dos casos de uso devem ser acompanhados por um descritivo. Existem várias propostas sobre como deve ser este descritivo. A que apresentaremos aqui propõe divisões, nome, atores envolvidos, pré-condições, fluxo básico e extensões.

Nome: é o nome do caso de uso, geralmente iniciando por um verbo.

Atores envolvidos: listar os atores e os papéis executados por eles no atual caso de uso.

Pré-condições: descrever o que é necessário para que se inicie a execução do caso de uso.

Fluxo básico: os passos a serem seguidos para a finalização correta do caso de uso.

Extensões: outras possibilidades de execução.

Observações: qualquer informação necessária para ajudar a compreender o funcionamento do caso de uso.

Agora vamos a um exemplo do caso de uso Registrar compra produtos. Veja:

Nome: Registrar compra produtos.

Atores envolvidos:

- Funcionário: ator que fará o registro da compra no sistema.
- Cliente: o cliente o qual a compra será registrada.

Pré-condições:

- Produto deve estar cadastrado.
- Cartão deve ser válido.
- Funcionário deve ter acesso ao sistema.
- Funcionário deve saber seu login e senha.
- A data e hora do sistema estão corretos.

Fluxo básico:

- Cliente chega para o funcionário com o produto e o cartão para registrar a compra.

- Funcionário recebe o cartão e o produto a ser registrado.
- Funcionário faz o login no sistema.
- Funcionário escolhe a opção de registro de compras no sistema.
- Funcionário passa o leitor de código de barras no cartão do cliente.
- Funcionário passa o leitor de código de barras no produto.
- Funcionário digita a quantidade comprada do produto.
- Funcionário confere dados cadastrados digitados.
- Funcionário confirma entrada da compra.

Extensões:

- A entrada do produto e da quantidade pode ser feita pela balança quando se tratar de produtos pesados na padaria.

Observações:

Não há.

4.3.2. Diagrama de classes

Um diagrama de classes mostra um conjunto de classes, interfaces e colaborações e seus relacionamentos. **Os diagramas de classes** abrangem a visão estática de projeto de um sistema. Expõem a coleção de elementos declarativos (estáticos) (BOOCH, RUMBAUGH e JACOBSON, 2005, *UML – Guia do usuário*.)

O Diagrama de classe não deve faltar em projetos orientados a objetos. Devemos prestar muita atenção ao criar um diagrama de classes, que será a base da nossa solução.

Principais componentes: classes, interfaces, relacionamentos.

O diagrama de classes fornece uma visão estática do modelo a ser criado. Como as classes são um dos componentes mais importantes da orientação a objetos, esse diagrama deve constar de todo projeto orientado a objetos.

Identificar uma classe não é tarefa das mais simples, mas o caminho é procurar itens que têm as mesmas informações e comportamentos. Nem sempre uma classe tem atributos e métodos. Pode ter apenas métodos ou apenas atributos.

Tente fazer uma lista do que você identificou como classes. Acrescente os atores, que geralmente são também classes de seu sistema.

Analise as candidatas a classes e tente achar atributos para elas e, se possível, alguma funcionalidade.

Coloque as classes em um diagrama e comece a analisar as relações entre elas, de acordo com os tipos de relacionamentos que você estudou.

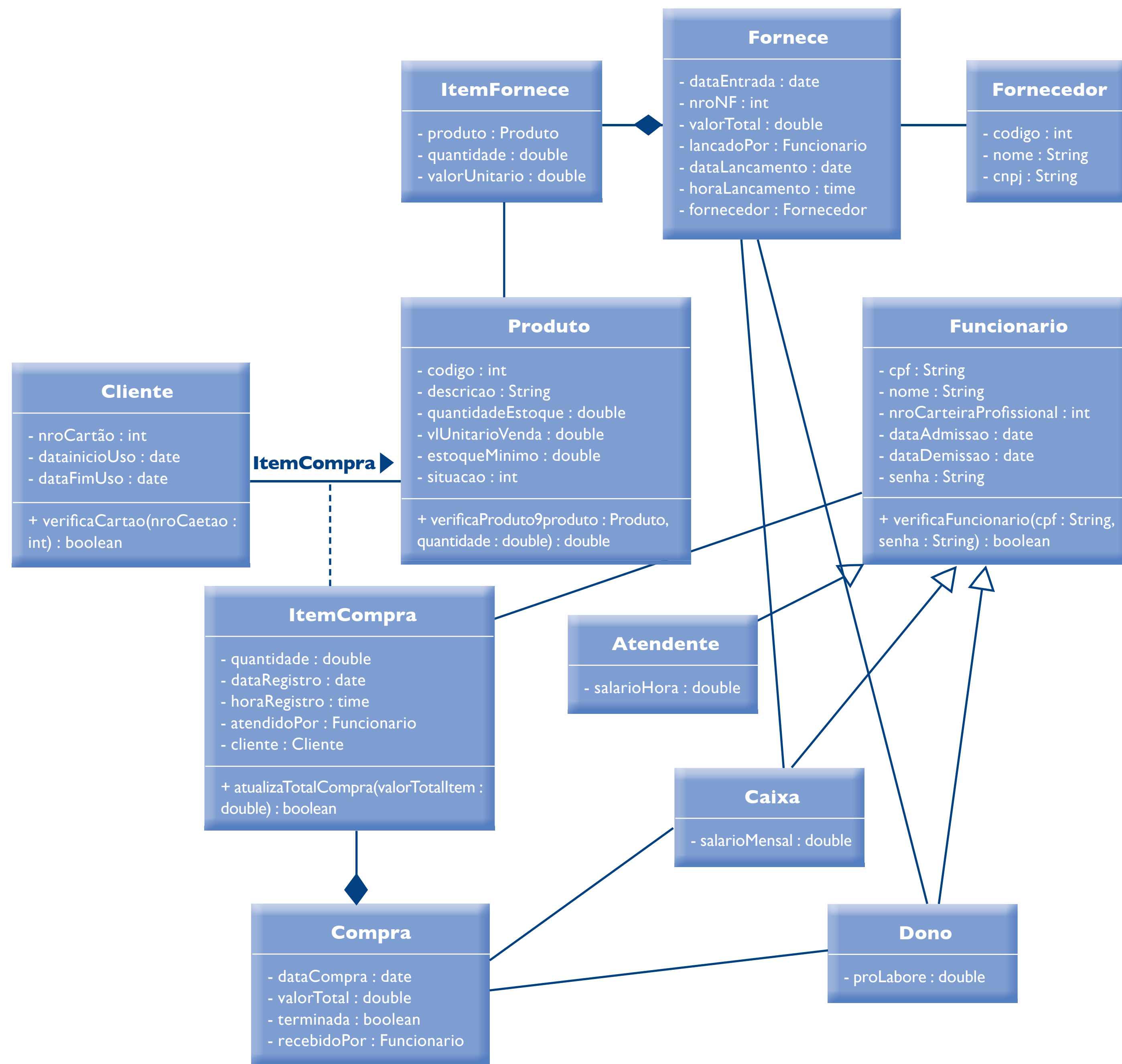
Lembre-se de que todos esses componentes foram definidos anteriormente. Se tiver dúvidas, volte a ler as definições sobre o tema.

Vamos agora analisar uma parte do diagrama de classes da padaria do senhor João. Essa parte foi montada com ênfase nos casos de uso Registrar compra e pagar compra. Por isso, serão mostrados somente os métodos envolvidos na

execução desses casos de uso (veja figura 115). As classes que não participam deles são apresentadas apenas como seus atributos.

Podemos identificar vários elementos da teoria de orientação a objetos nessa parte do diagrama. Vemos aí exemplos de generalização/especialização entre as classes Funcionario, Dono, Atendente e Caixa. E também de composição entre Compra/ItemCompra e Fornece/ItemFornece, além de uma classe de associação ItemCompra.

Figura 115
Diagrama de classes.



Podemos incluir a multiplicidade nos relacionamentos, se quisermos analisar esse requisito, para, por exemplo, projetarmos o banco de dados relativo a solução.

O diagrama de classes oferece inúmeras visões de nosso projeto, que vão desde a visão da relação entre as classes até a das abstrações utilizadas. E pode, até mesmo, ajudar na criação do banco de dados vinculado à solução.

Dependendo do foco da análise, podemos exibir os detalhes desse diagrama de forma diferente – os estereótipos das classes, seus atributos, métodos, responsabilidades ou apenas uma dessas características.

4.3.3. Diagrama de sequência

É um diagrama de interação que dá ênfase à ordenação temporal de mensagens. (BOOCH, RUMBAUGH e JACOBSON, 2005).

O diagrama de sequência permite a análise e definição da troca de mensagens entre os objetos e atores da solução e é muito utilizado para definição de parâmetros e classes dos métodos a serem criados.

Devemos estabelecer um diagrama de sequência para cada caso de uso cujo funcionamento tenhamos dificuldade de entender ou tenhamos dúvidas a respeito de como implementá-lo.

Principais componentes: atores, classes, objetos, mensagens e focos de controle.

Foco de controle é um componente do diagrama de sequência que permite a representação de comandos de decisão, de loop e de opção. A ideia é que todos os fluxos que se encontrarem dentro do foco de controle sejam executados quando ou enquanto a condição exibida for verdadeira. Veja a figura 116.

As mensagens representam a comunicação entre os objetos e atores do diagrama. São simbolizadas graficamente por setas. Existem quatro tipos de mensa-

Figura 116
Representação de um foco de controle.

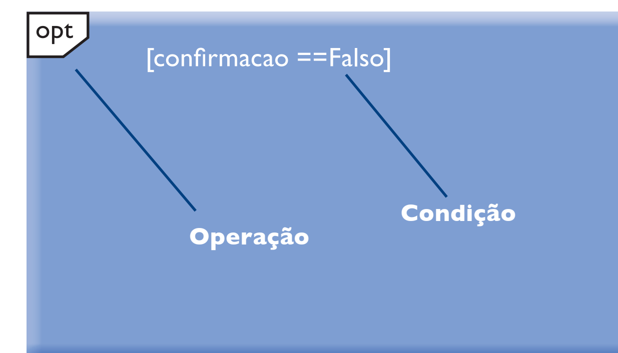
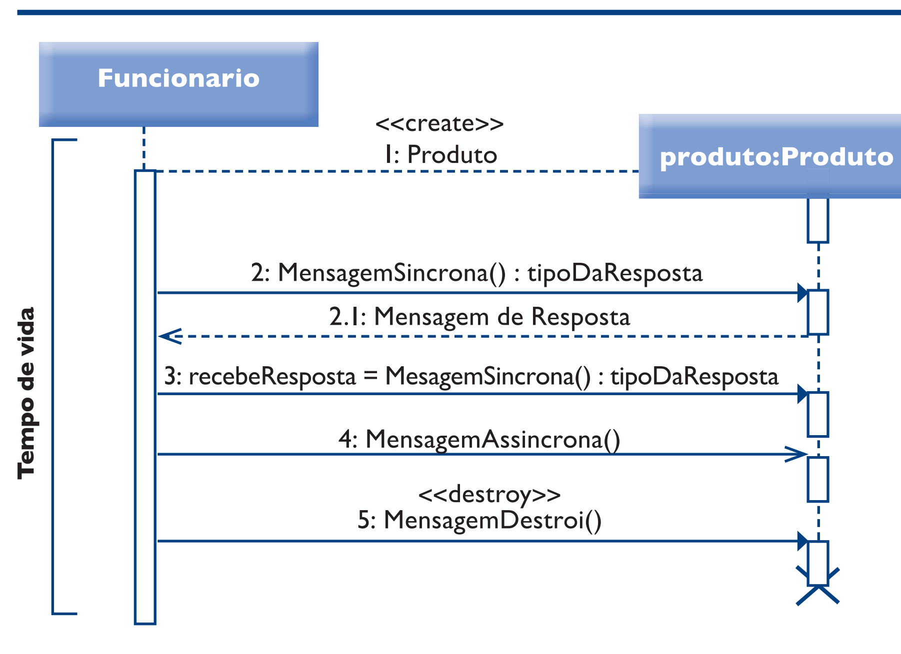


Figura 117
Troca de mensagens e tempo de vida em um diagrama de sequência.



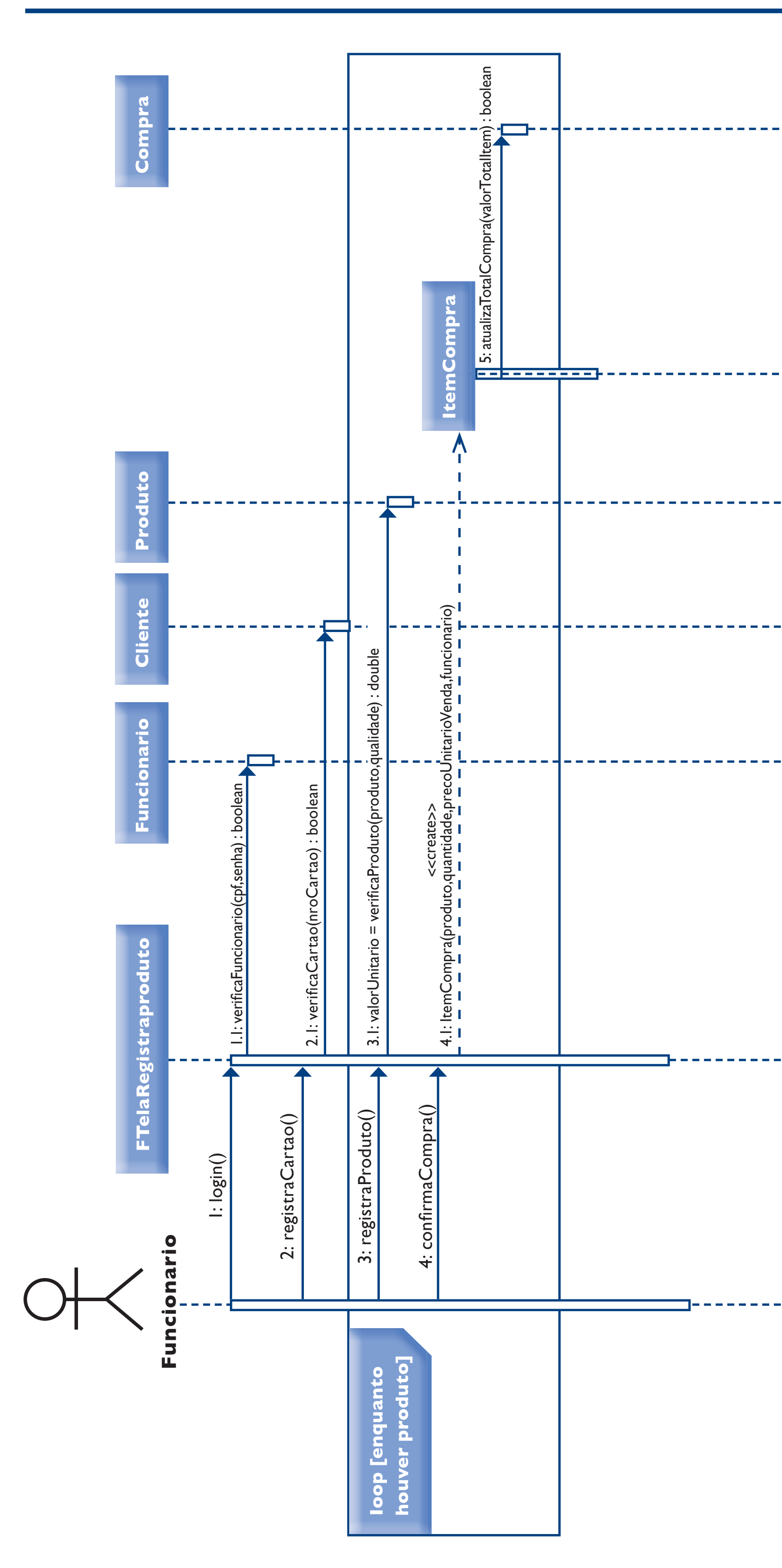
gens definidas. Mensagens síncronas são as que aguardam o retorno do fim de seu processamento para continuar a execução; mensagens assíncronas são aquelas que o remetente continua executando sem aguardar resposta; e há também as mensagens de create e destroy, que representam as chamadas dos métodos construtor e destrutor das classes.

A figura 117 mostra os quatro tipos de mensagem em um diagrama que representa a troca de mensagens entre Funcionario e Produto.

A primeira mensagem é a de criação de um objeto da classe Produto. Em seguida, vemos uma mensagem síncrona indicando que o remetente aguarda que o receptor processe a mensagem antes de continuar seu procedimento.
A mensagem número 1 é a de criação de um objeto da classe Produto, aquela que chama o método construtor da classe Produto.
A número 2 é uma mensagem síncrona, isto é, que aguarda o retorno de uma informação para continuar sua execução normal. Veja que o retorno pode ser por meio do final da execução do próprio método, da definição de um "tipoDaResposta, diferente de void ou ainda de uma mensagem de resposta, como a indicada na figura com o número 2.1.
A mensagem número 3 também é síncrona, mas nomeia o retorno pela execução do método. Isso permite melhor visualização da execução, principalmente quando se trata de retorno que define seu fluxo.
A quarta é uma mensagem assíncrona. Ou seja, é enviada, mas o emissor não aguarda o retorno da mensagem para continuar seu fluxo de execução.
A número 5 é uma mensagem que destrói o objeto criado. No exemplo, destrói o objeto produto criado e demonstra a chamada do destrutor da classe.
Podemos ver também, na figura 117, que a numeração das mensagens possibilita a compreensão da ordem em que são executadas.

Vamos agora analisar o diagrama de sequência que trata do caso de uso Registrar compra produtos (figura 118).

Figura 118
Diagrama de sequência do caso de uso Registrar compra produtos.



Analisando o diagrama, vemos que no início o funcionário faz o login no sistema e informa o número do cartão do cliente. Quando os dados do produto comprado foram digitados, as mensagens foram inseridas em um foco de controle que sugere a implementação de um loop, o qual, por sua vez, indica que aquela troca de mensagens ocorrerá enquanto houver produtos a serem lançados para o cliente.

Veja que agora sabemos exatamente quais métodos as classes envolvidas nesse caso de uso devem implementar.

Volte agora ao diagrama de classes. Observe que os métodos criados naquele diagrama saíram deste.

Note que a cada compra confirmada é criado um novo objeto itemCompra.

Neste exemplo podemos ver a definição da sequência de execução das classes como também da interface gráfica (GUI – Graphical User Interface), representada pela classe TelaRegistraProduto.

Volte agora ao exemplo do diagrama de classes e reveja os métodos definidos para as classes envolvidas no diagrama. Você perceberá que tais métodos foram definidos a partir desse diagrama de sequência.

4.3.4. Diagrama de comunicação

É um diagrama de interação que dá ênfase à organização estrutural de objetos que enviam e recebem mensagens; um diagrama que mostra as interações organizadas ao redor de instâncias e os vínculos entre elas. (BOOCH, RUMBAUGH e JACOBSON, 2005).

Principais componentes: objetos, mensagens.

O diagrama de comunicação mostra a relação entre os objetos, analisando a troca de mensagens entre eles. Mas não se preocupa necessariamente com a ordem em que elas ocorrem, e sim com quais objetos as mensagens são trocadas e quais são as mensagens.

Vamos analisar um diagrama de comunicação tomando por base o caso de uso Registrar compra produtos (figura 119).

Veja que ele traz, basicamente, as mesmas informações do diagrama de sequência, mas em ordem diferente, dando ênfase às mensagens trocadas pelos objetos, mas não necessariamente quando essas mensagens são trocadas.

Em muitos projetos, usa-se ou o diagrama de sequência ou o diagrama de comunicação, mas também pode-se criar ambos para análise de alguma situação particular.

4.3.5. Diagrama de atividades

É um diagrama que mostra o fluxo de controle e dados de uma atividade para outra; os diagramas de atividades abrangem a visão dinâmica do sistema. (BOOCH, RUMBAUGH e JACOBSON, 2005).

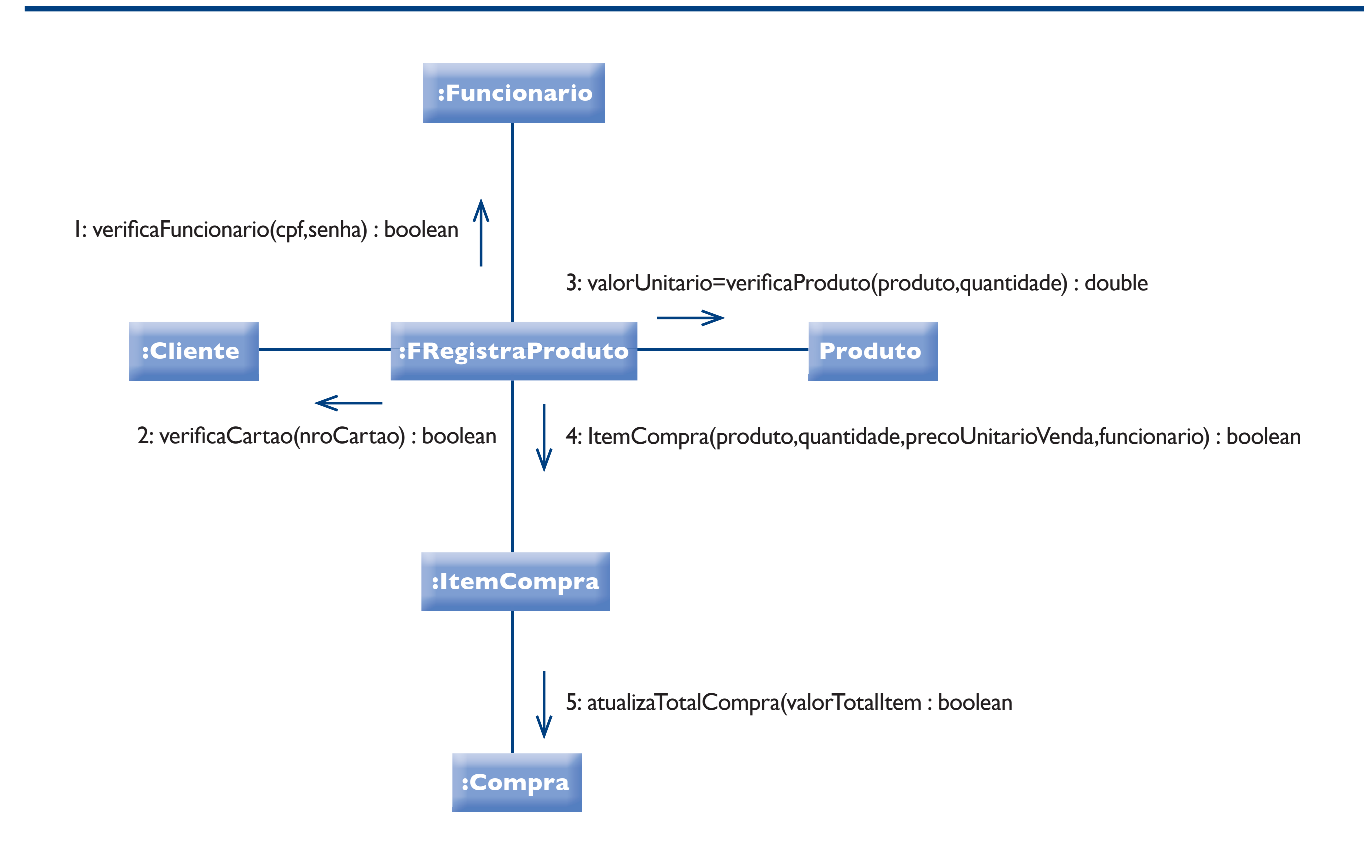


Figura 119
Diagrama de comunicação do caso de uso Registrar compra de produtos.

Por modelar aspectos dinâmicos do sistema, esse diagrama permite análise e documentação da sequência de atividades envolvidas em um caso de uso ou mesmo em um fluxo de trabalho. Possibilita a visão dos procedimentos efetuados para execução de um caso de uso, por exemplo, dando acesso a documentação dos procedimentos, tanto do sistema quanto das atividades extrassistema.

Todo diagrama de atividades deve possuir um início, marcado por um círculo preenchido, e um fim, representado por um círculo preenchido, porém com um aro branco na extremidade.

Existem também as ações que são representadas por um retângulo de bordas arredondadas, tendo em seu interior o nome da ação executada.

A execução de uma ação pode ser condicional a alguma ocorrência. Esses desvios condicionais são representados por um losango com as setas partindo para as ações a serem executadas, seja a condição satisfeita ou não.

Deve-se criar diagrama de atividades para os casos de uso cujo funcionamento não está claro ou para documentar os procedimentos a serem seguidos para sua execução.

Principais componentes: atividades, decisões, fluxos.

Veja na figura 120 o exemplo do diagrama de atividades gerado pelo caso de uso Registrar compra produtos.

Observe que o diagrama de atividades apresenta uma forma simples de documentar as ações executadas em cada caso de uso. É, asssim, uma importante ferramenta de documentação do software que está sendo produzido. Note também que você deve dividir o diagrama com linhas verticais para identificar quem deve fazer a ação.

4.3.6. Diagrama de pacotes

O diagrama de pacotes mostra a decomposição do próprio modelo em unidades organizacionais e suas dependências (BOOCH, RUMBAUGH e JACOBSON, 2005).

É um diagrama estrutural que permite uma visão da organização interna da aplicação que está sendo projetada.

Principais componentes: pacotes.

Como vimos anteriormente, dentro de um pacote podemos inserir quaisquer componentes da UML. Assim, podemos criar pacotes para estruturar nossa aplicação, usando sua modularização para organizá-la e facilitar sua compreeensão.

Veja o exemplo da figura 121.

Neste exemplo, usamos pacotes para organizar o projeto, separando as classes de projeto das de interface com o usuário (telas), também conhecidas como GUI (Graphical User Interface).

Como podemos colocar em pacotes todos os elementos da UML, devemos utilizá-los para organizar e modular nossos projetos, deixando-os mais claros e fáceis de compreender e manter.

4.3.7. Diagrama de gráficos de estados

Os diagramas de máquinas de estados abrangem a visão dinâmica de um sistema (BOOCH, RUMBAUGH e JACOBSON, 2005).

Principais componentes: estado, evento.

Esse diagrama mostra os estados que podem ser assumidos por um objeto em seu ciclo de vida. Geralmente o utilizamos para entender como tais mudanças acontecem de modo a podermos definir as trocas de mensagens e os métodos que as controlam.

O início da transição é representado por um círculo preenchido e o final, por um círculo preenchido, porém com um aro pintado de branco.

Utilizaremos como exemplo a classe Produto do estudo de caso da padaria do senhor João, que pode assumir três estados diferentes: 1 Ativo, 2 Ponto de encomenda e 3 Em falta. Esses valores são aplicados ao atributo situação quando a execução do caso de uso Registrar pagamento compra ou do caso de uso Registrar entrada de produtos (veja figura 122).

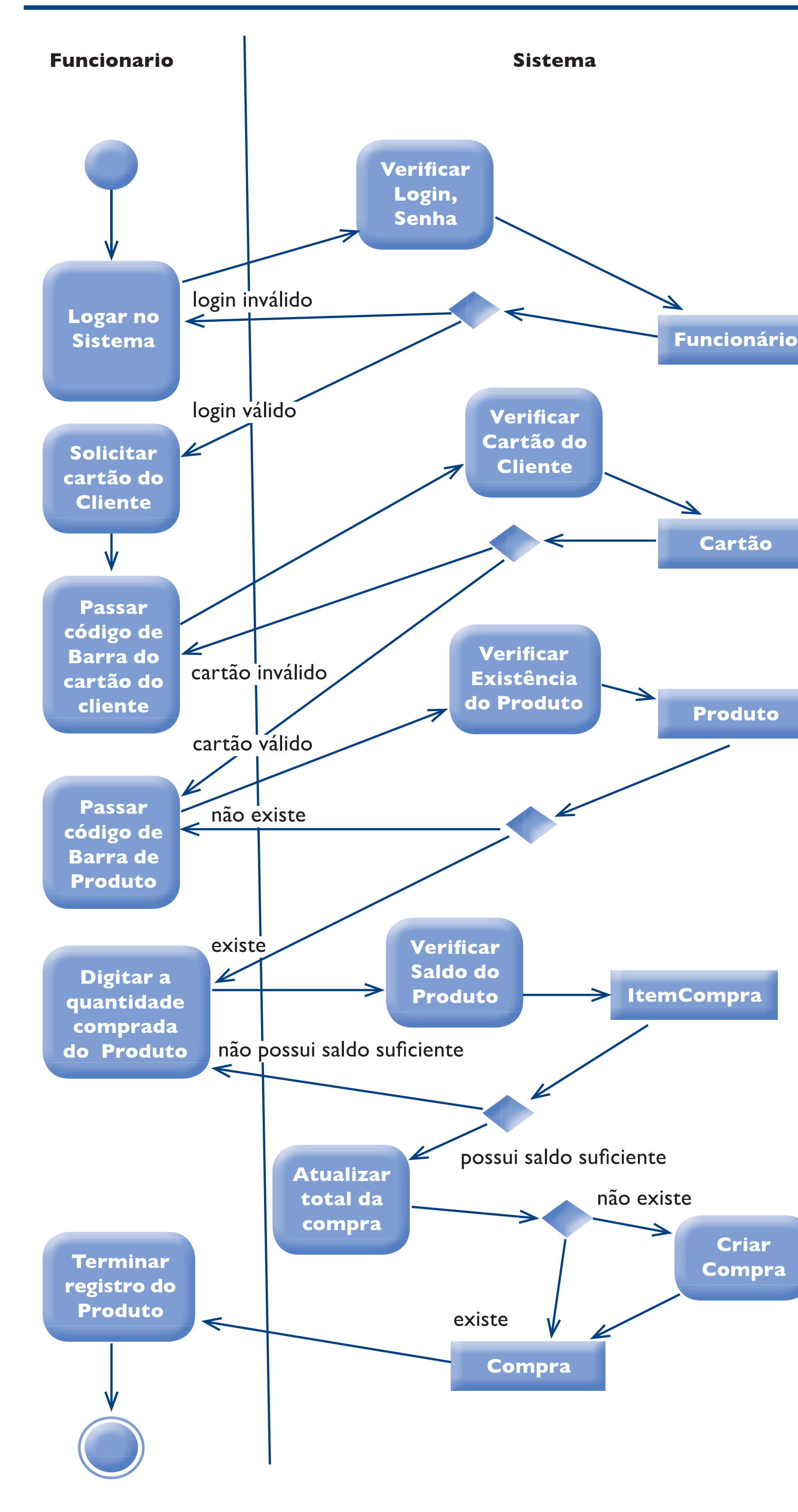


Figura 120
Diagrama de atividades do caso de uso Registrar compra produtos.

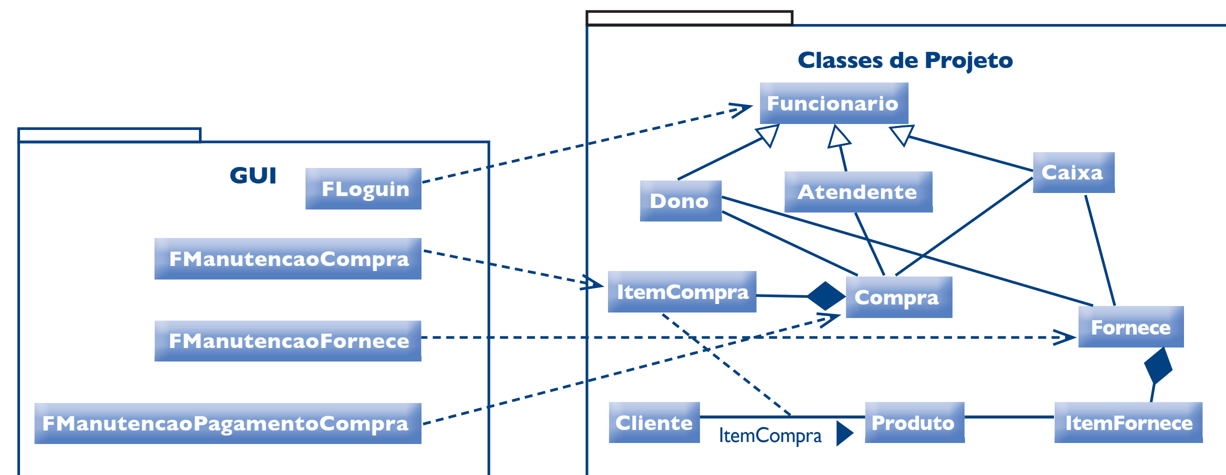


Figura 121 Diagrama de pacotes.

Analisando o diagrama podemos ver que a partir de um estado ativo, o produto poderá passar a ponto de encomenda ou em falta, dependendo das suas saídas e considerando-se que a regra para um produto chegar à condição de ponto de encomenda é seu saldo ser menor ou igual à indicada nesse campo.

Notamos também que só os métodos pagarCompra e receberProduto alteram o estado do produto e que a baixa do estoque só é efetuada quando se realiza o pagamento da compra.

Como pudemos observar, esse diagrama nos ajuda a entender e a definir melhor o funcionamento de nosso sistema quando há mudanças de estado dos objetos.

4.3.8. Diagrama de objetos

Mostra um conjunto de objetos e seus relacionamentos em um ponto do tempo; os diagramas de objetos abrangem a visão estática de projeto ou visão estática de processo de um sistema (BOOCH, RUMBAUGH e JACOBSON, 2005).

Principais componentes: objetos, relacionamentos.

Este diagrama nos dá uma visão de como ficarão os objetos em determinado momento da execução do sistema. É como se tirássemos uma fotografia do sistema em um momento para analisar os dados e os relacionamentos envolvidos, como você pode observar na figura 123.

Observe a notação desse diagrama: o objeto possui a mesma estrutura de uma classe, porém seu nome vem antes do nome da classe. func: Funcionario quer dizer objeto func da classe funcionário.

Podemos assim analisar as relações entre os objetos em um determinado ponto da execução do sistema.

4.3.9. Diagrama de componentes

Mostra a organização e as dependências existentes em um conjunto de componentes; os diagramas de componentes abrangem a visão estática de implementação de um sistema (BOOCH, RUMBAUGH e JACOBSON, 2005).

O diagrama de componentes é um diagrama estrutural que nos ajuda a analisar as partes do sistema que podem ser substituídas por outras que implementem as mesmas interfaces (de entrada e/ou de saída) sem alterar o seu funcionamento.

Principais componentes: componentes, interfaces, classes e relacionamentos.

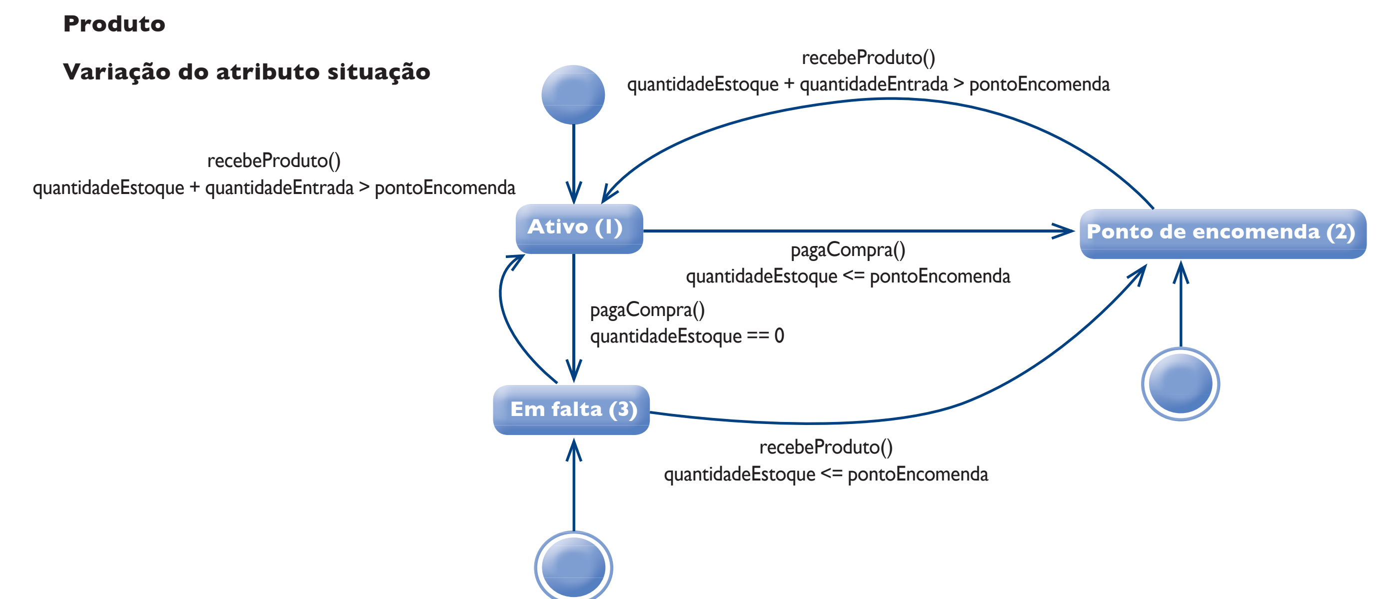
Todo componente, geralmente, pode ser substituído por uma classe, que implementa suas interfaces. Por isso é bastante difícil separar um do outro. Mas costumamos utilizar o diagrama de componentes quando precisamos documentar um componente que pode ser substituído no sistema.

Um claro exemplo de uso de componente e, consequentemente, do diagrama de componentes no estudo de caso da padaria do senhor João, é a representação da balança que pesa os produtos comercializados na padaria.

Como a balança vai interagir com os componentes do software, alimentando o sistema com informações sobre o produto vendido, como peso e valor, o senhor João poderá substituir a balança apenas por outra que possibilite as mesmas interfaces, tanto de entrada quanto de saída. Vejamos na figura 124 como representamos essa situação num diagrama de componentes.

Podemos aproveitar e definir as interfaces de entrada e saída da balança e deixá-las documentadas nesse diagrama.

Figura 122 Diagrama de gráfico de estados.



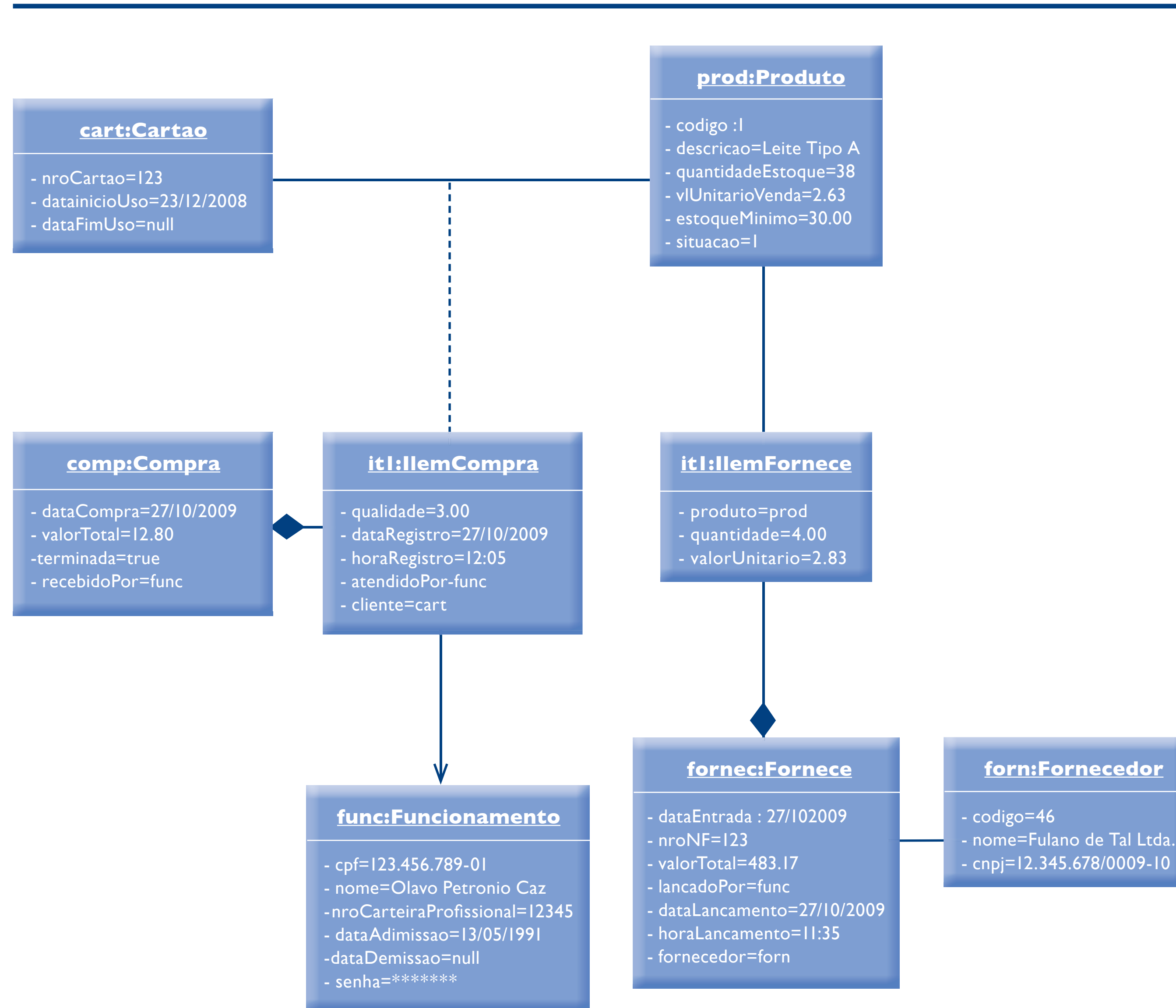


Figura 123
Diagrama de objetos.

4.3.10. Diagrama de implantação

Mostra a configuração dos nós de processamento em tempo de execução e os componentes envolvidos. Um diagrama de implantação abrange a visão estática de implantação de um sistema (BOOCH, RUMBAUGH e JACOBSON, 2005).

Trata-se de um diagrama estrutural que mostra como será criada a estrutura de software e hardware onde a solução será implementada. Podemos visualizar com esse diagrama toda a arquitetura da solução desde os servidores, sistemas operacionais, demais softwares e serviços requeridos, além dos protocolos de comunicação.

Principais componentes: nós, artefatos, relacionamentos.

Os nós podem representar dispositivos computacionais, como um computador ou um celular, ou um recurso de computação, como um sistema operacional, um sistema gerenciador de banco de dados, um servidor de aplicação ou quaisquer outros softwares que integrem a estrutura da aplicação. Possuem um nome e podem receber um estereótipo. Um nó é representado por um cubo. Os nós executam os artefatos.

Artefatos são os elementos executados pelos nós, geralmente os programas da solução criada. Possuem um nome e podem possuir um estereótipo.

Os relacionamentos são utilizados para representar o tipo de ligação entre os componentes do diagrama. Podem possuir estereótipos.

Veja que, na figura 125, demonstramos em detalhes a arquitetura da solução proposta.

4.3.11. Diagrama de temporização

É um diagrama de interação, que mostra os tempos reais em diferentes objetos e papéis, em vez de sequências de mensagens relativas (BOOCH, RUMBAUGH e JACOBSON, 2005).

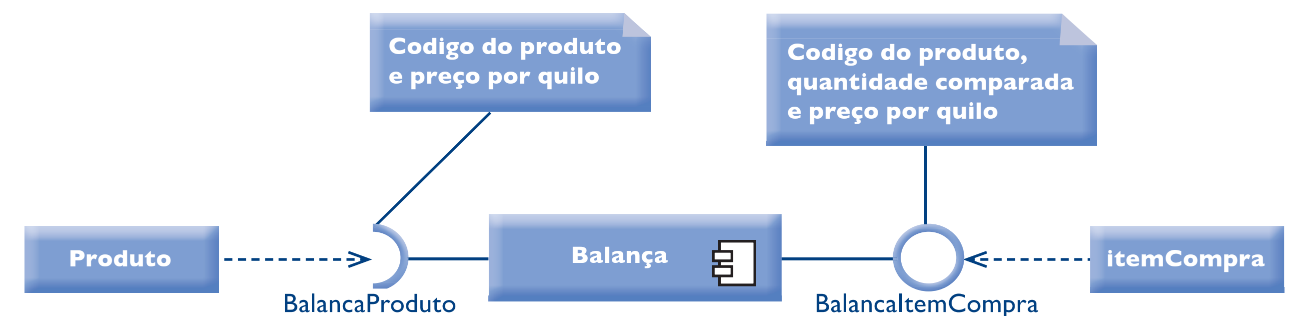
O diagrama de temporização, como o nome diz, tem seu foco principal no estudo do tempo gasto nas trocas de mensagens entre os componentes do sistema. É um diagrama de interação, pois auxilia o entendimento do processo de troca de mensagens entre os diversos componentes do sistema. Geralmente é utilizado em projetos de sistemas de tempo real, em que os tempos gastos nas trocas de mensagens e de estados na execução da tarefa são essenciais.

Esse é um diagrama de uso específico que não se utiliza em todos os projetos. Mas trata-se de um recurso que você deve conhecer caso precise analisar um sistema no qual o tempo seja um fator crucial.

Principais componentes: classes, linha de tempo, mensagens.

Criaremos um exemplo de diagrama de temporização para definir os tempos aceitáveis para as operações executadas pela balança da padaria do senhor João. A balança em questão vai interagir com o sistema, acessando os objetos da clas-

Figura 124
Diagrama de componentes.



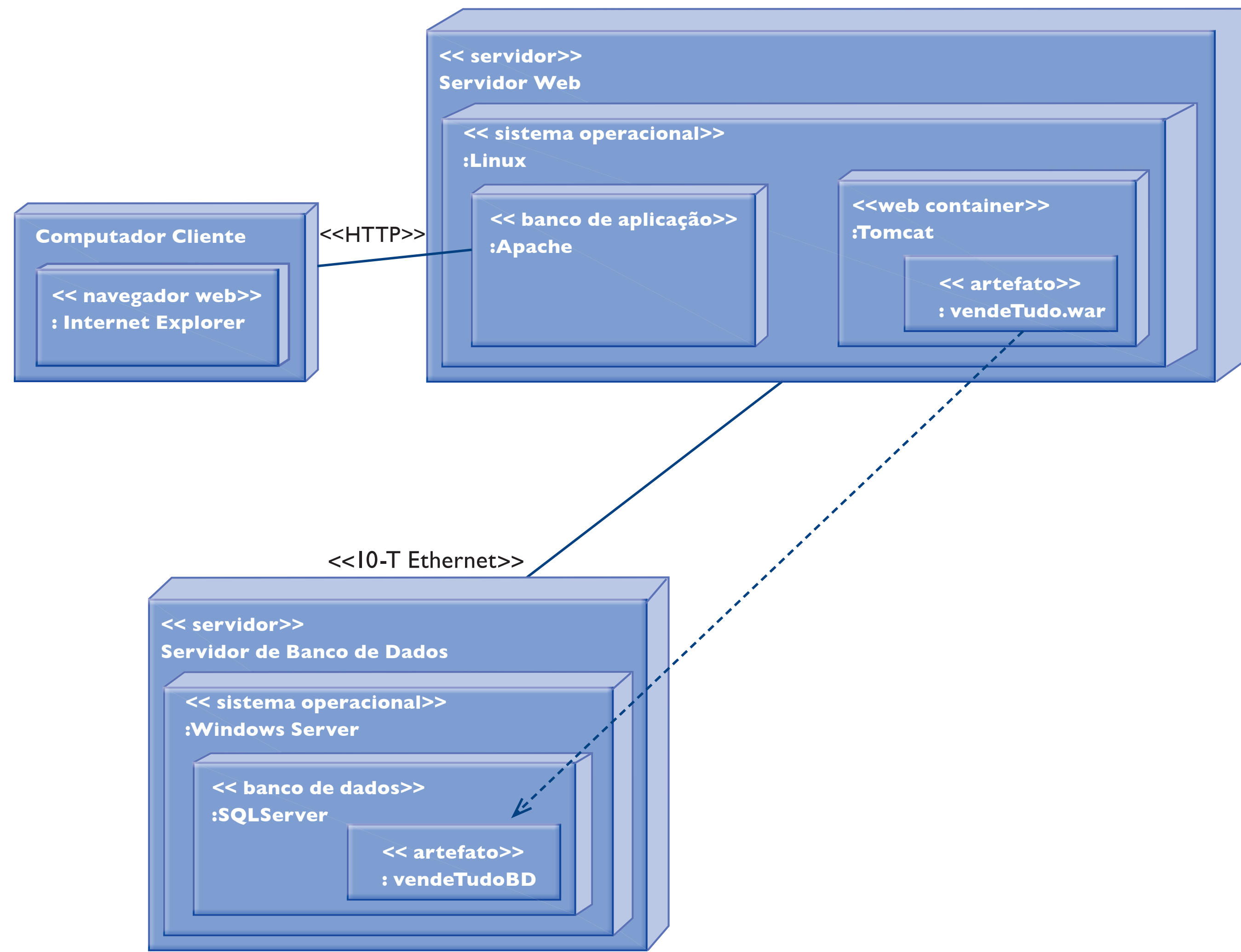


Figura 125
Exemplo de diagrama de implantação.

se produto, para pesquisar o preço por quilo. Depois da pesagem, calculará o valor do item e instanciará um objeto da classe ItemCompra. É exatamente essa situação que analisaremos no diagrama abaixo, avaliando os tempos aceitáveis para cada operação.

A balança possui um visor das informações digitadas/calculadas e um teclado numérico por meio do qual o atendente introduz os dados. Como apenas um atendente trabalha na pesagem a cada turno, ele faz o login na balança ao iniciar o trabalho e, assim, não terá de se identificar toda vez que precisar pesar algum produto.

Veja como representamos esse diagrama, na figura 126.

Podemos perceber que nosso foco na análise do tempo gasto é nas operações efetuadas pela balança, sem a intervenção do usuário.

As operações em foco são pesagem do produto, obtenção do preço por quilo,

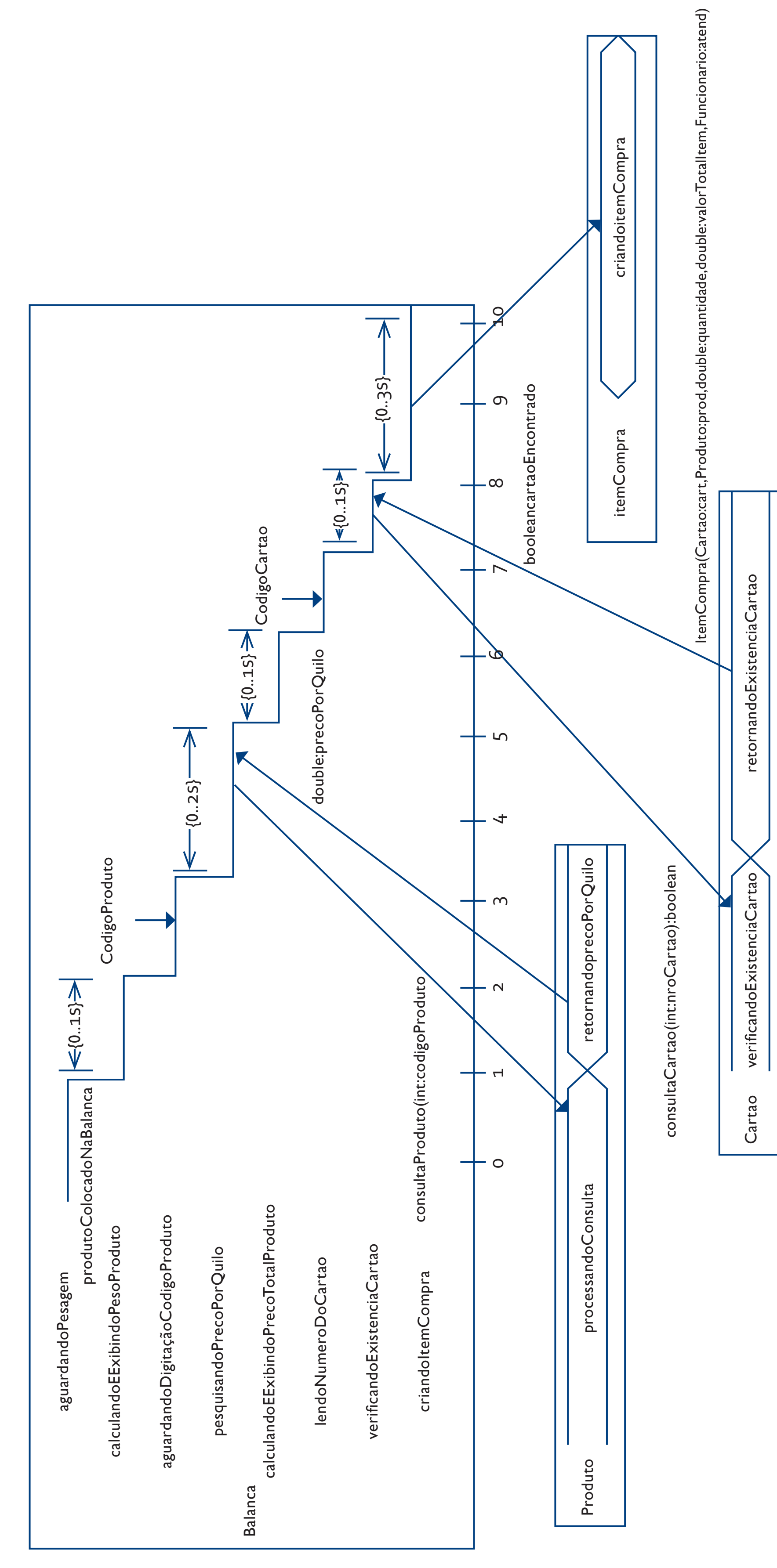


Figura 126
Diagrama de temporização.

cálculo do preço do item e geração do registro da compra (criação da instância da classe ItemCompra). São as únicas operações para as quais o diagrama demonstra um tempo máximo – as demais, que dependem da interação do atendente, têm os tempos estimados e apenas grafados na linha de tempo para podermos ter uma ideia do tempo total gasto com a operação.

Sabemos agora que requisitos de tempo básicos a balança deve realizar para que possa ser utilizada neste sistema.

4.3.12. Diagrama de estrutura composta

É utilizado para exibir as classes, interfaces e relacionamentos criados para implementar uma colaboração. Faz parte dos diagramas de interação.

Principais componentes: classes, relacionamentos e colaborações.

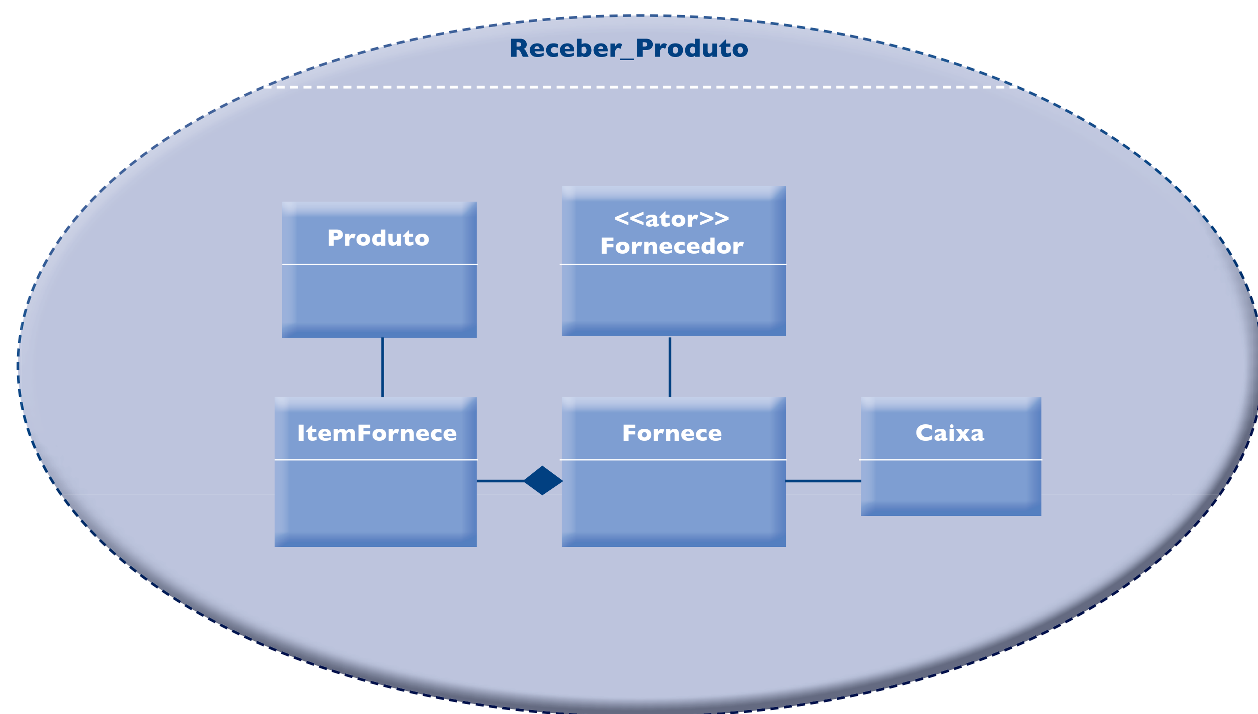
O diagrama de estrutura composta permite identificação e análise das classes e demais componentes que constituem uma colaboração.

Uma colaboração, como já vimos, é um agrupamento de classes, relacionamentos e interfaces que constituem uma unidade do sistema.

Vamos analisar o diagrama de estrutura composta (figura 127), que mostra as classes envolvidas na colaboração Receber produto.

Veja que no diagrama representado na figura 127, estamos analisando a colaboração Receber_Produto. Podemos verificar todas as classes envolvidas com a implementação dessa colaboração e as relações entre elas. Isso nos permite analisar cada colaboração em um nível mais baixo da abstração, isto é, de forma mais clara e detalhada. Podemos detalhar mais, se for necessário, incluindo ainda as classes de interação com o usuário e as mensagens.

Figura 127
Diagrama de estrutura composta.



Esse diagrama não é muito utilizado, mas pode ser útil para a compreensão da forma de implantar uma colaboração.

4.3.13. Diagrama de visão geral de interação

Tem por objetivo analisar as sequências necessárias para executar determinada funcionalidade do sistema que estamos projetando. Tal funcionalidade pode ser uma operação envolvendo vários casos de uso. Como seu nome diz, este diagrama faz parte dos diagramas de interação.

O diagrama de visão geral tem a mesma estrutura do diagrama de atividades, trocando as atividades por diagramas de sequência que mostram as classes, além de mensagens envolvidas em cada caso de uso.

Principais componentes: diagramas de sequência, decisões, fluxos.

Como utiliza os mesmos blocos de construção do diagrama de atividades, podemos verificar, passo a passo, as interações das classes em cada uma das sequências criadas no diagrama de visão geral de interação.

Vejamos um exemplo desse diagrama, modelando a sequência de atividades desde o registro até o pagamento da compra (figura 128).

Podemos observar passo a passo o registro de produtos comprados pelo cliente, tanto por meio do computador quanto da balança, analisando cada interação até o pagamento, que finaliza o processo.

4.4. Exemplo de desenvolvimento de projetos utilizando UML

Geralmente, desenvolvemos os diagramas de casos de uso para agrupar as funcionalidades mais importantes a serem implementadas. Sobre tais funcionalidades criamos o diagrama de classes, que será a estrutura de nossa aplicação ou o que chamamos de classes de projeto. No caso da padaria do senhor João, as classes de projeto são as classes de cliente, produto, funcionário, compra, fornece e suas classes filhas.

Definimos seus principais atributos e então fazemos o diagrama de sequência para os casos de uso mais relevantes – no exemplo da padaria, os casos de uso registrar compra, pagar compra, receber mercadoria. Usamos esse diagrama para definir os principais métodos de nossas classes e as trocas de mensagens entre elas. Com isso definido, voltamos ao diagrama de classes e o complementamos com os métodos definidos nos diagramas de sequência.

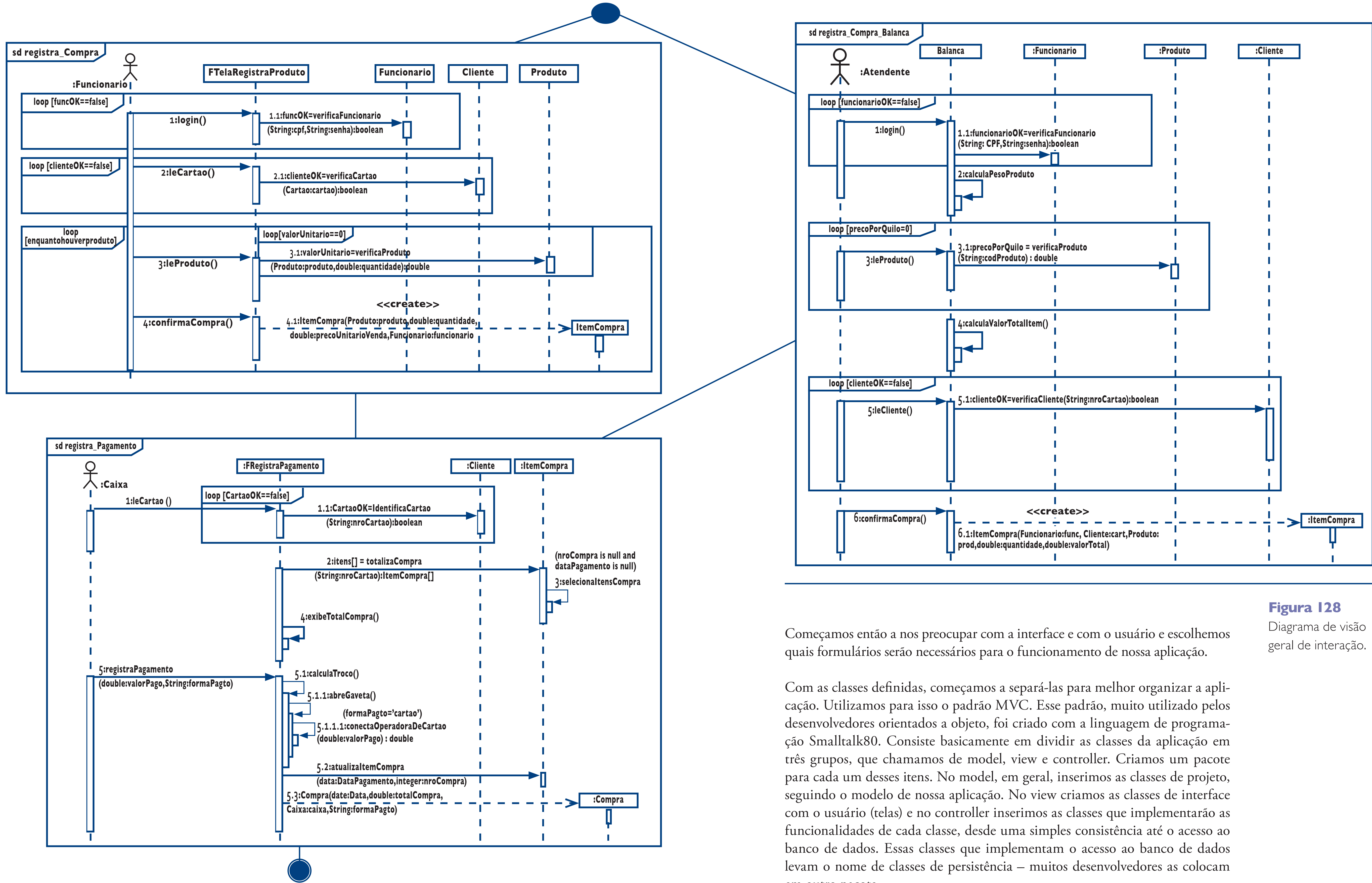


Figura 128
Diagrama de visão geral de interação.

Começamos então a nos preocupar com a interface e com o usuário e escolhemos quais formulários serão necessários para o funcionamento de nossa aplicação.

Com as classes definidas, começamos a separá-las para melhor organizar a aplicação. Utilizamos para isso o padrão MVC. Esse padrão, muito utilizado pelos desenvolvedores orientados a objeto, foi criado com a linguagem de programação Smalltalk80. Consiste basicamente em dividir as classes da aplicação em três grupos, que chamamos de model, view e controller. Criamos um pacote para cada um desses itens. No model, em geral, inserimos as classes de projeto, seguindo o modelo de nossa aplicação. No view criamos as classes de interface com o usuário (telas) e no controller inserimos as classes que implementarão as funcionalidades de cada classe, desde uma simples consistência até o acesso ao banco de dados. Essas classes que implementam o acesso ao banco de dados levam o nome de classes de persistência – muitos desenvolvedores as colocam em outro pacote.

Dentro do view voltamos a separar as classes em pacotes para implementar a modulação do sistema, criando pacotes para cadastro, movimentação, consultas, relatórios e demais rotinas da execução do sistema, às quais chamamos de ferramentas. Com isso organizamos internamente as classes nos mesmos padrões utilizados na aplicação.

DICA
Você pode pesquisar exemplos de implementação a partir de MVC em livros e sites sobre linguagens de programação. Consulte especificamente bibliografias que abordem a linguagem Java.

Terminada a separação das classes, criamos os diagramas de atividade para os casos de uso cujo procedimento e funcionamento pretendemos deixar documentados.

Verificamos então se é necessário documentar as interfaces de algum componente de software e elaboramos diagramas de componentes para isso.

Se houver classes que mudam de estado no decorrer da execução do sistema, desenvolvemos o diagrama de máquinas de estados para demonstrar qual a ideia da mudança de estado para cada uma delas.

Por fim, se o ambiente de implantação for um ambiente heterogêneo, isto é, que envolve arquitetura com vários servidores, como servidor de banco de dados e servidor de aplicações, entre outros, criamos o diagrama de implantação para demonstrar a arquitetura de software e hardware onde a aplicação deverá ser instalada.

Veja que nessa forma de desenvolvimento utilizamos os diagramas de casos de uso, os de classes, os de sequência, os de pacotes, os de atividades, o de máquina de estados, os de componentes e o de implantação.

Tudo depende do tamanho da aplicação a ser desenvolvida e das dificuldades que encontramos nas fases de análise e projeto de software.

É comum também surgirem alterações nos modelos na fase de programação do software. Nesse caso, voltamos ao modelo e incluímos as alterações que fizemos na fase de programação e testes. Mesmo depois de implantada a solução, sempre que for necessário fazer alguma alteração no sistema, devemos voltar ao modelo e fazer a inclusão, para que o modelo nunca fique diferente do **sistema criado**.

Pesquise sobre padrões de projeto (design patterns). São 24 padrões de desenvolvimento de software orientado a objetos que se propõem a resolver os problemas mais comuns nesse processo.

Considerações finais

Longe da tentativa de esgotar os assuntos aqui abordados, a intenção deste livro é ajudá-lo compreender um pouco melhor as fases de um projeto, o Modelo Relacional, o Modelo de Entidade e Relacionamento, o SGBD, o método orientação a objetos, o SQL e a UML.

Se você escolheu a área de informática para atuar profissionalmente, continue estudando e aprendendo sempre, pois nesse campo, dinâmico, as mudanças são constantes e quem não se atualiza vai ficando para trás. Esperamos que você tenha conseguido alcançar uma boa visão sobre os temas aqui abordados e que vá agora em busca de mais informações para aprofundar seus conhecimentos para avançar cada vez mais na sua carreira profissional.

Referências bibliográficas

AUGUST, Judy H. *JAD - Joint Application Design*. São Paulo: Makron Books, 1993.

BEZERRA, E. *Princípios de análise e projetos de sistema com UML*. Elsevier, 2007.

BOOCH, G., RUMBAUGH, J. e JACOBSON, I. – *UML Guia do Usuário*. 2ª edição. Elsevier, 2005.

COSTA, R. L. C. , *SQL Guia Prático*, 2ª edição. Rio de Janeiro: Brasport, 2006.

DA ROCHA, Ana Regina Cavalcanti et al. (org.). *Qualidade de software: Teoria e prática*. São Paulo: Pearson, 2004.

DALTON, Patrick. *Microsoft SQL Server Black Book*. 5ª edição. The Coriolis Group, 2008.

DATE, C. J. *Introdução a Sistemas de Banco de Dados*. 7ª edição. Rio de Janeiro: Campus, 2000.

ELMASRI, S. N.; NAVATHE, B. S. *Sistemas de Banco de Dados*: Fundamentos e Aplicações. 3ª edição. Rio de Janeiro: Livros Técnicos e Científicos, 2002.

ELMASRI, S. N.; NAVATHE, B. S. *Sistemas de Banco de Dados*. 4ª edição. São Paulo: Pearson Education, 2005.

FOWLER, Chad. *The Passionate Programmer*: Creating a Remarkable Career in Software Development, 2009.

KORTH, Henry F. e SILBERSCHATZ. *Sistemas de Bancos de Dados*, Ed. Mc.Graw-Hill, SP, 2ª edição revisada,1995.

MACHADO, F. N. R.; ABREU, M. *Projeto de Banco de Dados*, Uma Visão Prática. 2ª edição. São Paulo: Editora Érica, 1996.

OLIVEIRA, J. Wilson. *Oracle 8i & PL/SQL*. 1ª edição. Santa Catarina: Editora Visual Books, 2000.

OLIVEIRA, J. Wilson. *SQL Server 7 com Delphi*. 1ª edição. Santa Catarina: Editora Visual Books, 2001.

ORIT, Dubinsky Yael Hazzan. *Agile Software Engineering*. 1ª edição. Springer, 2008.

Project Management Body of Knowledge (PmBok). 3ª edição, 2004.

PRESSMAN, Roger S. *Engenharia de Software*. São Paulo: Pearson, 2006.

SILBERSCHATZ, Abraham; KORTH, H. F. ; SUDARSHAN, S. *Sistema de Banco de Dados*. 3ª edição. São Paulo: Makron Books, 1999.

SOMMERVILLE, Ian. *Engenharia de Software*. São Paulo: Pearson, 2004.

SWEBOK, *Software Engineering Body of Knowledge*, 2004.

LARMAN, C. *Utilizando UML e Padrões*. 3ª edição. Bookman, 2007.

WELLING, L. THOMSON L, *Tutorial MySQL*. 1ª edição. Rio de Janeiro: Editora Ciência Moderna, 2003.

YOURDON, Edward. *Declínio e Queda dos Analistas e Programadores*. São Paulo: Makron Books, 1995.

Excelência no ensino profissional

Administrador da maior rede estadual de educação profissional do país, o Centro Paula Souza tem papel de destaque entre as estratégias do Governo de São Paulo para promover o desenvolvimento econômico e a inclusão social no Estado, na medida em que capta as demandas das diferentes regiões paulistas. Suas Escolas Técnicas (Etecs) e Faculdades de Tecnologia (Fatecs) formam profissionais capacitados para atuar na gestão ou na linha de frente de operações nos diversos segmentos da economia.

Um indicador dessa competência é o índice de inserção dos profissionais no mercado de trabalho. Oito entre dez alunos formados pelas Etecs e Fatecs estão empregados um ano após concluírem o curso. Além da excelência, a instituição mantém o compromisso permanente de democratizar a educação gratuita e de qualidade. O Sistema de Pontuação Acrescida beneficia candidatos afrodescendentes e oriundos da Rede Pública. Mais de 70% dos aprovados nos processos seletivos das Etecs e Fatecs vêm do ensino público.

O Centro Paula Souza atua também na qualificação e requalificação de trabalhadores, por meio do Programa de Formação Inicial e Educação Continuada. E ainda oferece o Programa de Mestrado em Tecnologia, recomendado pela Capes e reconhecido pelo MEC, que tem como área de concentração a inovação tecnológica e o desenvolvimento sustentável.