Unidade IV

7 VISÃO ESTRUTURAL ESTÁTICA

Como vimos, a visão de caso de uso produz modelos que representam as necessidades do negócio, que são traduzidas nas funcionalidades de um sistema de *software*.

No entanto, no projeto desse sistema de *software*, necessitamos de mais do que especificar as funcionalidades, também precisamos especificar como os problemas serão resolvidos.

Se seguirmos uma boa prática de desenvolvimento, definir como os problemas serão resolvidos não se resume ao desenvolvimento de linhas de código.

Lembrete

Lembre-se que na Engenharia de Software temos modelos de processo de desenvolvimento e que cada modelo possui fases.

Em todos os processos de desenvolvimento, temos a clara divisão entre a fase de construção ou implementação e a fase de projeto, e isso não é por acaso.

É na fase de projetos, também chamada de fase de *design*, que definimos como os problemas do sistema de *software* serão resolvidos e como as funcionalidades serão implementadas.

Construir um sistema de *software* partindo diretamente para a fase de implementação é como construir uma casa sem uma planta. No exemplo da casa, teríamos as necessidades ou os requisitos de uma casa especificados e delimitados pela planta: metragem total, quantidade de quartos, quantidade de banheiros, metragem dos cômodos etc. Sem um projeto, uma planta, um modelo, a realização do projeto da casa ficará única e exclusivamente por conta do construtor.

Não há nada de tão errado nisso, pois o construtor pode ter habilidade suficiente para definir, durante a construção, a disposição das paredes, do telhado e a infraestrutura hidráulica e elétrica. No entanto, teríamos alguns riscos associados a essa situação.

Em primeiro lugar, nosso projeto teria uma grande dependência da habilidade do construtor ou de uma equipe de construtores e, caso alguma pessoa dessa equipe não tenha a senioridade que se espera, podemos ter problemas no projeto. Depender única e exclusivamente da habilidade dos integrantes da equipe quer dizer que não estamos falando de engenharia.

Engenharia é uma composição de pessoas, ferramentas e processos, que nos leva a um projeto que atinja tempo, custo e qualidade. No caso da construção da casa sem uma planta, são grandes as chances de termos paredes tortas, um sistema hidráulico ineficiente ou uma rede elétrica de baixa segurança. Ainda nesse exemplo, a correção das inconformidades da casa fatalmente acarretaria no aumento do orçamento inicial, ou seja, teríamos que gastar duas vezes: uma para construir e outra para corrigir o que foi construído. Por consequência, haveria um atraso no prazo inicial.

Podemos traçar um paralelo entre a engenharia civil, para a construção de uma casa e a engenharia de software para a elaboração de um sistema.

Construir um *software* sem um projeto é como construir uma casa sem uma planta, ou seja, estaríamos dependentes do conhecimento, da habilidade e da senioridade dos desenvolvedores. Correríamos o risco de termos requisitos funcionais e não funcionais não implementados ou implementados de maneira incorreta. Resultado: baixa qualidade, retrabalho e aumento do custo e do prazo.

Definir como as funcionalidades serão implementadas em um sistema de *software* é definir os componentes desse sistema, suas responsabilidades e como eles irão interagir entre si para resolver um determinado problema.

Quando estamos falando do paradigma da orientação a objetos, ou seja, de sistemas de *software* orientados a objetos, os componentes do sistema podem ser interpretados como objetos e a comunicação entre eles visa à execução de uma determinada tarefa.

Lembrete

O paradigma da orientação a objetos é uma forma de se desenvolver um sistema de *software*, pois este compreende que o dispositivo é um conjunto de componentes que interagem entre si para resolver um determinado problema. A cada componente, dá-se o nome de objeto.

À representação desses objetos e à relação entre eles, dá-se o nome de visão estrutural, que basicamente é a representação da estrutura dos objetos e a relação entre eles (BEZERRA, 2006).

Também chamado de aspecto estrutural estático, a visão estrutural estática representa a estrutura interna do sistema, quais são os objetos, suas responsabilidades e relacionamentos.

O termo estrutural se dá pela representação da estrutura do sistema e o estático se dá pelo fato de que nesse aspecto não temos a representação de informações desses objetos dentro de uma linha de tempo de execução (BEZERRA, 2006).

No exemplo da casa, é como se nessa visão representássemos todo o sistema elétrico, todos os componentes que o compõe e o relacionamento entre eles, mas não representaríamos uma

sequência de acendimento de lâmpadas de um morador que deseja se deslocar da sala para o quarto: quais lâmpadas seriam acesas em qual ordem e quais lâmpadas estariam acesas ou apagadas no decorrer desse processo.

Saiba mais

Para aprofundar a discussão da identificação de classes a partir de um domínio de problema, sugerimos a seguinte leitura:

WAZLAWICK, R. S. *Análise e projeto de sistemas de informação orientados a objetos*. Rio de Janeiro: Elsevier, 2011.

7.1 Conceitos fundamentais da orientação a objetos

Antes de abordarmos detalhadamente a modelagem estrutural de um sistema sob o ponto de vista do paradigma da orientação a objetos, vamos revisar alguns conceitos fundamentas da orientação a objetos.

Como vimos, um paradigma nos auxilia a trilhar um caminho, colabora para categorizar o que devemos fazer e o que não fazer para atingir um determinado objetivo (CHIAVENATO, 2008, p. 8).

Fazendo uma analogia para a orientação a objetos, o paradigma coloca um pouco de ordem no caos, definindo, dentro da orientação a objetos, o que devemos fazer e o que não devemos fazer.

Podemos dizer que o paradigma da orientação a objetos é uma forma de se desenvolver um sistema. Nesse paradigma o *software* é visto como um conjunto de componentes que interagem entre si para resolver um determinado problema, a cada componente dá-se o nome de objeto.

Observação

O paradigma da orientação a objetos não é o único paradigma aplicado ao desenvolvimento de *software*, existem outros, como o paradigma estruturado.

O principal motivador e diferenciador do paradigma orientado a objetos dos demais paradigmas de desenvolvimento de sistemas está na tentativa de aproximar o *software* do mundo real. Segundo o professor Bezzera (2006), a orientação a objetos é uma técnica para modelar sistemas que tem por objetivo a diminuição da diferença semântica entre a realidade do negócio, que está sendo modelado, e os modelos construídos para a solução dos problemas deste mesmo negócio. Devemos ter sempre em consideração que um objeto não é simplesmente um elemento do mundo real, mas, sim, que é um objeto do mundo real que possui relevância para solução de um determinado problema. Assim como no mundo real, um objeto possui características e executa determinadas ações, ou possui determinados comportamentos.

Às características de um objeto, damos o nome de atributos, e aos comportamentos, de métodos.

O paradigma da orientação a objetos é baseado nos seguintes pilares:

Classes

Podemos definir classe, ou classe de objetos, como um grupo de objetos com mesmas propriedades (atributos), comportamento (operações), relacionamentos e semântica. Uma classe deve possuir responsabilidades bem-definidas, cada responsabilidade representa um contrato ou obrigações dela.

Semanticamente, dizemos que uma classe é uma "especificação" de um objeto, pois nela está definido tudo o que o objeto possui (atributos) e tudo aquilo que o objeto pode fazer (métodos).

Objeto

Podemos afirmar que todo objeto possui uma identidade, representada por um conjunto de informações conferidas aos seus atributos, e que todo objeto possui um estado, definido também pelo conjunto dessas informações em um determinado espaço de tempo.

Encapsulamento

Encapsulamento significa deixar visível, ou deixar acessível a outros objetos, apenas o que é necessário.

Por exemplo, podemos ocultar dos demais objetos de um sistema detalhes da implementação ou a lógica algorítmica de um método de um determinado objeto, uma vez que esses detalhes não são importantes para os demais, apenas para o objeto que possui essa lógica.

Para os demais objetos que se comunicam com esse objeto, basta que ele faça o que deve fazer, não sendo importante o **como** será feito.

Abstração

Abstração é um dos principais conceitos aplicados à resolução de problemas, que é fundamental para a modelagem da estrutura de um sistema de *software*.

Abstração está ligada à nossa capacidade de selecionar determinados aspectos do problema e isolar o que é importante para algum propósito, ou seja, é dar ênfase àquilo que é necessário. Um conceito que parece simples à primeira vista, mas que faz toda a diferença na resolução e na modelagem de sistemas complexos.

Na modelagem de um sistema de *software*, abstração está relacionada à nossa capacidade, enquanto analistas, desenvolvedores e arquitetos se ocupam em estabelecer um modelo de objetos que resolva o problema da melhor forma possível, isso é, não basta apenas resolver o problema, mas, sim, pensar em

cada objeto como uma unidade autônoma, ou seja, fazendo única e exclusivamente aquilo que precisa ser feito e tendo apenas as dependências que deve ter.

A melhor forma de se representar a estrutura estática de um sistema orientado a objetos é utilizando o modelo de classes, são três os modelos utilizados (BEZERRA, 2006):

- Modelo de classe de domínio: desenvolvido na fase de análise, o modelo de classe de domínio representa os objetos, ou classes, inerentes ao domínio do problema que queremos resolver, deixando de lado, nessa visão, detalhes tecnológicos da solução do problema.

- Modelo de classe de especificação: construído na fase de desenvolvimento, o modelo de classe de especificação adiciona ao modelo de classes de domínio, objetos ou classes específicos para a solução do problema sob o aspecto tecnológico, ou seja, é uma extensão do modelo de classe de domínio.

- Modelo de classe de implementação: o modelo de implementação nada mais é que a implementação das classes, especificadas no modelo de especificação, construídas ou codificadas em alguma linguagem de desenvolvimento orientada a objetos, como a linguagem Java ou a linguagem C#.

Observação

Atualmente existe uma distorção grande quando falamos de orientação a objetos e linguagem de desenvolvimento, ou programação, orientada a objetos. Como vimos, orientação a objetos é um paradigma, não uma linguagem. Todavia, as linguagens possibilitam ao desenvolvedor implementar um *software* baseado nesse paradigma.

Nesta unidade daremos ênfase ao modelo de classes de domínio.

7.2 Modelo de classes de domínio

Desenvolvido na fase de análise, o modelo de classes de domínio representa os objetos, ou classes, inerentes ao domínio do problema que queremos resolver, deixando de lado, nessa visão, detalhes tecnológicos da solução do problema.

Iremos debater algumas técnicas de representação de classes de domínio utilizando o modelo de classes.

7.2.1 Conceitos fundamentais do modelo de domínio

Como vimos, um objeto é composto por atributos e métodos, e uma classe pode ser considerada como a especificação de um objeto.

Lembrete

Um objeto pode ser considerado como a instância de uma classe.

Normalmente identificamos uma classe pelo que ela representa dentro do domínio do negócio em que estamos trabalhando, por exemplo: a classe Cliente Pessoa Física representa todos os clientes do tipo pessoa física que podem efetuar saques em nosso terminal de autoatendimento.

No entanto, quando estamos representando em um modelo, seguimos determinadas regras de nomenclatura e, no caso da representação de classes, devemos nos atentar a algumas boas práticas, como observa Bezerra (2006):

- Procurar sempre representar uma classe por algum substantivo que tenha relevância para o domínio, por exemplo: Terminal de Autoatendimento.

- Nunca utilizar nomes de classes no plural. Por exemplo: para representar a classe de clientes de um banco, não utilizar a nomenclatura "Clientes", mas sim "Cliente".

- Para classes com nomes compostos, remover os espaços em branco, as acentuações ou caracteres especiais e iniciar sempre com letra maiúscula. Por exemplo: para representar a classe Terminal de Autoatendimento, identificar a classe como "TerminalAutoAtendimento", ou para a classe Clientes do Tipo Pessoa Física como "ClientePF" ou "ClientePessoaFisica".

Na UML, representamos uma classe, uma caixa sempre com a identificação do nome da classe na parte superior (ver a figura a seguir). A figura ainda mostra um exemplo da classe que representa o objeto cliente.

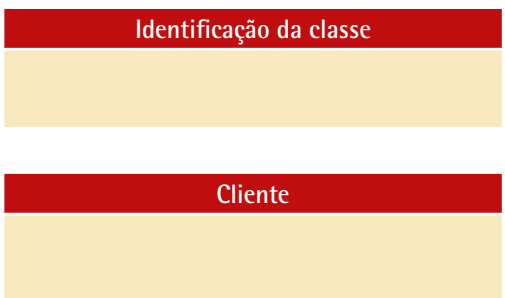

Figura 42 – Representação de uma classe na UML

Observação

Em modelagens pobres e incorretas, é comum que encontremos a representação da classe "sistema" ou da classe "*software*", sendo que esses objetos possuem todas as operações necessárias para o sistema modelado.

Essa abordagem é absolutamente incorreta, levando em consideração os conceitos básicos do paradigma orientado a objetos.

Uma classe, ou um objeto, possui atributos e, para representá-los ou representar adjetivos do objeto, também seguimos algumas boas práticas:

- Identificar o atributo com a primeira letra sempre em minúsculo e sem espaços, a exemplo da descrição de classes. Por exemplo, o atributo saldo de conta-corrente descreveríamos como "saldoContaCorrente".

- Nunca utilizar atributos no plural, como "preços".

Na UML, conforme mostra a figura a seguir, representamos os atributos de uma classe abaixo da identificação da classe, de forma sequencial e um abaixo do outro. A figura ainda mostra um exemplo da representação de uma lista de atributos da classe Cliente.

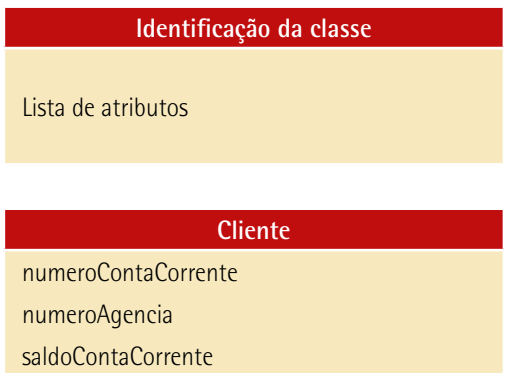

Figura 43 – Representação de atributos em uma classe UML

Assim como atributos, uma classe ou um objeto também possui métodos e, para representar esses comportamentos, também seguimos algumas boas práticas:

- Identificar um método sempre com um verbo no infinitivo seguido preferencialmente de um subjetivo que represente um atributo da classe. Por exemplo: "sacarContaCorrente", "informarSenha" ou "inserirCartao".

- Identificar o método com a primeira letra sempre em minúsculo e sem espaços, a exemplo da descrição de classes e de atributos.

Os métodos de uma classe são representados, na UML, abaixo da identificação de atributos, como podemos verificar na figura a seguir, que ainda mostra um exemplo da representação de uma lista de métodos da classe Cliente.

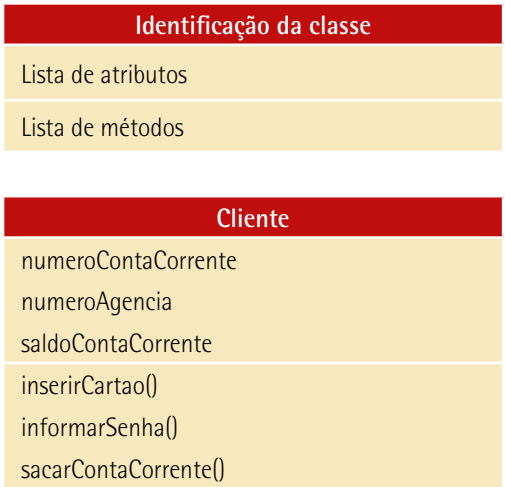

Figura 44 – Representação de métodos em uma classe UML

Um objeto é um elemento do mundo real que especificamos utilizando classes. Todavia, como vimos, um objeto possui um conjunto de informações que os caracterizam, chamados atributos. Quando damos valores a esses atributos, esse objeto passa a ter uma identidade.

Na UML, um objeto é representado da mesma forma que representamos uma classe, porém com o nome da classe sublinhado, podendo ou não constar o nome do objeto na própria identificação, como mostra a figura.

A representação de objetos na UML é utilizada no diagrama de objetos ou em modelos que representam uma visão dinâmica da estrutura do sistema, que veremos mais à frente. O objetivo neste momento é reafirmar o conceito de que um objeto é uma instância de uma classe.

Note que na figura representamos o nome do objeto, seus atributos e os respectivos valores, assim como seus métodos.

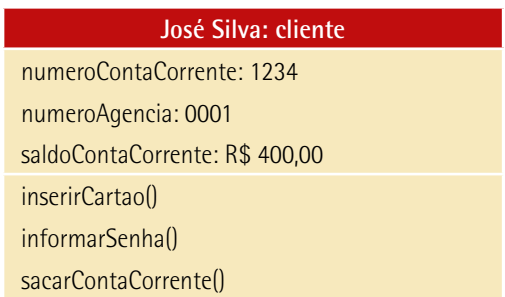

Figura 45 – Representação de um objeto UML

7.2.2 Relacionamento entre classes

Assim como no mundo real, objetos de sistema se relacionam entre si dentro de um determinado contexto para resolução de um determinado problema.

Existem cinco tipos de relacionamento entre objetos. Quatro deles, estudaremos na sequência. São eles: dependência, associação, agregação e composição. O quinto e último tipo, a herança, será debatido separadamente.

7.2.2.1 Dependência

Segundo Booch, Jacobson e Rumbaugh (2006), dependência é um relacionamento em que um objeto depende de informações de outro objeto para a execução de um determinado comportamento.

Na UML, representamos a dependência utilizando uma seta tracejada, como mostra a figura a seguir. Ainda na imagem, podemos fazer a leitura de que a Classe A depende da Classe B.

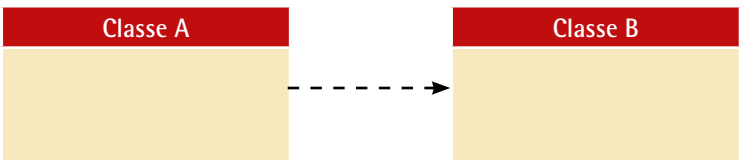

Figura 46 – Representação de dependência de classes

Comumente, a relação de dependência se dá quando uma classe utiliza informações de outra classe em um determinado método, como mostra a figura a seguir, na qual podemos notar que a classe Leitora de Cartão utiliza informações do objeto Cartão para realizar o comportamento "efetuarLeituraCartao".

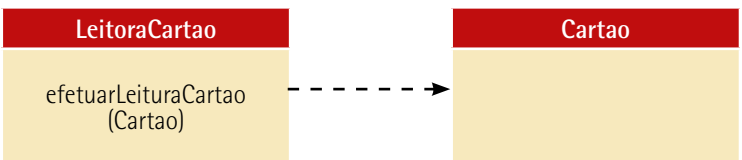

Figura 47 – Exemplo de dependência de classes

7.2.2.2 Associação

A associação mostra que um objeto pode se relacionar com outro, que existe uma conexão entre esses objetos. Existem dois tipos de associação:

- Associação binária: amplamente utilizada, é a associação entre duas classes apenas.
- Associação enésima: raramente utilizada, é a associação entre mais de duas classes (BOOCH; JACOBSON; RUMBAUGH, 2006).

Na UML, representamos a associação como uma reta, ou uma linha, que conecta as classes que se associam de alguma forma.

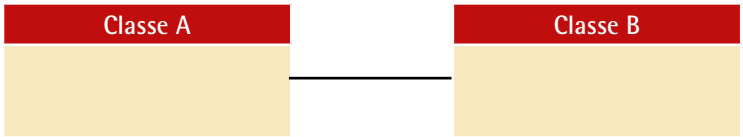

Figura 48 – Representação de associação de classes

Note que, embora representado corretamente, não conseguimos compreender, ou fazer a leitura, sobre o que se trata essa associação. Por isso, é comum que utilizemos algum tipo de notação que identifique a relação entre as classes em um determinado domínio, como mostra o quadro a seguir.

Essa notação, além de identificar a relação entre as classes, estabelece o papel que cada objeto ou classe estabelece nessa relação.

Quadro 18 – Exemplos de associação UML

Domínio	Representação UML	Interpretação
Vendas	Cliente — Compra — Produto	Um cliente compra produtos.
Bancário	ContaCorrente — Possui — Transacao	Uma conta-corrente possui transações efetuadas.
Hotelaria	Hospede — Ocupa — QuartoHotel	Um hóspede ocupa um quarto de hotel.

Adaptado de: Bezerra (2006, p. 99).

Note que no exemplo do domínio da hotelaria, tomando como base um exemplo real, poderíamos ter que um hóspede ocupa um quarto de hotel ou que, eventualmente, muitos hóspedes podem ocupar um quarto de hotel ou ainda que um quarto de hotel pode ser ocupado por no mínimo um hóspede e no máximo quatro. Outro exemplo, agora utilizando nosso caso do terminal de autoatendimento: um cliente pode efetuar infinitos saques.

Para quantificar essa associação, utilizamos o conceito de multiplicidade, que nada mais é que a representação da quantidade que um objeto pode possuir numa relação de associação.

Voltando ao exemplo do quadro, no domínio Hotelaria: suponhamos que, em um problema imaginário de reserva de quartos pela *web*, cada quarto de hotel possa ter, no máximo, quatro hóspedes,

sendo que eles podem, eventualmente, estar vazios. Sendo assim, teríamos o que é representado pela figura a seguir.

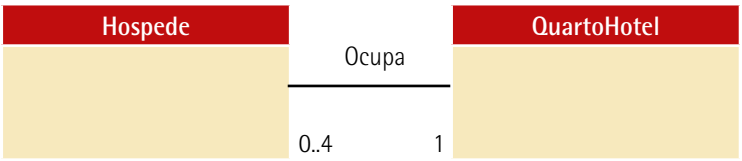

Figura 49 – Exemplo de multiplicidade na UML

Ainda na figura, temos duas interpretações possíveis a partir da representação, ambas corretas:

- Um hóspede ocupa no máximo um quarto de hotel.
- Um quarto de hotel pode ser ocupado por no mínimo zero ou no máximo quatro hóspedes.

Utilizando o caso do terminal de autoatendimento, no qual o cliente pode efetuar infinitos saques, poríamos representar o modelo conforme da seguinte forma:

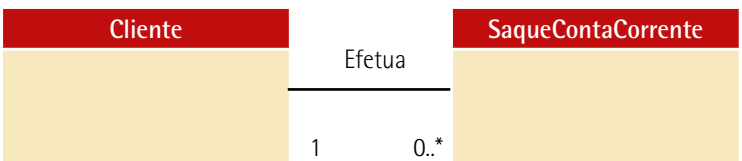

Figura 50 – Exemplo de multiplicidade em saque de conta-corrente

Temos duas interpretações possíveis a partir da representação:

- Um cliente pode efetuar no mínimo zero ou no máximo infinitos saques.
- Uma operação de saque de conta-corrente é feita por apenas um cliente.

Quando estamos tratando de multiplicidade, temos as possíveis opções de simbologia:

Quadro 19 – Opções para representações de multiplicidade

Representação	Interpretação
0..*	No mínimo zero e no máximo infinito OU Zero ou Muitos
1..*	No mínimo um e no máximo infinito OU Um ou Muitos
0..1	No mínimo zero e no máximo 1 OU Zero ou Um
1	Apenas Um
X..Y	Intervalo específico de no mínimo X e no máximo Y

Observação

O conceito de multiplicidade é muito semelhante ao conceito de cardinalidade, que pode ser observado no MER (Modelo Entidade Relacionamento) quando do desenvolvimento do modelo conceitual de um banco de dados. Mas é importante ter em mente que é uma semelhança apenas, uma vez que são utilizados em momentos e para finalidades distintas no projeto.

Existe ainda outra variação de associação, chamada associação reflexiva, que é quando objetos da mesma classe são associados, sendo que nessa associação esses objetos possuem papéis distintos, como mostra a figura a seguir.

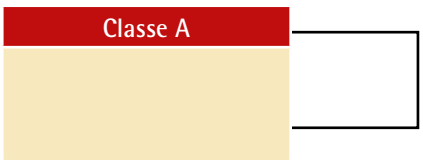

Figura 51 – Representação de associação reflexiva

Ainda a partir da figura, podemos tirar as seguintes conclusões:

- Um objeto X do tipo Classe A está associado a um objeto Y da mesma Classe A. No entanto, ambos desempenham papéis distintos nessa associação.

- Como observa Bezerra (2006), essa representação não significa que um objeto se associa com ele mesmo.

Lembrete

Na associação reflexiva, mais uma vez, reforça-se o conceito de que objeto é uma instância de uma classe.

A figura a seguir mostra um exemplo de aplicação da associação reflexiva, incluindo o conceito de multiplicidade. Note que a identificação da associação é fundamental para que não haja ambiguidade na interpretação dos papéis dos objetos.

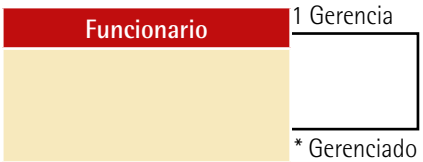

Figura 52 – Exemplo de associação reflexiva

A partir da figura, podemos chegar às seguintes interpretações:

- Uma instância da classe funcionário, como o objeto gerente, gerencia muitos funcionários ou instâncias da mesma classe.
- Uma instância da classe funcionário, como o objeto analista, é gerenciada por no máximo um gerente ou instância da mesma classe funcionário.
- Nesse caso, ambos, gerente e analista, são instâncias da classe funcionário, podemos dizer que ambos são tipos de funcionário.

7.2.2.3 Agregação

A agregação é utilizada para representar uma conexão entre dois objetos, sendo que essa conexão define uma relação todo-parte entre esses objetos, ou seja, um objeto está contido no outro (BEZERRA, 2006).

Na UML, representamos agregação utilizando uma reta saindo da classe que representa a parte e se conectando a um losango, também chamado de diamante, na classe que representa o todo.

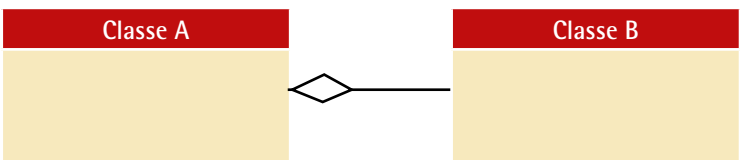

Figura 53 – Representação de agregação na UML

A partir da figura, alguns conceitos fundamentais da agregação podem ser interpretados:

- A Classe B é parte da Classe A e o inverso nunca é verdadeiro.
- A existência de um objeto da Classe B é independente da existência de um objeto da Classe A.

Assim como na associação, na agregação temos o conceito de multiplicidade, que define a quantidade de objetos agregados na relação todo-parte.

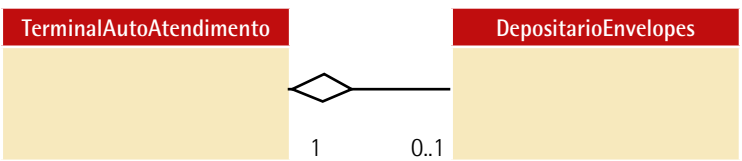

Figura 54 – Exemplo de agregação com multiplicidade na UML

Podemos interpretar que:

- A classe Depositário de Envelopes é parte da classe Terminal de Autoatendimento.
- Um objeto do tipo Depositário de Envelopes existe independentemente do terminal de autoatendimento (suponhamos um domínio de problema, mesmo que essa não seja a realidade).
- Um depositário pode fazer parte de, no máximo, um objeto terminal de autoatendimento.
- Em sua estrutura, um terminal de autoatendimento pode ter nenhum depositário ou, no máximo, um depositário de envelopes (pensemos em terminais de autoatendimento que não aceitem depósitos em envelope).

7.2.2.4 Composição

A composição tem exatamente o mesmo conceito da agregação, ou seja, estabelece uma relação todo-parte entre dois objetos.

A única diferença entre composição e agregação é que a classe que representa a parte da relação, para existir, depende da classe que representa o todo, ou seja, o ciclo de vida do objeto da classe parte depende do ciclo de vida do objeto da classe todo.

Na UML, representamos a composição de forma semelhante à agregação, utilizando uma reta saindo da classe que representa a parte e se conectando a um losango na classe que representa o todo. Todavia, na composição, o losango, ou diamante, é preenchido.

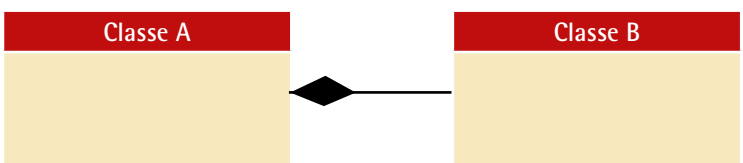

Figura 55 – Representação de composição na UML

Observando a figura, alguns conceitos fundamentais da composição podem ser interpretados:

- A Classe B é parte da Classe A e o inverso nunca é verdadeiro.
- A existência de um objeto da Classe B depende da existência de um objeto da Classe A, ou seja, o objeto da Classe B não existe sem um objeto da Classe A.

Assim como na associação e na agregação, na composição também temos o conceito de multiplicidade, que define a quantidade de objetos compostos na relação todo-parte.

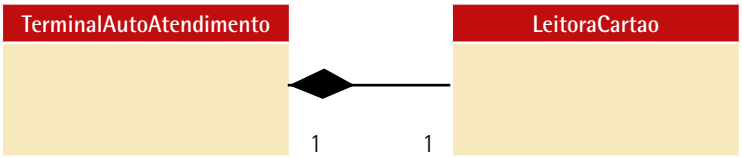

Figura 56 – Exemplo de composição com multiplicidade na UML

Na figura, podemos interpretar que:

- A classe Leitora de Cartão é parte da classe Terminal de Autoatendimento.
- Um objeto do tipo Leitora de Cartão não existe se não existir também um objeto do tipo Terminal de Autoatendimento (suponhamos um domínio de problema, mesmo que essa não seja a realidade).
- Em sua composição, um terminal de autoatendimento tem, no máximo, uma leitora de cartão, assim como apenas uma leitora de cartão pode compor um terminal de autoatendimento.

7.2.2.5 Classes associativas

Classes associativas são associações que possuem propriedades de classe (BOOCH; JACOBSON; RUMBAUGH, 2006), ou, como observa Bezerra (2006), são classes que, em vez de estarem ligadas à outras classes, estão ligadas a uma associação.

Normalmente utilizamos classes associativas quando desejamos armazenar informações importantes de uma associação.

Na UML, representamos uma classe associativa da mesma forma que representamos uma classe "normal", contudo, ela é ligada por uma linha tracejada, que termina na linha que define a associação de duas outras classes, como mostra a figura a seguir.

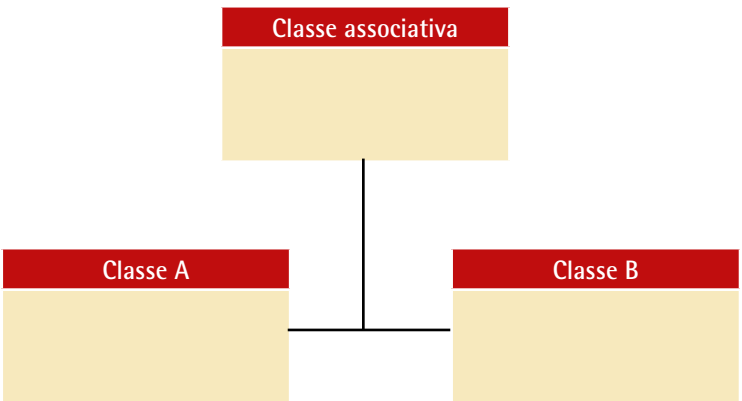

Figura 57 – Representação de classes associativas UML

Podemos afirmar, a partir da figura anterior, que a instância da Classe Associativa produz uma conexão da instância das classes que compõem essa associação: a Classe A e a Classe B.

Um exemplo clássico na utilização de classes associativas pode ser notado na figura a seguir.

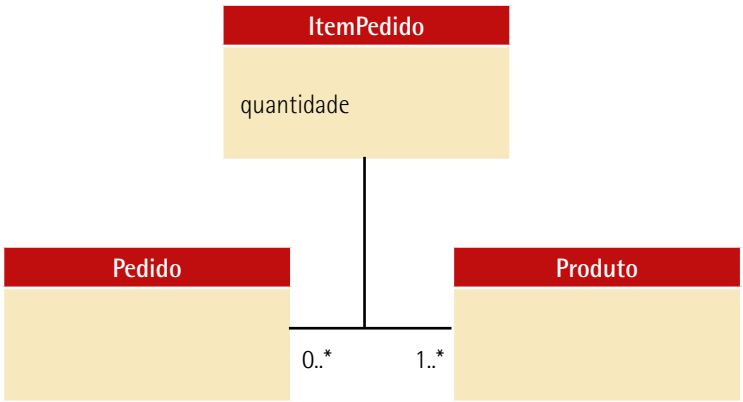

Figura 58 – Exemplo de utilização de classes associativas

No exemplo, um pedido pode conter um ou mais produtos e um produto pode estar apenas em estoque ou pode estar em inúmeros pedidos. A figura mostra a classe Item de Pedido como uma classe associativa que quantifica essa quantidade de produtos em um pedido.

Observação

Embora os conceitos de agregação, composição e associação (incluindo classes associativas) sejam semelhantes, existem diferenças substanciais no uso de cada um. O mau uso acarreta problemas na modelagem e na interpretação do domínio.

7.2.3 Herança

O conceito de herança é um dos conceitos mais importantes e mais utilizados no paradigma da orientação a objetos.

Para entender a herança em orientação a objetos, vamos fazer um paralelo com o conceito de herança no mundo real.

Nós, enquanto seres humanos, temos características e comportamentos comuns a todos os seres humanos. Temos um conjunto de órgãos e movimentos físicos que nos caracteriza como seres humanos. Isso nos permite dizer que somos tipos de seres humanos ou que todos nós herdamos comportamentos e características comuns a todos os seres humanos.

Na orientação a objetos, assim como no mundo real, um objeto pode herdar características e comportamentos de outro objeto. Traduzindo para a terminologia correta, um objeto pode herdar atributos e métodos de outro objeto.

Podemos dizer ainda que, se um objeto é uma instância de uma classe ou uma classe é uma especificação de um objeto, uma classe herda atributos e métodos de outra. Nessa situação temos dois papéis de classes na relação: classe mãe e classe filha.

Classe mãe, ou superclasse, possui atributos e métodos que podem ser herdados por uma ou mais classes filhas, também chamadas de subclasses.

Na UML, representamos herança com uma seta direcional, sendo que a ponta da seta é cheia, como mostra a figura.

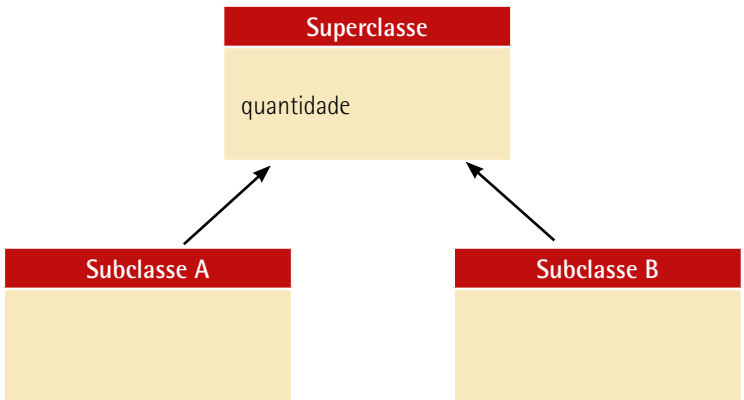

Figura 59 – Representação de herança na UML

Ainda de acordo com a figura, podemos afirmar:

- A superclasse é uma classe mãe das subclasses A e B.
- As subclasses A e B são classes filhas da Superclasse.
- A subclasse A é um tipo da superclasse, assim como a subclasse B.
- As subclasses A e B herdam atributos e métodos da superclasse. No entanto, podem ter seus próprios atributos e métodos, que não serão compartilhados entre elas ou com a superclasse.
- As subclasses A e B são especializações da superclasse.
- A superclasse é uma generalização das subclasses A e B.

A figura anterior mostra um exemplo claro de herança simples, no qual uma classe filha herda atributos e métodos de apenas uma classe mãe. No caso, as subclasses A e B herdam atributos e métodos apenas da superclasse.

Todavia, existe ainda outro tipo de herança: a herança múltipla. Diferentemente da herança simples, na herança múltipla uma classe filha pode herdar atributos e métodos de duas ou mais classes mãe.

A figura a seguir mostra a representação de uma herança múltipla, na qual é possível concluir que:

- A subclasse herda os atributos e métodos das superclasses A e B.
- A subclasse é um tipo das superclasses A e B.
- A subclasse é uma especialização tanto da superclasse A quanto da superclasse B.
- A superclasse A e a superclasse B são generalizações da subclasse.

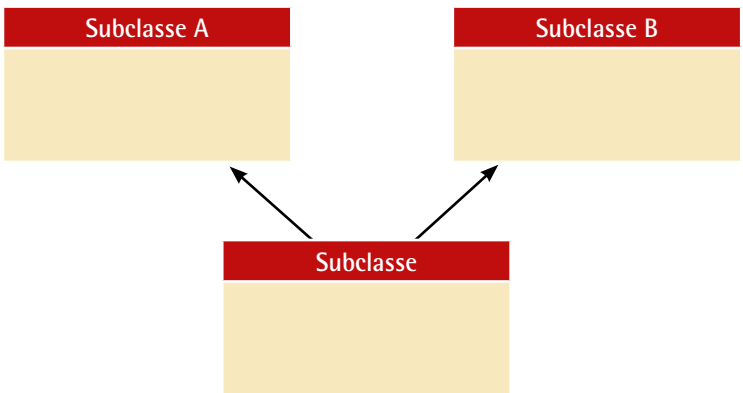

Figura 60 – Representação de herança múltipla na UML

Quando falamos em herança de classes, estamos falando em criar uma hierarquia, na qual são comuns os termos "um tipo de", "generalização" e "especialização", como pode ser notado anteriormente.

Hierarquia de classes pressupõe uma relação de "um tipo de", na qual uma classe filha é um tipo de uma classe mãe, ou seja, ela possui todas as características da classe mãe e mais as suas próprias.

O mesmo raciocínio pode ser empregado na terminologia "generalização" e "especialização". A partir do momento em que se cria uma hierarquia de classes na qual a classe filha possui todas as características da classe mãe e mais as suas próprias, podemos dizer que ela é uma especialização da classe mãe, que, por sua vez, é uma generalização de suas classes filhas.

A herança é dos principais conceitos da orientação a objetos, pois fecha com um dos principais benefícios do paradigma: o reuso. A partir do momento em que uma classe filha herda os atributos e métodos de uma classe mãe, não existe necessidade de reescrevermos essas mesmas características na classe filha, podendo reusar o código sem a necessidade de reimplementá-lo.

Observação

Um dos segredos para reuso em orientação a objetos é definir muito bem as responsabilidades de cada classe. Em geral, classes com muitas responsabilidades são pouco reutilizáveis e pouco manuteníveis.

A figura a seguir mostra um exemplo prático de herança no qual temos uma classe Cliente, que representa todos os clientes de um banco, que podem efetuar saque em terminais de autoatendimento, e duas especializações dessa classe: Cliente Pessoa Física e Cliente Pessoa Jurídica.

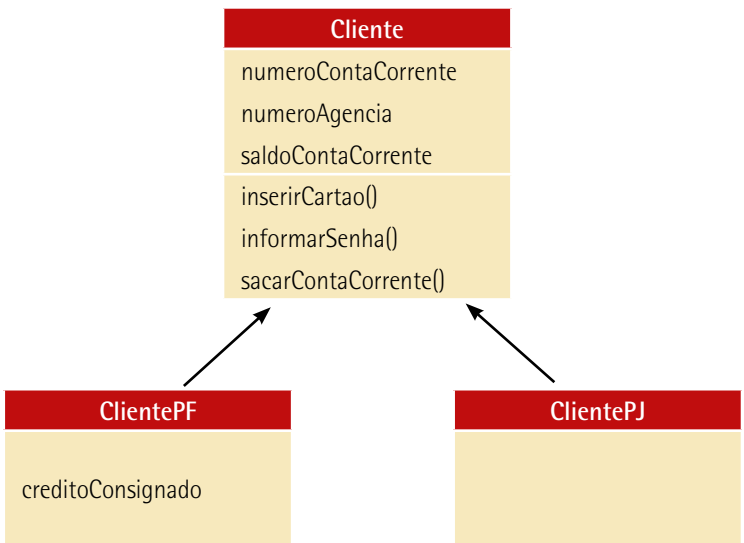

Figura 61 – Exemplo de herança na UML

Mesmo que não explicitamente representado na figura, é possível afirmar que:

- A classes ClientePF e ClientePJ possuem os atributos: numeroContaCorrente, numeroAgencia e saldoContaCorrente.
- A classes ClientePF e ClientePJ possuem os métodos: inserirCartao, informarSenha e sacarContaCorrente.
- Nada impede que uma classe filha possua seus próprios atributos e métodos. Sendo assim, a subclasse ClientePF possui todos os atributos da classe Cliente e mais o atributos de valor de crédito consignado: creditoConsignado.

7.2.4 Diagrama de classes

Segundo Booch, Jacobson e Rumbaugh (2006), diagrama de classes é um diagrama UML que tem como objetivo representar a estrutura estática das classes de um sistema de *software*.

Resumindo, um diagrama de classes é a representação das classes, seus atributos, métodos e o relacionamento entre essas classes, que debatemos na seção anterior.

No entanto, é importante que saibamos algumas diferenças entre modelo de classes e diagrama de classes.

O modelo de classes é mais abrangente. Nele não representamos apenas as classes, mas também devemos nos atentar em descrever detalhadamente o que cada classe representa dentro do domínio do problema.

Assim sendo, podemos considerar que um modelo de classes se utiliza do diagrama de classes como ferramenta para representar uma determinada visão ou um determinado tipo do modelo de classes.

Lembrete

Existem três modelos de classe: domínio, especificação e implementação. A representação do diagrama de classes deve estar alinhada com o objetivo de cada um deles.

A Figura a seguir mostra um exemplo de diagrama de classes e o quadro em seguida mostra os conceitos e as interpretações do modelo de classes representado pelo diagrama de classe da figura.

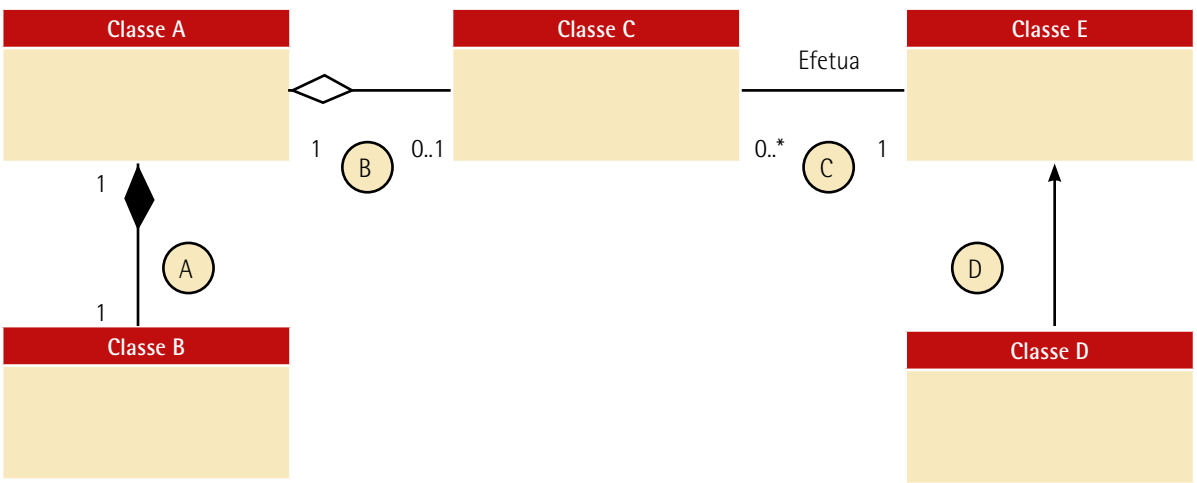

Figura 62 – Exemplo de diagrama de classes na UML

Quadro 20 – Conceitos e interpretações para o diagrama de classes

Letra	Conceitos	Interpretações
A	– Composição – Multiplicidade	– Um objeto da Classe A é composto por no máximo um objeto da Classe B. – Um objeto da Classe B compõe no máximo um objeto da Classe A. – O objeto da Classe B só existe se um objeto da Classe A também existir.
B	– Agregação – Multiplicidade	– Um objeto da Classe A agrega em sua estrutura no mínimo zero e no máximo um objeto da Classe C. – Um objeto da Classe C agrega no máximo um objeto da Classe A. – O objeto da Classe C existe independentemente do objeto da Classe A.
C	– Associação – Multiplicidade – Navegabilidade – Papéis	– Um objeto da Classe C possui uma associação com um objeto da Classe E. Essa associação é identificada utilizando o verbo "Efetua". – A identificação da associação define os papéis dos objetos nessa associação. No exemplo, quem é o sujeito e o predicado do "efetuar". – A direção da associação define o sentido da associação, a navegabilidade da associação. – A identificação da multiplicidade indica que um objeto da Classe C efetua no máximo um objeto da Classe E da mesma forma que um objeto da Classe E é efetuado por no mínimo zero objetos da Classe C e no máximo infinitos objetos dessa classe.
D	– Herança – Hierarquia de Classes – Especialização – Generalização	– Um objeto da Classe D herda os atributos e métodos de um objeto da Classe E. – A Classe D é um tipo da Classe E. – A Classe E é uma generalização da Classe D. – A Classe D é uma especialização da Classe E.

A figura a seguir mostra um possível diagrama de classes do modelo de domínio para o problema de saque em um terminal de autoatendimento. Note que estamos representando apenas a visão da operação de saque, embora, em um terminal de autoatendimento, tenhamos outras operações, como depósito, e outros periféricos, como impressora ou depositário de envelopes.

Observação

Um diagrama de classes pode ser tão grande quanto exige a complexidade do problema a ser modelado. É comum que tenhamos diversos diagramas que representem visões parciais do domínio, mas que em conjunto representem a visão do todo.

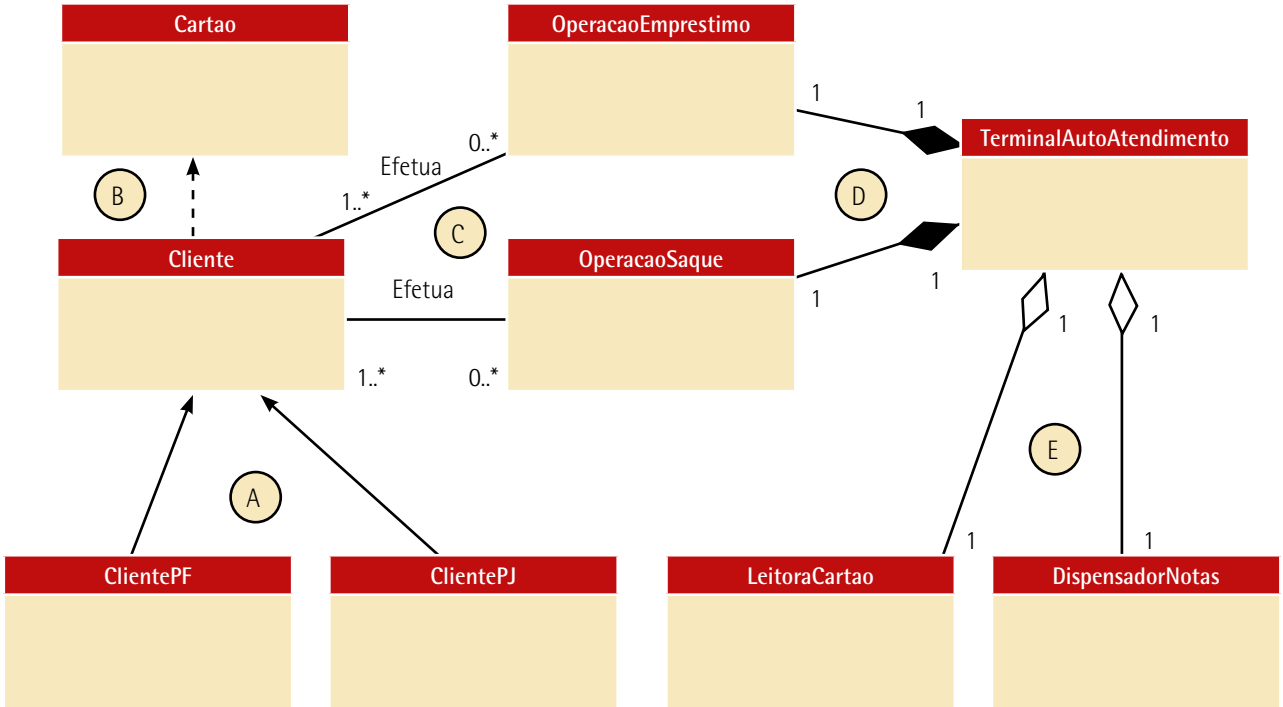

Figura 63 – Diagrama de classes do saque em autoatendimento

O quadro a seguir mostra os conceitos e as interpretações do modelo de classes representado pelo diagrama de classes da figura.

Quadro 21 – Conceitos e interpretações diagrama de classes: efetuar saque

Letra	Conceitos	Interpretações
A	– Herança – Hierarquia de Classes – Especialização – Generalização	– Cliente Pessoa Jurídica e Cliente Pessoa Física herdam atributos e métodos de Cliente. – Cliente Pessoa Jurídica e Cliente Pessoa Física herdam as mesmas associações e dependência de Cliente. – Cliente Pessoa Jurídica e Cliente Pessoa Física são tipos de Cliente. – Cliente Pessoa Jurídica e Cliente Pessoa Física são especializações de Cliente. – Cliente é uma generalização de Cliente Pessoa Jurídica e Cliente Pessoa Física.
B	– Dependência – Navegabilidade	– Cliente depende de um objeto do tipo Cartão para realização de algum comportamento. – A direção da seta da dependência dá sentido e navegabilidade na dependência.
C	– Associação – Multiplicidade – Navegabilidade – Papéis	– Um Cliente efetua no mínimo zero e no máximo inúmeras operações de empréstimo. – Um Cliente efetua no mínimo zero e no máximo inúmeras operações de saque. – Operações de Saque e de Empréstimo são efetuadas por no mínimo um Cliente e no máximo por inúmeros.

D	– Composição – Multiplicidade	– Um terminal de autoatendimento é composto por algumas operações, entre elas: Empréstimo e Saque (*). – Falando em multiplicidade, um terminal é composto por apenas uma operação de Empréstimo e uma operação de Saque. Assim como um objeto que represente essas operações compõe apenas um objeto do tipo Terminal de Autoatendimento. – Os objetos das operações de Empréstimo e Saque não existem se um objeto do tipo Terminal de Autoatendimento não existir.
E	– Agregação – Multiplicidade	– Um objeto Terminal de Autoatendimento agrega em sua estrutura objetos do tipo Leitor de Cartão e Dispensador de Notas. – Falando em multiplicidade, um Terminal de Autoatendimento agrega em sua estrutura apenas uma Leitora de Cartão e apenas um Dispensador de Notas. Assim como uma Leitora de Cartão ou um Dispensador de Notas agrega apenas um objeto do tipo Terminal de Autoatendimento. – Objetos do tipo Leitora de Cartão e do tipo Dispensador de Notas existem independentemente da existência de objetos do tipo Terminal de Autoatendimento.

(*) O diagrama dá ênfase apenas às operações envolvidas com Saque, domínio inicial do exemplo proposto.

Observação

Podemos notar que, além da representação da visão estática do projeto, o diagrama de classes também envolve a modelagem do vocabulário do sistema (BOOCH; JACOBSON; RUMBAUGH, 2006).

7.2.5 Diagrama de objetos

O objetivo de um diagrama de objetos é representar instâncias de classes, ou seja, objetos e suas respectivas ligações ou associações. Podemos dizer, então, que a base do diagrama de objetos é o diagrama de classes.

O diagrama de objetos fornece uma visão dos objetos, suas informações e seus relacionamentos em um determinado cenário em determinado momento de execução desse cenário.

O fato desse diagrama representar um estado momentâneo de um conjunto de objetos nos permite pensar que podemos ter, na modelagem de um domínio específico, diversos diagramas de objetos, uma vez que o estado de um objeto pode mudar com o passar do tempo.

Lembrete

A instância de uma classe é um objeto, e seus atributos correspondem a um conjunto de informações que possuem determinados valores que, por sua vez, compõem o estado desse objeto em um determinado momento.

Se um objeto é originado de uma classe, logo, na UML, um objeto é representado da mesma forma que representamos uma classe, mas com alguns pequenos detalhes.

A figura a seguir mostra três possíveis representações para objetos, instâncias:

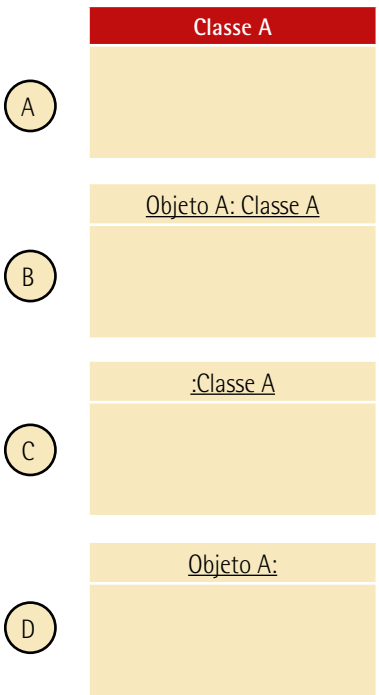

Figura 64 – Representação de Objetos na UML

A) A classe **ClasseA** foi identificada no modelo de classes como sendo do domínio e possui atributos e métodos.

B) Representa um objeto de nome **Objeto A**, que é uma instância da classe **ClasseA**. Essa é a representação mais utilizada de objetos:

Nome do Objeto + Separador : + Nome da Classe

C) Representa um objeto não identificado, também chamado de anônimo, que é uma instância da classe **ClasseA**.

D) Representa um objeto de nome **Objeto A** cuja classe não foi especificada. Esse tipo de representação é pouco utilizado pelo grau de ambiguidade que acaba gerando. Utilizamos essa notação normalmente em um determinado momento da modelagem, quando ainda não temos a definição pronta da classe e desejamos fazer uma simulação de um determinado cenário do qual esse objeto faz parte.

Voltando ao nosso exemplo do saque em terminais de autoatendimento, observe o diagrama de classes representado na figura 63. Apenas como exemplo, vamos dar ênfase na representação de dois objetos: Cliente e Operação de Saque.

A figura a seguir mostra uma "fotografia" dos objetos Cliente e Operação de Saque representados em um diagrama de objeto. Note que os atributos de cada objeto possuem seus respectivos valores, sendo possível verificar o estado desses objetos em um determinado momento no tempo.

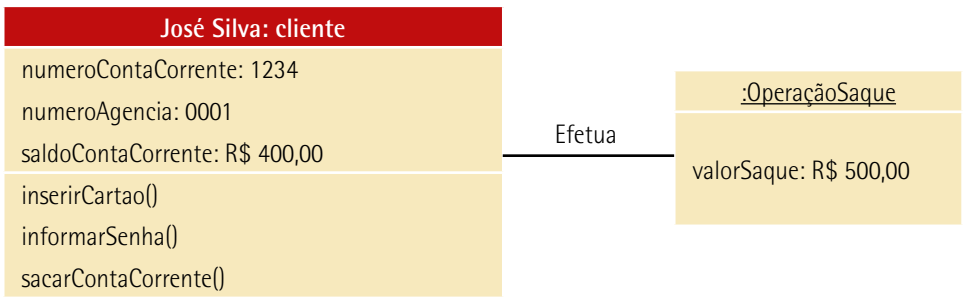

Figura 65 – Exemplo de Diagrama de Objetos UML

É possível notar que o cliente deseja efetuar uma operação de saque de valor superior ao valor do atributo "saldoContaCorrente", sendo possível tirarmos algumas pequenas conclusões sobre o comportamento dos objetos em vista do comportamento esperado, modelado nos casos de uso.

Saiba mais

Existem outros diagramas que representam a estrutura estática de um sistema, como: Diagrama de Pacotes, Diagrama de Componentes e Diagrama de Instalação.

Saiba mais em:

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.

Os diagramas de classes e de objetos são os principais diagramas que representam a visão estrutural estática de um sistema, todavia, em um sistema existem aspectos dinâmicos, reações e respostas que o sistema produz e que não são captadas nessas representações.

8 VISÃO ESTRUTURAL DINÂMICA

Um sistema de *software*, dentro do paradigma da orientação a objetos, pode ser considerado como uma coleção de objetos que possuem atributos e métodos e que interagem entre si para resolver um determinado problema.

Já vimos que o modelo estrutural estático, representa a coleção desses objetos, seus atributos, métodos e seus relacionamentos, mas a forma como eles interagem entre si para resolver um problema não pode ser visualizada em um diagrama de classes, tampouco em um diagrama de objetos.

A visão estrutural dinâmica, também chamada de visão comportamental, tem como objetivo representar a interação dos objetos para atingir um determinado objetivo.

O objetivo a ser atingido por um conjunto de objetos, como vimos, está representado nos casos de uso, nas regras de negócio. Logo, podemos dizer que um conjunto de objetos interage entre si para a realização de casos de uso e a forma como essa interação se dá é representada pelo modelo estrutural dinâmico.

O modelo estrutural dinâmico é originado do modelo estático. Não existe comportamento dinâmico que não tenha sido representado nos diagramas de classe ou de objeto. Qualquer tipo de alteração no modelo dinâmico acarreta mudança no estático e vice-versa.

8.1 Realização de casos de uso

O fundamental para a realização de casos de uso é saber que essa realização se dá a partir da interação entre objetos, todavia, apenas isso não basta. Façamos um paralelo com o nosso dia a dia: quando estamos trabalhando em equipe e precisamos realizar determinada atividade, o primeiro passo que damos é definir as responsabilidades de cada um.

Em orientação a objetos não é diferente, precisamos, em primeiro lugar, definir as responsabilidades de cada objeto. A abordagem para definição de responsabilidades de objetos é chamada de método dirigido a responsabilidades.

O método dirigido a responsabilidades se baseia em um dos conceitos fundamentais da orientação a objetos: encapsulamento do comportamento e das características do objeto (WIRFS-BROCK e WILKERSON, 1989).

Lembrete

Encapsulamento significa deixar visível apenas o que é necessário para a comunicação entre dois objetos, como detalhes da implementação ou a lógica algorítmica de um método.

Nesta seção iremos debater como a abordagem dirigida a responsabilidades nos auxilia na divisão das responsabilidades dos objetos, alguns aspectos e conceitos fundamentais quando falamos em comportamento de objetos e como, efetivamente, se dá a interação ou comunicação entre esses objetos.

Saiba mais

Além da abordagem dirigida a responsabilidades, existe outra abordagem: a orientada a dados. Saiba mais sobre o tema "*Data Driven Design*" no *site* <http://www.dataorienteddesign.com/dodmain/>.

8.2 Classes de análise

Como observa Bezerra (2006), a responsabilidade de um objeto é a "obrigação" que ele deve cumprir no sistema de *software* do qual faz parte. Todavia, em alguns casos, o objeto não consegue cumprir com a sua "obrigação" de forma autônoma, precisando da colaboração de outros objetos.

Voltando para o nosso exemplo do terminal de autoatendimento, o objeto Cliente deseja efetuar um Saque em um Terminal de Autoatendimento. Assim sendo, o objeto "TerminalAutoatendimento" tem a responsabilidade de efetuar o saque e dispensar as cédulas para o Cliente. No entanto, ele necessita da colaboração do objeto "DispensadorNotas" para cumprir com a sua "obrigação".

Os objetos são divididos e categorizados em três grupos de acordo com seu tipo de responsabilidade: classe entidade, classe de controle e classe de fronteira.

A forma de organizar essas classes, também chamadas de classes de análise, vai ao encontro de um dos princípios fundamentais da orientação a objetos: divisão de responsabilidades.

A divisão de responsabilidades é uma das características fundamentais em uma boa modelagem de sistemas. Objetos com responsabilidades bem definidas aumentam a sua capacidade de reúso.

Organizar e dividir os objetos por responsabilidade é a base para o conceito de padrões de projeto, que vem a ser um conjunto de soluções e organização sistêmica com um objetivo específico.

No caso, a divisão de responsabilidades pode ser encarada como um padrão de projeto com o objetivo de aumentar o reúso e diminuir o acoplamento entre objetos de um sistema. Esse conceito é a base para o padrão de projeto MVC (*Model-View-Controller*).

Saiba mais

O padrão MVC propõe a divisão lógica da aplicação em três camadas: *Model, View* e *Controller*. Sobre o assunto, sugerimos a leitura de:

COUTAZ, J. PAC, an Object-Oriented Model for Dialog Design. *In:* BULLINGER, H-J.; SHACKEL, B. (eds.). *INTERACT 87* – 2nd IFIP International Conference on Human-Computer Interaction. Estugarda: Elsevier Science Publishers, 1987. p. 431-436. Disponível em: <http://iihm.imag.fr/publs/1987/Interact87.PAC.pdf>. Acesso em: 15 dez. 2014.

A respeito de padrões de projeto, leia:

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

8.2.1 Entidade

Uma entidade, classe de entidade ou ainda objeto de entidade, são objetos mais próximos do domínio do mundo real que o sistema representa, são abstrações do mundo real que normalmente conseguimos identificar nos casos de uso.

Esses objetos têm como objetivo principal manter informações referentes ao domínio de problema que queremos resolver.

Nas classes de entidade, representamos informações e comportamentos que são, de alguma forma, armazenados no sistema. Por exemplo: dentro do domínio do nosso problema de saque em terminal de autoatendimento, temos as classes de entidade cliente, cartão e terminal de autoatendimento.

8.2.2 Fronteira

Classes de fronteira ou objetos de fronteira, como o próprio nome já diz, têm como responsabilidade dividir o ambiente interno do sistema e suas interações externas.

Podemos interpretar interações externas a um sistema como toda e qualquer comunicação que um sistema faz com atores do sistema ou ainda alimentar informações de outros sistemas. Por exemplo: dentro do domínio do nosso problema de saque, o sistema deve se comunicar com outro sistema, externo, responsável por efetuar a autenticação de segurança da senha de acesso do cliente.

Outros exemplos clássicos de interface externa:

- Envio de informações para sistemas governamentais, como informações fiscais.
- Consumo de informações de outros sistemas, como informações de crédito.
- Interface com SGBD (Sistema Gerenciador de Banco de Dados).

Lembrete

Lembrando que vimos, em casos de uso, que atores de um sistema são agentes externos ao sistema, que executam determinada ação e que esperam algum resultado, podendo ser um usuário, um dispositivo de *hardware* ou outro sistema.

8.2.3 Controle

Classes de controle, objetos de controle ou ainda controladores são objetos que têm como objetivo realizar o sequenciamento da execução de um caso de uso na estrutura de objetos do sistema, fazer a coordenação entre as camadas internas do sistema, representadas pelas classes de entidade, com as

camadas externas ao sistema, representadas pelas classes de fronteira. Alguns autores também chamam esse movimento de orquestração.

Utilizando a abordagem da divisão de responsabilidades e o exemplo do saque em um terminal de autoatendimento, a figura a seguir mostra um exemplo de representação de classes de entidade, controle e fronteira.

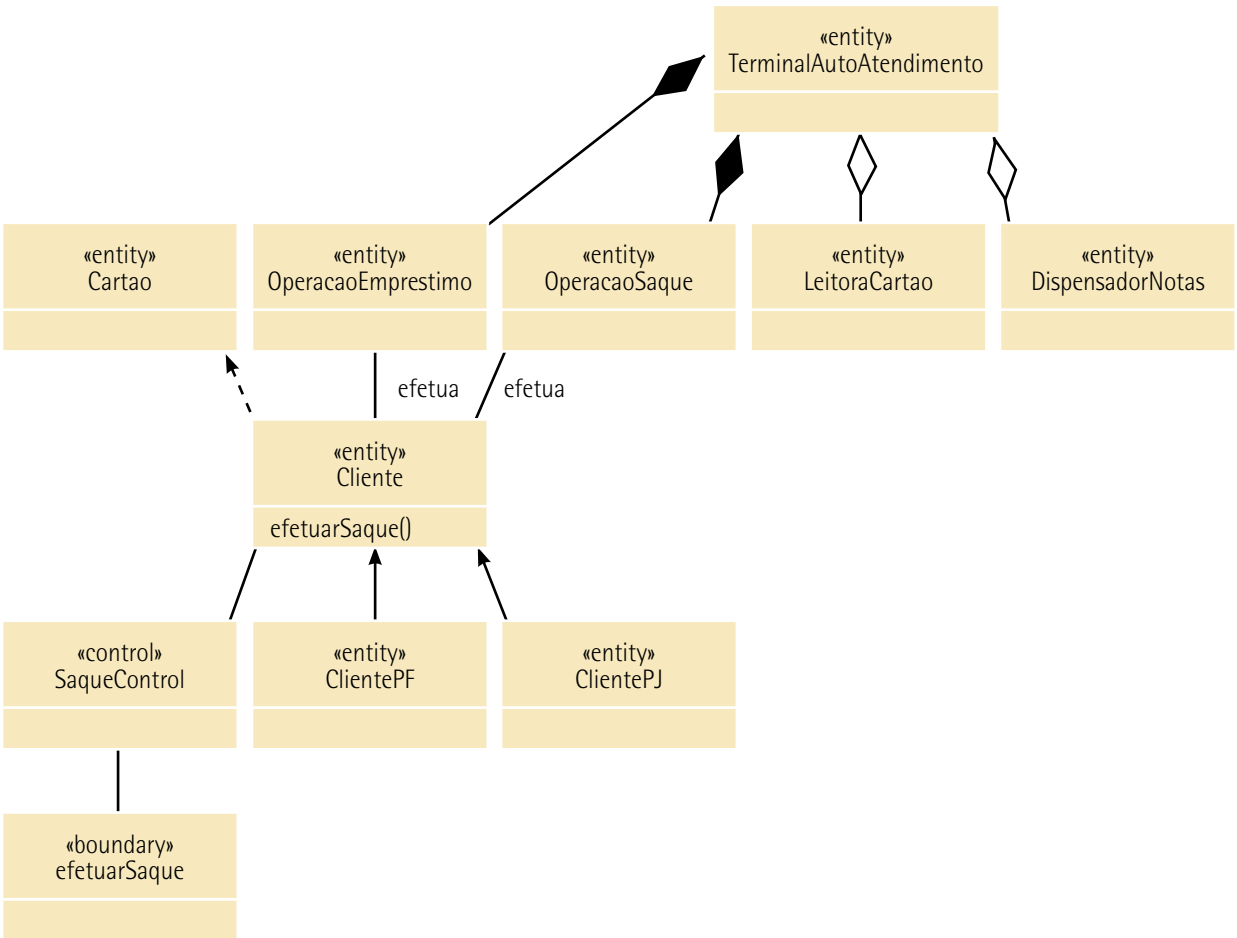

Figura 66 – Representação UML com classes de análise

Note ainda que esse modelo representa outra visão do modelo representado, anteriormente, na figura 63, pois aquele modelo nos dá a visão das classes de domínio e este, o modelo de classes de análise.

Ocorre que o modelo de classes de domínio está sempre associado ao modelo de classes de análise, sendo que durante o desenvolvimento de um é possível e, inclusive, bastante comum, que tenhamos o refinamento do outro.

Como vimos, em um projeto, podemos ter diversas visões do sistema, a abordagem com ênfase na divisão de responsabilidades é uma representação mais próxima do modelo de implementação, pois representa classes e objetos que farão parte da solução final.

No exemplo da figura 66, é possível notar duas classes que são próximas à implementação:

- EfetuarSaque (<<*Boundary*>>): *Boundary*, ou fronteira, como vimos, divide o ambiente externo da aplicação com o ambiente interno. Podemos interpretar que essa classe representa uma interface do usuário (ator).

- SaqueControl (<<*Control*>>): *Control*, ou controle, como vimos, tem a função de orquestrar a divisão entre a classe mais externa, fronteira, das classes internas do sistema: entidade (*entity*).

Observação

Os estereótipos são utilizados para identificarmos, em um diagrama de classes, as classes do tipo entidade (*entity*), controle (*control)* e fronteira (*boundary*).

Note que na representação do modelo de classes de análise, alguns aspectos importantes a um sistema de *software* não foram representados, como o comportamento da cada objeto e como eles interagem entre si, ou se comunicam, para atingir um determinado objetivo.

8.3 Comportamento de objetos

Como vimos, objetos possuem características e comportamentos que são representados nas suas respectivas classes por seus atributos e métodos.

Métodos têm como objetivo a realização de alguma atividade ou algum comportamento por parte do objeto, que, na maioria das vezes, inclui a modificação do valor de algum atributo do próprio objeto ou de objetos relacionados, consequentemente o estado desse(s) objeto(s) será alterado.

Comportamentos, ou métodos de objetos, possuem certos aspectos a serem conceituados, como a forma que os representamos e características próprias da orientação a objetos que veremos na sequência.

8.3.1 Representando métodos

Na representação, ou na forma de representar um método, podemos fazer uma analogia com funções algorítmicas.

Saiba mais

Saiba mais sobre os conceitos básicos de algoritmos, incluindo funções, em:

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

Ao escrever uma função, em algoritmos, temos três elementos a serem observados:

- Nome da função: toda função tem um nome que é utilizado em outros pontos do algoritmo quando desejamos chamar essa função. Por exemplo: na linguagem C, temos a função *printf*, que tem como objetivo escrever um texto qualquer em uma aplicação console.

- Parâmetros de entrada: funções algorítmicas podem ou não possuir parâmetros de entrada e esses parâmetros necessariamente devem ser de um determinado tipo de dado. Utilizando o exemplo anterior, a função *printf* da linguagem C tem como parâmetro de entrada o texto a ser impresso no console da aplicação.

- Saída: funções algorítmicas podem ou não possuir uma única saída, algum valor que será "devolvido" àquele que "chamou a função". Essa saída também possui um determinado tipo de dado. Por exemplo: a função *strlen* na linguagem C tem como objetivo calcular a quantidade de caracteres em um texto. Dessa forma, essa função possui um parâmetro de entrada do tipo texto, que representa o texto cujo tamanho será calculado, e possui uma saída do tipo inteiro, que determinará o tamanho do texto.

Representamos métodos da mesma forma que representamos funções. Um método, bem como uma função, possui um nome que o identifique e pode ou não possuir parâmetros de entrada e ou saídas com determinados tipos de dados.

Lembrete

Lembre-se das boas práticas para se identificar um método: verbos no infinitivo seguido de um substantivo que é diretamente afetado ou tem ligação à ação praticada.

A figura a seguir mostra dois exemplos de representação de métodos, ou comportamentos, de um objeto. No caso, um objeto da classe Cliente, proveniente do domínio do problema efetuar saque em terminal de autoatendimento.

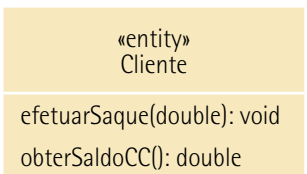

Figura 67 – Exemplo de Representação de Métodos

Ainda podemos notar dois métodos:

- efetuarSaque: o método efetuar saque possui um parâmetro de entrada do tipo *double*, que representa a quantidade de dinheiro que se deseja sacar. O método não possui nenhum retorno, assim, indicamos o método como *void*, assim como fazemos em funções algorítmicas.

- obterSaldoCC: o método não possui nenhum parâmetro de entrada e possui uma saída, ou retorno, do tipo *double*, que representa o saldo em conta-corrente.

O significado dos parâmetros de entrada ou das saídas não são representados no diagrama de classes, como podemos notar na figura, todavia, precisam ser especificados dentro do modelo de classes para que o diagrama e o modelo não deem margem para dúvidas.

8.3.2 Polimorfismo

Como vimos, conceitualmente, polimorfismo é quando um objeto tem um comportamento diferente para a mesma ação. Agora, com os conceitos de orientação a objetos e, principalmente, de herança, podemos aprofundar na definição de polimorfismo.

Pensemos no modelo de classes do domínio saque em terminal de autoatendimento, representado, por exemplo, na figura 66.

Nesse modelo é possível ver a relação de herança entre as classes *Cliente*, *ClientePJ* e *ClientePF*, na qual podemos interpretar que as classes Cliente Pessoa Jurídica e Cliente Pessoa Física herdam atributos e métodos da classe Cliente.

A complementação desse modelo, exibido na Figura 67, mostra que a classe *Cliente* possui dois métodos: *efetuarSaque* e *obterSaldoCC*, logo, as classes *ClientePJ* e *ClientePF* também possuem esses métodos exatamente com a mesma assinatura.

Suponhamos que a forma de cálculo do saldo de conta-corrente, especificados nas regras de negócio e nos casos de uso, para Cliente Pessoa Jurídica e Cliente Pessoa Física sejam diferentes e sigam as seguintes regras:

- Cliente Pessoa Jurídica: o saldo para saque será calculado pelo valor do saldo em conta-corrente somado ao limite pré-definido para pessoa jurídica.

- Cliente Pessoa Física: o saldo será calculado pelo valor do saldo em conta-corrente somado ao crédito pessoal pré-aprovado.

Note que nessa situação temos o mesmo resultado dos objetos *ClientePJ* e *ClientePF* quanto a saldo de conta corrente, ou seja, o método *obterSaldoCC* continua sem receber nenhum parâmetro e continua tendo como saída um número com virgula, ou seja, a assinatura do método se mantém. Todavia, o comportamento dos objetos *ClientePJ* e *ClientePF* diferem quando cada um executa o método *obterSaldoCC*.

Polimorfismo é quando uma classe filha herda um método de uma classe mãe, ou seja, o método na classe filha possui a mesma assinatura do método da classe mãe, todavia possuem implementações diferentes, reagem de maneiras diferentes à mesma ação.

Polimorfismos nos conferem duas características importantes na orientação a objetos (LARMAN, 2007):

- Facilidade na introdução de novas implementações, possibilitando manutenibilidade e extensão ao sistema.

- Encapsulamento das variações de implementação.

Na UML, não temos uma representação específica para polimorfismo, como mostra a figura a seguir. No entanto, vale ressaltar que a especificação do método polimórfico pode não constar literalmente no diagrama de classes da UML, mas deve necessariamente estar documentado no modelo de classes.

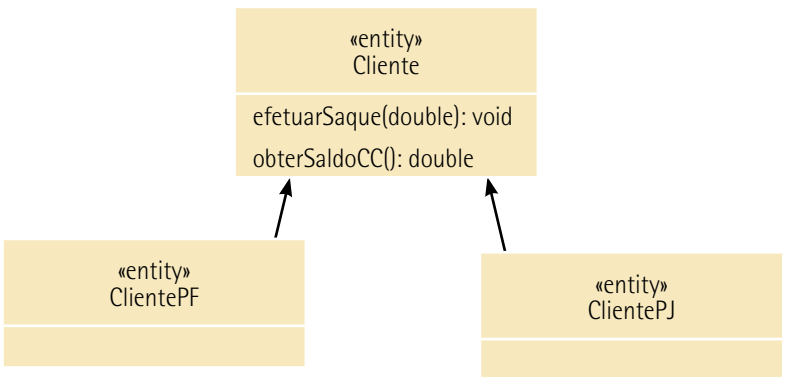

Figura 68 – Exemplo de métodos polimórficos

8.3.3 Visibilidade de atributos e métodos

Como vimos, objetos possuem atributos, métodos e se relacionam com outros objetos a partir de herança, associação, composição, agregação e relação de dependência.

Vimos também que encapsulamento é a capacidade que um objeto tem de "esconder" de outros objetos que se relacionam com ele detalhes internos, como detalhes da implementação de um método.

Visibilidade indica quando e em que nível um atributo ou um método de um objeto pode ser acessível aos objetos que se relacionam com ele. Em orientação a objetos, temos três níveis de visibilidade de atributos e métodos:

- Público: o atributo ou o método pode ser acessado por qualquer classe. Na UML, indicamos que um atributo ou um método é público utilizando o sinal + (positivo), conforme exemplo a seguir.

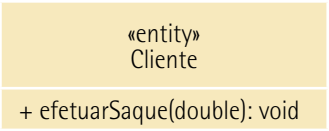

Figura 69 – Exemplo de representação de um método público

Ainda de acordo com a figura, podemos dizer que o método *efetuarSaque* pode ser chamado por qualquer outro objeto.

- Privado: um atributo ou método privado pode ser acessado somente na própria classe em que está declarado. Na UML, indicamos que um atributo ou um método é privado utilizando o sinal – (negativo), conforme exemplo a seguir.

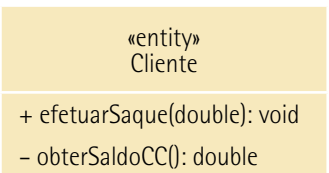

Figura 70 – Exemplo de representação de um método privado

Ainda a partir da figura, podemos dizer que o método *obterSaldoCC* pode ser chamado apenas dentro da própria classe Cliente. Por exemplo: poderíamos ter uma chamada a esse método dentro do método *efetuarSaque*.

Observação

Atributos e métodos marcados como privado são a chave para implementação de encapsulamento.

- Protegido: um atributo ou um método protegido pode ser acessado apenas na classe em que está declarado e em suas classes filhas. Na UML, indicamos que um atributo ou um método é protegido utilizando o sinal # (sustenido), conforme exemplo a seguir.

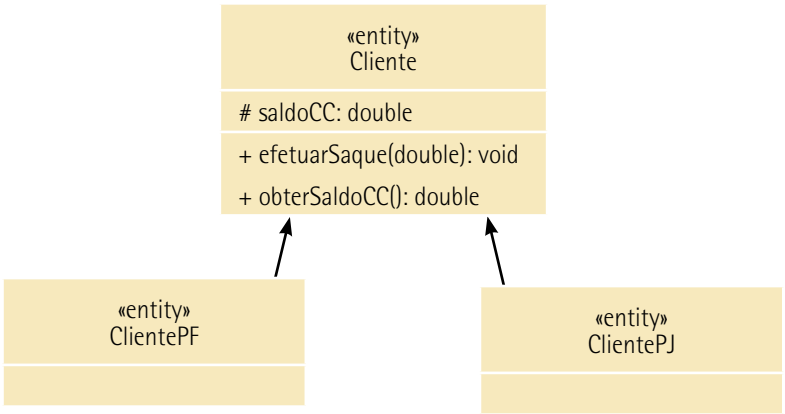

Figura 71 – Exemplo de representação de um atributo protegido

Podemos dizer, de acordo com a figura, que o atributo *saldoCC* é acessível apenas na classe *Cliente* e nas suas classes filhas: *ClientePF* e *ClientePJ*.

8.3.4 IInterfaces

Para definir o que é uma interface, em orientação a objetos, podemos fazer uma analogia com um contrato. Por exemplo, um contrato para a construção de uma casa entre duas partes, cliente e construtor, define o que cada um deve fazer, mas, em tese, não determina como. Um construtor deve entregar uma casa de acordo com as especificações entregues a ele. Em teoria, para o cliente, basta que ele entregue a casa dentro daquilo que se espera, não sendo relevante para a situação como ele irá executar o trabalho. Caso o cliente deseje trocar de construtor, basta entregar o mesmo contrato a outro construtor e esta relação seguirá a mesma lógica, ou seja, o construtor deverá entregar a casa ao cliente seguindo as especificações da casa, não sendo importante como ele irá construir, contanto que efetivamente construa.

Uma interface, em orientação a objetos, pode ser definida como um contrato que define quais métodos devem ser implementados por uma classe. Uma interface define o que deve ser implementado, mas não como deve ser implementado, define a assinatura de um método, mas não a forma de implementação, que fica a cargo da classe.

Resumindo, interface é um contrato que contém apenas assinaturas de métodos (não contém atributos) e que podem ser implementados por mais de uma classe.

Observação

É importante que não se confunda os conceitos de interface da orientação a objetos com interface do usuário ou interface para outros sistemas. Nesses casos, interface dá sentido à comunicação com algo que está fora da fronteira do sistema.

Na UML, representamos interfaces com o estereótipo *<<interface>>* e, como boa prática, o nome sempre começa com a letra I (interface), como mostra a figura a seguir.

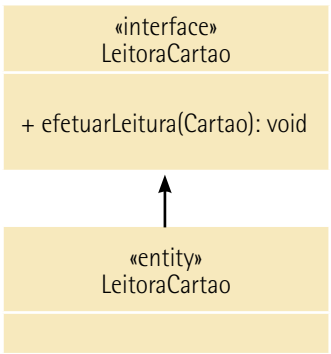

Figura 72 – Exemplo de representação de interface

Ainda podemos fazer algumas interpretações a partir da figura:

- A interface *ILeitoraCartao* possui um método, *efetuarLeitura*, que recebe como parâmetro um objeto do tipo *Cartao* e não retorna nenhum valor como saída. Esse método não possui implementação na interface, apenas a assinatura.

- A classe *LeitoraCartao* realiza, ou implementa, a interface *ILeitoraCartao*, como mostra a seta tracejada com ponta cheia, representação do termo "realização" na UML.

- A classe *LeitoraCartao* necessariamente terá um método *efetuarLeitura*, que recebe como parâmetro um objeto do tipo *Cartao* e não retorna nenhum valor como saída. Esse método necessariamente possui implementação na classe e deve obrigatoriamente seguir a assinatura proposta na interface.

8.3.5 Ciclo de vida de objetos

Dentro de um sistema de *software*, um objeto é criado, utilizado e destruído, obrigatoriamente nessa sequência. Isso é o que chamamos de ciclo de vida de objetos e que envolvem alguns conceitos.

Primeiro há a criação, ou a construção, de um objeto. Construir um objeto significa dar identidade a um objeto ou criar uma instância de uma classe. Instanciar uma classe envolve o conceito de construtor.

Construtor é um método da própria classe que tem como objetivo a criação da estrutura do objeto em memória.

Geralmente, nas linguagens de desenvolvimento orientado a objetos, não há a necessidade do desenvolvedor efetivamente implementar um construtor manualmente, basta saber alguns conceitos básicos:

- Um construtor é um método da própria classe.
- Um construtor é um método público.
- Um construtor não possui saída.
- Um construtor pode ou não receber parâmetros de entrada.

No diagrama de classes da UML, representamos construtores como representamos um método normal, a única diferença é que o construtor não possui saída, pode ou não receber parâmetros e o nome do método deve ser igual ao nome da classe, como mostra o exemplo a seguir.

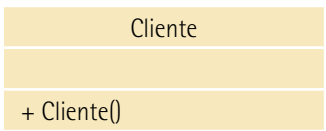

Figura 73 – Exemplo de representação de construtores UML

Se para construir um objeto temos o construtor, para destruir o objeto temos o destrutor, que tem como objetivo remover o objeto da memória. Assim como o construtor, o destrutor também é um método da própria classe.

Diferentemente do construtor, o destrutor obrigatoriamente não possui saída e não possui parâmetros de entrada.

Na UML, raramente necessitamos representar um destrutor, isso porque, na maioria das linguagens e plataformas de desenvolvimento orientadas a objeto, a destruição de objetos da memória fica a cargo da própria plataforma. Todavia, caso haja necessidade de representação, o destrutor é representado da mesma forma que o construtor, porém acrescido de um ~ (til), como mostra a figura a seguir.

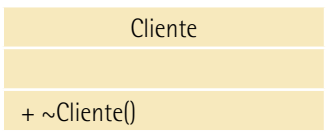

Figura 74 – Representação de destrutor na UML

Saiba mais

O mecanismo *garbage collector*, presente na plataforma Java e na plataforma Microsoft.Net, é responsável por remover os objetos da memória de um sistema de *software*.

Saiba mais em:

MICROSOFT DEVELOPER NETWORK. *Garbage collection*. 2014. Disponível em: <http://msdn.microsoft.com/en-us/library/0xy59wtx(v=vs.110).aspx>. Acesso em: 16 dez. 2014.

ORACLE. *Java garbage collection basics*. [s.d.]. Disponível em: <http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html>. Acesso em: 16 dez. 2014.

Após um objeto ser criado, ele é utilizado, ou seja, ele está apto a receber algum tipo de estímulo que o faça executar determinado comportamento. Um objeto só pode ser utilizado por outro objeto e essa "utilização" só é possível devido à existência, na orientação a objetos, de mecanismos de comunicação que possibilitam essa troca de informações entre objetos.

8.4 Comunicação entre objetos

Basicamente, a comunicação entre objetos se dá pela chamada de métodos. Para isso, é fundamental os conceitos de encapsulamento e visibilidade de métodos.

8.4.1 Mensagem

Para que um objeto execute um método, ou seja, tenha um determinado comportamento, é necessário enviar a este objeto uma mensagem solicitando a execução do método em questão.

Uma mensagem pode ser considerada um meio de comunicação entre objetos e tem como objetivo a ativação de um processamento.

Uma mensagem pode ter três significados (STADZISZ, 2002):

- Chamada de função ou procedimento: é o tipo mais utilizado de mensagem. Significa que um objeto está solicitando a execução de um método de outro objeto.

- Envio explícito de mensagem utilizando um tipo de serviço bem específico e especializado para envio e recebimento de mensagens, por exemplo: *Message Queue* (fila de mensagens).

- Evento: quando uma mensagem é enviada para um objeto do sistema originária de um evento externo ao sistema. É comum que esse tipo de mensagem seja próprio da comunicação entre atores e objetos. Por exemplo: um clique de *mouse* pode ser considerado um evento externo ao sistema e que deve ser interpretado por um objeto interno do mesmo sistema.

Existem alguns tipos de mensagens que acabam por definir padrões de comunicação entre objetos, para visualizar melhor cada tipo de mensagem, utilizaremos uma abordagem utilizando o Diagrama de Sequência da UML.

8.4.2 Diagrama de sequência

O diagrama de sequência da UML representa a interação de um conjunto de objetos, a troca de mensagens entre eles para resolver um problema específico. Uma característica positiva do diagrama de sequência é que podemos visualizar a troca de mensagens de forma sequencial e encaixada em uma linha de tempo.

Em um diagrama de sequência podemos representar todos os objetos e atores que fazem parte do cenário do problema que desejamos representar. Idealmente, representamos em um diagrama de sequência um cenário específico de um problema que queremos resolver, podemos fazer uma quebra por casos de uso ou por regras de negócio, a depender da complexidade de cada um, com o intuito de deixar o diagrama de sequência o mais inteligível possível.

A figura a seguir mostra os principais elementos de um diagrama de sequência.

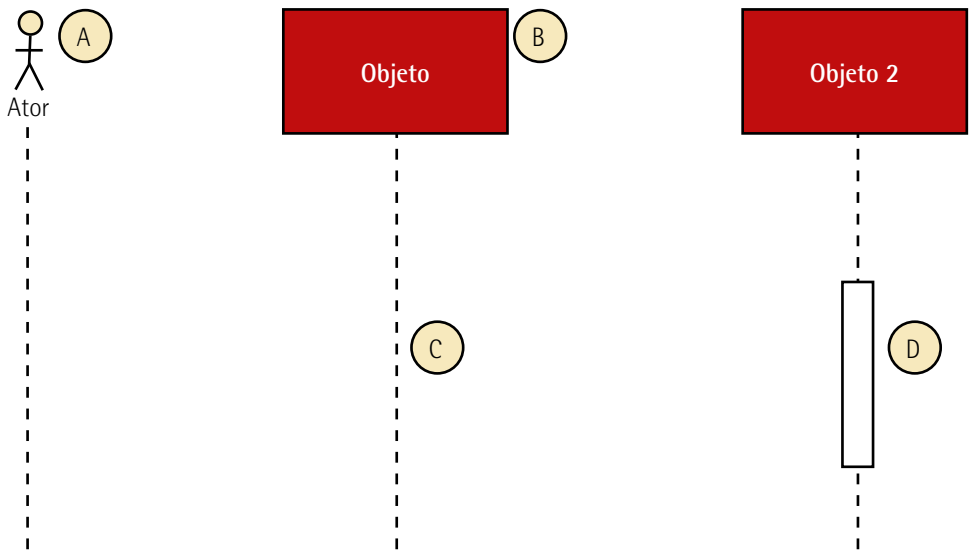

Figura 75 – Principais elementos do diagrama de sequência da UML

O quadro a seguir mostra os conceitos do diagrama de sequência representado pela figura anterior.

Quadro 22 – Conceitos e elementos do diagrama de sequência

Letra	Conceito
A	Representa o ator, envolvido no contexto. Como vimos, um ator pode enviar uma mensagem para um objeto. Essa mensagem tem o significado de evento.
B	O retângulo representa um objeto que faz parte da execução do cenário. Note que estamos falando de objetos e não de classes, pois quando estamos representando a troca de mensagens, no aspecto dinâmico do sistema, o objeto já está instanciado. Obrigatoriamente, os objetos do diagrama de sequência devem constar no modelo de classes.
C	A linha tracejada abaixo dos atores e dos objetos representa a linha de tempo
D	O retângulo posicionado abaixo do objeto significa que naquele espaço de tempo iniciou-se o ciclo de vida do objeto. Como vimos, um objeto é criado, utilizado e destruído, e nesse espaço de tempo representado pelo retângulo o objeto está ativo, ou seja, está criado, e está apto a receber mensagens.

A próxima figura mostra um exemplo de troca de mensagens, representado pelo diagrama de sequência, associado ao modelo de classes. Note que temos duas classes, Aluno e Professor, representadas no diagrama de classes. Essas classes originaram dois objetos no diagrama de sequencia: Pedro (Aluno) e João (Professor). A classe Aluno possui dois métodos públicos: *informarMatricula* e *receberNota*, que são representados no diagrama de sequência como mensagens enviadas de professor (João) para aluno (Pedro).

Note ainda que o nome da mensagem segue o mesmo padrão da declaração do método na classe envolvida, bem como os parâmetros ou argumentos e retornos dos métodos.

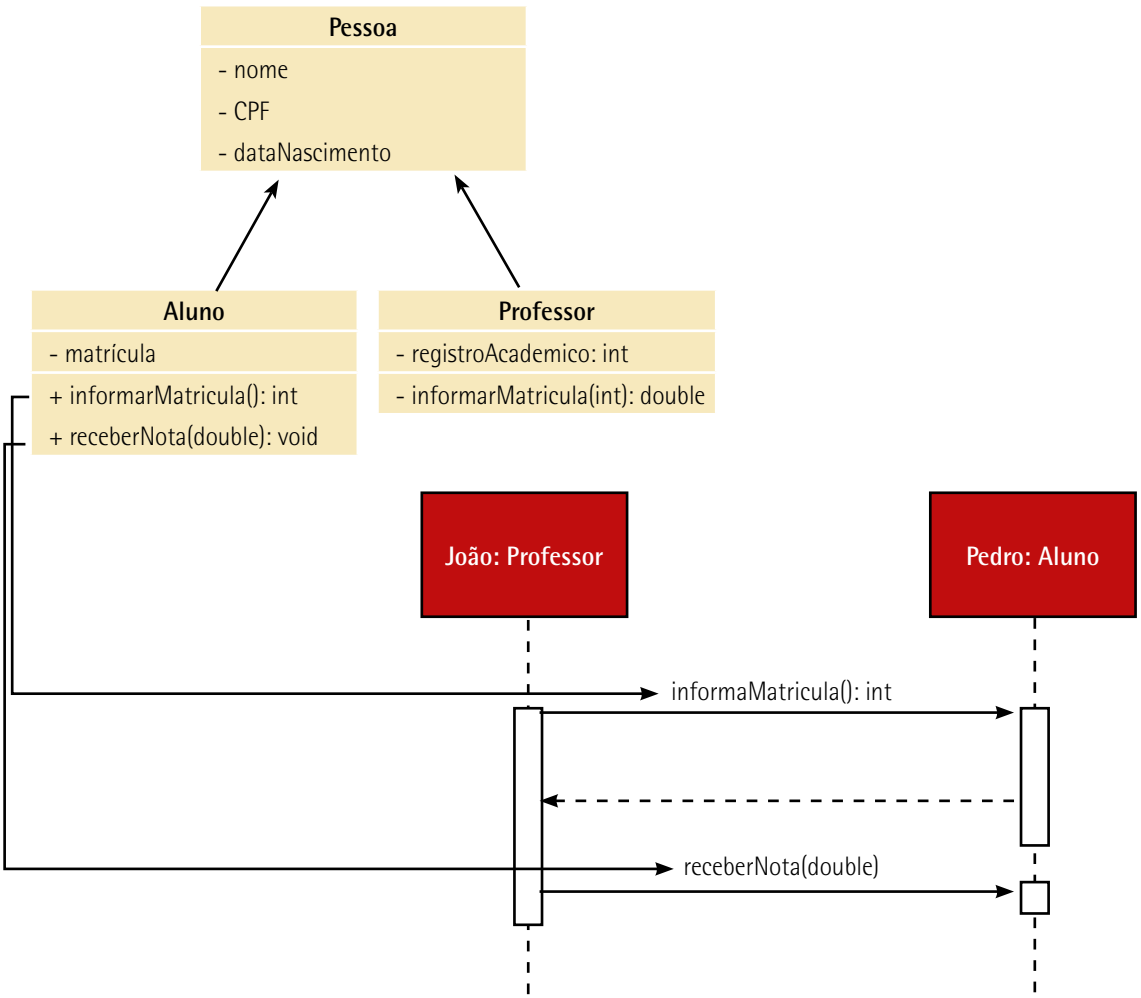

Figura 76 – Exemplo de troca de mensagens no diagrama de sequência

Existem dois tipos de mensagens (STADZISZ, 2002):

Mensagens síncronas

Criam uma dependência do estado do objeto que enviou uma mensagem com o estado do objeto que a recebe. Isso significa que o objeto que enviou a mensagem não tem seu estado alterado, não executa nenhuma ação, até que o objeto que recebeu a mensagem permita.

Fazendo uma analogia com funções algorítmicas, é como uma chamada de função com retorno, no qual o algoritmo que chamou a função não executa nenhuma ação até que a função finalize seu processamento e retorna um valor.

Em orientação a objetos, o exemplo mais comum de mensagens síncronas é a chamada de um método com retorno, como mostra o exemplo a seguir.

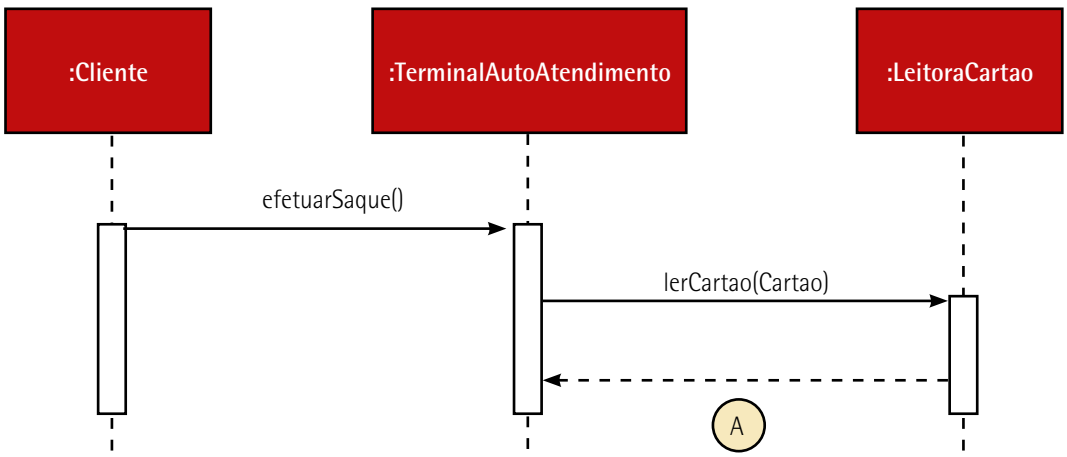

Figura 77 – Exemplo de mensagens síncronas

O ponto indicado com a letra A, a seta tracejada, indica o retorno da mensagem *lerCartao*. Se olharmos sob o ponto de vista do sincronismo, o objeto *TerminalAutoAtendimento* não pode executar nenhuma ação até que não haja o retorno da mensagem por parte do objeto *LeitoraCartao*.

Mensagens assíncronas

Ao contrário das mensagens síncronas, as assíncronas não criam uma dependência do estado do objeto que enviou a mensagem com o estado do objeto que a recebe. Ou seja, o objeto que enviou a mensagem pode executar qualquer ação independentemente da reação do objeto que recebeu a mensagem.

A figura a seguir mostra um exemplo de mensagem assíncrona no diagrama de sequência.

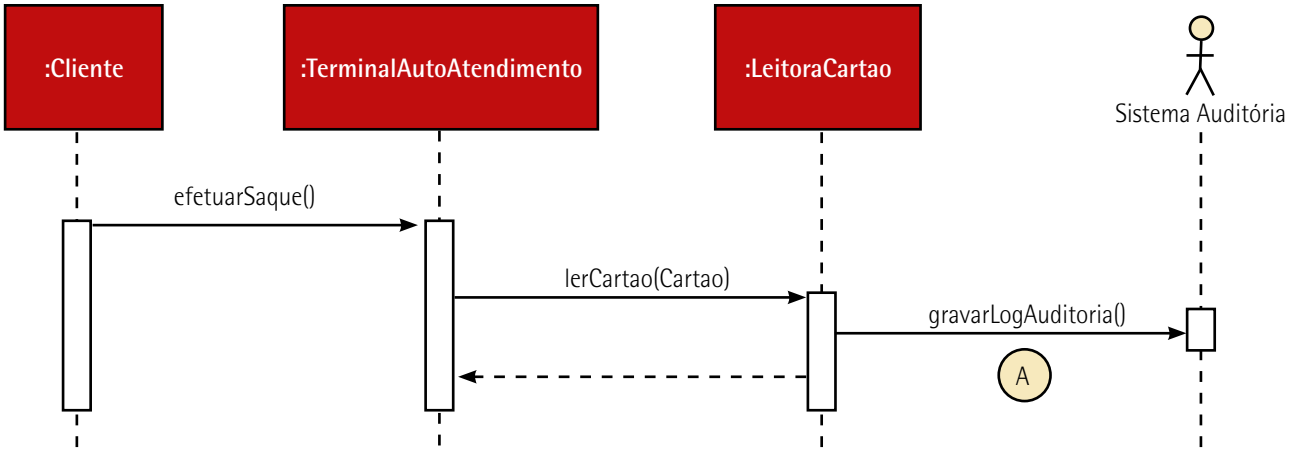

Figura 78 – Exemplo de mensagem assíncrona

O ponto indicado com a letra A, a seta contínua com flecha na ponta, indica que a mensagem *gravarLogAuditoria* é assíncrona. Se olharmos sob o ponto de vista do sincronismo, o objeto *LeitoraCartao* pode executar qualquer ação independentemente do ator externo *Sistema Auditoria*.

Além da notação de mensagens síncronas e assíncronas, temos algumas notações complementares (LARMAN, 2007):

Criação e destruição de objetos

Como vimos no ciclo de vida de um objeto, existem dois métodos, construtores e destrutores, que têm como objetivo criar e remover um objeto da estrutura de memória de um sistema.

A figura a seguir mostra a representação da criação e destruição de um objeto. No caso, temos o *Objeto A* criando o *Objeto B*, como indicado pela letra A, e temos o mesmo *Objeto A* destruindo o *Objeto B*, como indicado pela letra B. Note que a remoção, por completo, de um objeto da memória é representado por um X ao fim da sua linha de tempo, indicando que o ciclo de vida do objeto se encerrou.

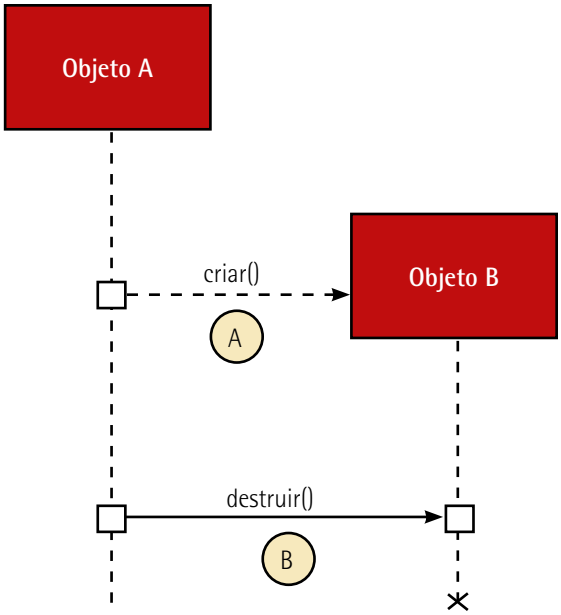

Figura 79 – Exemplo de representação de construção e destruição de objetos

Observação

Em orientação a objetos, quando um objeto cria outro para utilizá-lo ou para enviar-lhe uma mensagem, dizemos que o primeiro possui uma referência à instância do objeto criado.

Autodelegação de mensagens

Como vimos em visibilidade de métodos, existem métodos públicos e métodos privados. Métodos públicos são acessíveis por qualquer objeto no modelo, logo, são passíveis de envio de mensagens.

Um método privado de um objeto não pode ser acessado por outros objetos, apenas pelo objeto na qual está declarado, logo, esse método não é passível de receber mensagens de outros objetos.

Todavia, um objeto pode enviar uma mensagem para ele mesmo, solicitando a execução de um método privado. Esse movimento é chamado de autodelegação e pode ser visto, como exemplo, na figura a seguir, no ponto indicado com a letra A.

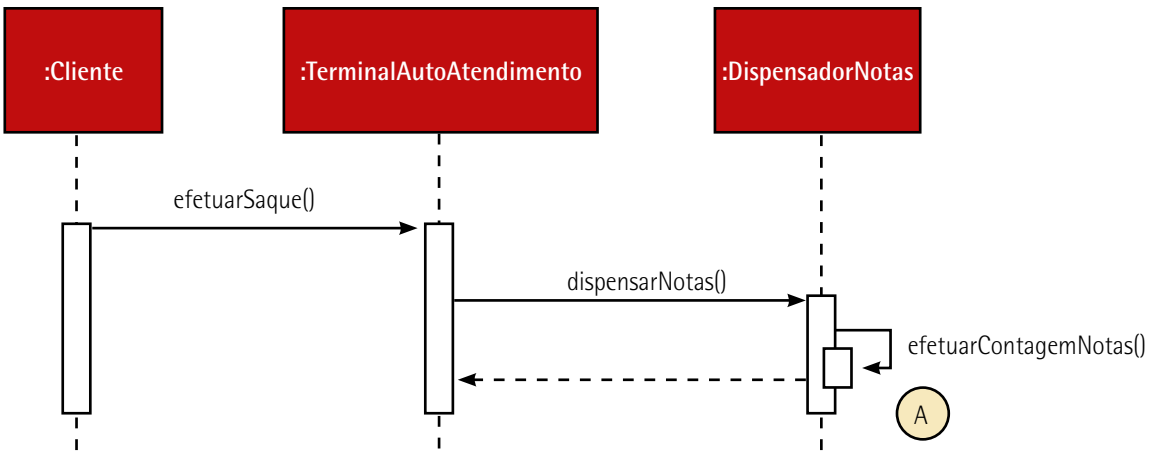

Figura 80 – Exemplo de autodelegação de mensagens

Notação de estruturas de decisão e de repetição

Assim como em algoritmos, temos estruturas de decisão e laços de repetição. A interpretação é a mesma que damos em lógica de programação, ou seja: em estruturas de decisão, uma mensagem só é enviada para um objeto se uma determinada condição for satisfeita e, em um laço de repetição, uma mensagem é enviada sucessivas vezes, tantas quantas forem especificadas no laço.

A figura a seguir mostra um exemplo de estrutura de decisão. No diagrama de sequência, temos o elemento operador de controle, que pode ser observado no ponto indicado com a letra A. O indicativo *alt* no operador de controle significa fluxo alternativo, ou seja, só será executado se uma condição for satisfeita, no caso, a senha informada deve ser correta.

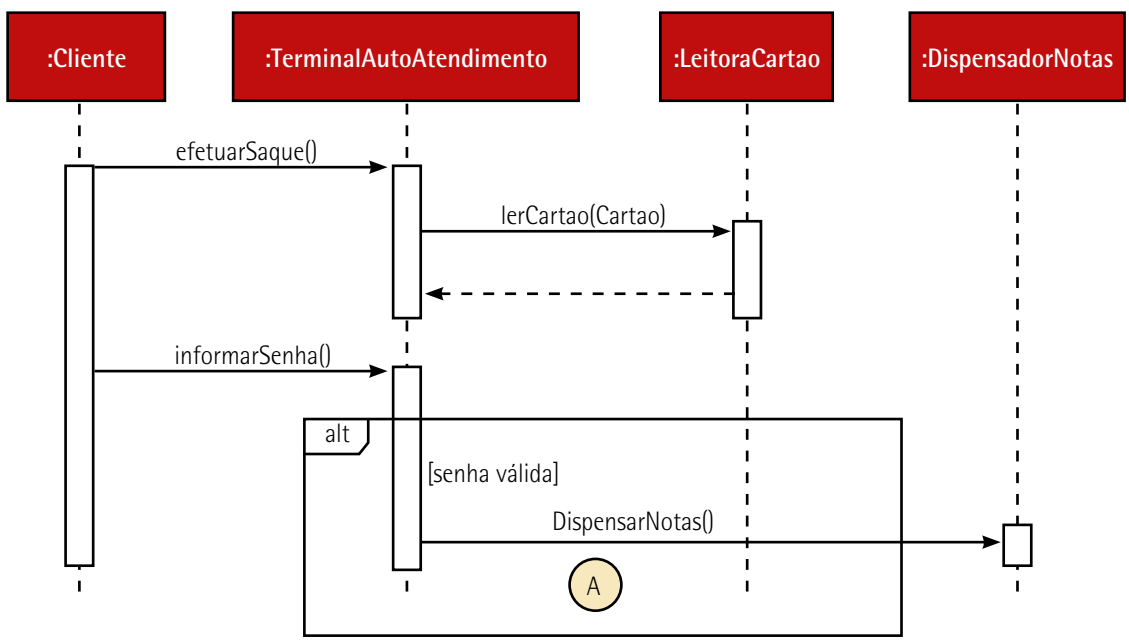

Figura 81 – Exemplo de fluxo alternativo em diagrama de sequência

A próxima figura mostra um exemplo de laço de repetição. O indicativo *loop* no operador de controle, indicado pela letra A, sinaliza uma repetição: no caso, a mensagem *informarSenha* será enviada no mínimo uma vez e no máximo três vezes, conforme indicado no operador de controle.

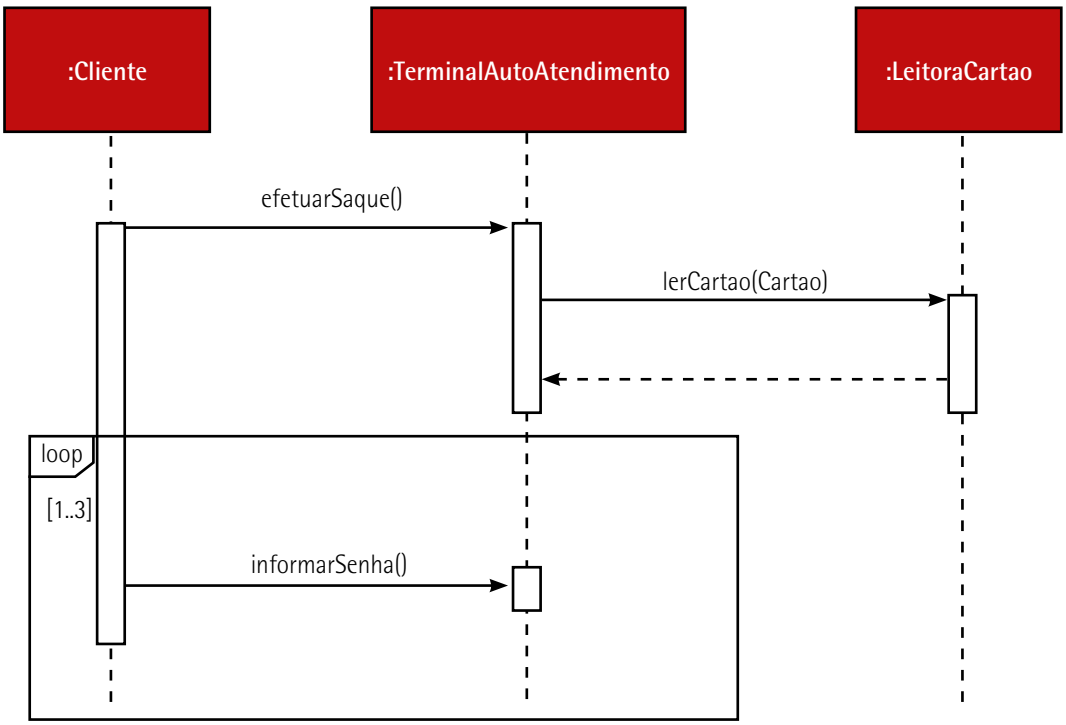

Figura 82 – Exemplo de estrutura de repetição em diagrama de sequência.

Estereótipos de classes de análise

Como vimos em classes de análise, as classes podem ser classificadas e organizadas de acordo com a responsabilidade que possuem dentro do modelo: entidade, controle e fronteira.

No diagrama de sequência podemos utilizar os estereótipos que auxiliam na interpretação das responsabilidades dos objetos, esses estereótipos podem ser vistos na figura a seguir.

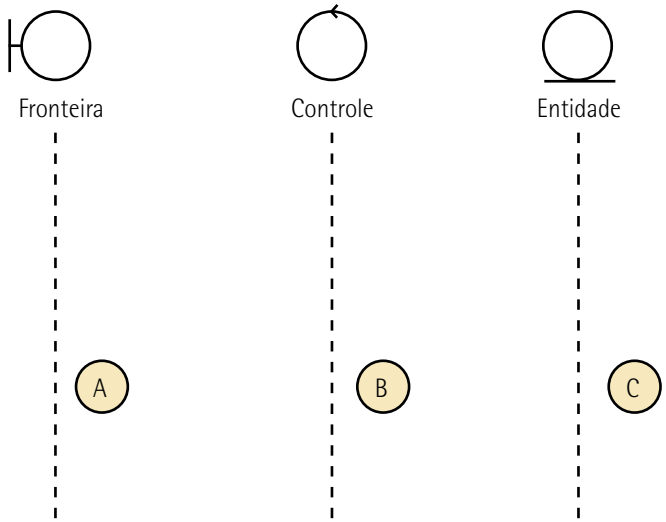

Figura 83 – Estereótipos de classes de análise

O quadro a seguir indica os estereótipos representados na figura.

Quadro 23 – Estereótipos no diagrama de sequência

Letra	Estereótipo
A	Classe de Fronteira
B	Classe de Controle
C	Entidade

Voltando para o exemplo do cenário de saque em terminal de autoatendimento (refinado para o diagrama de classes de análise que pode ser visto na figura 66), a figura a seguir mostra o diagrama de sequência com estereótipos que indicam as responsabilidades dos objetos.

Note que o cenário representado não leva em consideração fluxos alternativos e as regras de negócio previstas na fase de engenharia de requisitos.

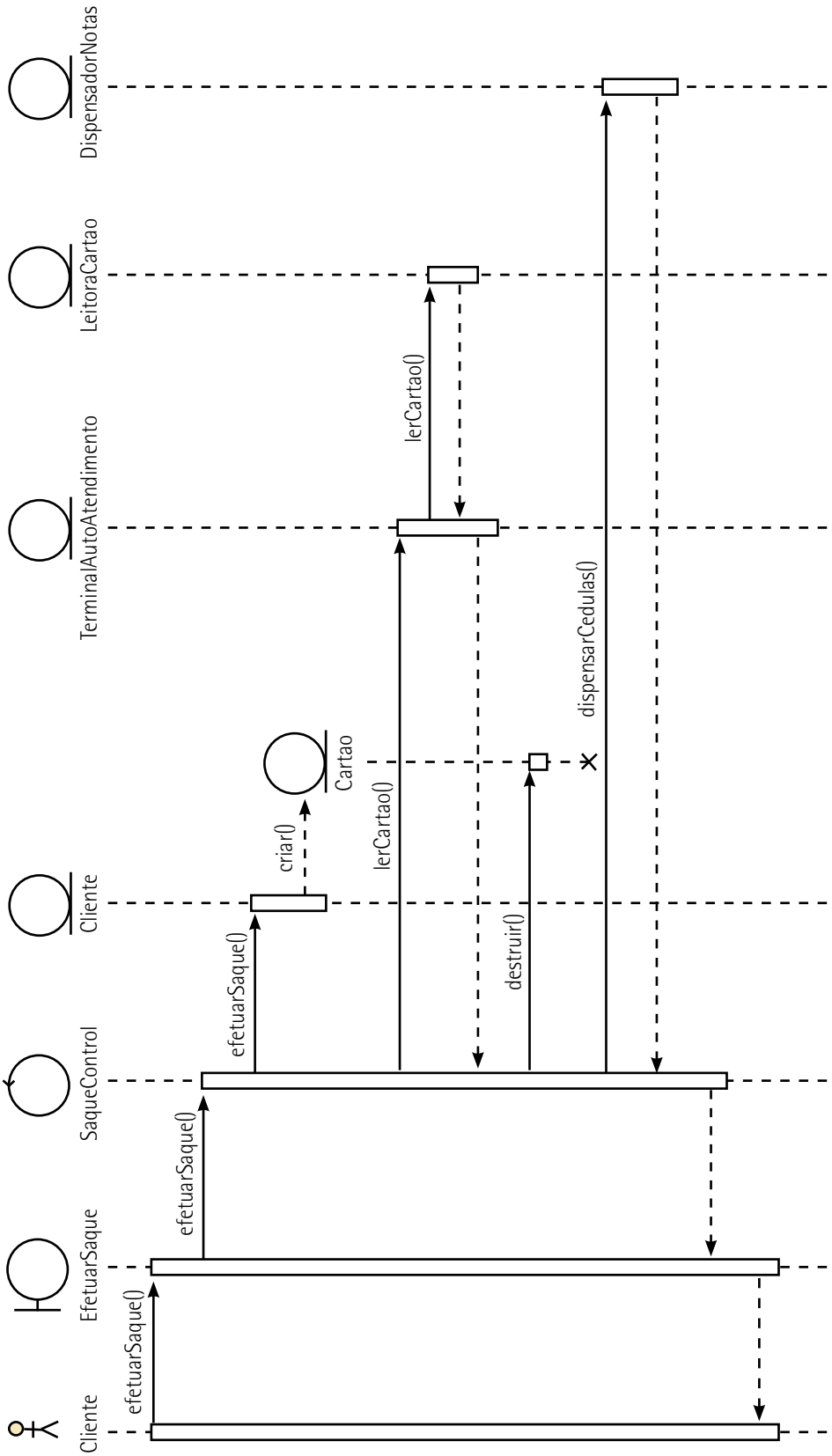

Figura 84 – Exemplo de diagrama de sequência com estereótipos

Saiba mais

Além do diagrama de sequência, o diagrama de colaboração também tem como objetivo representar a interação entre objetos. No diagrama de colaboração, a notação é mais complexa e não temos visualização clara da sequência de execução com relação ao tempo, como temos no diagrama de sequência. Para conhecer mais sobre o diagrama de colaboração em:

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

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.

Exemplo de aplicação

Estudo de caso proposto: desenvolva a análise orientada a objetos para o problema "Consultar Saldo em um Terminal de Autoatendimento".

Utilize um modelo de processo de desenvolvimento, atentando-se aos artefatos que devem ser produzidos, conforme debatemos ao longo do livro-texto.

Indique e justifique o modelo de processo utilizado, as características que o levou, enquanto analista de sistemas, a adotar determinado modelo em detrimento de outro.

Independentemente de qual for o modelo adotado, como resultado da análise deve-se produzir:

- o escopo dos requisitos do cenário proposto;
- classificação dos requisitos entre funcionais e não funcionais;
- descrição e documentação dos requisitos;
- modelagem e descrição dos casos de usos utilizando o diagrama de casos de uso e a descrição de casos de uso vistos na Unidade II;
- modelagem e especificação das regras e do processo de negócio conforme debatido na Unidade III;
- modelagem das visões estrutural estática e estrutural dinâmica, utilizando os conceitos da orientação a objetos e os diagramas de classe e sequência conforme visto na Unidade IV.

Dica: utilizar como base, ou guia de referência, tudo o que foi produzido neste livro-texto e que girou em torno do problema "Efetuar saque em terminal de autoatendimento".

Resumo

Conhecemos as definições nas diversas visões que os modelos de sistema proporcionam aos envolvidos em um projeto, com ênfase nas visões estruturais de um sistema de *software*.

Modelar a estrutura de um sistema é definir como as funcionalidades serão implementadas e definir os componentes desse sistema, suas responsabilidades e como eles irão interagir entre si para resolver um determinado problema.

Debatemos que os componentes desse sistema, dentro do paradigma da orientação a objetos, podem ser considerados objetos.

Um objeto é um elemento do mundo real, que possui relevância para a solução de um determinado problema e que possui atributos e métodos que são especificados em uma classe. Logo, podemos interpretar que um objeto é uma instância de uma classe.

Modelos de classes são a melhor forma de se representar a estrutura estática de um sistema orientado a objetos. Temos três tipos de modelos de classe:

- Modelo de classe de domínio: desenvolvido na fase de análise, o modelo de classe de domínio representa os objetos, ou classes, inerentes ao domínio do problema que queremos resolver, deixando de lado, nessa visão, detalhes tecnológicos da solução do problema.

- Modelo de classe de especificação: construído na fase de desenvolvimento, o modelo de classes de especificação adiciona ao modelo de classes de domínio objetos ou classes específicas para a solução do problema sob o aspecto tecnológico, ou seja, é uma extensão do modelo de classe de domínio.

- Modelo de classe de implementação: nada mais é que a implementação das classes, especificadas no modelo de especificação, construídas, ou codificadas, em alguma linguagem de desenvolvimento orientada a objetos, por exemplo, a linguagem Java ou a linguagem C#.

Qualquer que seja o modelo de classes, temos alguns conceitos fundamentais da orientação a objetos que devemos sempre ter em mente: abstração, encapsulamento, atributos e métodos.

Com relação a atributos e métodos, vimos as boas práticas na representação de atributos e métodos de uma classe.

Em um modelo de classes, objetos se relacionam entre si. São exemplos de relacionamento entre objetos: dependência, associação, agregação, composição e herança.

Utilizando os diagramas da UML, podemos representar um modelo de classes a partir dos diagramas de Classe e Objetos. No entanto, apenas os diagramas não representam um modelo, sendo necessária a descrição do significado das classes e seus respectivos atributos e métodos para que o modelo ganhe inteligibilidade.

A partir da visão estrutural estática, representada pelos modelos de classe, avançamos para a realização dos casos de uso, ou seja, como efetivamente os objetos se relacionam entre si de forma dinâmica, o modelo estrutural dinâmico.

O primeiro passo para se definir como uma tarefa será realizada é definir as responsabilidades de cada objeto. Debatemos os conceitos de classe de análise, que é a classificação das classes a partir de sua responsabilidade dentro do modelo. Classes podem assumir o papel de classes de fronteira, controle ou entidade.

Definidas as responsabilidades dos objetos, avançamos para a definição dos comportamentos de cada objeto, inicialmente explorando os padrões para representação de métodos, pedra fundamental no comportamento de objetos.

Foram explorados os conceitos de orientação a objetos ligados ao comportamento de objetos, como o polimorfismo, que é a capacidade que um objeto tem de implementar de maneira diferente um método herdado de uma classe mãe, sem que seja alterada a assinatura do método. Já o conceito das visibilidades de atributos e métodos é um grau de visibilidade em relação aos outros objetos do modelo. São eles: privado, público ou protegido.

Temos ainda a interface, que pode ser interpretada como um contrato que uma classe deve cumprir, uma especificação de quais métodos e suas assinaturas devem necessariamente ser implementados por uma ou mais classes.

Em um sistema de *software*, objetos são criados, utilizados e destruídos. Esse ciclo é chamado de ciclo de vida de objetos. Uma vez que o objeto seja criado, ele está apto para ser acionado.

A comunicação, ou o acionamento do comportamento de um objeto, se dá pelo envio e recebimento de mensagens por parte dos objetos envolvidos na resolução de um determinado problema.

Mensagens possuem diferentes significados, sendo o mais comum a chamada de um método de um objeto. Existem dois tipos de mensagem: síncronas e assíncronas, sendo que essa classificação se dá pela dependência dos estados do objeto que envia a mensagem para com o estado do objeto que recebe a mensagem.

O diagrama de sequência da UML é uma boa alternativa para a representação da troca de mensagens entre objetos, sempre em um cenário limitado pelo domínio do problema que se deseja resolver.

Exercícios

Questão 1. (Enade 2011) Analise as seguintes afirmativas sobre a Linguagem de Modelagem Unificada (UML):

I – A UML é uma metodologia para o desenvolvimento de *software* orientado a objetos, uma vez que fornece um conjunto de representações gráficas e sua semântica para a modelagem de *software*.

II – O diagrama de casos de uso procura, por meio de uma linguagem simples, demonstrar o comportamento externo do sistema. Esse diagrama apresenta o sistema sob a perspectiva do usuário e é, entre todos da UML, o mais abstrato, flexível e informal.

III – Um relacionamento de extensão de um caso de uso "A" para um caso de uso "B" significa que toda vez que "A" for executado ele incorporará o comportamento definido em "B".

IV – Os diagramas de comportamento da UML demonstram como ocorrem as trocas de mensagens entre os objetos do sistema para se atingir um determinado objetivo.

É correto apenas o que apenas se afirma em:

A) I e II.

B) II e IV.

C) III e IV.

D) I, II e III.

E) II, III e IV.

Resposta correta: alternativa B.

Análise da questão

A UML é, na verdade, uma linguagem de modelagem e não uma metodologia de desenvolvimento.

Já um relacionamento de extensão de um caso de uso "A" para um caso de uso "B" significa que **em algumas situações** em que "A" for executado ele incorporará o comportamento definido em "B".

Sendo assim, somente as afirmativas II e IV estão corretas.

Questão 2. (Enade 2011) O paradigma de programação orientada a objetos tem sido largamente utilizado no desenvolvimento de sistemas. Considerando o conceito de herança, avalie as afirmativas a seguir:

I – Herança é uma propriedade que facilita a implementação de reúso.

II – Quando uma subclasse é criada, ela herda todas as características da superclasse, não podendo possuir propriedades e métodos próprios.

III – Herança múltipla é uma propriedade na qual uma superclasse possui diversas subclasses.

IV – Extensão é uma das formas de se implementar herança.

É correto apenas o que se afirma em:

A) I.

B) III.

C) I e IV.

D) II e III.

E) II e IV.

Resolução desta questão na plataforma.

FIGURAS E ILUSTRAÇÕES

Figura 1

STALLINGS, W. *Arquitetura de Computadores*. São Paulo: Prentice Hall, 2003. p. 14.

Figura 3

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. 40. (Adaptado).

Figura 5

SOMMERVILLE, I. Engenharia de Software. São Paulo: Pearson, 2010. p. 45. (Adaptado).

Figura 6

BOEHM, B. W. *A spiral model of software development and enhancement computer.* California, v. 21, n. 5, maio 1988. p. 2.

Figura 10

THE STANDISH GROUP INTERNATIONAL. *The chaos report (1994)*. [S.l.]: The Standish Group, 1995. p. 4. Disponível em: <http://www.standishgroup.com/sample_research_files/chaos_report_1994.pdf>. Acesso em: 17 dez. 2014.

Figura 11

SOMMERVILLE, I. Engenharia de Software. São Paulo: Pearson, 2010. p. 88. (Adaptado).

Figura 12

INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. *25010*: Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models. Geneve: ISO, 2011. p. 14.

Figura 13

KOTONYA, G.; SOMMERVILLE, I. *Requirements engineering*: processes and techniques. Chichester: John Wiley, 2000. p. 28. (Adaptado).

Figura 14

SOMMERVILLE, I. Engenharia de Software. São Paulo: Pearson, 2010. p. 101. (Adaptado).

Figura 15

SOMMERVILLE, I. Engenharia de Software. São Paulo: Pearson, 2010. p. 109. (Adaptado).

Figura 16

BOOCH, G.; JACOBSON, I.; RUMBAUGH, J. *UML*: guia do usuário. 2. ed. Rio de Janeiro: Campus, 2006. p. 142. (Adaptado).

Figura 18

BOOCH, G.; JACOBSON, I.; RUMBAUGH, J. *UML*: guia do usuário. 2. ed. Rio de Janeiro: Campus, 2006. p. 236. (Adaptado).

Figura 20

BOOCH, G.; JACOBSON, I.; RUMBAUGH, J. *UML*: guia do usuário. 2. ed. Rio de Janeiro: Campus, 2006. p. 236. (Adaptado).

Figura 22

BOOCH, G.; JACOBSON, I.; RUMBAUGH, J. *UML*: guia do usuário. 2. ed. Rio de Janeiro: Campus, 2006. p. 236. (Adaptado).

Figura 24

BOOCH, G.; JACOBSON, I.; RUMBAUGH, J. *UML*: guia do usuário. 2. ed. Rio de Janeiro: Campus, 2006. p. 246. (Adaptado).

Figura 26

ZACHMAN, J. A. Framework for information systems architecture. *IBM Systems Journal*, Los Angeles, California, v. 26, n. 3, p. 276-292, 1987. Disponível em: <http://www.zachman.com/images/ZI_PIcs/ibmsj2603e.pdf>. Acesso em: 3 nov. 2014. p. 292. (Adaptado).

Figura 28

ERIKSSON, H. E.; PENKER, M. *Business modeling with UML*: business patterns at work. New York: John Wiley & Sons, 2000. p. 55.

Figura 29

ERIKSSON, H. E.; PENKER, M. *Business modeling with UML*: business patterns at work. New York: John Wiley & Sons, 2000. p. 60.

Figura 30

ERIKSSON, H. E.; PENKER, M. *Business modeling with UML*: business patterns at work. New York: John Wiley & Sons, 2000. p. 60.

Figura 31

ERIKSSON, H. E.; PENKER, M. *Business modeling with UML*: business patterns at work. New York: John Wiley & Sons, 2000. p. 63.

Figura 33

BOOCH, G.; JACOBSON, I.; RUMBAUGH, J. *UML*: guia do usuário. 2. ed. Rio de Janeiro: Campus, 2006. p. 302.

Figura 38

ERIKSSON, H. E.; PENKER, M. *Business modeling with UML*. New York: John Wiley & Sons, 2000. p. 24. (Adaptado).

Figura 41

KRUCHTEN, P. The 4+1 View Model of Architecture. *IEEE Software*, Washington, v. 12, n. 6, nov. 1995. p. 2.

Figura 42

BOOCH, G.; JACOBSON, I.; RUMBAUGH, J. *UML*: guia do usuário. 2. ed. Rio de Janeiro: Campus, 2006. p. 31.

Figura 43

BOOCH, G.; JACOBSON, I.; RUMBAUGH, J. *UML*: guia do usuário. 2. ed. Rio de Janeiro: Campus, 2006. p. 31.

Figura 44

BOOCH, G.; JACOBSON, I.; RUMBAUGH, J. *UML*: guia do usuário. 2. ed. Rio de Janeiro: Campus, 2006. p. 31.

Figura 46

BOOCH, G.; JACOBSON, I.; RUMBAUGH, J. *UML*: guia do usuário. 2. ed. Rio de Janeiro: Campus, 2006. p. 66.

Figura 48

BOOCH, G.; JACOBSON, I.; RUMBAUGH, J. *UML*: guia do usuário. 2. ed. Rio de Janeiro: Campus, 2006. p. 67.

Figura 49

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. 99.

Figura 51

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. 108.

Figura 53

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. 104.

Figura 55

BOOCH, G.; JACOBSON, I.; RUMBAUGH, J. *UML*: guia do usuário. 2. ed. Rio de Janeiro: Campus, 2006. p. 150.

Figura 57

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. 105.

Figura 59

BOOCH, G.; JACOBSON, I.; RUMBAUGH, J. *UML*: guia do usuário. 2. ed. Rio de Janeiro: Campus, 2006. p. 65.

Figura 60

BOOCH, G.; JACOBSON, I.; RUMBAUGH, J. *UML*: guia do usuário. 2. ed. Rio de Janeiro: Campus, 2006. p. 145.

Figura 69

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.

Figura 70

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.

Figura 71

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.

Figura 72

BOOCH, G.; JACOBSON, I.; RUMBAUGH, J. *UML*: guia do usuário. 2. ed. Rio de Janeiro: Campus, 2006. p. 161.

Figura 73

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. 175.

Figura 74

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. 175.

Figura 75

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. 148.

Figura 77

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. 151.

Figura 78

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. 151.

Figura 79

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. 152.

Figura 80

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. 151.

Figura 81

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

Figura 82

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

Figura 83

WTHREEX. *Conceito-chave*: papel. [s.d.]. Disponível em: <http://www.wthreex.com/rup/portugues/manuals/intro/kc_role.htm>. Acesso em: 18 dez. 2014.

REFERÊNCIAS

Textuais

BASS, L.; CLEMENTS, P.; KAZMAN, R. *Software architecture in practice*. Boston: Addison-Wesley, 2003.

BELLOQUIM, A. Analistas de processos de negócio: 5 competências fundamentais. *Gnosis IT Knowledge Solutions*, São Paulo, 2007. Disponível em: <http://gnosisbr.com.br/analistas-de-processos-de-negocios-5-competencias-fundamentais>. Acesso em: 3 nov. 2014.

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.

BOEHM, B.; BROWN, J.R.; LIPOW, M. Quantitative Evaluation of Software Quality. *In: Proedures. of 2nd International Conference on Soft. Eng. San Francisco*. 1976. p. 592-605.

BOEHM, B. W. A spiral model of software development and enhancement. *Computer*, California, v..21, n. 5, p. 61-72, maio 1988.

BOEHM, B.; HANSEN, W. J. *Spiral development*: experience, principles, and refinements. Pittsburg: CMU/SEI, 2000. Disponível em: <http://csse.usc.edu/csse/TECHRPTS/2000/usccse2000-525/usccse2000-525.pdf>. Acesso em: 18 dez. 2014.

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

BOURQUE, P.; FAIRLEY, R. E. (Eds.). *Guide to the Software Engineering Body of Knowledge, Version 3.0*. Washington: IEEE Computer Society. Disponível em: <http://www.swebok.org/>. Acesso em: 30 out. 2014.

BUSINESS RULES GROUP. *Defining business rules*: what are they really? [S.l.], 2000. Disponível em: <http://www.businessrulesgroup.org/first_paper/BRG-whatisBR_3ed.pdf>. Acesso em: 3 nov. 2014.

CARDOSO, E.; ALMEIDA, J. P. A.; GUIZZARDI, G. Uma experiência com engenharia de requisitos baseada em modelos de processos. *In:* WORKSHOP IBEROAMERICANO DE ENGENHARIA DE REQUISITOS E AMBIENTES DE SOFTWARE, 11., 2008, Recife. *Artigos publicados no CIbSE: IDEAS08*, 2008. Disponível em: <http://cibse.inf.puc-rio.br/CIBSEPapers/artigos/artigos_IDEAS08/27_paper_67_EvellinCardoso-fmra.pdf>. Acesso em: 17 dez. 2014.

CARROL, J. M. *Scenario-based design:* envisioning work and technology in system development. New York: John Wiley, 1995.

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

COCKBURN, A.. *Escrevendo casos de uso eficazes*: um guia prático para desenvolvedores de software. Porto Alegre: Bookman, 2005.

CORTÊS, M. L.; CHIOSSI, T. C. dos 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:* BULLINGER, H-J.; SHACKEL, B. (eds.). *INTERACT 87* – 2nd IFIP International Conference on Human-Computer Interaction. Estugarda: Elsevier Science Publishers, 1987. p. 431-436. Disponível em: <http://iihm.imag.fr/publs/1987/Interact87.PAC.pdf>. Acesso em: 15 dez. 2014.

ERIKSSON, H. E.; PENKER, M.. *Business modeling with UML*: business patterns at work. New York: John Wiley & Sons, 2000.

FOWLER, M. *UML essencial*: um breve guia para a linguagem-padrão de modelagem de objetos. Porto Alegre: Bookman, 2000.

___. The unpredictability of requirements. *In: The new methodology*. 2005. Disponível em: <http://www.martinfowler.com/articles/newMethodology.html#TheUnpredictabilityOfRequirements>. Acesso em: 30 out. 2014.

FURLAN, J. D.. Modelagem de objetos através da UML: the Unified Modeling Language. São Paulo: Makron Books, 1998.

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

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, 27., maio 2005, St. Louis. *Proceedings*. St. Louis: ACM/IEEE, 2005 p. 460-469.

GRANDI, M. A. de. *Uma abordagem de identificação e modelagem de regras de negócio e seus relacionamentos transversais*. 2008. 120 f. Dissertação (Mestrado em Ciência da Computação)–Centro Universitário Eurípedes de Marília, 2008.

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

HICKEY, A. M.; DAVIS, A. M. Elicitation technique selection: how do experts do it? *In:* IEEE INTERNATIONAL REQUIREMENTS ENGINEERING CONFERENCE, 11., 2003, Monterey Bay. *Proceedings of...* Monterey Bay: IEEE, 2003. p. 169-178.

___. The role of requirements elicitation techniques in achieving software quality. *In:* INTERNATIONAL WORKSHOP ON REQUIREMENTS ENGINEERING: FOUNDATION FOR SOFTWARE QUALITY, 8., 2002, Essen (Alemanha). *Proceedings of...* Essen: CRI – Centre de Reserche em Informatique, 2002.

INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS. *12207:* standard for information technology – software life cycle processes. New York: IEEE, 2008.

INTERNATIONAL INSTITUTE OF BUSINESS ANALISYS. *O guia para o corpo de conhecimento de análise de negócios (Guia BABOK)*. Versão 2.0. São Paulo, SP, 2011.

INTERNATIONAL ORGANIZATION FOR STANDARDIZATION/INTERNATIONAL ELETROTECNINICAL COMISSION. *14102*: information technology – guideline for the evaluation and selection of case tools. Geneve: ISO, 2008.

INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. *25010*: Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models.. Geneve: ISO, 2011.

ISD BRASIL. *Perguntas frequentes:* o que é CMMI-DEV? Disponível em: <http://www.isdbrasil.com.br/o-que-e-cmmi-dev.php>. Acesso em: 18 dez. 2014.

JACOBSON, I.; RUMBAUGH, J.; BOOCH, G. *The unified software development process*. Boston: Addison-wesley, 1999.

KOTONYA, G.; SOMMERVILLE, I. *Requirements engineering*: processes and techniques. Chichester: John Wiley, 1998.

KRUCHTEN, P. The 4+1 View Model of Architecture. *IEEE Software*, Washington, v. 12, n. 6, p. 42-50, nov. 1995

___. *The rational unified process: an introduction*. Boston: Addison-wesley, 2003.

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.

MAC KNIGHT, D. *Elicitação de requisitos de software a partir do modelo de negócio*. 2004. 210 f. Dissertação (Mestrado em Informática) – Universidade Federal do Rio de Janeiro, Rio de Janeiro, 2004.

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

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

MICHAELIS: moderno dicionário da língua portuguesa. São Paulo: Companhia Melhoramentos, 1998.

MICROSOFT DEVELOPER NETWORK. *Garbage collection*. 2014. Disponível em: <http://msdn.microsoft.com/en-us/library/0xy59wtx(v=vs.110).aspx>. Acesso em: 16 dez. 2014.

MOORE G. E. Cramming more components onto integrated circuits, electronics. *Electronics Magazine*, EUA, v. 38, n. 8, p. 114-117, abr. 1965.

ORACLE. *Java garbage collection basics*. [s.d.]. Disponível em: <http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html>. Acesso em: 16 dez. 2014.

PAETSCH, F.; EBERLEIN, A.; MAURER, F. Requirements Engineering and Agile Software Development. *In:* IEEE INTERNATIONAL WORKSHOP ON ENABLING TECHNOLOGIES: INFRASTRUCTURE FO COLLABORATIVE ENTERPRISES, 12., 2003, Linz. *Proceedings*... Linz: IEEE, 2003. p. 308-313, 2003.

PALMER J. D. Traceability. *In:* THAYER, R. H.; DORFMAN, R. *Software Requirements Engineering* [Eds.]. Michigan: IEEE Computer Society Press, 1997.

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.

REZENDE, A. D. Engenharia de Software *e sistemas de informação*. 3. ed. Rio de Janeiro: Brasport, 2005.

ROCHA, H. V.; BARANAUSKAS, M. C. C. *Design e avaliação de interfaces humano-computador*. Campinas: Unicamp, 2003.

ROCHA, A. R. C.; MALDONADO, J. C.; WEBER, K. C. *Qualidade de software: teoria e prática. São Paulo: Prentice Hall, 2001*.

ROSEMBERG, C. *et al*. Prototipação de software e design participativo: uma experiência do atlântico. *In:* SIMPÓSIO BRASILEIRO DE FATORES HUMANOS EM SISTEMAS COMPUTACIONAIS, 8.,2008, Porto Alegre. *Proceedings of the VIII Brazilian Symposium on Human Factors in Computing Systems*. Porto Alegre: Sociedade Brasileira de Computação, 2008.

ROSS, R. G. *Principles of the business rule approach*. Boston: Addison-wesley, 2003.

SILVA, L. N. de C. *Engenharia imunológica*: desenvolvimento e aplicação de ferramentas computacionais inspiradas em sistemas imunológicos artificiais. 2001. 277 f. Tese (Doutorado) do Curso de Engenharia Elétrica, Universidade Estadual de Campinas (Unicamp). Campinas, 2001.

SOMMERVILLE, I. Engenharia de Software. São Paulo: Pearson, 2010.

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

STALLINGS, W. *Arquitetura e Organização de Computadores*. São Paulo: Prentice Hall, 2003.

TETILA, E. C. *Processos de estimativa de software com a métrica use case points, PMBOK e RUP*. 2007. 140 f. Dissertação (Mestrado em Engenharia de Produção) da Universidade Paulista (UNIP), 2007. Disponível em: <http://www3.unip.br/ensino/pos_graduacao/strictosensu/eng_producao/download/eng_evertoncastelaotetila.swf>. Acesso em: 3 nov. 2014.

THE STANDISH GROUP INTERNATIONAL. *The chaos report (1994)*. [S.l.]: The Standish Group, 1995. Disponível em: <http://www.standishgroup.com/sample_research_files/chaos_report_1994.pdf>. Acesso em: 17 dez. 2014.

WAZLAWICK, R. S. *Análise e projeto de sistemas de informação orientados a objetos*. Rio de Janeiro: Elsevier, 2011.

WIEGERS, K. E. *Software requirements:* second edition. Washington: Microsoft Press, 2003.

WIRFS-BROCK, R.; WILKERSON, B., Object-Oriented Design: A Responsibility-Driven Approach. *In:* OBJECT-ORIENTED PROGRAMMING, SYSTEMS, LANGUAGES, AND APPLICATIONS CONFERENCE (OOPSLA 89), 4., 1989 Nova Orleans. *Proceedings of... Nova Orleans: ACM, 1989.*

WTHREEX. *Conceito-chave*: papel. [s.d.]. Disponível em: <http://www.wthreex.com/rup/portugues/manuals/intro/kc_role.htm>. Acesso em: 18 dez. 2014.

WTHREEX. *Diretrizes*: classe de análise. [s.d.]. Disponível em: <http://www.wthreex.com/rup/portugues/process/modguide/md_acls2.htm#Analysis Class Stereotypes>. Acesso em: 02 jan. 2015.

ZACHMAN, J. A. A framework for information systems architecture. *IBM Systems Journal*, v. 26, n. 3, p. 276-292, 1987.

Sites

<http://agilemanifesto.org/iso/ptbr/>.

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

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

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

<http://www.omg.org/spec/index.htm>

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

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

<http://www.softex.br/mpsbr/guias/>.

Exercícios

Unidade I – Questão 1: INSTITUTO NACIONAL DE ESTUDOS E PESQUISAS EDUCACIONAIS ANÍSIO TEIXEIRA (INEP). *Exame Nacional de Desempenho dos Estudantes (Enade) 2008*: Computação. Questão 16. Disponível em: <http://download.inep.gov.br/download/Enade2008_RNP/COMPUTACAO.pdf>. Acesso em: 10 out. 2018.

Unidade II – Questão 1: INSTITUTO NACIONAL DE ESTUDOS E PESQUISAS EDUCACIONAIS ANÍSIO TEIXEIRA (INEP). *Exame Nacional de Desempenho dos Estudantes (Enade) 2008*: Computação. Questão 69. Disponível em: <http://download.inep.gov.br/download/Enade2008_RNP/COMPUTACAO.pdf>. Acesso em: 10 out. 2018.

Unidade II – Questão 2: INSTITUTO NACIONAL DE ESTUDOS E PESQUISAS EDUCACIONAIS ANÍSIO TEIXEIRA (INEP). *Exame Nacional de Desempenho dos Estudantes (Enade) 2011*: Computação. Questão 21. Disponível em: <http://download.inep.gov.br/educacao_superior/enade/provas/2011/COMPUTACAO.pdf>. Acesso em: 10 out. 2018.

Unidade III – Questão 1: FUNDAÇÃO CESGRANRIO. *Petrobras 2008*: Analista de Sistemas Junior – Processos de Negócio. Questão 65. Disponível em: <https://arquivo.pciconcursos.com.br/provas/11093956/3c7847942b99/prova31.pdf>. Acesso em: 10 out. 2018.

Unidade III – Questão 2: INSTITUTO NACIONAL DE ESTUDOS E PESQUISAS EDUCACIONAIS ANÍSIO TEIXEIRA (INEP). *Exame Nacional de Desempenho dos Estudantes (Enade) 2008*: Computação. Questão 64. Disponível em: <http://download.inep.gov.br/download/Enade2008_RNP/COMPUTACAO.pdf>. Acesso em: 10 out. 2018.

Unidade IV – Questão 1: INSTITUTO NACIONAL DE ESTUDOS E PESQUISAS EDUCACIONAIS ANÍSIO TEIXEIRA (INEP). *Exame Nacional de Desempenho dos Estudantes (Enade) 2011*: Tecnologia em Análise e Desenvolvimento de Sistemas. Questão 12. Disponível em: <http://download.inep.gov.br/educacao_superior/enade/provas/2011/ANALISE_E_DESENVOLVIMENTO_DE_SISTEMAS.pdf>. Acesso em: 10 out. 2018.

Unidade IV – Questão 2: INSTITUTO NACIONAL DE ESTUDOS E PESQUISAS EDUCACIONAIS ANÍSIO TEIXEIRA (INEP). *Exame Nacional de Desempenho dos Estudantes (Enade) 2011*: Tecnologia em Análise e Desenvolvimento de Sistemas. Questão 21. Disponível em: <http://download.inep.gov.br/educacao_superior/enade/provas/2011/ANALISE_E_DESENVOLVIMENTO_DE_SISTEMAS.pdf>. Acesso em: 10 out. 2018.

Interativa

Informações:
www.sepi.unip.br ou 0800 010 9000