UML 2

UMA ABORDAGEM PRÁTICA

Gilleanes T. A. Guedes

Novatec

Editor: Rubens Prates
Revisão de texto: Lia Gabriele Regius
Editoração eletrônica: Camila Kuwabata e Carolina Kuwabata
Capa: Victor Bittow

ISBN: 978-85-7522-281-2

Esta obra foi revisada, atualizada e ampliada tomando como base o livro:

"UML – Uma Abordagem Prática", ISBN 978-7522-149-5

Histórico de impressões:

Junho/2011 Segunda edição
Maio/2009 Primeira edição (ISBN: 978-85-7522-193-8)

Novatec Editora Ltda.
Rua Luís Antônio dos Santos 110
02460-000 – São Paulo, SP – Brasil
Tel.: +55 11 2959-6529
Fax: +55 11 2950-8869
E-mail: novatec@novatec.com.br
Site: www.novatec.com.br
Twitter: twitter.com/novateceditora
Facebook: facebook.com/novatec
LinkedIn: linkedin.com/in/novatec

Dados Internacionais de Catalogação na Publicação (CIP)
(Câmara Brasileira do Livro, SP, Brasil)

Guedes, Gilleanes T. A.
UML 2 : uma abordagem prática / Gilleanes T. A. Guedes. -- 2. ed. -- São Paulo : Novatec Editora, 2011.

Bibliografia.
ISBN 978-85-7522-281-2

1. Componentes - Programas de computador 2. Programação orientada para o objeto (Ciência da computação) 3. Software - Desenvolvimento 4. UML (Ciência da computação) I. Título.

07-9482 CDD-005.3

Índices para catálogo sistemático:

1. Linguagem de modelagem unificada : Ciência da computação 005.3
2. UML : Unified Modeling Language : Programas : Ciência da computação 005.3

CRS20110602

Agradecimentos

Agradeço primeiramente ao professor Carlos Emilio Padilla Severo, pelas muitas ideias e esclarecimentos que foram produzidos durante nossas conversas.

Agradeço também aos meus alunos das disciplinas de Engenharia de Software e afins do curso de Licenciatura Plena em Informática da Universidade Federal de Mato Grosso, UFMT, Campus de Rondonópolis; aos meus alunos do curso de Sistemas de Informação do CESUR/FACSUL, Faculdade do Sul de Mato Grosso, no qual lecionei antes de ingressar na UFMT, e aos meus alunos do curso de pós-graduação em Processo e Desenvolvimento de Software da Faculdade Exponencial, FIE, de Chapecó, em Santa Catarina.

Sobre o Autor

Gilleanes Thorwald Araujo Guedes é mestre em Ciência da Computação pela Universidade Federal do Rio Grande do Sul (UFRGS) e bacharel em Informática pela Universidade da Região da Campanha (URCAMP). É professor de Engenharia de Software no curso de Licenciatura Plena em Informática na Universidade Federal de Mato Grosso (UFMT), estando atualmente cursando o doutorado em Ciência da Computação na Universidade Federal do Rio Grande do Sul. Já ministrou diversas palestras e cursos sobre UML em eventos científicos, cursos técnicos e cursos de pós-graduação "lato sensu". É autor dos livros UML – Uma abordagem prática, UML 2 – Guia de consulta rápida e UML 2 – Guia prático, publicados pela Novatec Editora. Pode ser contatado pelo e-mail `gtag@novatec.com.br`.

Prefácio

A UML (Unified Modeling Language – Linguagem de Modelagem Unificada) tornou-se, nos últimos anos, a linguagem-padrão de modelagem adotada internacionalmente pela indústria de engenharia de software. Em decorrência disso, existe hoje uma grande valorização e procura no mercado por profissionais que dominem essa linguagem.

O objetivo deste livro é ensinar ao leitor como modelar software por meio dos diversos diagramas que compõem a UML. No entanto, é importante destacar que a UML é uma linguagem de modelagem totalmente independente, não estando vinculada a nenhum processo de desenvolvimento específico e menos ainda a qualquer linguagem de programação.

Apesar de a UML oferecer um grande número de diagramas que enfoquem tanto características estruturais quanto comportamentais de um software, o leitor não deve se sentir obrigado a utilizar todos os diagramas propostos na modelagem de seus sistemas, pois cada um deles tem uma função específica e, algumas vezes, alguns deles não são necessários em determinadas situações ou domínios.

Esta obra foi revisada, atualizada e ampliada tomando como base o livro UML – Uma Abordagem Prática, que teve sua primeira edição lançada no início de 2004. Contudo, nessa primeira obra utilizamos principalmente a UML em sua versão 1.5, enquanto neste novo livro empregamos totalmente as notações definidas na UML 2, que apresenta grandes inovações com relação às versões anteriores. Assim, embora alguns exemplos sejam parecidos com os do primeiro livro, estes foram todos revisados e atualizados sempre que isso se mostrou necessário. Além disso, o livro contém uma grande quantidade de novos exemplos e propõe exercícios igualmente inéditos, sendo que todos os capítulos utilizam a notação da UML 2 e demonstram ainda os novos componentes e características acrescidos a cada diagrama. Alguns desses capítulos se referem a diagramas novos, que só passaram a existir ou se tornaram independentes de outros com o advento da UML 2. O estudo de caso apresentado ao final do livro também é totalmente inédito, onde modelamos um sistema para controle de pizzaria online.

O livro está estruturado da seguinte forma:

O capítulo 1 apresenta uma explanação a respeito da necessidade de se modelar software, além de introduzir a UML, destacando em linhas gerais as funções de cada diagrama.

O capítulo 2 enfoca o paradigma de orientação a objetos, uma vez que a UML é uma linguagem baseada nesse paradigma e utilizada principalmente para a modelagem de softwares orientados a objetos.

Os capítulos 3 a 15 abordam, respectivamente, os diagramas de casos de uso, de classes, objetos, pacotes, sequência, comunicação, máquina de estados, atividade, visão geral de interação, componentes, implantação, estrutura composta e tempo. Em cada um desses capítulos procuramos descrever a função de cada diagrama, detalhando seus componentes e apresentando exemplos de como utilizar cada diagrama. Ao final da maioria dos capítulos são sugeridos alguns exercícios para que o leitor possa praticar seu conhecimento. Todos os exercícios encontram-se resolvidos e explicados no final de seus respectivos capítulos.

Ao longo dos capítulos 3 a 14 foi modelado um sistema de controle bancário como ilustração e, ao longo dos exercícios, foram parcialmente modelados cinco sistemas relativamente simples. O capítulo 16 apresenta um estudo de caso referente a um sistema maior e mais complexo, no qual modelamos um sistema para controle de pizzaria online, onde os pedidos dos clientes poderão ser feitos pela internet.

Finalmente, no último capítulo enfocamos a arquitetura da linguagem, discorrendo sobre a infraestrutura e superestrutura da UML 2.

CAPÍTULO 1

Introdução à UML

A UML – Unified Modeling Language ou Linguagem de Modelagem Unificada – é uma linguagem visual utilizada para modelar softwares baseados no paradigma de orientação a objetos. É uma linguagem de modelagem de propósito geral que pode ser aplicada a todos os domínios de aplicação. Essa linguagem tornou-se, nos últimos anos, a linguagem-padrão de modelagem adotada internacionalmente pela indústria de engenharia de software.

Deve ficar bem claro, porém, que a UML não é uma linguagem de programação, e sim uma linguagem de modelagem, uma notação, cujo objetivo é auxiliar os engenheiros de software a definirem as características do sistema, tais como seus requisitos, seu comportamento, sua estrutura lógica, a dinâmica de seus processos e até mesmo suas necessidades físicas em relação ao equipamento sobre o qual o sistema deverá ser implantado. Tais características podem ser definidas por meio da UML antes do software começar a ser realmente desenvolvido. Além disso, cumpre destacar que a UML não é um processo de desenvolvimento de software e tampouco está ligada a um de forma exclusiva, sendo totalmente independente, podendo ser utilizada por muitos processos de desenvolvimento diferentes ou mesmo da forma que o engenheiro considerar mais adequada.

1.1 Breve Histórico da UML

A UML surgiu da união de três métodos de modelagem: o método de Booch, o método OMT (Object Modeling Technique) de Jacobson, e o método OOSE (Object-Oriented Software Engineering) de Rumbaugh. Estes eram, até meados da década de 1990, os métodos de modelagem orientada a objetos mais populares entre os profissionais da área de desenvolvimento de software. A união desses métodos contou com o amplo apoio da Rational Software, que a incentivou e financiou.

O esforço inicial do projeto começou com a união do método de Booch ao OMT de Jacobson, o que resultou no lançamento do Método Unificado no final de 1995. Logo em seguida, Rumbaugh juntou-se a Booch e Jacobson na

Rational Software, e seu método OOSE começou também a ser incorporado à nova metodologia. O trabalho de Booch, Jacobson e Rumbaugh, conhecidos popularmente como "Os Três Amigos", resultou no lançamento, em 1996, da primeira versão da UML propriamente dita.

Tão logo a primeira versão foi lançada, muitas empresas atuantes na área de modelagem e desenvolvimento de software passaram a contribuir para o projeto, fornecendo sugestões para melhorar e ampliar a linguagem. Finalmente, a UML foi adotada, em 1997, pela OMG (Object Management Group ou Grupo de Gerenciamento de Objetos), como uma linguagem-padrão de modelagem.

A versão 2.0 da linguagem foi oficialmente lançada em julho de 2005, encontrando-se esta atualmente na versão 2.3 beta. A documentação oficial da UML pode ser consultada no site da OMG em `www.omg.org` ou mais exatamente em `www.uml.org`.

1.2 Por Que Modelar Software?

Qual a real necessidade de se modelar um software? Muitos "profissionais" podem afirmar que conseguem determinar todas as necessidades de um sistema de informação de cabeça, e que sempre trabalharam assim. Qual a real necessidade de se projetar uma casa? Um pedreiro experiente não é capaz de construí-la sem um projeto? Isso pode ser verdade, mas a questão é muito mais ampla, envolvendo fatores extremamente complexos, como levantamento e análise de requisitos, prototipação, tamanho do projeto, complexidade, prazos, custos, documentação, manutenção e reusabilidade, entre outros.

Existe uma diferença gritante entre construir uma pequena casa e construir um prédio de vários andares. Obviamente, para se construir um edifício é necessário, em primeiro lugar, desenvolver um projeto muito bem-elaborado, cujos cálculos têm de estar extremamente corretos e precisos. Além disso, é preciso fornecer uma estimativa de custos, determinar em quanto tempo a construção estará concluída, avaliar a quantidade de profissionais necessária à execução da obra, especificar a quantidade de material a ser adquirida para a construção, escolher o local onde o prédio será erguido etc. Grandes projetos não podem ser modelados de cabeça, nem mesmo a maioria dos pequenos projetos pode sê-lo, exceto, talvez, aqueles extremamente simples.

Na realidade, por mais simples que seja, todo e qualquer sistema deve ser modelado antes de se iniciar sua implementação, entre outras coisas, porque os sistemas de informação frequentemente costumam ter a propriedade de "crescer", isto é, aumentar em tamanho, complexidade e abrangência. Muitos profissionais costumam afirmar que sistemas de informação são "vivos", porque nunca estão completamente finalizados. Na verdade, o termo correto seria "dinâmicos", pois

normalmente os sistemas de informação estão em constante mudança. Tais mudanças são devidas a diversos fatores, como, por exemplo:

- Os clientes desejam constantemente modificações ou melhorias no sistema.
- O mercado está sempre mudando, o que força a adoção de novas estratégias por parte das empresas e, consequentemente, de seus sistemas.
- O governo seguidamente promulga novas leis e cria novos impostos e alíquotas ou, ainda, modifica as leis, os impostos e alíquotas já existentes, o que acarreta a manutenção no software.

Assim, um sistema de informação precisa ter uma documentação extremamente detalhada, precisa e atualizada para que possa ser mantido com facilidade, rapidez e correção, sem produzir novos erros ao corrigir os antigos. Modelar um sistema é uma forma bastante eficiente de documentá-lo, mas a modelagem não serve apenas para isso: a documentação é apenas uma das vantagens fornecidas pela modelagem. Existem muitas outras que serão discutidas nas próximas seções.

1.2.1 Modelo de Software – Uma Definição

A modelagem de um software implica em criar modelos de software, mas o que é realmente um modelo de software? Um modelo de software captura uma visão de um sistema físico, é uma abstração do sistema com um certo propósito, como descrever aspectos estruturais ou comportamentais do software. Esse propósito determina o que deve ser incluído no modelo e o que é considerado irrelevante. Assim um modelo descreve completamente aqueles aspectos do sistema físico que são relevantes ao propósito do modelo, no nível apropriado de detalhe.

Dessa forma, um modelo de casos de uso fornecerá uma visão dos requisitos necessários ao sistema, identificando as funcionalidades do software e os atores que poderão utilizá-las, não se preocupando em detalhar nada além disso. Já um modelo conceitual irá identificar as classes relacionadas ao domínio do problema, sem detalhar seus métodos, enquanto um modelo de domínio ampliará o modelo conceitual, incluindo informações relativas à solução do problema, incluindo, entre outras coisas, os métodos necessários a essa solução.

1.2.2 Levantamento e Análise de Requisitos

Uma das primeiras fases de um processo de desenvolvimento de software consiste no Levantamento de Requisitos. As outras etapas, sugeridas por muitos autores, são: Análise de Requisitos, Projeto, que se constitui na principal fase da modelagem, Codificação, Testes e Implantação. Dependendo do método/processo adotado, essas etapas ganham, por vezes, nomenclaturas diferentes, podendo algumas delas ser condensadas em uma etapa única, ou uma etapa pode ser

dividida em duas ou mais etapas. Se tomarmos como exemplo o Processo Unificado (Unified Process), um método de desenvolvimento de software, veremos que este se divide em quatro fases: Concepção, onde é feito o levantamento de requisitos; Elaboração, onde é feita a análise dos requisitos e o projeto do software; Construção, onde o software é implementado e testado; e Transição, onde o software será implantado. As fases de Elaboração e Construção ocorrem, sempre que possível, em ciclos iterativos, dessa forma, sempre que um ciclo é completado pela fase de Construção, volta-se à fase de Elaboração para tratar do ciclo seguinte, até todo o software ser finalizado.

As etapas de levantamento e análise de requisitos trabalham com o domínio do problema e tentam determinar "o que" o software deve fazer e se é realmente possível desenvolver o software solicitado. Na etapa de levantamento de requisitos, o engenheiro de software busca compreender as necessidades do usuário e o que ele deseja que o sistema a ser desenvolvido realize. Isso é feito sobretudo por meio de entrevistas, nas quais o engenheiro tenta compreender como funciona atualmente o processo a ser informatizado e quais serviços o cliente precisa que o software forneça.

Devem ser realizadas tantas entrevistas quantas forem necessárias para que as necessidades do usuário sejam bem-compreendidas. Durante as entrevistas, o engenheiro deve auxiliar o cliente a definir quais informações deverão ser produzidas, quais deverão ser fornecidas e qual o nível de desempenho exigido do software.

Um dos principais problemas enfrentados na fase de levantamento de requisitos é o de comunicação. A comunicação constitui-se em um dos maiores desafios da engenharia de software, caracterizando-se pela dificuldade em conseguir compreender um conjunto de conceitos vagos, abstratos e difusos que representam as necessidades e os desejos dos clientes e transformá-los em conceitos concretos e inteligíveis.

A fase de levantamento de requisitos deve identificar dois tipos de requisitos: os funcionais e os não-funcionais. Os requisitos funcionais correspondem ao que o cliente quer que o sistema realize, ou seja, as funcionalidades do software. Já os requisitos não-funcionais correspondem às restrições, condições, consistências, validações que devem ser levadas a efeito sobre os requisitos funcionais. Por exemplo, em um sistema bancário deve ser oferecida a opção de abrir novas contas correntes, o que é um requisito funcional. Já determinar que somente pessoas maiores de idade possam abrir contas corrente é um requisito não-funcional.

Podem existir diversos tipos de requisitos não-funcionais, como de usabilidade, desempenho, confiabilidade, segurança ou interface. Alguns requisitos não-funcionais identificam regras de negócio, ou seja, as políticas, normas e condições estabelecidas pela empresa que devem ser seguidas na execução de

uma funcionalidade. Por exemplo, estabelecer que depois de abrir uma conta é necessário depositar um valor mínimo inicial é uma regra de negócio adotada por um determinado banco e que não necessariamente é seguida por outras instituições bancárias. Outro exemplo de regra de negócio seria determinar que, em um sistema de videolocadora, só se poderia realizar uma nova locação para um sócio depois de ele ter devolvido as cópias locadas anteriormente.

Logo após o levantamento de requisitos, passa-se à fase em que as necessidades apresentadas pelo cliente são analisadas. Essa etapa é conhecida como análise de requisitos. Aqui o engenheiro examina os requisitos enunciados pelos usuários, verificando se estes foram especificados corretamente e se foram realmente bem-compreendidos. A partir da etapa de análise de requisitos são determinadas as reais necessidades do sistema de informação.

A grande questão é: como saber se as necessidades dos usuários foram realmente bem-compreendidas? Um dos objetivos da análise de requisitos consiste em determinar se as necessidades dos usuários foram entendidas de maneira correta, verificando se alguma questão deixou de ser abordada, determinando se algum requisito foi especificado incorretamente ou se algum conceito precisa ser melhor explicado. Durante a análise de requisitos, uma linguagem de modelagem auxilia a levantar questões que não foram concebidas durante as entrevistas iniciais. Tais questões devem ser sanadas o quanto antes, para que o projeto do software não tenha que sofrer modificações quando seu desenvolvimento já estiver em andamento, o que pode causar significativos atrasos no desenvolvimento do software, sendo por vezes necessário remodelar por completo o projeto.

Além daquele concernente à comunicação, outro grande problema encontrado durante as entrevistas consiste no fato de que, na maioria das vezes, os usuários não têm realmente certeza do que querem e não conseguem enxergar as reais potencialidades de um sistema de informação. Em geral, os engenheiros de software precisam sugerir inúmeras características e funções do sistema que o cliente não sabia como formular ou sequer havia imaginado. Na realidade, na maior parte das vezes, esses profissionais precisam reestruturar o modo como as informações são geridas e utilizadas pela empresa e apresentar maneiras de combiná-las e apresentá-las de maneira que possam ser melhor aproveitadas pelos usuários.

Em muitos casos é realmente isso o que os clientes esperam dos engenheiros de software, porém, em outros, os engenheiros encontram fortes resistências a qualquer mudança na forma como a empresa manipula suas informações, fazendo-se necessário um significativo esforço para provar ao cliente que as modificações sugeridas permitirão um melhor desempenho do software, além

de ser útil para a própria empresa, obviamente. Na realidade, nesse último caso é fundamental trabalhar bastante o aspecto social da implantação de um sistema informatizado na empresa, pois muitas vezes a resistência não é tanto por parte da gerência, mas pelos usuários finais, que serão obrigados a mudar a forma como estavam acostumados a trabalhar e aprender a utilizar uma nova tecnologia.

1.2.3 Prototipação

A prototipação é uma técnica bastante popular e de fácil aplicação. Essa técnica consiste em desenvolver rapidamente um "rascunho" do que seria o sistema de informação quando ele estivesse finalizado. Um protótipo normalmente apresenta pouco mais do que a interface do software a ser desenvolvido, ilustrando como as informações seriam inseridas e recuperadas no sistema, apresentando alguns exemplos com dados fictícios de quais seriam os resultados apresentados pelo software, principalmente em forma de relatórios. A utilização de um protótipo pode, assim, evitar que, após meses ou até anos de desenvolvimento, descubra-se, ao implantar o sistema, que o software não atende completamente às necessidades do cliente devido, sobretudo, a falhas de comunicação durante as entrevistas iniciais.

Hoje em dia, é possível desenvolver protótipos com extrema rapidez e facilidade, por meio da utilização de ferramentas conhecidas como RAD (Rapid Application Development ou Desenvolvimento Rápido de Aplicações). Essas ferramentas são encontradas na maioria dos ambientes de desenvolvimento das linguagens de programação atuais, como NetBeans, Delphi, Visual Basic ou C++ Builder, entre outras. Essas ferramentas disponibilizam ambientes de desenvolvimento que permitem a construção de interfaces de forma muito rápida, além de permitirem também modificar tais interfaces de maneira igualmente veloz, na maioria das vezes sem a necessidade de alterar qualquer código porventura já escrito.

As ferramentas RAD permitem a criação de formulários e a inserção de componentes nos mesmos, de uma forma muito simples, rápida e fácil, bastando ao desenvolvedor selecionar o componente (botões, caixas de texto, labels, combos etc.) em uma barra de ferramentas e clicar com o mouse sobre o formulário. Alternativamente, o usuário pode clicar duas vezes sobre o componente desejado, fazendo com que um componente do tipo selecionado surja no centro do formulário. Além disso, tais ferramentas permitem ao usuário mudar a posição dos componentes depois de terem sido colocados no formulário simplesmente selecionando o componente com o mouse e o arrastando para a posição desejada.

Esse tipo de ferramenta é extremamente útil no desenvolvimento de protótipos pela facilidade de produzir e modificar as interfaces. Assim, depois de determinar quais as modificações necessárias ao sistema de informação após o

protótipo ter sido apresentado aos usuários, pode-se modificar a interface do protótipo de acordo com as novas especificações e reapresentá-lo ao cliente de forma muito rápida.

Seguindo esse raciocínio, a etapa de análise de requisitos deve, obrigatoriamente, produzir um protótipo para demonstrar como se apresentará e comportará o sistema em essência, bem como quais informações deverão ser inseridas no sistema e que tipo de informações deverão ser fornecidas pelo software. Um protótipo é de extrema importância durante as primeiras fases de engenharia de um sistema de informação. Por meio da ilustração que um protótipo pode apresentar, a maioria das dúvidas e erros de especificação pode ser sanada, devido ao fato de um protótipo demonstrar visualmente um exemplo de como funcionará o sistema depois de concluído, como será sua interface, de que maneira os usuários interagirão com ele, que tipo de relatórios serão fornecidos etc., facilitando a compreensão do cliente.

Apesar das grandes vantagens advindas do uso da técnica de prototipação, é necessária ainda uma ressalva: um protótipo pode induzir o cliente a acreditar que o software encontra-se em um estágio bastante avançado de desenvolvimento. Com frequência ocorre de o cliente não compreender o conceito de um protótipo. Para ele, o esboço apresentado já é o próprio sistema praticamente acabado. Por isso, muitas vezes o cliente não compreende nem aceita prazos longos, os quais considera absurdos, já que o sistema foi-lhe apresentado já funcionando, necessitando de alguns poucos ajustes. Por isso, é preciso deixar bem claro ao usuário que o software que lhe está sendo apresentado é apenas um "rascunho" do que será o sistema de informação quando estiver finalizado e que seu desenvolvimento ainda não foi realmente iniciado.

1.2.4 Prazos e Custos

Em seguida, vem a espinhosa e desagradável, porém extremamente importante, questão dos prazos e custos. Como determinar o prazo real de entrega de um software? Quantos profissionais deverão trabalhar no projeto? Qual será o custo total de desenvolvimento? Qual deverá ser o valor estipulado para produzir o sistema? Como determinar a real complexidade de desenvolvimento do software? Geralmente, após as primeiras entrevistas, os clientes estão bastante interessados em saber quanto vai lhes custar o sistema de informação e em quanto tempo eles o terão implantado e funcionando em sua empresa.

A estimativa de tempo é realmente um tópico extremamente complexo da engenharia de software. Na realidade, por melhor modelado que um sistema tenha sido, ainda assim fica difícil determinar com exatidão os prazos finais de entrega do software. Uma boa modelagem auxilia a estimar a complexidade de

desenvolvimento de um sistema, e isso, por sua vez, ajuda – e muito – a determinar o prazo final em que o software será entregue. No entanto, é preciso ter diversos sistemas de informação com níveis de dificuldade e características semelhantes ao software que está para ser construído, já previamente desenvolvidos e bem-documentados, para determinar com maior exatidão a estimativa de prazos.

Contudo, mesmo com o auxílio dessa documentação, ainda é muito difícil estipular uma data exata. O máximo que se pode conseguir é apresentar uma que seja aproximada, com base na experiência documentada de desenvolvimento de outros softwares. Assim, é recomendável acrescentar alguns meses à data de entrega, o que serve como margem de segurança para possíveis erros de estimativa.

Para poder auxiliar na estimativa de prazos e custos de um software, a documentação da empresa desenvolvedora deverá ter registros das datas de início e término de cada projeto já concluído, além do custo real de desenvolvimento que tais projetos acarretaram, envolvendo inclusive os custos com manutenção e o número de profissionais envolvidos em cada projeto. Na verdade, uma empresa de desenvolvimento de software que nunca tenha desenvolvido um sistema de informação antes e, portanto, não tenha documentação histórica de projetos anteriores, dificilmente será capaz de apresentar uma estimativa correta de prazos e custos, principalmente porque a equipe de desenvolvimento não saberá com certeza quanto tempo levará desenvolvendo o sistema, já que o tempo de desenvolvimento influencia diretamente o custo de desenvolvimento do sistema e, logicamente, o valor a ser cobrado pelo software.

Se a estimativa de prazo estiver errada, cada dia a mais de desenvolvimento do projeto acarretará prejuízos para a empresa que desenvolve o sistema, decorrentes, por exemplo, de pagamentos de salários aos profissionais envolvidos no projeto que não haviam sido previstos e desgaste dos equipamentos utilizados. Isso sem levar em conta prejuízos mais difíceis de contabilizar, como manter inúmeros profissionais ocupados em projetos que já deveriam estar concluídos, que deixam de trabalhar em novos projetos, além da insatisfação dos clientes por não receberem o produto no prazo estimado e a propaganda negativa daí decorrente.

1.2.5 Projeto

Enquanto a fase de análise trabalha com o domínio do problema, a fase de projeto trabalha com o domínio da solução, procurando estabelecer "como" o sistema fará o que foi determinado na fase de análise, ou seja, qual será a solução para o problema identificado. É na etapa de projeto que é realizada a maior parte da modelagem do software a ser desenvolvido, ou seja, é nessa etapa que é produzida a arquitetura do sistema.

A etapa de projeto toma a modelagem iniciada na fase de análise e lhe acrescenta profundos acréscimos e detalhamentos. Enquanto na análise foram identificadas as funcionalidades necessárias ao software e suas restrições, na fase de projeto será estabelecido como essas funcionalidades deverão realizar o que foi solicitado.

A fase de projeto leva em consideração os recursos tecnológicos existentes para que o problema apresentado pelo cliente possa ser solucionado. É nesse momento que será selecionada a linguagem de programação a ser utilizada, o sistema gerenciador de banco de dados a ser empregado, como será a interface final do sistema e até mesmo como o software será distribuído fisicamente na empresa, especificando o hardware necessário para a sua implantação e funcionamento correto.

1.2.6 Manutenção

Possivelmente a questão mais importante que todas as outras já enunciadas é a da manutenção. Alguns autores afirmam que muitas vezes a manutenção de um software pode representar de 40 a 60% do custo total do projeto. Alguém poderá então dizer que a modelagem é necessária para diminuir os custos com manutenção – se a modelagem estiver correta o sistema não apresentará erros e, então, não precisará sofrer manutenções.

Embora um dos objetivos de modelar um software seja realmente diminuir a necessidade de mantê-lo, a modelagem não serve apenas para isso. Na maioria dos casos, a manutenção de um software é inevitável, pois, como já foi dito, as necessidades de uma empresa são dinâmicas e mudam constantemente, o que faz surgir novas necessidades que não existiam na época em que o software foi projetado, isso sem falar nas frequentes mudanças em leis, alíquotas, impostos, taxas ou formato de notas fiscais, por exemplo. Levando isso em consideração, é bastante provável que um sistema de informação, por mais bem-modelado que esteja, precise sofrer manutenções.

Nesse caso, a modelagem não serve apenas para diminuir a necessidade de futuras manutenções, mas também para facilitar a compreensão do sistema por quem tiver que mantê-lo, já que, em geral, a manutenção de um sistema é considerada uma tarefa ingrata pelos profissionais de desenvolvimento, por normalmente exigir que estes despendam grandes esforços para compreender códigos escritos por outros cujos estilos de desenvolvimento são diferentes e que, via de regra, não se encontram mais na empresa.

Esse tipo de código é conhecido como "código alienígena" ou "software legado". O termo refere-se a códigos que não seguem as regras atuais de desenvolvimento da empresa, não foram modelados e, por conseguinte, têm pouca ou nenhuma

documentação. Além disso, nenhum dos profissionais da equipe atual trabalhou em seu projeto inicial e, para piorar, o código já sofreu manutenções anteriores por outros profissionais que também não se encontram mais na empresa, sendo que cada um deles tinha um estilo de desenvolvimento diferente, ou seja, como se diz no meio de desenvolvimento, o código encontra-se "remendado".

Assim, uma modelagem correta aliada a uma documentação completa e atualizada de um sistema de informação torna mais rápido o processo de manutenção e impede que erros sejam cometidos, já que é muito comum que, depois de manter uma rotina ou função de um software, outras rotinas ou funções do sistema que antes funcionavam perfeitamente passem a apresentar erros ou simplesmente deixem de funcionar. Tais erros são conhecidos como "efeitos colaterais" da manutenção.

Além disso, qualquer manutenção a ser realizada em um sistema deve ser também modelada e documentada, para não desatualizar a documentação do sistema e prejudicar futuras manutenções, já que muitas vezes uma documentação desatualizada pode ser mais prejudicial à manutenção do sistema do que nenhuma documentação.

Pode-se fornecer uma analogia de "manutenção" na vida real responsável pela produção de um efeito colateral no meio ambiente, o que não deixa de ser um sistema: muito pelo contrário, é "o" sistema. Na realidade, esse exemplo não identifica exatamente uma manutenção, e sim uma modificação em uma região. Descobri recentemente, ao assistir a uma reportagem, que a formiga saúva vinha se tornando uma praga em algumas regiões do país porque estava se reproduzindo demais. Esse crescimento desordenado era causado pelos tratores que, ao arar a terra, destruíam os formigueiros da formiga Lava-Pés, que ficam próximos à superfície, mas não afetavam os formigueiros de saúvas, por estes se encontrarem em um nível mais profundo do solo.

Entretanto, as lava-pés consumiam os ovos das saúvas, o que impedia que estas aumentassem demais. Assim, a diminuição das formigas lava-pés resultou no crescimento desordenado das saúvas. Isso é um exemplo de manutenção com efeito colateral na vida real. No caso, foi aplicada uma espécie de "manutenção", onde modificou-se o ambiente para arar a terra e produziu-se uma praga que antes constituía-se apenas em uma espécie controlada. Se a "função" da formiga lava-pés estivesse devidamente documentada, ela não teria sido eliminada, e a saúva, por conseguinte, não teria se tornado uma praga.

1.2.7 Documentação Histórica

Finalmente, existe a questão da perspectiva histórica, ou seja, novamente a já tão falada documentação de software. Neste caso, referimo-nos à documentação histórica dos projetos anteriores já concluídos pela empresa. É por meio dessa documentação histórica que a empresa pode responder a perguntas como:

- A empresa está evoluindo?
- O processo de desenvolvimento tornou-se mais rápido?
- As metodologias hoje adotadas são superiores às práticas aplicadas anteriormente?
- A qualidade do software produzido está melhorando?

Uma empresa ou setor de desenvolvimento de software necessita de um registro detalhado de cada um de seus sistemas de informação antes desenvolvidos para poder determinar, entre outros, fatores como:

- A média de manutenções que um sistema sofre normalmente dentro de um determinado período de tempo.
- Qual a média de custo de modelagem.
- Qual a média de custo de desenvolvimento.
- Qual a média de tempo despendido até a finalização do projeto.
- Quantos profissionais são necessários envolver normalmente em um projeto.

Essas informações são computadas nos orçamentos de desenvolvimento de novos softwares e são de grande auxílio no momento de determinar prazos e custos mais próximos da realidade.

Além disso, a documentação pode ser muito útil em outra área: a Reusabilidade. Um dos grandes desejos e muitas vezes necessidades dos clientes é que o software esteja concluído o mais rápido possível. Uma das formas de agilizar o processo de desenvolvimento é a reutilização de rotinas, funções e algoritmos previamente desenvolvidos em outros sistemas. Nesse caso, a documentação correta do sistema pode auxiliar a sanar questões como:

- Onde as rotinas se encontram?
- Para que foram utilizadas?
- Em que projetos estão documentadas?
- Elas são adequadas ao software atualmente em desenvolvimento?
- Qual o nível necessário de adaptação destas rotinas para que possam ser utilizadas na construção do sistema atual?

1.3 Por que tantos Diagramas?

Por que a UML é composta por tantos diagramas? O objetivo disso é fornecer múltiplas visões do sistema a ser modelado, analisando-o e modelando-o sob diversos aspectos, procurando-se, assim, atingir a completitude da modelagem, permitindo que cada diagrama complemente os outros.

Cada diagrama da UML analisa o sistema, ou parte dele, sob uma determinada óptica. É como se o sistema fosse modelado em camadas, sendo que alguns diagramas enfocam o sistema de forma mais geral, apresentando uma visão externa do sistema, como é o objetivo do Diagrama de Casos de Uso, enquanto outros oferecem uma visão de uma camada mais profunda do software, apresentando um enfoque mais técnico ou ainda visualizando apenas uma característica específica do sistema ou um determinado processo. A utilização de diversos diagramas permite que falhas sejam descobertas, diminuindo a possibilidade da ocorrência de erros futuros.

Tomando novamente o exemplo da construção de um edifício, percebemos que ao se projetar uma construção, esta não tem apenas uma planta, mas diversas, enfocando o projeto de construção do prédio sob diferentes formas, algumas referentes ao layout dos andares, outras apresentando a planta hidráulica e outras ainda abordando a planta elétrica, por exemplo. Isso torna o projeto do edifício completo, abrangendo todas as características da construção. Da mesma maneira, os diversos diagramas fornecidos pela UML permitem analisar o sistema em diferentes níveis, podendo enfocar a organização estrutural do sistema, o comportamento de um processo específico, a definição de um determinado algoritmo ou até mesmo as necessidades físicas para a implantação do sistema.

1.4 Rápido Resumo dos Diagramas da UML

A seguir descreveremos rapidamente cada um dos diagramas oferecidos pela UML, destacando suas principais características.

1.4.1 Diagrama de Casos de Uso

O diagrama de casos de uso é o diagrama mais geral e informal da UML, utilizado normalmente nas fases de levantamento e análise de requisitos do sistema, embora venha a ser consultado durante todo o processo de modelagem e possa servir de base para outros diagramas. Apresenta uma linguagem simples e de fácil compreensão para que os usuários possam ter uma ideia geral de como o sistema irá se comportar. Procura identificar os atores (usuários, outros sistemas ou até mesmo algum hardware especial) que utilizarão de alguma forma

o software, bem como os serviços, ou seja, as funcionalidades que o sistema disponibilizará aos atores, conhecidas nesse diagrama como casos de uso. Veja na figura 1.1 um exemplo desse diagrama.

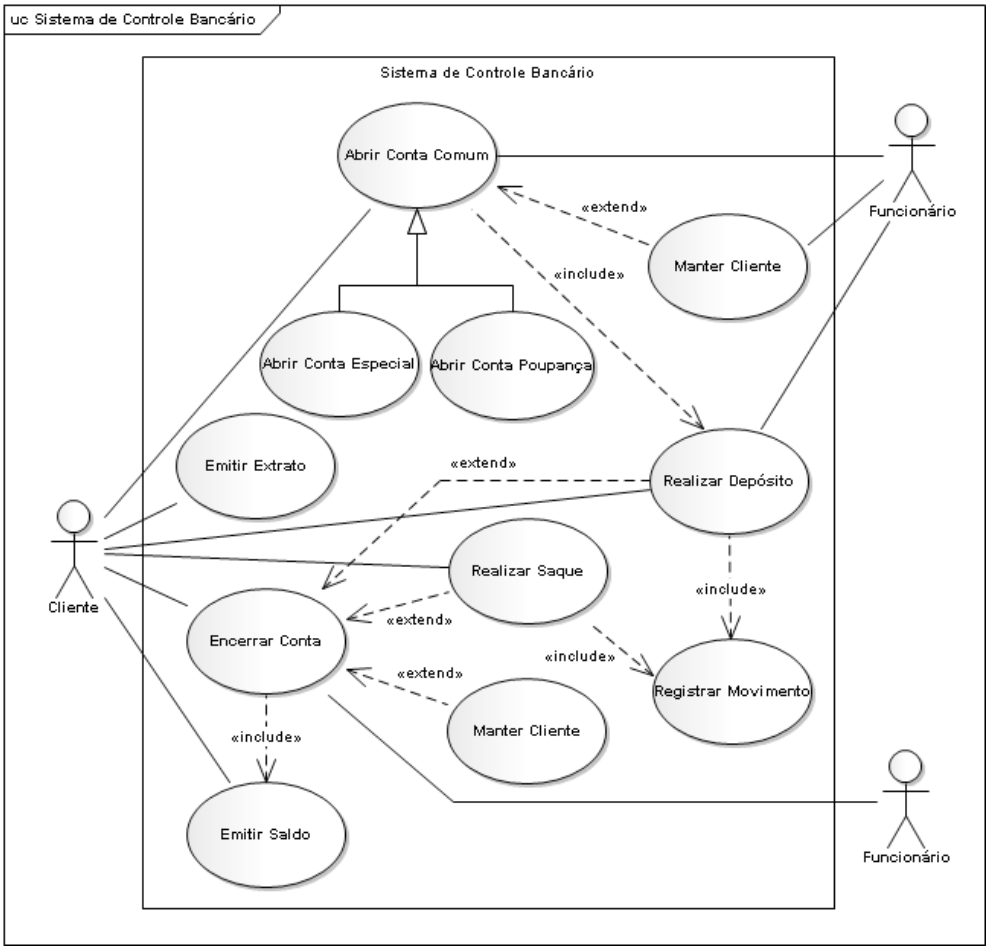

Figura 1.1 – Exemplo de Diagrama de Casos de Uso.

1.4.2 Diagrama de Classes

O diagrama de classes é provavelmente o mais utilizado e é um dos mais importantes da UML. Serve de apoio para a maioria dos demais diagramas. Como o próprio nome diz, define a estrutura das classes utilizadas pelo sistema, determinando os atributos e métodos que cada classe tem, além de estabelecer como as classes se relacionam e trocam informações entre si. A figura 1.2 apresenta um exemplo desse diagrama.

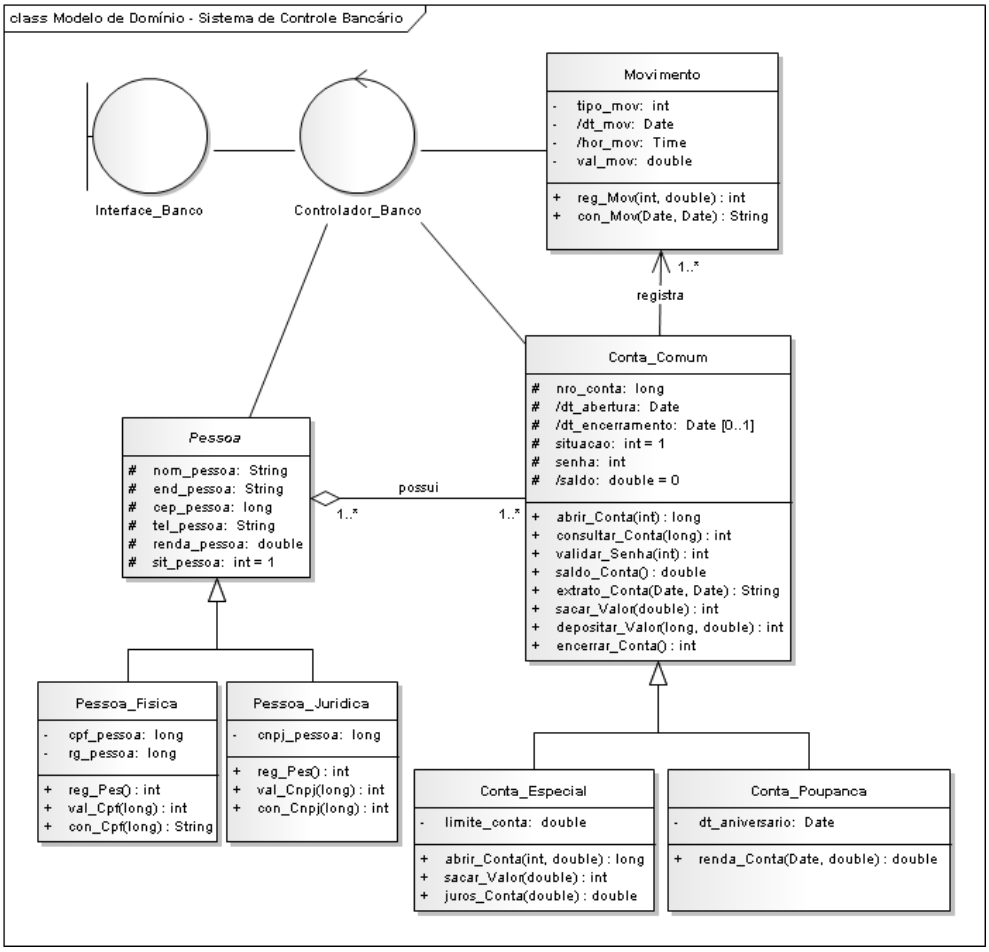

Figura 1.2 – Exemplo de Diagrama de Classes.

1.4.3 Diagrama de Objetos

O diagrama de objetos está amplamente associado ao diagrama de classes. Na verdade, o diagrama de objetos é praticamente um complemento do diagrama de classes e bastante dependente deste. O diagrama fornece uma visão dos valores armazenados pelos objetos de um diagrama de classes em um determinado momento da execução de um processo do software. A figura 1.3 apresenta um exemplo de diagrama de objetos.

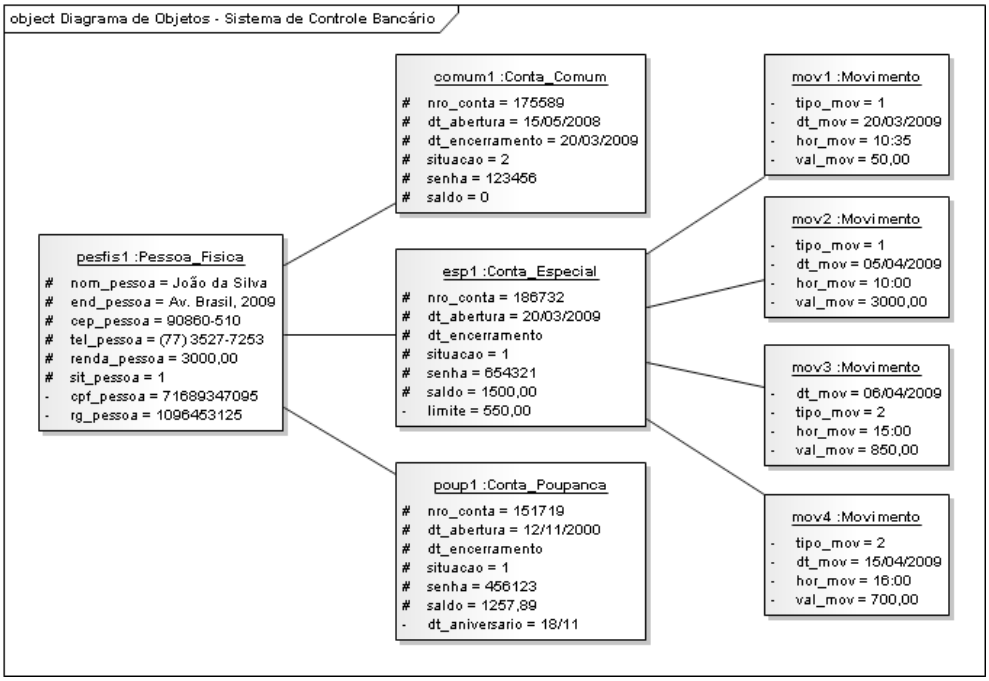

Figura 1.3 – Exemplo de Diagrama de Objetos.

1.4.4 Diagrama de Pacotes

O diagrama de pacotes é um diagrama estrutural que tem por objetivo representar os subsistemas ou submódulos englobados por um sistema de forma a determinar as partes que o compõem. Pode ser utilizado de maneira independente ou associado com outros diagramas. Esse diagrama pode ser utilizado também para auxiliar a demonstrar a arquitetura de uma linguagem, como ocorre com a própria UML ou ainda para definir as camadas de um software ou de um processo de desenvolvimento. A figura 1.4 apresenta um exemplo do mesmo.

1.4.5 Diagrama de Sequência

O diagrama de sequência é um diagrama comportamental que preocupa-se com a ordem temporal em que as mensagens são trocadas entre os objetos envolvidos em um determinado processo. Em geral, baseia-se em um caso de uso definido pelo diagrama de mesmo nome e apoia-se no diagrama de classes para determinar os objetos das classes envolvidas em um processo. Um diagrama de sequência costuma identificar o evento gerador do processo modelado, bem como o ator responsável por esse evento, e determina como o processo deve se desenrolar e ser concluído por meio da chamada de métodos disparados por mensagens enviadas entre os objetos. A figura 1.5 apresenta um exemplo desse diagrama.

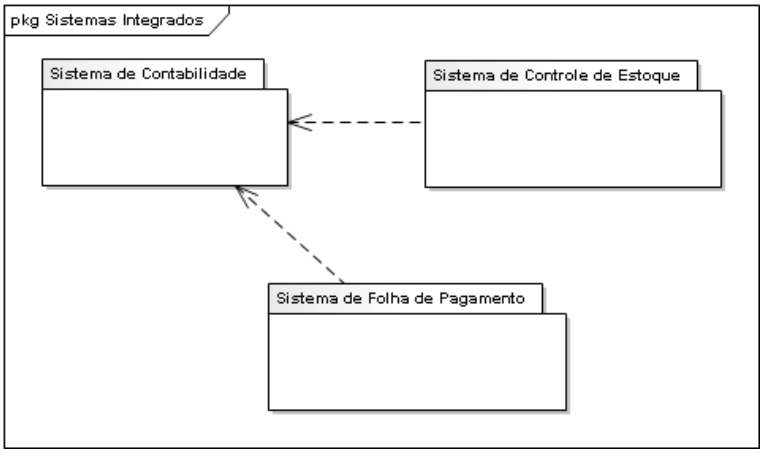

Figura 1.4 – Exemplo de Diagrama de Pacotes.

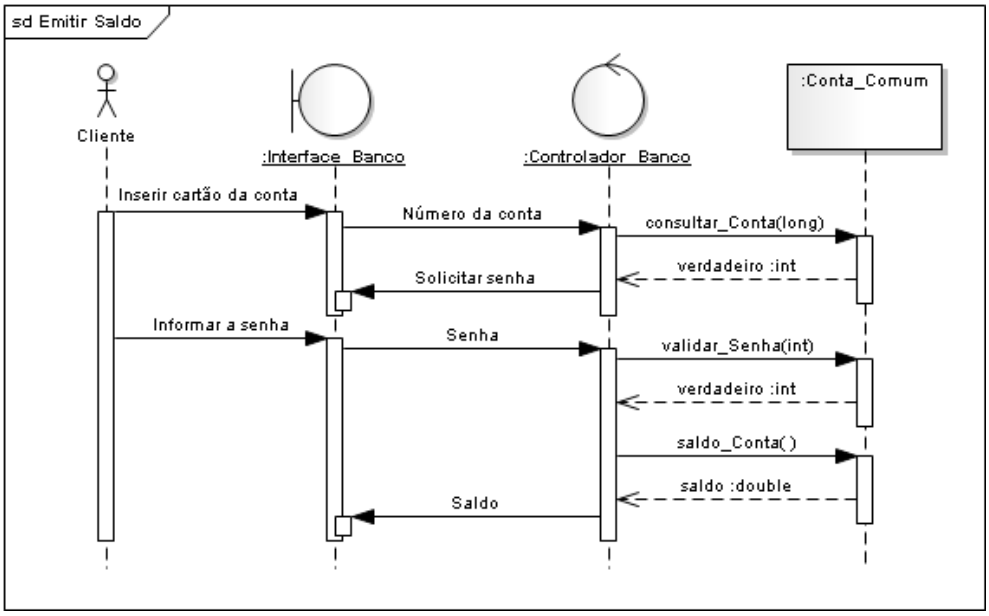

Figura 1.5 – Exemplo de Diagrama de Sequência.

1.4.6 Diagrama de Comunicação

O diagrama de comunicação era conhecido como de colaboração até a versão 1.5 da UML, tendo seu nome modificado para diagrama de comunicação a partir da versão 2.0. Está amplamente associado ao diagrama de sequência: na verdade, um complementa o outro. As informações mostradas no diagrama de comunicação com frequência são praticamente as mesmas apresentadas no de sequência, porém com um enfoque distinto, visto que esse diagrama não se preocupa com a temporalidade do processo, concentrando-se em como os elementos do diagrama estão vinculados e quais mensagens trocam entre si durante o processo. A figura 1.6 apresenta um exemplo de diagrama de comunicação.

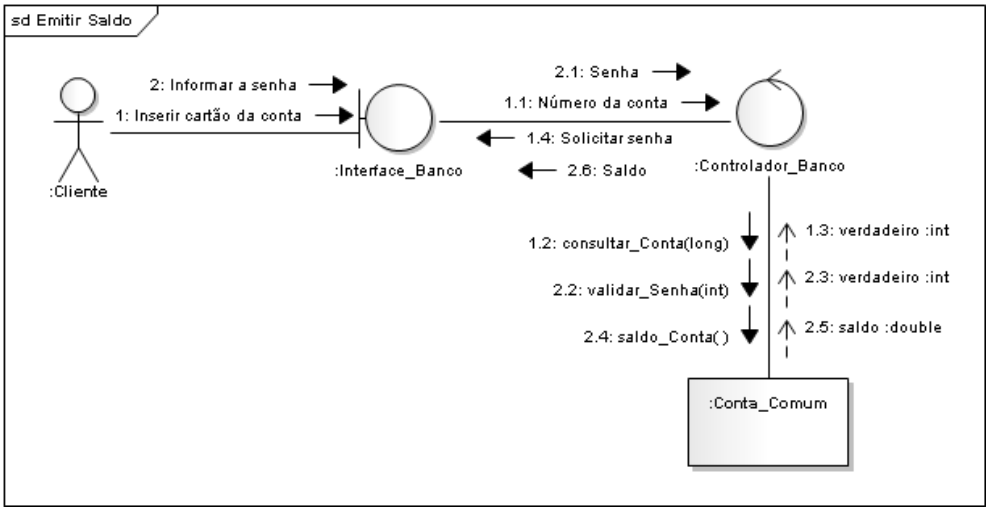

Figura 1.6 – Exemplo de Diagrama de Comunicação.

1.4.7 Diagrama de Máquina de Estados

O diagrama de máquina de estados demonstra o comportamento de um elemento por meio de um conjunto finito de transições de estado, ou seja, uma máquina de estados. Além de poder ser utilizado para expressar o comportamento de uma parte do sistema, quando é chamado de máquina de estado comportamental, também pode ser usado para expressar o protocolo de uso de parte de um sistema, quando identifica uma máquina de estado de protocolo. Como o diagrama de sequência, o de máquina de estados pode basear-se em um caso de uso, mas também pode ser utilizado para acompanhar os estados de outros elementos, como, por exemplo, uma instância de uma classe. A figura 1.7 apresenta um exemplo de diagrama de máquina de estados.

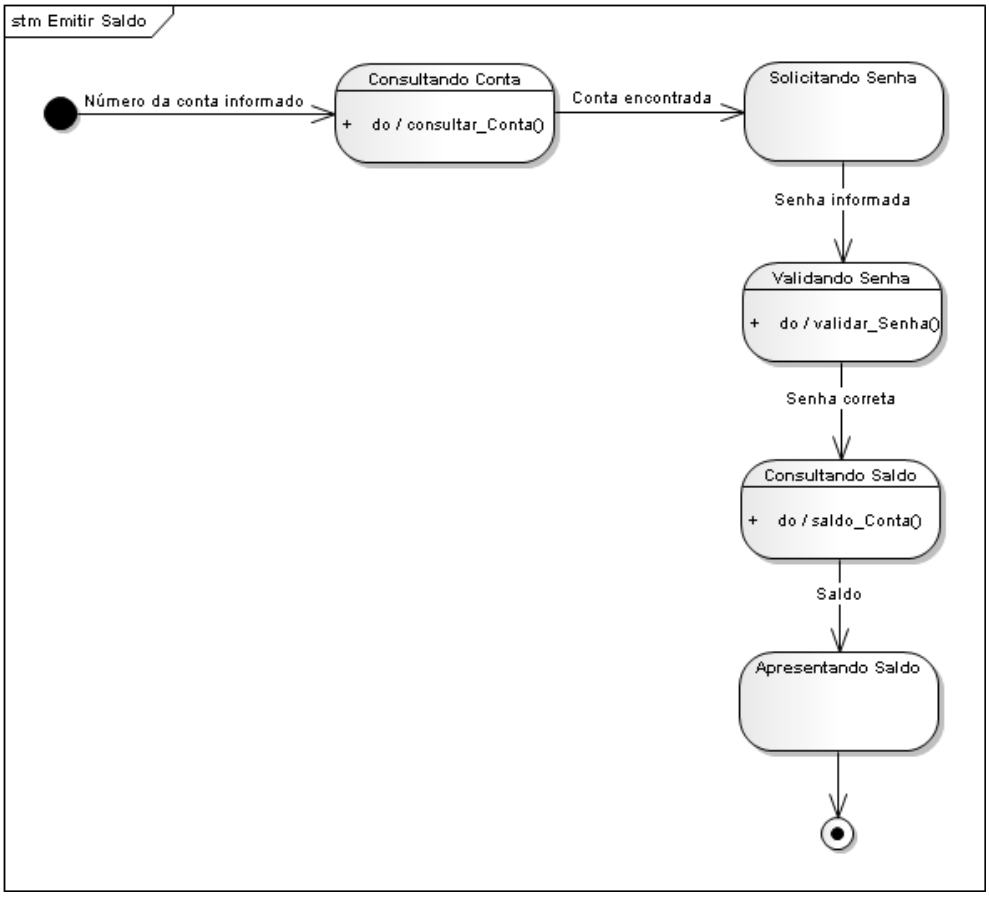

Figura 1.7 – Exemplo de Diagrama de Gráfico de Estados.

1.4.8 Diagrama de Atividade

O diagrama de atividade era considerado um caso especial do antigo diagrama de gráfico de estados, hoje conhecido como diagrama de máquina de estados, conforme descrito na seção anterior. A partir da UML 2.0, foi considerado independente do diagrama de máquina de estados. O diagrama de atividade preocupa-se em descrever os passos a serem percorridos para a conclusão de uma atividade específica, podendo esta ser representada por um método com certo grau de complexidade, um algoritmo, ou mesmo por um processo completo. O diagrama de atividade concentra-se na representação do fluxo de controle de uma atividade. A figura 1.8 apresenta um exemplo desse diagrama.

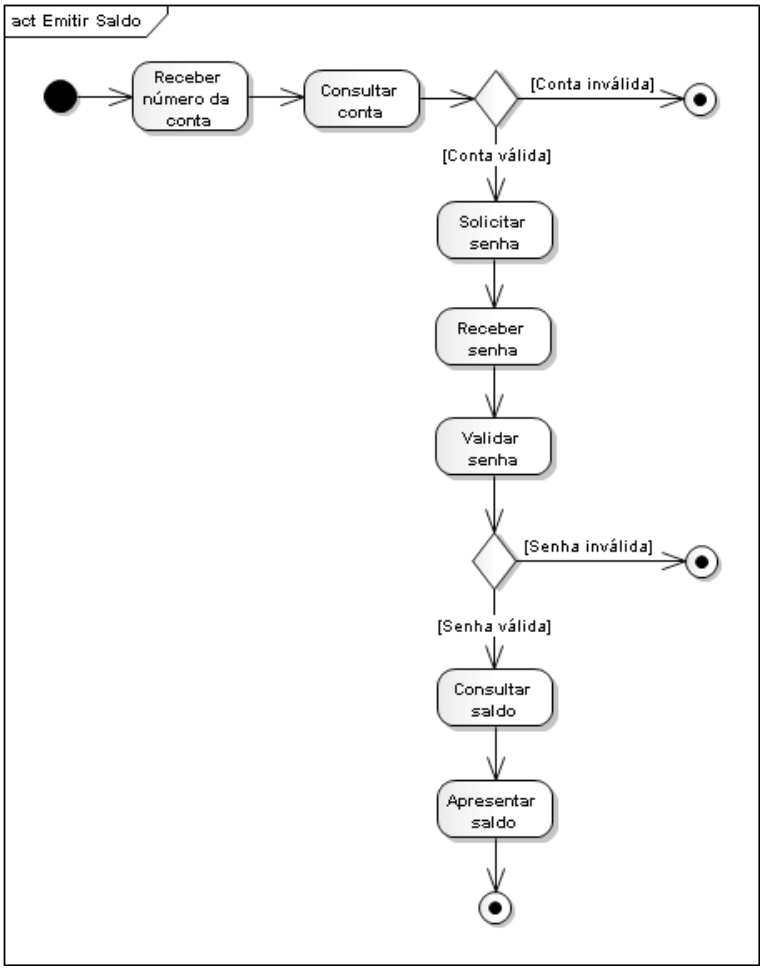

Figura 1.8 – Exemplo de Diagrama de Atividade.

1.4.9 Diagrama de Visão Geral de Interação

O diagrama de visão geral de interação é uma variação do diagrama de atividade que fornece uma visão geral dentro de um sistema ou processo de negócio. Esse diagrama passou a existir apenas a partir da UML 2. A figura 1.9 apresenta um exemplo do diagrama em questão.

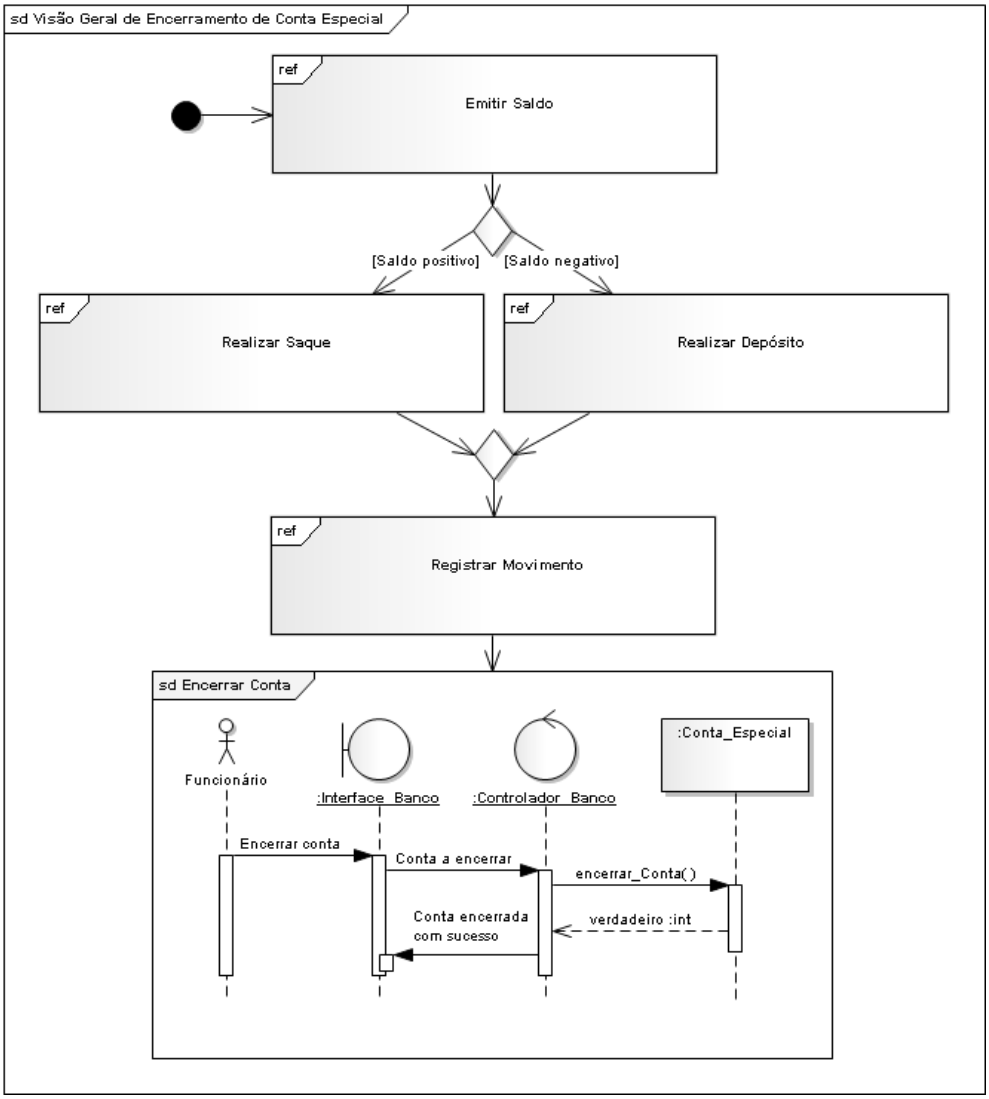

Figura 1.9 – Exemplo de Diagrama de Visão Geral de Interação.

1.4.10 Diagrama de Componentes

O diagrama de componentes está amplamente associado à linguagem de programação que será utilizada para desenvolver o sistema modelado. Esse diagrama representa os componentes do sistema quando o mesmo for ser implementado em termos de módulos de código-fonte, bibliotecas, formulários, arquivos de ajuda, módulos executáveis etc. e determina como tais componentes estarão estruturados e irão interagir para que o sistema funcione de maneira adequada. A figura 1.10 apresenta um exemplo desse diagrama.

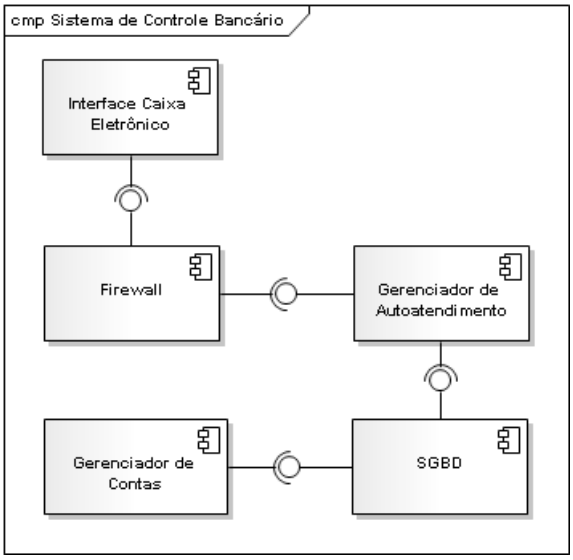

Figura 1.10 – Exemplo de Diagrama de Componentes.

1.4.11 Diagrama de Implantação

O diagrama de implantação determina as necessidades de hardware do sistema, as características físicas como servidores, estações, topologias e protocolos de comunicação, ou seja, todo o aparato físico sobre o qual o sistema deverá ser executado. Esse diagrama permite demonstrar também como se dará a distribuição dos módulos do sistema, em situações em que estes forem ser executados em mais de um servidor. A figura 1.11 apresenta um exemplo desse diagrama.

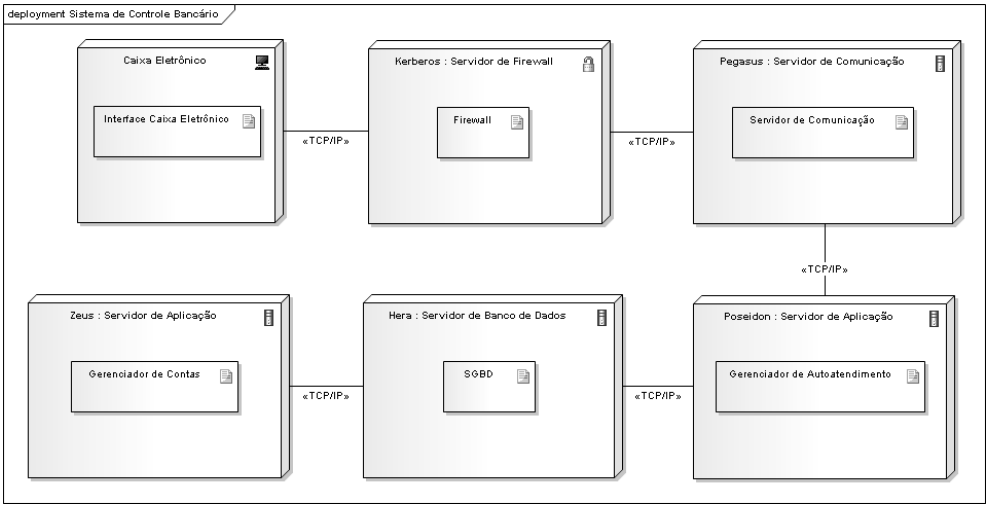

Figura 1.11 – Exemplo de Diagrama de Implantação.

1.4.12 Diagrama de Estrutura Composta

O diagrama de estrutura composta descreve a estrutura interna de um classificador, como uma classe ou componente, detalhando as partes internas que o compõem, como estas se comunicam e colaboram entre si. Também é utilizado para descrever uma colaboração em que um conjunto de instâncias cooperam entre si para realizar uma tarefa. A figura 1.12 mostra um exemplo de diagrama de estrutura composta.

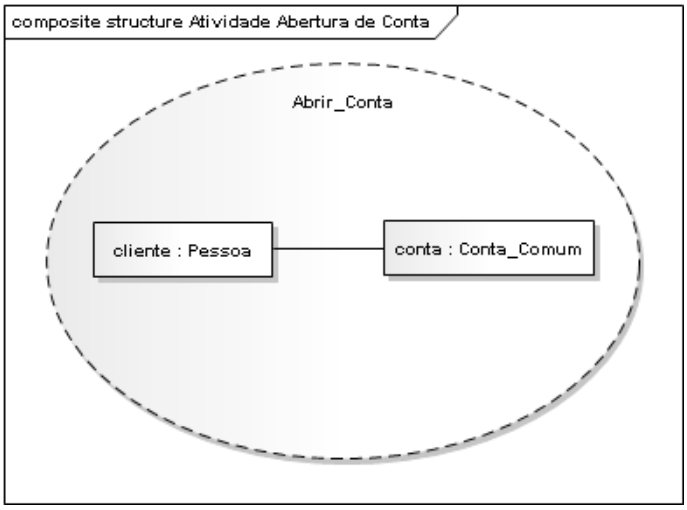

Figura 1.12 – Exemplo de Diagrama de Estrutura Composta.

1.4.13 Diagrama de Tempo ou de Temporização

O diagrama de tempo descreve a mudança no estado ou condição de uma instância de uma classe ou seu papel durante um período. Tipicamente utilizado para demonstrar a mudança no estado de um objeto no tempo em resposta a eventos externos. A figura 1.13 apresenta um exemplo desse diagrama.

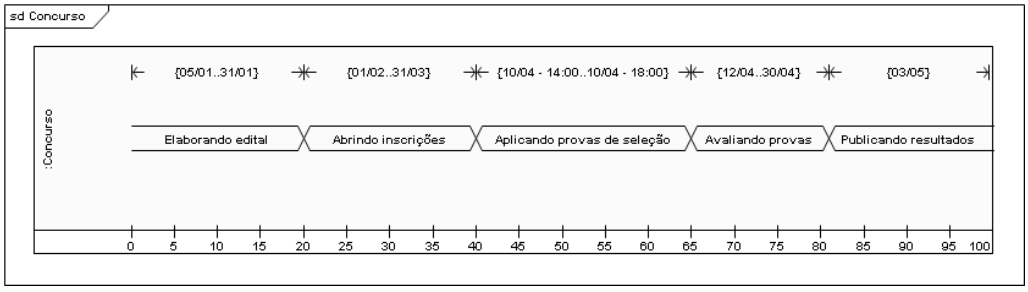

Figura 1.13 – Exemplo de Diagrama de Tempo.

1.4.14 Síntese Geral dos Diagramas

Os diagramas da UML dividem-se em diagramas estruturais e diagramas comportamentais, sendo que os últimos contêm ainda uma subdivisão representada pelos diagramas de interação, conforme pode ser verificado na figura 1.14.

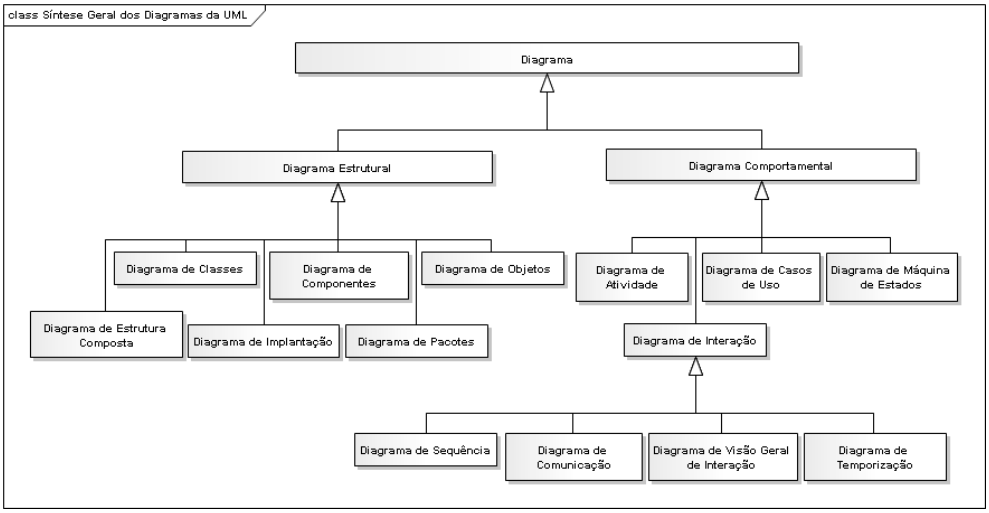

Figura 1.14 – Diagramas da UML.

Como podemos observar, os diagramas estruturais abrangem os diagramas de classes, de estrutura composta, de objetos, de componentes, de implantação e de pacotes, enquanto os comportamentais englobam os de casos de uso, atividade, máquina de estados, sequência, comunicação, visão geral de interação e tempo, sendo que os últimos quatro correspondem aos diagramas da subdivisão de diagramas de interação.

1.5 Ferramentas CASE Baseadas na Linguagem UML

Ferramentas CASE (Computer-Aided Software Engineering ou Engenharia de Software Auxiliada por Computador) são softwares que, de alguma maneira, colaboram para a execução de uma ou mais atividades realizadas durante o processo de engenharia de software. A maioria das ferramentas CASE atuais suporta a UML, sendo essa, em geral, sua principal característica. Entre as diversas ferramentas existentes hoje no mercado podemos citar:

- **Enterprise Architect** – é uma ferramenta muito fácil de utilizar. Embora não seja livre nem ofereça uma edição para a comunidade, é uma das ferramentas que mais oferecem recursos compatíveis com a UML em sua última versão. Apesar de não dispor de uma edição para a comunidade, a

Sparx Systems, a empresa que produz a Enterprise Architect, disponibiliza uma versão trial, que pode ser utilizada por cerca de 60 dias, no site www.sparxsystems.com.au. Praticamente todos os exemplos apresentados neste livro foram produzidos por meio dessa ferramenta.

- **Visual Paradigm for UML ou VP-UML** – a ferramenta pode ser encontrada no site www.visual-paradigm.com e oferece uma edição para a comunidade, ou seja, uma versão da ferramenta que pode ser baixada gratuitamente de sua página. Logicamente, a edição para a comunidade não suporta todos os serviços e opções disponíveis nas suas versões standard ou professional. No entanto, para quem deseja praticar a UML, a edição para a comunidade é uma boa alternativa, apresentando um ambiente amigável e de fácil compreensão. Além disso, a Visual-Paradigm oferece ainda uma cópia acadêmica da versão standard para instituições de ensino superior, que podem consegui-la solicitando-a na própria página da empresa. Tão logo a Visual-Paradigm comprovar a veracidade das informações fornecidas pela instituição, ela enviará uma licença de um ano para uso pelos professores e seus alunos. A licença precisa ser renovada anualmente.
- **Poseidon for UML** – esta ferramenta também tem uma edição para a comunidade, apresentando bem menos restrições que a edição para a comunidade da Visual-Paradigm, mas a interface da Poseidon é sensivelmente inferior à VP-UML, além de apresentar um desempenho um pouco inferior, embora ambas as ferramentas tenham sido desenvolvidas em Java. Uma cópia da Poseidon for UML pode ser adquirida no site www.gentleware.com.
- **ArgoUML** – trata-se de uma ferramenta um tanto limitada, e sua interface não é das mais amigáveis e intuitivas. Porém, apresenta uma característica bastante interessante e atrativa: é totalmente livre. O projeto ArgoUML constitui-se em um projeto acadêmico, no qual os códigos-fonte dessa ferramenta podem ser baixados e utilizados até mesmo para o desenvolvimento de ferramentas comercias, como foi o caso da Poseidon for UML. Os usuários dessa ferramenta podem perceber muitas semelhanças entre as duas ferramentas, mas a Poseidon tem uma interface muito melhor e é, em geral, superior à ArgoUML. O projeto de código aberto ArgoUML exige apenas que quaisquer empresas que utilizarem seus códigos-fonte como base para uma nova ferramenta disponibilizem uma edição para a comunidade gratuitamente. Uma cópia da ArgoUML pode ser encontrada no site www.argouml.tigris.org.