UML
Uma Abordagem Prática

Gilleanes T. A. Guedes

Novatec Editora

Capítulo 1

Introdução à UML

A UML (Unified Modeling Language ou Linguagem de Modelagem Unificada) é uma linguagem visual utilizada para modelar sistemas computacionais por meio do paradigma de Orientação a Objetos. Essa linguagem tornou-se, nos últimos anos, a linguagem padrão de modelagem de software adotada internacionalmente pela indústria de Engenharia de Software.

Deve ficar bem claro, no entanto, que a UML não é uma linguagem de programação e sim uma linguagem de modelagem, cujo objetivo é auxiliar os engenheiros de software a definir as características do software, 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. Todas essas características são definidas por meio da UML antes do software começar a ser realmente desenvolvido.

1.1 Breve Histórico da UML

A UML surgiu da união de três metodologias de modelagem: o método de Booch, o método OMT (Object Modeling Technique) de Jacobson e o método OOSE (Object-Oriented Software Engineering) de Rumbaugh. Estas eram, até meados da década de 90, as três metodologias de modelagem orientada a objetos mais populares entre os profissionais da área de desenvolvimento de software. A união dessas metodologias contou com o amplo apoio da Rational Software, que incentivou e financiou a união das três metodologias.

O esforço inicial do projeto começou com a união do método de Booch com o método 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, diversas grandes empresas atuantes na área de modelagem e desenvolvimento de software passaram a contribuir com o projeto, fornecendo sugestões para melhorar e ampliar a linguagem. Finalmente a UML foi adotada pela OMG (Object Management Group ou Grupo de Gerenciamento de Objetos) em 1997, como uma linguagem padrão de modelagem. Até pouco tempo atrás, a UML encontrava-se na versão 1.5, tendo sido recentemente substituída pela versão 2.0. Esta nova versão traz grandes novidades em relação à estrutura geral da linguagem principalmente com relação a abordagem de quatro camadas e a possibilidade de se desenvolver "perfis" particulares a partir da UML, conforme detalharemos neste livro. A documentação oficial da UML pode ser consultada no site da OMG em www.omg.com.

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 primeiramente desenvolver um projeto muito bem elaborado, cujos cálculos têm que 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 necessários 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 podem sê-lo, exceto talvez os extremamente simples.

Na realidade, por mais simples que seja, todo e qualquer sistema deve ser modelado antes de se iniciar a implementação, entre outras coisas, por que os sistemas de informação freqüentemente costumam possuir a propriedade de "crescer", isto é, aumentar em tamanho, complexidade e abrangência. Muitos profissionais costumam afirmar que sistemas de informação são "vivos", por que nunca estão completamente finalizados. Na verdade, o termo correto seria "dinâmicos", por que normalmente os sistemas de informação estão em constante mudança. Essas mudanças se devem a diversos fatores, como por exemplo:

- Os clientes desejarem constantemente modificações ou melhorias no sistema;

- O mercado estar sempre mudando, o que força a adoção de novas estratégias por parte das empresas e conseqüentemente de seus sistemas;
- O governo freqüentemente promulgar novas leis e criar novos impostos e alíquotas ou ainda modificar as leis, impostos e alíquotas já existentes, o que acarreta a manutenção no software.

Dessa forma, um sistema de informação precisa possuir uma documentação extremamente detalhada, precisa e atualizada para que possa ser mantido com facilidade, rapidez e de maneira correta, 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 discutiremos no decorrer das próximas seções.

1.2.1 Levantamento e Análise de Requisitos

Uma das primeiras fases de engenharia de um software consiste no Levantamento de Requisitos (as outras etapas, sugeridas por muitos autores, são: Análise de Requisitos, Projeto, que se constitui na principal etapa da modelagem, Codificação, Testes e Implantação). Nesta etapa, o engenheiro de software busca compreender as necessidades do usuário e o que ele deseja que o sistema a ser desenvolvido realize. Isto é feito principalmente por meio de entrevistas, onde o engenheiro de software 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 informações deverão ser fornecidas e qual o nível de desempenho exigido do software.

Logo após o Levantamento dos Requisitos, passa-se à fase em que as necessidades apresentadas pelo cliente são analisadas; esta etapa é conhecida como Análise de Requisitos, onde 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.

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 desejos dos clientes e transformá-los em conceitos concretos e inteligíveis.

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 corretamente, verificando se algum tópico deixou de ser abordado, determinando se algum item 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. Estas 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 grandes atrasos no desenvolvimento do software, sendo por vezes necessário remodelar totalmente o projeto.

Além do problema de comunicação, outro grande problema encontrado durante as entrevistas consiste no fato de que, na grande 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 muitas características e funções do sistema que o cliente não sabia como formular ou sequer havia imaginado. Na realidade, na maioria das vezes, os engenheiros de software precisam reestruturar o modo como as informações são geridas e utilizadas pela empresa e apresentar maneiras de combiná-las e apresentá-las de modo 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 e é preciso um grande 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, neste último caso é preciso trabalhar bastante o aspecto social da implantação de um sistema informatizado na empresa, porque 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.

1.2.2 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 este 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 e 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 mesmo anos de desenvolvimento, descubra-se, ao implantar o sistema, que o software não atende completamente as necessidades do cliente devido principalmente a falhas de comunicação durante as entrevistas iniciais.

Nos dias de hoje, 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), como 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 também permitirem modificar essas 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, estas ferramentas permitem ao usuário mudar a posição dos componentes após terem sido colocados no formulário simplesmente selecionando o componente com o mouse e o arrastando para a posição desejada.

Este 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 apresentá-lo novamente 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 podem ser sanadas, 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 o mesmo, 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 se encontra em um estágio bastante avançado de desenvolvimento. Com freqüência ocorre do 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 e para ele absurdos, já que o sistema lhe foi apresentado já funcionando, necessitando de alguns poucos ajustes. Por esse motivo 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.

1.2.3 Prazos e Custos

Em seguida existe 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 isto por sua vez ajuda em muito a determinar os prazos finais em que o software será entregue, no entanto, é preciso possuir 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.

Porém, mesmo com o auxílio desta documentação, ainda é muito difícil estipular uma data exata, o máximo que se pode conseguir é apresentar uma data aproximada, baseada na experiência documentada de desenvolvimento de outros sistemas de informação. Assim é recomendável acrescentar alguns meses à data de entrega, o que serve como uma 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 possuir registros das datas de início e término de cada projeto concluído anteriormente, bem como o custo real de desenvolvimento que estes projetos acarretaram, envolvendo inclusive os custos com manutenção, além do 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 possua documentação histórica de projetos anteriores, dificilmente será capaz de apresentar uma estimativa correta de prazos e custos, principalmente por que 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 pedido 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 diversos profissionais ocupados em projetos que já deveriam estar concluídos, deixando de trabalhar em novos projetos, além da insatisfação dos clientes por não receberem o produto no prazo estimado e a propaganda negativa decorrente disto.

1.2.4 Manutenção

Possivelmente mais importante que todas as outras questões já enunciadas anteriormente, existe a questão 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, porque, como já foi dito anteriormente, as necessidades de uma empresa são dinâmicas e elas mudam constantemente, o que faz surgir novas necessidades que não existiam na época em que o software foi projetado, isso sem falar nas freqüentes mudanças em leis, alíquotas, impostos, taxas ou no formato de notas fiscais, por exemplo. Levando isto em consideração, é extremamente provável que um sistema de informação, por mais bem modelado que esteja, precise sofrer manutenções.

Nesse caso a modelagem não serve unicamente 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 profissionais com estilos de desenvolvimento diferentes e que normalmente não se encontram mais na empresa.

Este tipo de código é conhecido como “código alienígena”. Este termo se refere a códigos que não seguem as regras atuais de desenvolvimento da empresa, não foram modelados e, por conseguinte possuem 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 e cada um deles possuía um estilo de desenvolvimento diferente, ou seja, como se diz no meio de desenvolvimento, o código se encontra “remendado”.

Dessa forma, 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 durante o mesmo, já que é muito comum que, após 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. Estes 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, que não deixa de ser um sistema, muito pelo contrário, é "o" sistema. Na realidade este exemplo não identifica exatamente uma manutenção e sim uma modificação em uma região. Descobri recentemente, ao assistir uma reportagem, que a formiga Saúva vinha se tornando uma praga em algumas regiões do país, por estar se reproduzindo demais. Este crescimento desordenado era causado pelos tratores que, ao arar a terra, destruíam os formigueiros da formiga Lava-Pés, que ficam próximos à superfície, porém não afetavam os formigueiros de Saúvas por estes se encontrarem em um nível mais profundo do solo.

Acontece que as formigas Lava-Pés consumiam os ovos das formigas Saúvas, o que impedia que estas aumentassem demais. Dessa maneira, a diminuição das formigas Lava-Pés resultou no crescimento desordenado das Saúvas. Isso é um exemplo de manutenção com efeito colateral na vida real. Neste 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 formiga Saúva por conseguinte não se tornaria uma praga.

1.2.5 Documentação Histórica

Finalmente existe a questão da perspectiva histórica, ou seja, novamente a já tão falada documentação de software. Neste caso nos referimos 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 ágil?
- As metodologias atualmente 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 anteriormente 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.

Estas 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 uma 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. Neste 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 disto 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 ótica, 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 projetar-se uma construção, esta não possui apenas uma planta, mas sim diversas, enfocando o projeto de construção do prédio sob diversas 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 do sistema para que este funcione adequadamente.

1.4 Rápido Resumo dos Diagramas da UML

A seguir descreveremos rapidamente cada um dos diagramas oferecidos pela UML, destacando suas principais características. Convém alertar que a nova versão UML 2.0 recentemente lançada criou três novos diagramas que não são encontrados nas versões anteriores, além de "promover" dois sub-diagramas a diagramas independentes, bem como trocar a nomenclatura de outros dois diagramas que já existiam nas versões anteriores da linguagem.

1.4.1 Diagrama de Casos de Uso

O Diagrama de Casos de Uso é o diagrama mais geral e informal da UML, sendo 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 idéia 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 opções, que o sistema disponibilizará aos atores, conhecidas neste diagrama como Casos de Uso. Veja na figura 1.1 um exemplo desse diagrama.

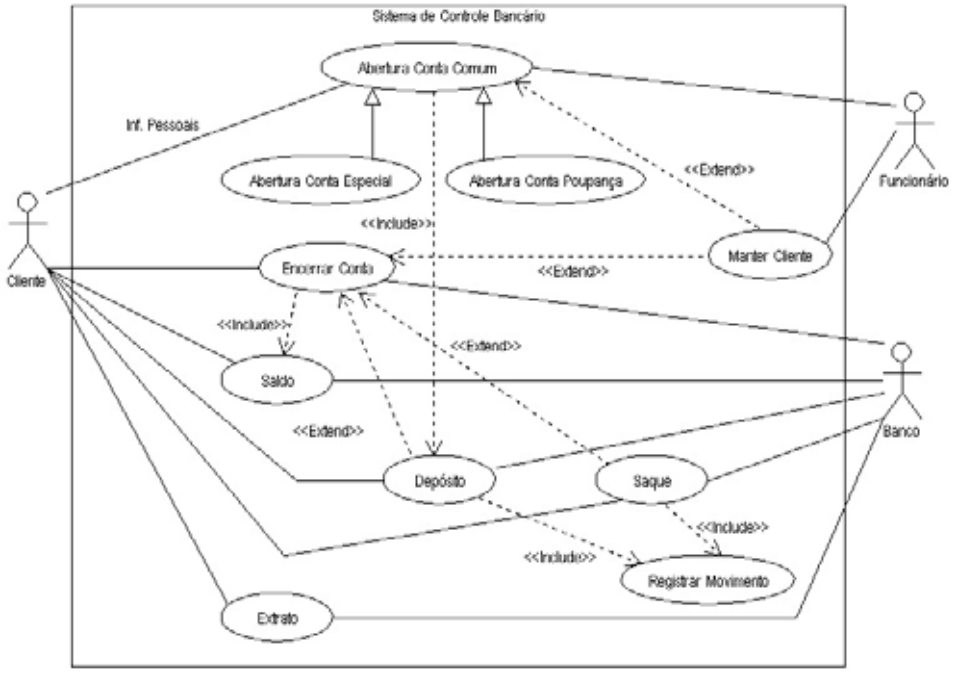

Figura 1.1 – Exemplo de Diagrama de Casos de Uso.

1.4.2 Diagrama de Classes

O Diagrama de Classes é o diagrama mais utilizado e o mais importante da UML, servindo de apoio para a maioria dos outros diagramas. Como o próprio nome diz, define a estrutura das classes utilizadas pelo sistema, determinando os atributos e métodos possuídos por cada classe, além de estabelecer como as classes se relacionam e trocam informações entre si. Veja na figura 1.2 um exemplo desse diagrama.

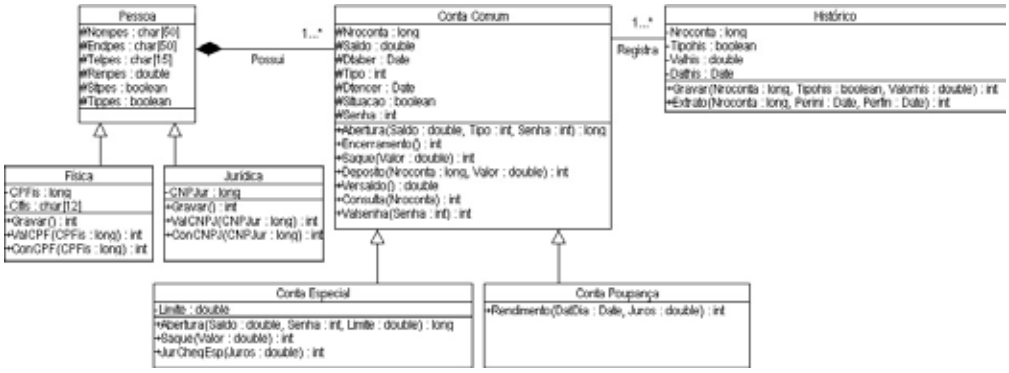

Figura 1.2 – Exemplo de Diagrama de Classes.

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, sendo bastante dependente deste. O Diagrama de Objetos fornece uma visão dos valores armazenados pelos objetos de um Diagrama de Classes em um determinado momento da execução de um processo do software. Este foi um dos diagramas tornados independentes pela UML 2, apesar de já existir anteriormente, era considerado apenas uma extensão do Diagrama de Classes. A figura 1.3 apresenta um exemplo de Diagrama de Objetos.

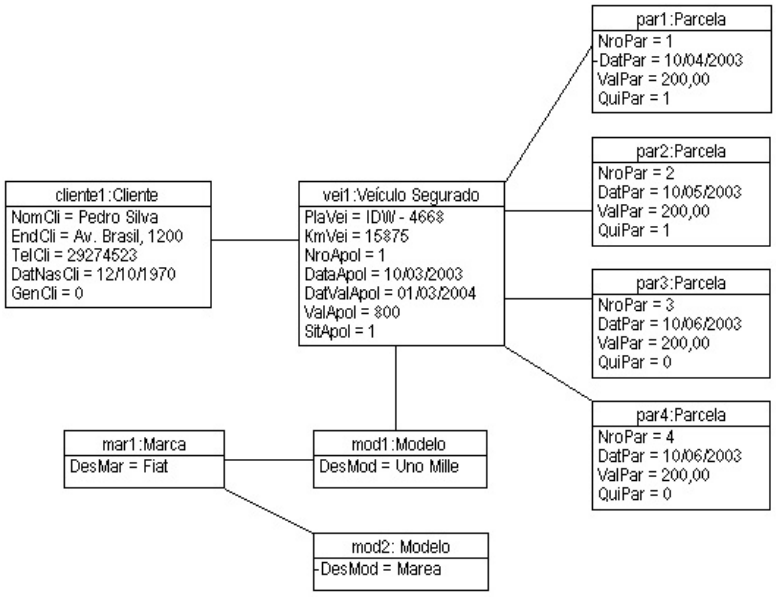

Figura 1.3 – Exemplo de Diagrama de Objetos.

1.4.4 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 onde um conjunto de instâncias cooperam entre si para realizar uma tarefa. Este é um dos três novos diagramas propostos pela UML 2. A figura 1.4 apresenta m exemplo de Diagrama de Estrutura Composta.

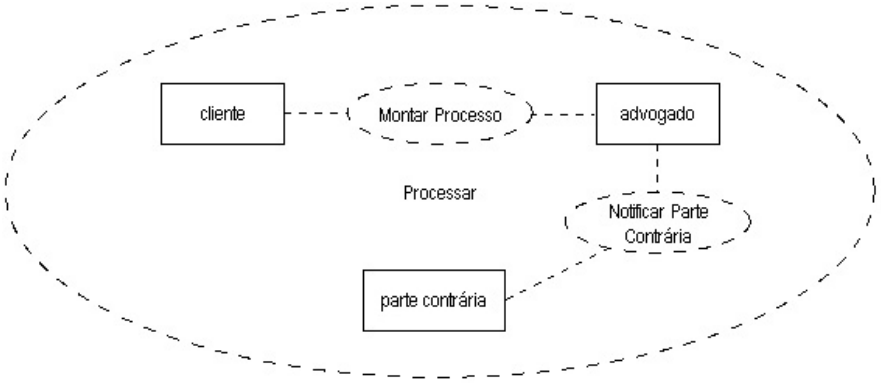

Figura 1.4 – Exemplo de Diagrama de Estrutura Composta.

1.4.5 Diagrama de Seqüência

O Diagrama de Seqüência 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 apóia-se no Diagrama de Classes para determinar os objetos das classes envolvidas em um processo. Um Diagrama de Seqüência costuma identificar o evento gerador do processo modelado, bem como, o ator responsável por este 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.

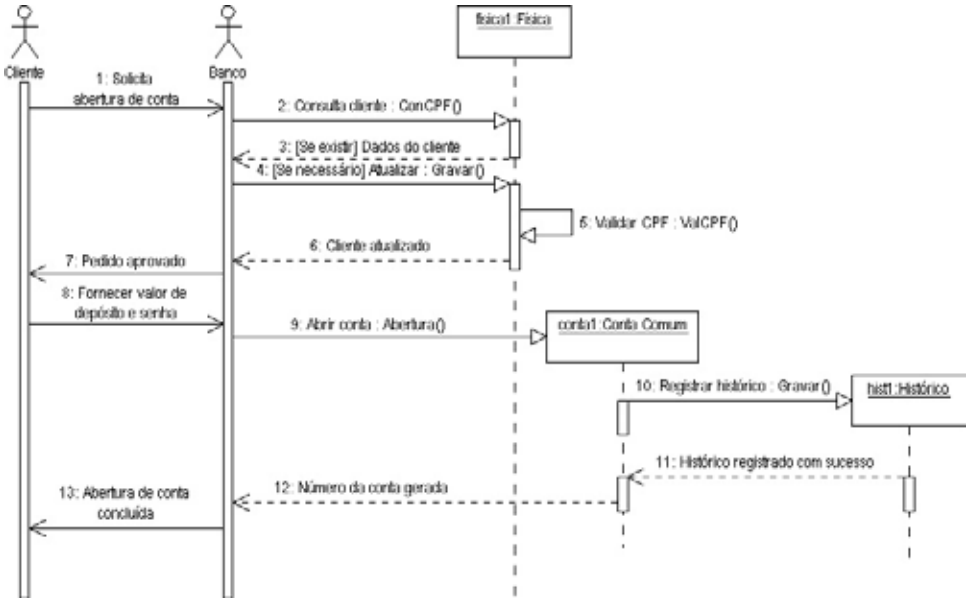

Figura 1.5 – Exemplo de Diagrama de Seqüência.

1.4.6 Diagrama de Colaboração (Comunicação na UML 2)

O Diagrama de Comunicação era conhecido como Diagrama de Colaboração até a versão 1.5 da UML, tendo seu nome modificado para Diagrama de Comunicação a partir da versão 2.0. Esse diagrama está amplamente associado ao Diagrama de Seqüência, na verdade, um complementa o outro. As informações mostradas no Diagrama de Comunicação são, com freqüência, praticamente as mesmas apresentadas no Diagrama de Seqüência, porém com um enfoque diferente, visto que este diagrama não se preocupa com a temporalidade do processo, concentrando-se em como os objetos estão vinculados e quais mensagens trocam entre si durante o processo. A figura 1.6 apresenta um exemplo de Diagrama de Comunicação.

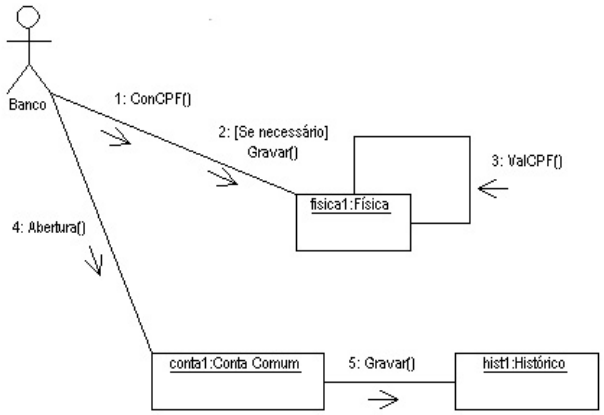

Figura 1.6 – Exemplo de Diagrama de Comunicação.

1.4.7 Diagrama de Gráfico de Estados (Máquina de Estados na UML 2)

O Diagrama de Máquina de Estados era conhecido nas versões anteriores da linguagem como Diagrama de Gráfico de Estados ou simplesmente como Diagrama de Estados, tendo assumido esta nova nomenclatura a partir da versão 2.0. Este diagrama procura acompanhar as mudanças sofridas por um objeto dentro de um determinado processo. Como o Diagrama de Seqüência, o Diagrama de Máquina de Estados muitas vezes baseia-se em um Caso de Uso descrito em um Diagrama de Casos de Uso e apóia-se no Diagrama de Classes. O Diagrama de Máquina de Estados é utilizado normalmente para acompanhar os estados por que passa uma instância de uma classe, no entanto pode ser utilizado para representar os estados de um caso de uso ou mesmo os estados gerais de um sub-sistema ou de um sistema completo. A figura 1.7 apresenta um exemplo de Diagrama de Máquina de Estados.

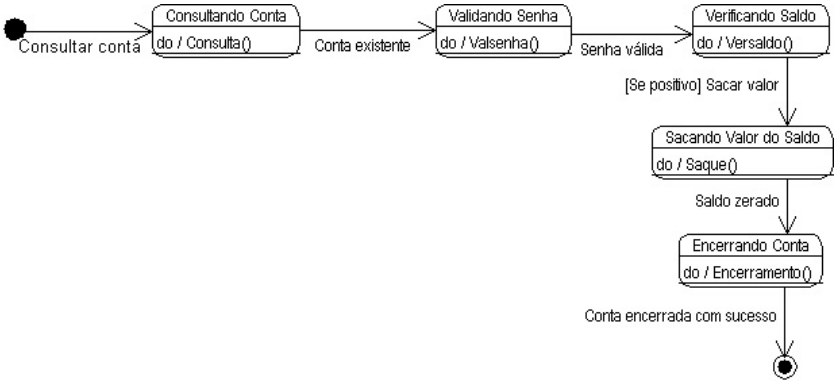

Figura 1.7 – Exemplo de Diagrama de Gráfico de Estados.

1.4.8 Diagrama de Atividades

O Diagrama de Atividades era considerado um caso especial do antigo Diagrama de Gráfico de Estados, atualmente conhecido como Diagrama de Máquina de Estados, conforme foi descrito na seção anterior. A partir da UML 2.0 o Diagrama de Atividades foi considerado independente do Diagrama de Máquina de Estados. Esse diagrama preocupa-se em descrever os passos a serem percorridos para a conclusão de uma atividade específica, muitas vezes representada por um método com um certo grau de complexidade e não de um processo completo como é o caso dos Diagramas de Seqüência ou Colaboração, embora também possa ser utilizado para este fim. O Diagrama de Atividades concentra-se na representação do fluxo de controle de um atividade. A figura 1.8 apresenta um exemplo desse diagrama.

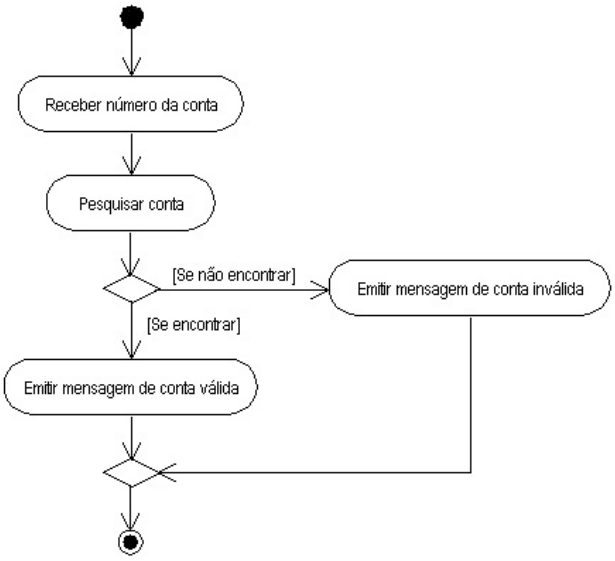

Figura 1.8 – Exemplo de Diagrama de Atividades.

1.4.9 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 este for ser implementado em termos de módulos de código-fonte, bibliotecas, formulários, arquivos de ajuda, módulos executáveis etc. e determina como esses componentes estarão estruturados e interagirão para que o sistema funcione de maneira adequada. A figura 1.9 apresenta um exemplo desse diagrama.

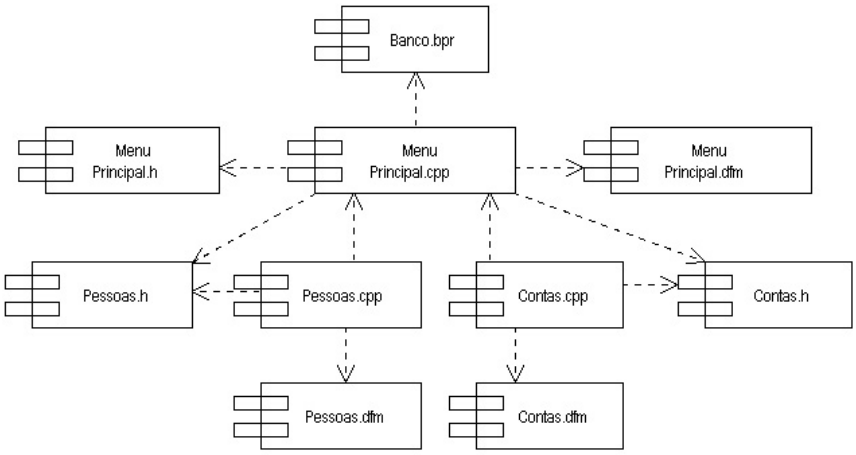

Figura 1.9 – Exemplo de Diagrama de Componentes.

1.4.10 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. O Diagrama de Componentes e de Implantação são bastante associados, podendo ser representados em separado ou em conjunto. A figura 1.10 apresenta um exemplo desse diagrama.

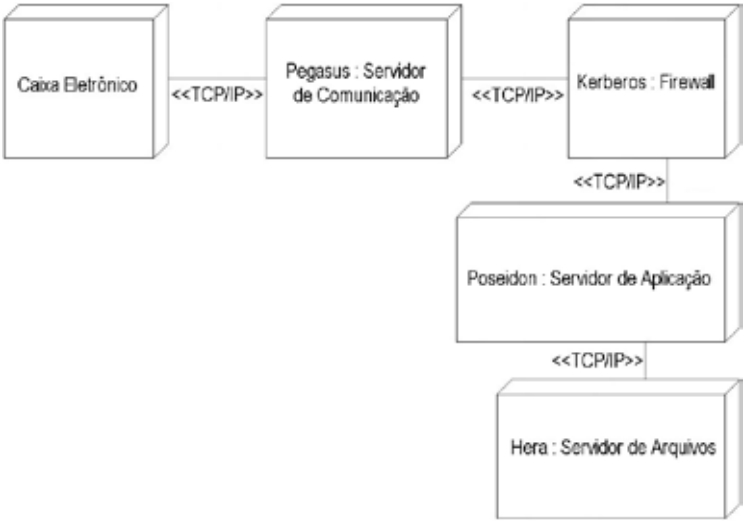

Figura 1.10 – Exemplo de Diagrama de Implantação.

1.4.11 Diagrama de Pacotes

O Diagrama de Pacotes já existia nas versões anteriores da UML, mas foi considerado totalmente independente a partir da UML 2. Tem por objetivo representar os sub-sistemas englobados por um sistema de forma a determinar as partes que o compõem. Pode ser utilizado de maneira independente ou associado com outros diagramas conforme será visto ao longo deste livro. A maioria das figuras iniciais do capítulo 11 são Diagramas de Pacotes utilizados para demonstrar a estrutura a linguagem. A figura 1.11 apresenta um exemplo desse diagrama.

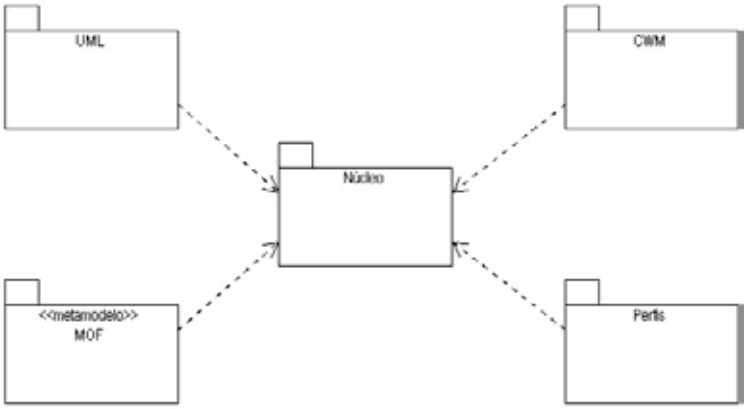

Figura 1.11 – Exemplo de Diagrama de Pacotes.

1.4.12 Diagrama de Interação Geral

O Diagrama de Interação Geral é uma variação do Diagrama de Atividades que fornece uma visão geral dentro de um sistema ou processo de negócio. Esse diagrama passou a existir somente a partir da UML 2. A figura 1.12 apresenta um exemplo desse diagrama.

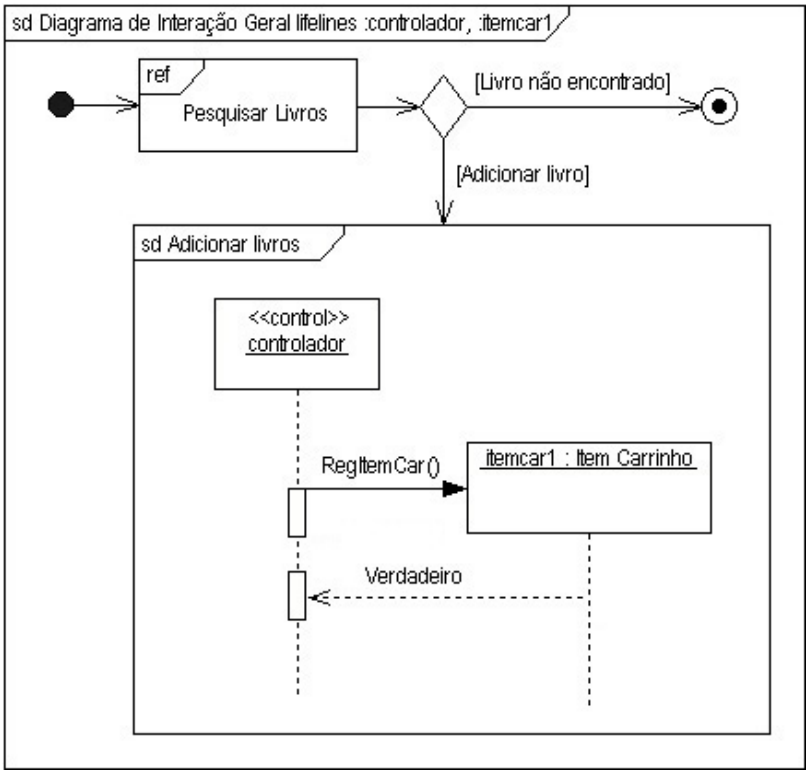

Figura 1.12 – Exemplo de Diagrama de Interação Geral.

1.4.13 Diagrama de Tempo

O Diagrama de Tempo descreve a mudança no estado ou condição de uma instância de uma classe ou seu papel durante um tempo. Tipicamente utilizada para demonstrar a mudança no estado de um objeto no tempo em resposta a eventos externos. Esse é o terceiro diagrama criado a partir da nova versão da linguagem. 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 2.0 dividem-se em Diagramas Estruturais e Diagramas Comportamentais, sendo que estes últimos possuem ainda uma sub-divisão representada pelos Diagramas de Interação, conforme pode ser verificado na figura 1.14.

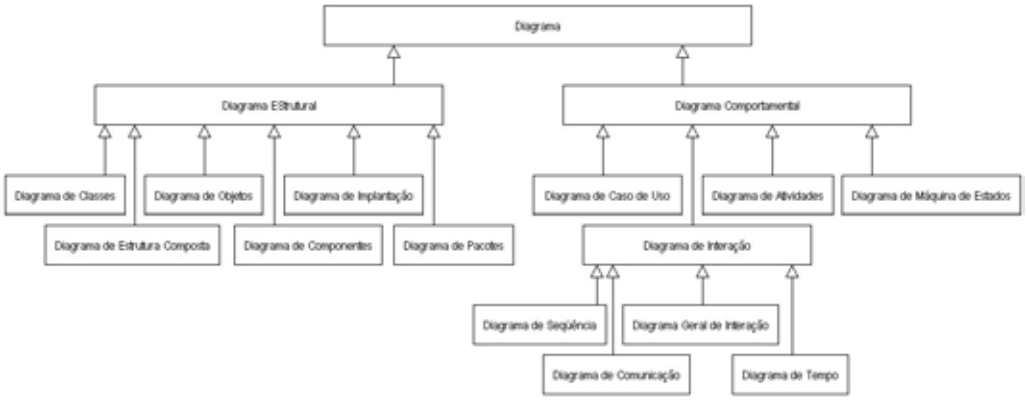

Figura 1.14 – Diagramas da UML.

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 Diagramas Comportamentais englobam os Diagramas de Casos de Uso, Atividade, Máquina de Estados, Seqüência, Comunicação, de Interação Geral e de Tempo, sendo que estes últimos 4 correspondem aos diagramas da sub-divisão 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 suportam a UML, sendo esta, em geral, sua principal característica. Entre as diversas ferramentas existentes atualmente no mercado podemos citar:

- Rational Rose – É a ferramenta mais conhecida e possivelmente a mais utilizada atualmente. É uma das ferramentas mais completas disponíveis no mercado, é totalmente orientada à UML, no entanto suporta igualmente os métodos de Booch e OMT. A Rational Software foi a empresa que incentivou a criação da UML e foi a primeira lançar uma ferramenta CASE baseada nesta linguagem. Uma cópia da ferramenta Rational Rose pode ser adquirida no site www.rational.com.

- Visual Paradigm for UML ou VP-UML – Esta ferramenta pode ser encontrada no site www.visual-paradigm.com e oferece inclusive 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 versões Standard ou Professional da ferramenta, 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. Atualmente a VP-UML encontra-se na versão 5.0.

- Poseidon for UML – Esta ferramenta também possui uma edição para a comunidade, apresentando bem menos restrições que a edição para a comunidade da Visual-Paradigm, no entanto a interface da Poseidon é sensivelmente inferior à VP-UML, além de apresentar um desempenho um pouco inferior também, embora ambas as ferramentas tenham sido desenvolvidas em Java. Uma cópia da Poseidon for UML pode ser adquirida no site www.gentleware.com. Atualmente a Poseidon encontra-se na versão 3.1.

- ArgoUML – É 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, onde os códigos-fonte dessa ferramenta podem até mesmo ser baixados e utilizados 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 possui uma interface muito melhor e é, em geral, superior à ArgoUML. No entanto o projeto de código aberto ArgoUML exige que quaisquer empresas que utilizarem seus códigos 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. A ArgoUML encontra-se na versão 0.19.1 atualmente.

- Enterprise Architect – Embora a ferramenta Enterprise Architect não ofereça uma edição para a comunidade como as anteriores, ela é uma das ferramentas que mais oferecem recursos compatíveis com a UML 2. 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.