UNIFIED MODELING LANGUAGE

Linguagem de Modelagem Unificada

Em Português

1. **Introdução**
2. **Desenvolvimento de Softwares orientado a objetos**
3. **UML – A unificação dos métodos para a criação de um novo padrão**
4. **Uso da UML**
5. **Fases do Desenvolvimento de um Sistema em UML**
    1. Análise de Requisitos
    2. Análise
    3. Design (Projeto)
    4. Programação
    5. Testes
6. **A Notação da Linguagem de Modelagem Unificada – UML**
7. **Visões**
8. **Modelos de Elementos**
    1. Classes
    2. Objetos
    3. Estados
    4. Pacotes
    5. Componentes
    6. Relacionamentos
    7. Mecanismos Gerais
9. **Diagramas**
    1. Diagrama Use-Case
    2. Diagrama de Classes
    3. Diagrama de Objetos
    4. Diagrama de Estado
    5. Diagrama de Sequência
    6. Diagrama de Colaboração
    7. Diagrama de Atividade
    8. Diagrama de Componente
    9. Diagrama de Execução
10. **Um processo para utilizar a UML**
11. **O Futuro da UML**
12. **Um estudo de caso em UML - Sistema de Controle de Contas Correntes, Poupança e Aplicações Pré-fixadas**
    1. Análise de Requisitos
    2. Análise
    3. Design
    4. Implementação
    5. Testes
13. **Conclusão**

1. Introdução

O grande problema do desenvolvimento de novos sistemas utilizando a orientação a objetos nas fases de análise de requisitos, análise de sistemas e design é que não existe uma notação padronizada e realmente eficaz que abranja qualquer tipo de aplicação que se deseje. Cada simbologia existente possui seus próprios conceitos, gráficos e terminologias, resultando numa grande confusão, especialmente para aqueles que querem utilizar a orientação a objetos não só sabendo para que lado aponta a seta de um relacionamento, mas sabendo criar modelos de qualidade para ajudá-los a construir e manter sistemas cada vez mais eficazes.

Quando a "Unified Modeling Language" (UML) foi lançada, muitos desenvolvedores da área da orientação a objetos ficaram entusiasmados já que essa padronização proposta pela UML era o tipo de força que eles sempre esperaram.

A UML é muito mais que a padronização de uma notação. É também o desenvolvimento de novos conceitos não normalmente usados. Por isso e muitas outras razões, o bom entendimento da UML não é apenas aprender a simbologia e o seu significado, mas também significa aprender a modelar orientado a objetos no estado da arte.

UML foi desenvolvida por Grady Booch, James Rumbaugh, e Ivar Jacobson que são conhecidos como "os três amigos". Eles possuem uma extenso conhecimento na área de modelagem orientado a objetos já que as três mais conceituadas metodologias de modelagem orientado a objetos foram eles que desenvolveram e a UML é a junção do que havia de melhor nestas três metodologias adicionado novos conceitos e visões da linguagem. Veremos características de cada uma destas metodologias no desenvolver deste trabalho.

Veremos como a UML aborda o caráter estático e dinâmico do sistema a ser analisado levando em consideração, já no período de modelagem, todas as futuras características do sistema em relação à utilização de "packages" próprios da linguagem a ser utilizada, utilização do banco de dados bem como as diversas especificações do sistema a ser desenvolvido de acordo com as métricas finais do sistema.

Não é intuito deste trabalho definir e explicar os significados de classes, objetos, relacionamentos, fluxos, mensagens e outras entidades comuns da orientação a objetos, e sim apresentarmos como essas entidades são criadas, simbolizadas, organizadas e como serão utilizadas dentro de um desenvolvimento utilizando a UML.

2. Desenvolvimento de Softwares orientado a objetos

Os conceitos da orientação a objetos já vêm sido discutidos há muito tempo, desde o lançamento da 1ª linguagem orientada a objetos, a SIMULA. Vários "papas" da engenharia de software mundial como Peter Coad, Edward Yourdon e Roger Pressman abordaram extensamente a análise orientada a objetos como realmente um grande avanço no desenvolvimento de sistemas. Mas mesmo assim, eles citam que não existe (ou que não existia no momento de suas publicações) uma linguagem que possibilitasse o desenvolvimento de qualquer software utilizando a análise orientada a objetos.

Os conceitos que Coad, Yourdon, Pressman e tantos outros abordaram, discutiram e definiram em suas publicações foram que:

- A orientação a objetos é uma tecnologia para a produção de modelos que especifiquem o domínio do problema de um sistema.

- Quando construídos corretamente, sistemas orientados a objetos são flexíveis a mudanças, possuem estruturas bem conhecidas e provêm a oportunidade de criar e implementar componentes totalmente reutilizáveis.

- Modelos orientado a objetos são implementados convenientemente utilizando uma linguagem de programação orientada a objetos. A engenharia de software orientada a objetos é muito mais que utilizar mecanismos de sua linguagem de programação, é saber utilizar da melhor forma possível todas as técnicas da modelagem orientada a objetos.

- A orientação a objetos não é só teoria, mas uma tecnologia de eficiência e qualidade comprovadas usada em inúmeros projetos e para construção de diferentes tipo de sistemas.

A orientação a objetos requer um método que integre o processo de desenvolvimento e a linguagem de modelagem com a construção de técnicas e ferramentas adequadas

3. UML – A unificação dos métodos para a criação de um novo padrão

A UML é uma tentativa de padronizar a modelagem orientada a objetos de uma forma que qualquer sistema, seja qual for o tipo, possa ser modelado corretamente, com consistência, fácil de se comunicar com outras aplicações, simples de ser atualizado e compreensível.

Existem várias metodologias de modelagem orientada a objetos que até o surgimento da UML causavam uma guerra entre a comunidade de desenvolvedores orientado a objetos. A UML acabou com esta guerra trazendo as melhores idéias de cada uma destas metodologias, e mostrando como deveria ser a migração de cada uma para a UML.

Falaremos sobre algumas das principais metodologias que se tornaram populares nos anos 90:

- Booch – O método de Grady Booch para desenvolvimento orientado a objetos está disponível em muitas versões. Booch definiu a noção de que um sistema é analisado a partir de um número de visões, onde cada visão é descrita por um número de modelos e diagramas. O Método de Booch trazia uma simbologia complexa de ser desenhada a mão, continha também o processo pelo qual sistemas são analisados por macro e micro visões.

- OMT – Técnica de Modelagem de Objetos (Object Modelling Technique) é um método desenvolvido pela GE (General Electric) onde James Rumbaugh trabalhava. O método é especialmente voltado para o teste dos modelos, baseado nas especificações da análise de requisitos do sistema. O modelo total do sistema baseado no método OMT é composto pela junção dos modelos de objetos, funcional e use-cases.

- OOSE/Objectory – Os métodos OOSE e o Objectory foram desenvolvidos baseados no mesmo ponto de vista formado por Ivar Jacobson. O método OOSE é a visão de Jacobson de um método orientado a objetos, já o Objectory é usado para a construção de sistemas tão diversos quanto eles forem. Ambos os métodos são baseados na utilização de use-cases, que definem os requisitos iniciais do sistema, vistos por um ator externo. O método Objectory também foi adaptado para a engenharia de negócios, onde é usado para modelar e melhorar os processos envolvidos no funcionamento de empresas.

Cada um destes métodos possui sua própria notação (seus próprios símbolos para representar modelos orientado a objetos), processos (que atividades são desenvolvidas em diferentes partes do desenvolvimento), e ferramentas (as ferramentas CASE que suportam cada uma destas notações e processos).

Diante desta diversidade de conceitos, "os três amigos", Grady Booch, James Rumbaugh e Ivar Jacobson decidiram criar uma Linguagem de Modelagem Unificada. Eles disponibilizaram inúmeras versões preliminares da UML para a comunidade de desenvolvedores e a resposta incrementou muitas novas idéias que melhoraram ainda mais a linguagem.

Os objetivos da UML são:

- A modelagem de sistemas (não apenas de software) usando os conceitos da orientação a objetos;

- Estabelecer uma união fazendo com que métodos conceituais sejam também executáveis;

- Criar uma linguagem de modelagem usável tanto pelo homem quanto pela máquina.

A UML está destinada a ser dominante, a linguagem de modelagem comum a ser usada nas indústrias. Ela está totalmente baseada em conceitos e padrões extensivamente testados provenientes das metodologias existentes anteriormente, e também é muito bem documentada com toda a especificação da semântica da linguagem representada em meta-modelos.

4. Uso da UML

A UML é usada no desenvolvimento dos mais diversos tipos de sistemas. Ela abrange sempre qualquer característica de um sistema em um de seus diagramas e é também aplicada em diferentes fases do desenvolvimento de um sistema, desde a especificação da análise de requisitos até a finalização com a fase de testes.

O objetivo da UML é descrever qualquer tipo de sistema, em termos de diagramas orientado a objetos. Naturalmente, o uso mais comum é para criar modelos de sistemas de software, mas a UML também é usada para representar sistemas mecânicos sem nenhum software. Aqui estão alguns tipos diferentes de sistemas com suas características mais comuns:

- Sistemas de Informação: Armazenar, pesquisar, editar e mostrar informações para os usuários. Manter grandes quantidades de dados com relacionamentos complexos, que são guardados em bancos de dados relacionais ou orientados a objetos.

- Sistemas Técnicos: Manter e controlar equipamentos técnicos como de telecomunicações, equipamentos militares ou processos industriais. Eles devem possuir interfaces especiais do equipamento e menos programação de software de que os sistemas de informação. Sistemas Técnicos são geralmente sistemas real-time.

- Sistemas Real-time Integrados: Executados em simples peças de hardware integrados a telefones celulares, carros, alarmes etc. Estes sistemas implementam programação de baixo nível e requerem suporte real-time.

- Sistemas Distribuídos: Distribuídos em máquinas onde os dados são transferidos facilmente de uma máquina para outra. Eles requerem mecanismos de comunicação sincronizados para garantir a integridade dos dados e geralmente são construídos em mecanismos de objetos como CORBA, COM/DCOM ou Java Beans/RMI.

- Sistemas de Software: Definem uma infra-estrutura técnica que outros softwares utilizam. Sistemas Operacionais, bancos de dados, e ações de usuários que executam ações de baixo nível no hardware, ao mesmo tempo em que disponibilizam interfaces genéricas de uso de outros softwares.

- Sistemas de Negócios: descreve os objetivos, especificações (pessoas, computadores etc.), as regras (leis, estratégias de negócios etc.), e o atual trabalho desempenhado nos processos do negócio.

É importante perceber que a maioria dos sistemas não possuem apenas uma destas características acima relacionadas, mas várias delas ao mesmo tempo. Sistemas de informações de hoje, por exemplo, podem ter tanto características distribuídas como real-time. E a UML suporta modelagens de todos estes tipos de sistemas.

5. Fases do Desenvolvimento de um Sistema em UML

Existem cinco fases no desenvolvimento de sistemas de software: análise de requisitos, análise, design (projeto), programação e testes. Estas cinco fases não devem ser executadas na ordem descrita acima, mas concomitantemente de forma que problemas detectados numa certa fase modifiquem e melhorem as fases desenvolvidas anteriormente de forma que o resultado global gere um produto de alta qualidade e performance. A seguir falaremos sobre cada fase do desenvolvimento de um sistema em UML:

5.1. Análise de Requisitos

Esta fase captura as intenções e necessidades dos usuários do sistema a ser desenvolvido através do uso de funções chamadas "use-cases". Através do desenvolvimento de "use-case", as entidades externas ao sistema (em UML chamados de "atores externos") que interagem e possuem interesse no sistema são modelados entre as funções que eles requerem, funções estas chamadas de "use-cases". Os atores externos e os "use-cases" são modelados com relacionamentos que possuem comunicação associativa entre eles ou são desmembrados em hierarquia. Cada "use-case" modelado é descrito através de um texto, e este especifica os requerimentos do ator externo que utilizará este "use-case". O diagrama de "use-cases" mostrará o que os atores externos, ou seja, os usuários do futuro sistema deverão esperar do aplicativo, conhecendo toda sua funcionalidade sem importar como esta será implementada. A análise de requisitos também pode ser desenvolvida baseada em processos de negócios, e não apenas para sistemas de software.

5.2. Análise

A fase de análise está preocupada com as primeiras abstrações (classes e objetos) e mecanismos que estarão presentes no domínio do problema. As classes são modeladas e ligadas através de relacionamentos com outras classes, e são descritas no Diagrama de Classe. As colaborações entre classes também são mostradas neste diagrama para desenvolver os "use-cases" modelados anteriormente, estas colaborações são criadas através de modelos dinâmicos em UML. Na análise, só serão modeladas classes que pertençam ao domínio principal do problema do software, ou seja, classes técnicas que gerenciem banco de dados, interface, comunicação, concorrência e outros não estarão presentes neste diagrama.

5.3. Design (Projeto)

Na fase de design, o resultado da análise é expandido em soluções técnicas. Novas classes serão adicionadas para prover uma infra-estrutura técnica: a interface do usuário e de periféricos, gerenciamento de banco de dados, comunicação com outros sistemas, dentre outros. As classes do domínio do problema modeladas na fase de análise são mescladas nessa nova infra-estrutura técnica tornando possível alterar tanto o domínio do problema quanto à infra-estrutura. O design resulta no detalhamento das especificações para a fase de programação do sistema.

5.4. Programação

Na fase de programação, as classes provenientes do design são convertidas para o código da linguagem orientada a objetos escolhida (a utilização de linguagens procedurais é extremamente não recomendada). Dependendo da capacidade da linguagem usada, essa conversão pode ser uma tarefa fácil ou muito complicada. No momento da criação de modelos de análise e design em UML, é melhor evitar traduzi-los mentalmente em código. Nas fases anteriores, os modelos criados são o significado do entendimento e da estrutura do sistema, então, no momento da geração do código onde o analista conclua antecipadamente sobre modificações em seu conteúdo, seus modelos não estarão mais demonstrando o real perfil do sistema. A programação é uma fase separada e distinta onde os modelos criados são convertidos em código.

5.5. Testes

Um sistema normalmente é rodado em testes de unidade, integração, e aceitação. Os testes de unidade são para classes individuais ou grupos de classes e são geralmente testados pelo programador. Os testes de integração são aplicados já usando as classes e componentes integrados para se confirmar se as classes estão cooperando uma com as outras como especificado nos modelos. Os testes de aceitação observam o sistema como uma " caixa preta" e verificam se o sistema está funcionando como o especificado nos primeiros diagramas de "use-cases".

O sistema será testado pelo usuário final e verificará se os resultados mostrados estão realmente de acordo com as intenções do usuário final.

<PAROU AQUI>

6. A Notação da Linguagem de Modelagem Unificada – UML

Tendo em mente as cinco fases do desenvolvimento de softwares, as fases de análise de requisitos, análise e design utilizam-se em seu desenvolvimento cinco tipos de visões, nove tipos de diagramas e vários modelos de elementos que serão utilizados na criação dos diagramas e mecanismos gerais que todos em conjunto especificam e exemplificam a definição do sistema, tanto a definição no que diz respeito à funcionalidade estática e dinâmica do desenvolvimento de um sistema.

Antes de abordarmos cada um destes componentes separadamente, definiremos as partes que compõem a UML:

- Visões: As Visões mostram diferentes aspectos do sistema que está sendo modelado. A visão não é um gráfico, mas uma abstração consistindo em uma série de diagramas. Definindo um número de visões, cada uma mostrará aspectos particulares do sistema, dando enfoque a ângulos e níveis de abstrações diferentes e uma figura completa do sistema poderá ser construída. As visões também podem servir de ligação entre a linguagem de modelagem e o método/processo de desenvolvimento escolhido.

- Modelos de Elementos: Os conceitos usados nos diagramas são modelos de elementos que representam definições comuns da orientação a objetos como as classes, objetos, mensagem, relacionamentos entre classes incluindo associações, dependências e heranças.

- Mecanismos Gerais: Os mecanismos gerais provém comentários suplementares, informações, ou semântica sobre os elementos que compõem os modelos; eles provém também mecanismos de extensão para adaptar ou estender a UML para um método/processo, organização ou usuário específico.

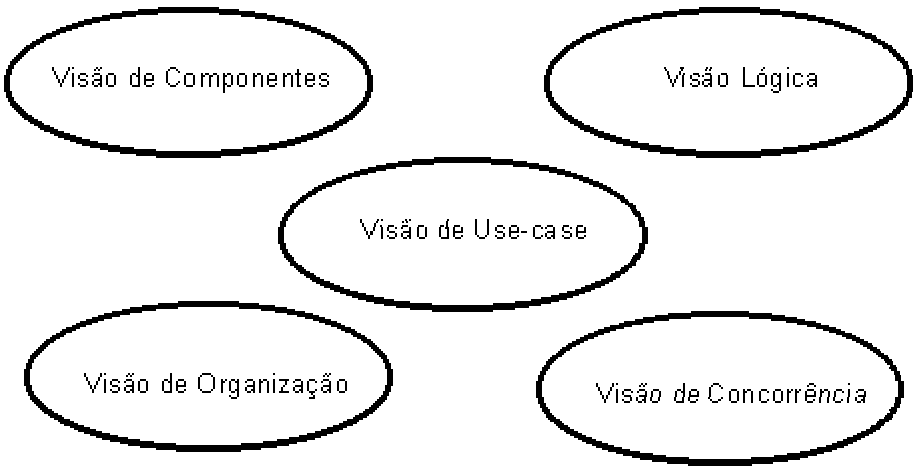
- Diagramas: Os diagramas são os gráficos que descrevem o conteúdo em uma visão. UML possui nove tipo de diagramas que são usados em combinação para prover todas as visões do sistema.

7. Visões

O desenvolvimento de um sistema complexo não é uma tarefa fácil. O ideal seria que o sistema inteiro pudesse ser descrito em um único gráfico e que este representasse por completo as reais intenções do sistema sem ambiguidades, sendo facilmente interpretável. Infelizmente, isso é impossível. Um único gráfico é incapaz de capturar todas as informações necessárias para descrever um sistema.

Um sistema é composto por diversos aspectos: funcional (que é sua estrutura estática e suas interações dinâmicas), não funcional (requisitos de tempo, confiabilidade, desenvolvimento, etc.) e aspectos organizacionais (organização do trabalho, mapeamento dos módulos de código, etc.). Então o sistema é descrito em um certo número de visões, cada uma representando uma projeção da descrição completa e mostrando aspectos particulares do sistema.

Cada visão é descrita por um número de diagramas que contém informações que dão ênfase aos aspectos particulares do sistema. Existe em alguns casos uma certa sobreposição entre os diagramas o que significa que um deste pode fazer parte de mais de uma visão. Os diagramas que compõem as visões contém os modelos de elementos do sistema. As visões que compõem um sistema são:



- Visão "use-case": Descreve a funcionalidade do sistema desempenhada pelos atores externos do sistema (usuários). A visão use-case é central, já que seu conteúdo é base do desenvolvimento das outras visões do sistema. Essa visão é montada sobre os diagramas de use-case e eventualmente diagramas de atividade.

- Visão Lógica: Descreve como a funcionalidade do sistema será implementada. É feita principalmente pelos analistas e desenvolvedores. Em contraste com a visão use-case, a visão lógica observa e estuda o sistema internamente. Ela descreve e especifica a estrutura estática do sistema (classes, objetos, e relacionamentos) e as colaborações dinâmicas quando os objetos enviarem mensagens uns para os outros para realizarem as funções do sistema. Propriedades como persistência e concorrência são definidas nesta fase, bem como as interfaces e as estruturas de classes. A estrutura estática é descrita pelos diagramas de classes e objetos. O modelamento dinâmico é descrito pelos diagramas de estado, sequência, colaboração e atividade.

- Visão de Componentes: É uma descrição da implementação dos módulos e suas dependências. É principalmente executado por desenvolvedores, e consiste nos componentes dos diagramas.

- Visão de concorrência: Trata a divisão do sistema em processos e processadores. Este aspecto, que é uma propriedade não funcional do sistema, permite uma melhor utilização do ambiente onde o sistema se encontrará, se o mesmo possui execuções paralelas, e se existe dentro do sistema um gerenciamento de eventos assíncronos. Uma vez dividido o sistema em linhas de execução de processos concorrentes (threads), esta visão de concorrência deverá mostrar como se dá a comunicação e a concorrência destas threads. A visão de concorrência é suportada pelos diagramas dinâmicos, que são os diagramas de estado, sequência, colaboração e atividade, e pelos diagramas de implementação, que são os diagramas de componente e execução.

- Visão de Organização: Finalmente, a visão de organização mostra a organização física do sistema, os computadores, os periféricos e como eles se conectam entre si. Esta visão será executada pelos desenvolvedores, integradores e testadores, e será representada pelo diagrama de execução.

8. Modelos de Elementos

Os conceitos usados nos diagramas são chamados de modelos de elementos. Um modelo de elemento é definido com a semântica, a definição formal do elemento com o exato significado do que ele representa sem definições duvidosas ou ambíguas e também define sua representação gráfica que é mostrada nos diagramas da UML. Um elemento pode existir em diversos tipos de diagramas, mas existem regras que definem que elementos podem ser mostrados em que tipos de diagramas. Alguns exemplos de modelos de elementos são as classes, objetos, estados, pacotes e componentes. Os relacionamentos também são modelos de elementos, e são usados para conectar outros modelos de elementos entre si. Todos os modelos de elementos serão definidos e exemplificados a seguir.

8.1. Classes

Uma classe é a descrição de um tipo de objeto. Todos os objetos são instâncias de classes, onde a classe descreve as propriedades e comportamentos daquele objeto. Objetos só podem ser instanciados de classes. Usam-se classes para classificar os objetos que identificamos no mundo real. Tomando como exemplo Charles Darwin, que usou classes para classificar os animais conhecidos, e combinou suas classes por herança para descrever a "Teoria da Evolução". A técnica de herança entre classes é também usada em orientação a objetos.

Uma classe pode ser a descrição de um objeto em qualquer tipo de sistema – sistemas de informação, técnicos, integrados real-time, distribuídos, software etc. Num sistema de software, por exemplo, existem classes que representam entidades de software num sistema operacional como arquivos, programas executáveis, janelas, barras de rolagem, etc.

Identificar as classes de um sistema pode ser complicado, e deve ser feito por experts no domínio do problema a que o software modelado se baseia. As classes devem ser retiradas do domínio do problema e serem nomeadas pelo que elas representam no sistema. Quando procuramos definir as classes de um sistema, existem algumas questões que podem ajudar a identificá-las:

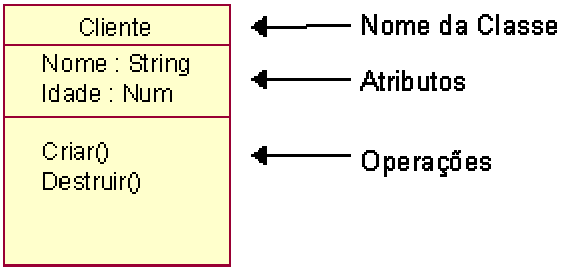
- Existem informações que devem ser armazenadas ou analisadas? Se existir alguma informação que tenha que ser guardada, transformada ou analisada de alguma forma, então é uma possível candidata para ser uma classe.

- Existem sistemas externos ao modelado? Se existir, eles deverão ser vistos como classes pelo sistema para que possa interagir com outros externos.

- Existem classes de bibliotecas, componentes ou modelos externos a serem utilizados pelo sistema modelado? Se sim, normalmente essas classes, componentes e modelos conterão classes candidatas ao nosso sistema.

- Qual o papel dos atores dentro do sistema? Talvez o papel deles possa ser visto como classes, por exemplo, usuário, operador, cliente e daí por diante.

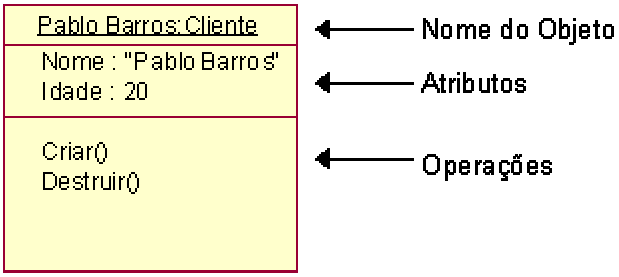
Em UML as classes são representadas por um retângulo dividido em três compartimentos: o compartimento de nome, que conterá apenas o nome da classe modelada, o de atributos, que possuirá a relação de atributos que a classe possui em sua estrutura interna, e o compartimento de operações, que serão o métodos de manipulação de dados e de comunicação de uma classe com outras do sistema. A sintaxe usada em cada um destes compartimentos é independente de qualquer linguagem de programação, embora pode ser usadas outras sintaxes como a do C++, Java, e etc.



8.2. Objetos

Um objeto é um elemento que podemos manipular, acompanhar seu comportamento, criar, destruir etc. Um objeto existe no mundo real. Pode ser uma parte de qualquer tipo de sistema, por exemplo, uma máquina, uma organização, ou negócio. Existem objetos que não encontramos no mundo real, mas que podem ser vistos de derivações de estudos da estrutura e comportamento de outros objetos do mundo real.

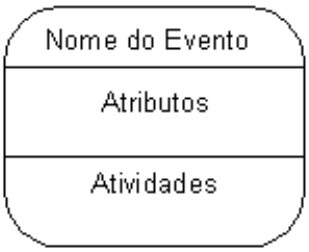
Em UML um objeto é mostrado como uma classe só que seu nome (do objeto) é sublinhado, e o nome do objeto pode ser mostrado opcionalmente precedido do nome da classe.



8.3. Estados

Todos os objetos possuem um estado que significa o resultado de atividades executadas pelo objeto, e é normalmente determinada pelos valores de seus atributos e ligações com outros objetos.

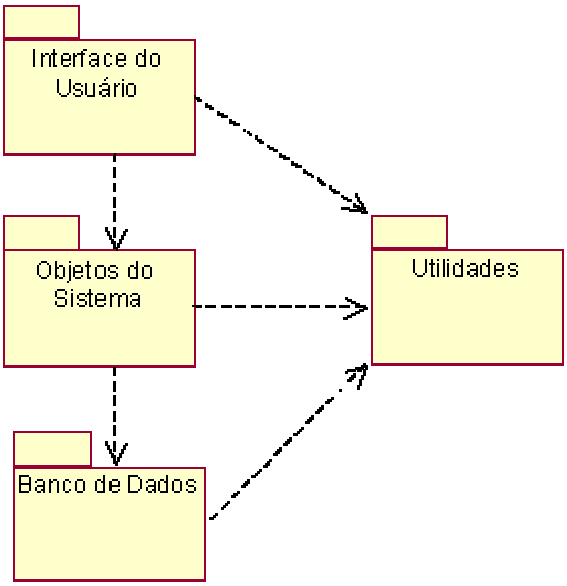
Um objeto muda de estado quando acontece algo, o fato de acontecer alguma coisa com o objeto é chamado de evento. Através da análise da mudança de estados dos tipos de objetos de um sistema, podemos prever todos os possíveis comportamentos de um objetos de acordo com os eventos que o mesmo possa sofrer.



Um estado, em sua notação, pode conter três compartimentos. O primeiro mostra o nome do estado. O segundo é opcional e mostra a variável do estado, onde os atributos do objeto em questão podem ser listados e atualizados. Os atributos são aqueles mostrados na representação da classe, e em algumas vezes, podem ser mostradas também as variáveis temporárias, que são muito úteis em diagramas de estado, já que através da observância de seus valores podemos perceber a sua influência na mudança de estados de um objeto. O terceiro compartimento é opcional e é chamado de compartimento de atividade, onde eventos e ações podem ser listadas. Três eventos padrões podem ser mostrados no compartimento de atividades de um estado: entrar, sair e fazer. O evento entrar pode ser usado para definir atividades no momento em que o objeto entra naquele estado. O evento sair, define atividades que o objeto executa antes de passar para o próximo estado e o evento fazer define as atividades do objeto enquanto se encontra naquele estado.

8.4. Pacotes

Pacote é um mecanismo de agrupamento, onde todos os modelos de elementos podem ser agrupados. Em UML, um pacote é definido como: "Um mecanismo de propósito geral para organizar elementos semanticamente relacionados em grupos." Todos os modelos de elementos que são ligados ou referenciados por um pacote são chamados de "Conteúdo do pacote". Um pacote possui vários modelos de elementos, e isto significa que estes não podem ser incluídos em outros pacotes.

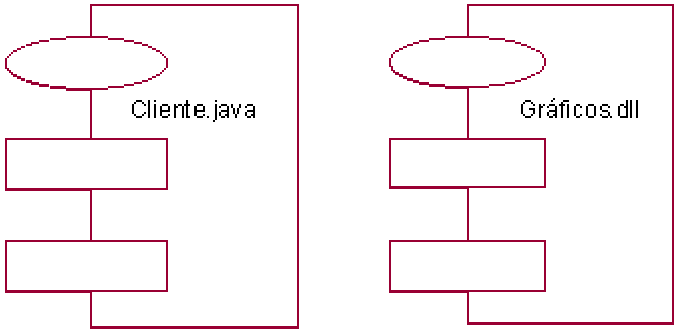


Pacotes podem importar modelos de elementos de outros pacotes. Quando um modelo de elemento é importado, refere-se apenas ao pacote que possui o elemento. Na grande maioria dos casos, os pacotes possuem relacionamentos com outros pacotes. Embora estes não possuam semânticas definidas para suas instâncias. Os relacionamentos permitidos entre pacotes são de dependência, refinamento e generalização (herança).

O pacote tem uma grande similaridade com a agregação (relacionamento que será tratado em seguida). O fato de um pacote ser composto de modelos de elementos cria uma agregação de composição. Se este for destruído, todo o seu conteúdo também será.

8.5. Componentes

Um componente pode ser tanto um código em linguagem de programação como um código executável já compilado. Por exemplo, em um sistema desenvolvido em Java, cada arquivo.*Java* ou.*Class* é um componente do sistema, e será mostrado no diagrama de componentes que os utiliza.



8.6. Relacionamentos

Os relacionamentos ligam as classes/objetos entre si criando relações lógicas entre estas entidades. Os relacionamentos podem ser dos seguintes tipos:

- Associação: É uma conexão entre classes, e também significa que é uma conexão entre objetos daquelas classes. Em UML, uma associação é definida com um relacionamento que descreve uma série de ligações, onde a ligação é definida como a semântica entre as duplas de objetos ligados.

- Generalização: É um relacionamento de um elemento mais geral e outro mais específico. O elemento mais específico pode conter apenas informações adicionais. Uma instância (um objeto é uma instância de uma classe) do elemento mais específico pode ser usada onde o elemento mais geral seja permitido.

- Dependência e Refinamentos: Dependência é um relacionamento entre elementos, um independente e outro dependente. Uma modificação é um elemento independente afetará diretamente elementos dependentes do anterior. Refinamento é um relacionamento entre duas descrições de uma mesma entidade, mas em níveis diferentes de abstração.

Abordaremos agora cada tipo de relacionamento e suas respectivas sub-divisões:

8.6.1 Associações

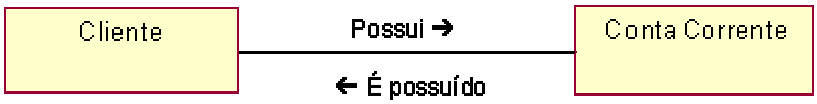
Uma associação representa que duas classes possuem uma ligação (link) entre elas, significando, por exemplo, que elas "conhecem uma a outra", "estão conectadas com", "para cada X existe um Y" e assim por diante. Classes e associações são muito poderosas quando modeladas em sistemas complexos.

Associações Normais

O tipo mais comum de associação é apenas uma conexão entre classes. É representada por uma linha sólida entre duas classes. A associação possui um nome (junto à linha que representa a associação), normalmente um verbo, mas substantivos também são permitidos.

Pode-se também colocar uma seta no final da associação indicando que esta só pode ser usada para o lado onde a seta aponta. Mas associações também podem possuir dois nomes, significando um nome para cada sentido da associação.

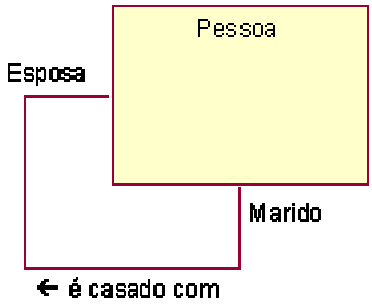
Para expressar a multiplicidade entre os relacionamentos, um intervalo indica quantos objetos estão relacionados no link. O intervalo pode ser de zero para um (0..1), zero para vários (0..* ou apenas *), um para vários (1..*), dois (2), cinco para 11 (5..11) e assim por diante. É também possível expressar uma série de números como (1, 4, 6..12). Se não for descrito nenhuma multiplicidade, então é considerado o padrão de um para um (1..1 ou apenas 1).



No exemplo acima vemos um relacionamento entre as classes Cliente e Conta Corrente se relacionam por associação.

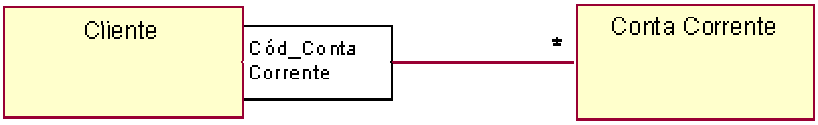
Associação Recursiva

É possível conectar uma classe a ela mesma através de uma associação e que ainda representa semanticamente a conexão entre dois objetos, mas os objetos conectados são da mesma classe. Uma associação deste tipo é chamada de associação recursiva.



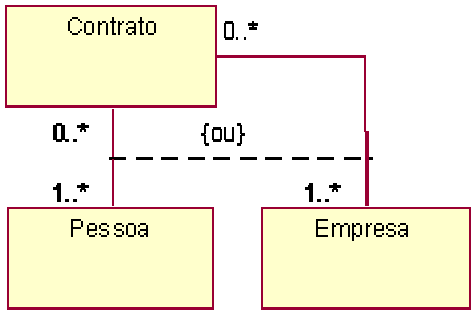
Associação Qualificada

Associações qualificadas são usadas com associações de um para vários (1..*) ou vários para vários (*). O "qualificador" (identificador da associação qualificada) especifica como um determinado objeto no final da associação "n" é identificado, e pode ser visto como um tipo de chave para separar todos os objetos na associação. O identificador é desenhado como uma pequena caixa no final da associação junto à classe de onde a navegação deve ser feita.



Associação Exclusiva

Em alguns modelos nem todas as combinações são válidas, e isto pode causar problemas que devem ser tratados. Uma associação exclusiva é uma restrição em duas ou mais associações. Ela especifica que objetos de uma classe podem participar de no máximo uma das associações em um dado momento. Uma associação exclusiva é representada por uma linha tracejada entre as associações que são parte da associação exclusiva, com a especificação "{ou}" sobre a linha tracejada.



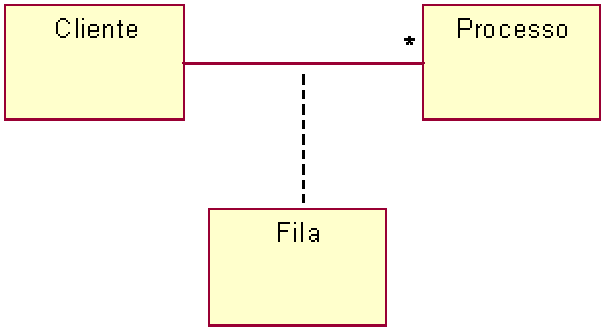
No diagrama acima um contrato não pode se referir a uma pessoa e a uma empresa ao mesmo tempo, significando que o relacionamento é exclusivo a somente uma das duas classes.

Associação Ordenada

As associações entre objetos podem ter uma ordem implícita. O padrão para uma associação é desordenada (ou sem nenhuma ordem específica). Mas uma ordem pode ser especificada através da associação ordenada. Este tipo de associação pode ser muito útil em casos como este: janelas de um sistema têm que ser ordenadas na tela (uma está no topo, uma está no fundo e assim por diante). A associação ordenada pode ser escrita apenas colocando "{ordenada}" junto à linha de associação entre as duas classes.

Associação de Classe

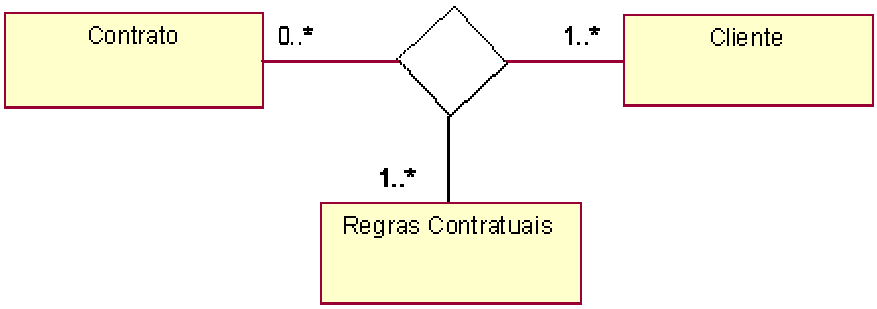
Uma classe pode ser associada a uma outra associação. Este tipo de associação não é conectada a nenhuma das extremidades da associação já existente, mas na própria linha da associação. Esta associação serve para se adicionar informações extra a associação já existente.



A associação da classe Fila com a associação das classes Cliente e Processo pode ser estendida com operações de adicionar processos na fila, para ler e remover da fila e de ler o seu tamanho. Se operações ou atributos são adicionados a associação, ela deve ser mostrada como uma classe.

Associação Ternária

Mais de duas classes podem ser associadas entre si, a associação ternária associa três classes. Ela é mostrada como um grade losango (diamante) e ainda suporta uma associação de classe ligada a ela, traçaria-se, então, uma linha tracejada a partir do losango para a classe onde seria feita a associação ternária.



No exemplo acima a associação ternária especifica que um cliente poderá possuir 1 ou mais contratos e cada contrato será composto de 1 ou várias regras contratuais.

Agregação

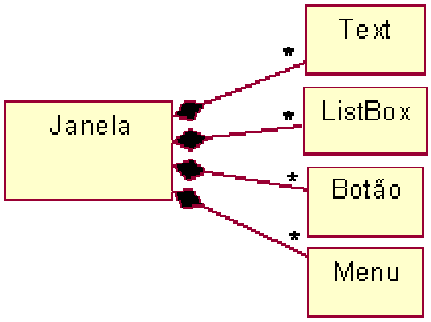
A agregação é um caso particular da associação. A agregação indica que uma das classes do relacionamento é uma parte, ou está contida em outra classe. As palavras chaves usadas para identificar uma agregação são: "consiste em", "contém", "é parte de".

Existem tipos especiais de agregação que são as agregações compartilhadas e as compostas.

- Agregação Compartilhada: É dita compartilhada quando uma das classes é uma parte, ou está contida na outra, mas esta parte pode fazer estar contida na outra várias vezes em um mesmo momento.

  No exemplo acima uma pessoa pode ser membro de um time ou vários times e em determinado momento.

- Agregação de Composição: É uma agregação onde uma classe que está contida na outra "vive" e constitui a outra. Se o objeto da classe que contém for destruído, as classes da agregação de composição serão destruídas juntamente já que as mesmas fazem parte da outra.



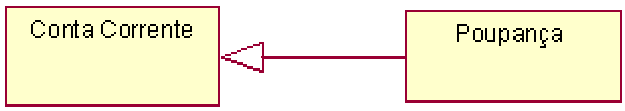
8.6.2. Generalizações

A generalização é um relacionamento entre um elemento geral e um outro mais específico. O elemento mais específico possui todas as características do elemento geral e contém ainda mais particularidades. Um objeto mais específico pode ser usado como uma instância do elemento mais geral. A generalização, também chamada de herança, permite a criação de elementos especializados em outros.

Existem alguns tipos de generalizações que variam em sua utilização a partir da situação. São elas: generalização normal e restrita. As generalizações restritas se dividem em generalização de sobreposição, disjuntiva, completa e incompleta.

Generalização Normal

Na generalização normal a classe mais específica, chamada de subclasse, herda tudo da classe mais geral, chamada de superclasse. Os atributos, operações e todas as associações são herdadas.



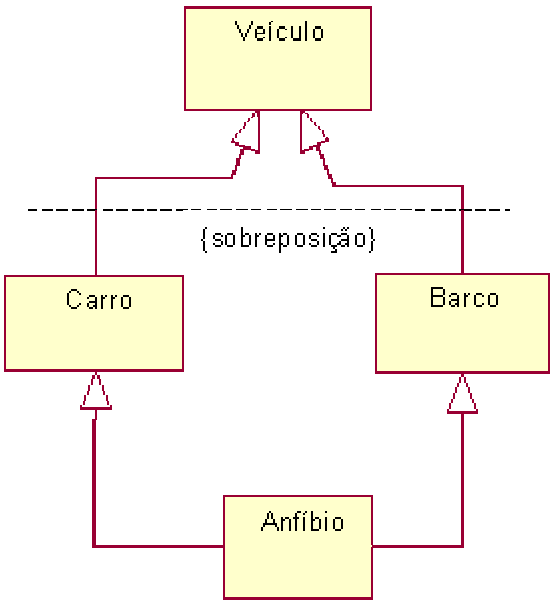
Uma classe pode ser tanto uma subclasse quanto uma superclasse, se ela estiver numa hierarquia de classes que é um gráfico onde as classes estão ligadas através de generalizações.

A generalização normal é representada por uma linha entre as duas classes que fazem o relacionamento, sendo que coloca-se um seta no lado da linha onde encontra-se a superclasse indicando a generalização.

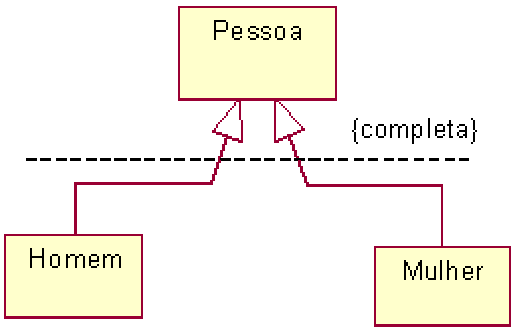
Generalização Restrita

Uma restrição aplicada a uma generalização especifica informações mais precisas sobre como a generalização deve ser usada e estendida no futuro. As restrições a seguir definem as generalizações restritas com mais de uma subclasse:

- Generalizações de Sobreposição e Disjuntiva: Generalização de sobreposição significa que quando subclasses herdam de uma superclasse por sobreposição, novas subclasses destas podem herdar de mais de uma subclasse. A generalização disjuntiva é exatamente o contrário da sobreposição e a generalização é utilizada como padrão.



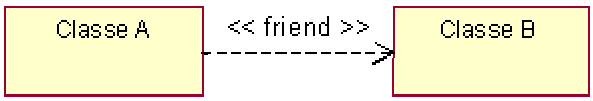
- Generalizações Completa e Incompleta: Uma restrição simbolizando que uma generalização é completa significa que todas as subclasses já foram especificadas, e não existe mais possibilidade de outra generalização a partir daquele ponto. A generalização incompleta é exatamente o contrário da completa e é assumida como padrão da linguagem.



8.6.3. Dependência e Refinamentos

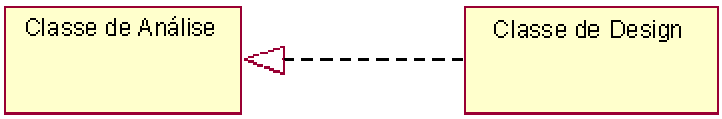
Além das associações e generalizações, existem ainda dois tipos de relacionamentos em UML. O relacionamento de dependência é uma conexão semântica entre dois modelos de elementos, um independente e outro dependente. Uma mudança no elemento independente irá afetar o modelo dependente. Como no caso anterior com generalizações, os modelos de elementos podem ser uma classe, um pacote, um use-case e assim por diante. Quando uma classe recebe um objeto de outra classe como parâmetro, uma classe acessa o objeto global da outra. Nesse caso existe uma dependência entre estas duas classes, apesar de não ser explícita.

Uma relação de dependência é simbolizada por uma linha tracejada com uma seta no final de um dos lados do relacionamento. E sobre essa linha o tipo de dependência que existe entre as duas classes. As classes "Amigas" provenientes do C++ são um exemplo de um relacionamento de dependência.



Os refinamentos são um tipo de relacionamento entre duas descrições de uma mesma coisa, mas em níveis de abstração diferentes e podem ser usados para modelar diferentes implementações de uma mesma coisa (uma implementação simples e outra mais complexa, mas também mais eficiente).

Os refinamentos são simbolizados por uma linha tracejada com um triângulo no final de um dos lados do relacionamento e são usados em modelos de coordenação. Em grandes projetos, todos os modelos que são feitos devem ser coordenados. Coordenação de modelos pode ser usada para mostrar modelos em diferentes níveis de abstração que se relacionam e mostram também como modelos em diferentes fases de desenvolvimento se relacionam.

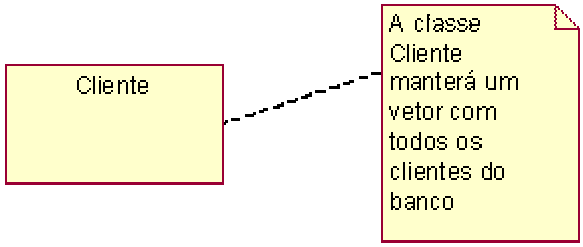


8.7. Mecanismos Gerais

A UML utiliza alguns mecanismos em seus diagramas para tratar informações adicionais.

- Ornamentos: Ornamentos gráficos são anexados aos modelos de elementos em diagramas e adicionam semânticas ao elemento. Um exemplo de um ornamento é o da técnica de separar um tipo de uma instância. Quando um elemento representa um tipo, seu nome é mostrado em negrito. Quando o mesmo elemento representa a instância de um tipo, seu nome é escrito sublinhado e pode significar tanto o nome da instância quanto o nome do tipo. Outros ornamentos são os de especificação de multiplicidade de relacionamentos, onde a multiplicidade é um número ou um intervalo que indica quantas instâncias de um tipo conectado pode estar envolvido na relação.

- Notas: Nem tudo pode ser definido em uma linguagem de modelagem, sem importar o quanto extensa ela seja. Para permitir adicionar informações a um modelo não poderia ser representado de outra forma, UML provê a capacidade de adicionar Notas. Uma Nota pode ser colocada em qualquer lugar em um diagrama, e pode conter qualquer tipo de informação.



9. Diagramas

Os diagramas utilizados pela UML são compostos de nove tipos: diagrama de use case, de classes, de objeto, de estado, de sequência, de colaboração, de atividade, de componente e o de execução.

Todos os sistemas possuem uma estrutura estática e um comportamento dinâmico. A UML suporta modelos estáticos (estrutura estática), dinâmicos (comportamento dinâmico) e funcional. A Modelagem estática é suportada pelo diagrama de classes e de objetos, que consiste nas classes e seus relacionamentos. Os relacionamentos podem ser de associações, herança (generalização), dependência ou refinamentos. Os modelamentos dinâmicos são suportados pelos diagramas de estado, sequência, colaboração e atividade. E o modelamento funcional é suportado pelos diagramas de componente e execução. Abordaremos agora cada um destes tipos de diagrama:

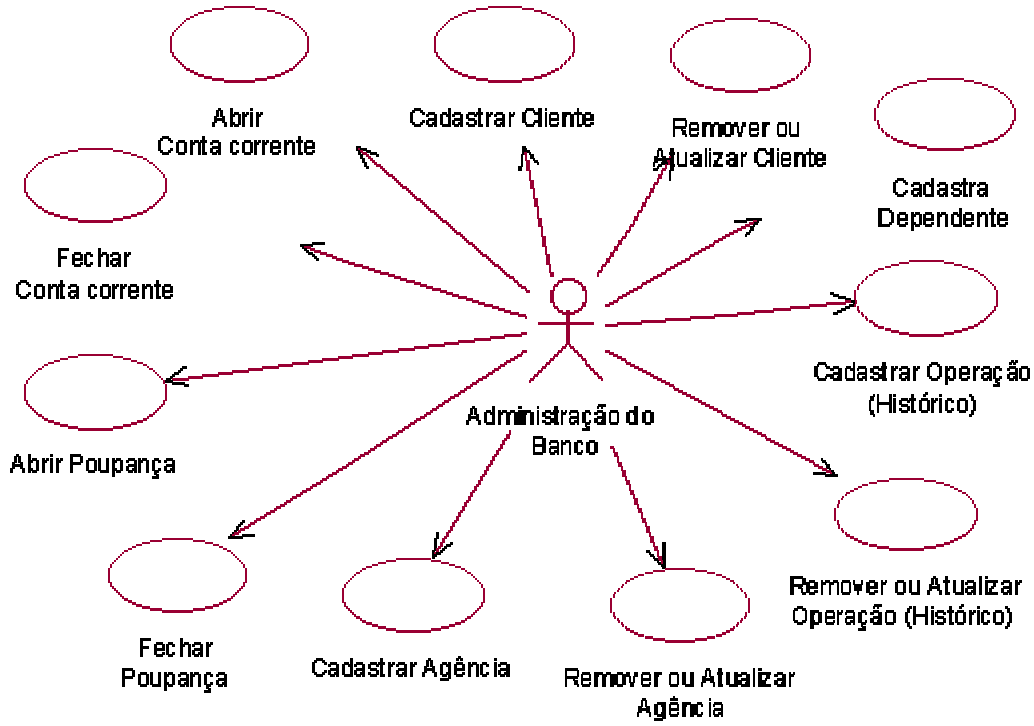
9.1. Diagrama Use-Case

A modelagem de um diagrama use-case é uma técnica usada para descrever e definir os requisitos funcionais de um sistema. Eles são escritos em termos de atores externos, use-cases e o sistema modelado. Os atores representam o papel de uma entidade externa ao sistema como um usuário, um hardware, ou outro sistema que interage com o sistema modelado. Os atores iniciam a comunicação com o sistema através dos use-cases, onde o use-case representa uma sequência de ações executadas pelo sistema e recebe do ator que lhe utiliza dados tangíveis de um tipo ou formato já conhecido, e o valor de resposta da execução de um use-case (conteúdo) também já é de um tipo conhecido, tudo isso é definido juntamente com o use-case através de texto de documentação.

Atores e use-cases são classes. Um ator é conectado a um ou mais use-cases através de associações, e tanto atores quanto use-cases podem possuir relacionamentos de generalização que definem um comportamento comum de herança em superclasses especializadas em subclasses.

O uso de use-cases em colaborações é muito importante, onde estas são a descrição de um contexto mostrando classes/objetos, seus relacionamentos e sua interação exemplificando como as classes/objetos interagem para executar uma atividade específica no sistema. Uma colaboração é descrita por diagramas de atividades e um diagrama de colaboração.

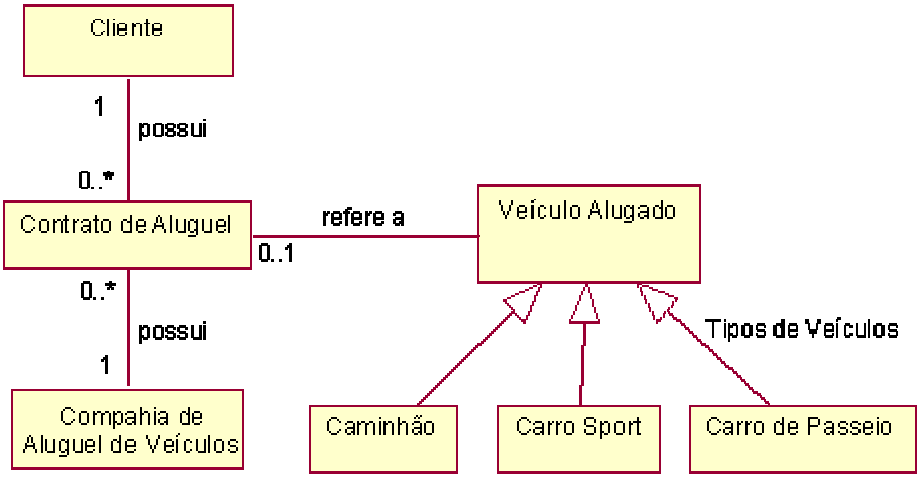
Quando um use-case é implementado, a responsabilidade de cada passo da execução deve ser associada às classes que participam da colaboração, tipicamente especificando as operações necessárias dentro destas classes juntamente com a definição de como elas irão interagir. Um cenário é uma instância de um use-case, ou de uma colaboração, mostrando o caminho específico de cada ação. Por isso, o cenário é um importante exemplo de um use-case ou de uma colaboração. Quando visto a nível de um use-case, apenas a interação entre o ator externo e o use-case é vista, mas já observando a nível de uma colaboração, toda as interações e passos da execução que implementam o sistema serão descritos e especificados.



O diagrama de use-cases acima demonstra as funções de um ator externo de um sistema de controle bancário de um banco fictício que foi modelado no estudo de caso no final deste trabalho. O diagrama especifica que funções o administrador do banco poderá desempenhar. Pode-se perceber que não existe nenhuma preocupação com a implementação de cada uma destas funções, já que este diagrama apenas se resume a determinar que funções deverão ser suportadas pelo sistema modelado.

9.2 Diagrama de Classes

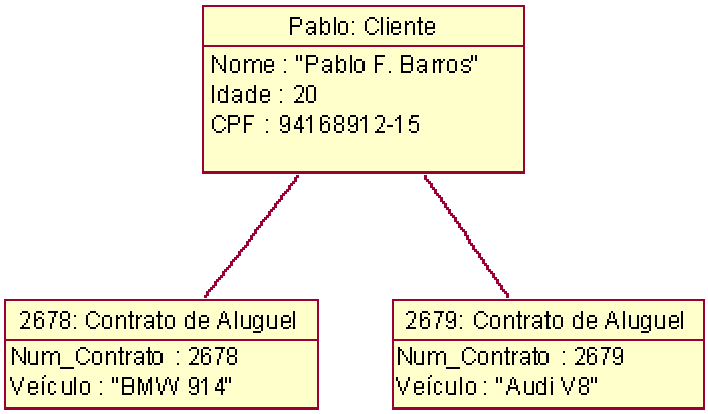
O diagrama de classes demonstra a estrutura estática das classes de um sistema onde estas representam as "coisas" que são gerenciadas pela aplicação modelada. Classes podem se relacionar com outras através de diversas maneiras: associação (conectadas entre si), dependência (uma classe depende ou usa outra classe), especialização (uma classe é uma especialização de outra classe), ou em pacotes (classes agrupadas por características similares). Todos estes relacionamentos são mostrados no diagrama de classes juntamente com as suas estruturas internas, que são os atributos e operações. O diagrama de classes é considerado estático já que a estrutura descrita é sempre válida em qualquer ponto do ciclo de vida do sistema. Um sistema normalmente possui alguns diagramas de classes, já que não são todas as classes que estão inseridas em um único diagrama e uma certa classes pode participar de vários diagramas de classes.



Uma classe num diagrama pode ser diretamente implementada utilizando-se uma linguagem de programação orientada a objetos que tenha suporte direto para construção de classes. Para criar um diagrama de classes, as classes têm que estar identificadas, descritas e relacionadas entre si.

9.3. Diagrama de Objetos

O diagrama de objetos é uma variação do diagrama de classes e utiliza quase a mesma notação. A diferença é que o diagrama de objetos mostra os objetos que foram instanciados das classes. O diagrama de objetos é como se fosse o perfil do sistema em um certo momento de sua execução. A mesma notação do diagrama de classes é utilizada com 2 exceções: os objetos são escritos com seus nomes sublinhados e todas as instâncias num relacionamento são mostradas. Os diagramas de objetos não são tão importantes como os diagramas de classes, mas eles são muito úteis para exemplificar diagramas complexos de classes ajudando muito em sua compreensão. Diagramas de objetos também são usados como parte dos diagramas de colaboração, onde a colaboração dinâmica entre os objetos do sistema são mostrados.

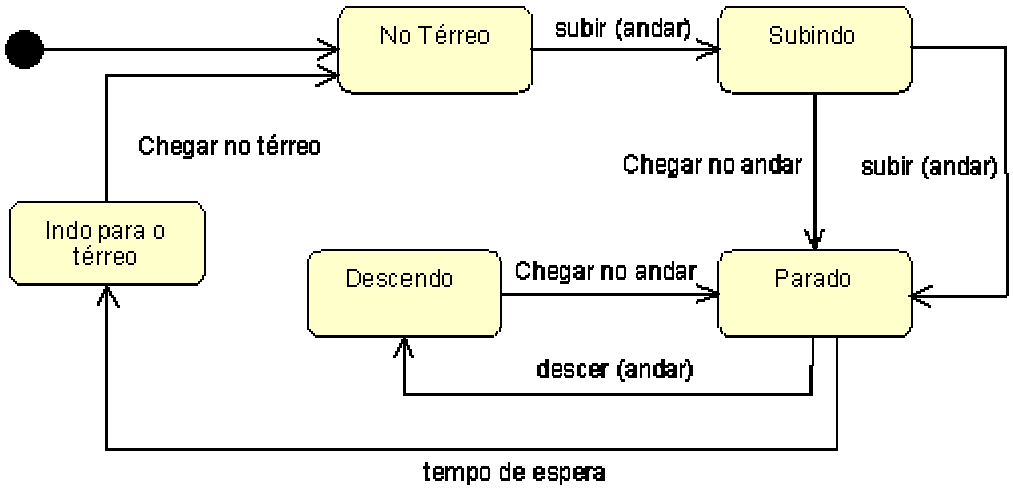


9.4. Diagrama de Estado

O diagrama de estado é tipicamente um complemento para a descrição das classes. Este diagrama mostra todos os estados possíveis que objetos de uma certa classe podem se encontrar e mostra também quais são os eventos do sistemas que provocam tais mudanças. Os diagramas de estado não são escritos para todas as classes de um sistema, mas apenas

para aquelas que possuem um número definido de estados conhecidos e onde o comportamento das classes é afetado e modificado pelos diferentes estados.

Diagramas de estado capturam o ciclo de vida dos objetos, subsistemas e sistemas. Eles mostram os estados que um objeto pode possuir e como os eventos (mensagens recebidas, timer, erros, e condições sendo satisfeitas) afetam estes estados ao passar do tempo.



Diagramas de estado possuem um ponto de início e vários pontos de finalização. Um ponto de início (estado inicial) é mostrado como um círculo todo preenchido, e um ponto de finalização (estado final) é mostrado como um círculo em volta de um outro círculo menor preenchido. Um estado é mostrado como um retângulo com cantos arredondados. Entre os estados estão as transições, mostrados como uma linha com uma seta no final de um dos estados. A transição pode ser nomeada com o seu evento causador. Quando o evento acontece, a transição de um estado para outro é executada ou disparada.

Uma transição de estado normalmente possui um evento ligado a ela. Se um evento é anexado a uma transição, esta será executada quando o evento ocorrer. Se uma transição não possuir um evento ligado a ela, a mesma ocorrerá quando a ação interna do código do estado for executada (se existir ações internas como entrar, sair, fazer ou outras ações definidas pelo desenvolvedor). Então quando todas as ações forem executadas pelo estado, a transição será disparada e serão iniciadas as atividades do próximo estado no diagrama de estados.

9.5. Diagrama de Sequência

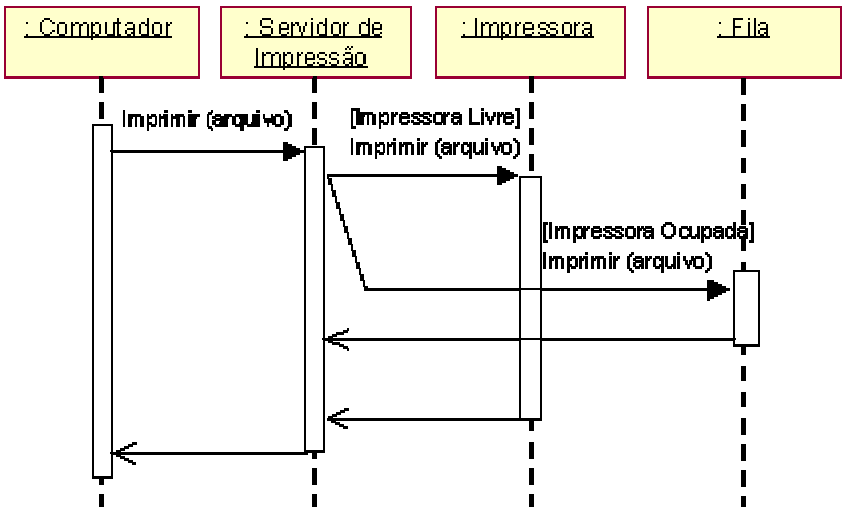
Um diagrama de sequência mostra a colaboração dinâmica entre os vários objetos de um sistema. O mais importante aspecto deste diagrama é que a partir dele percebe-se a sequência de mensagens enviadas entre os objetos. Ele mostra a interação entre os objetos, alguma coisa que acontecerá em um ponto específico da execução do sistema. O diagrama de sequência consiste em um número de objetos mostrado em linhas verticais. O decorrer do tempo é visualizado observando-se o diagrama no sentido vertical de cima para baixo. As mensagens enviadas por cada objeto são simbolizadas por setas entre os objetos que se relacionam.

Diagramas de sequência possuem dois eixos: o eixo vertical, que mostra o tempo e o eixo horizontal, que mostra os objetos envolvidos na sequência de uma certa atividade. Eles também mostram as interações para um cenário específico de uma certa atividade do sistema.

No eixo horizontal estão os objetos envolvidos na sequência. Cada um é representado por um retângulo de objeto (similar ao diagrama de objetos) e uma linha vertical pontilhada chamada de linha de vida do objeto, indicando a execução do objeto durante a sequência, como exemplo

citamos: mensagens recebidas ou enviadas e ativação de objetos. A comunicação entre os objetos é representada como linha com setas horizontais simbolizando as mensagens entre as linhas de vida dos objetos. A seta especifica se a mensagem é síncrona, assíncrona ou simples. As mensagens podem possuir também números sequenciais, eles são utilizados para tornar mais explícito as sequência no diagrama.

Em alguns sistemas, objetos rodam concorrentemente, cada um com sua linha de execução (thread). Se o sistema usa linhas concorrentes de controle, isto é mostrado como ativação, mensagens assíncronas, ou objetos assíncronos.



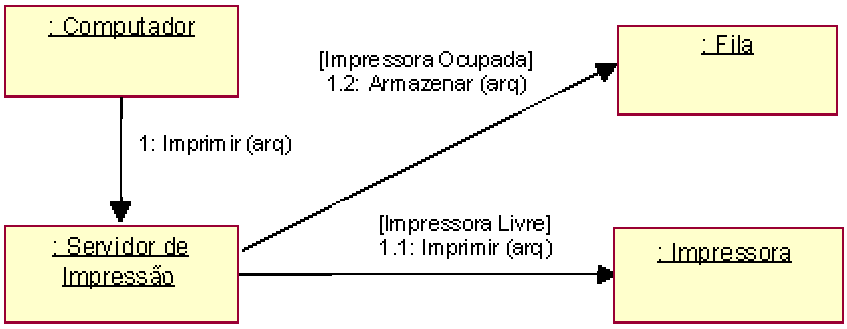
Os diagramas de sequência podem mostrar objetos que são criados ou destruídos como parte do cenário documentado pelo diagrama. Um objeto pode criar outros objetos através de mensagens. A mensagem que cria ou destroi um objeto é geralmente síncrona, representada por uma seta sólida.

9.6. Diagrama de Colaboração

Um diagrama de colaboração mostra de maneira semelhante ao diagrama de sequência, a colaboração dinâmica entre os objetos. Normalmente pode-se escolher entre utilizar o diagrama de colaboração ou o diagrama de sequência.

No diagrama de colaboração, além de mostrar a troca de mensagens entre os objetos, percebe-se também os objetos com os seus relacionamentos. A interação de mensagens é mostrada em ambos os diagramas. Se a ênfase do diagrama for o decorrer do tempo, é melhor escolher o diagrama de sequência, mas se a ênfase for o contexto do sistema, é melhor dar prioridade ao diagrama de colaboração.

O diagrama de colaboração é desenhado como um diagrama de objeto, onde os diversos objetos são mostrados juntamente com seus relacionamentos. As setas de mensagens são desenhadas entre os objetos para mostrar o fluxo de mensagens entre eles. As mensagens são nomeadas, que entre outras coisas mostram a ordem em que as mensagens são enviadas. Também podem mostrar condições, interações, valores de resposta, e etc. O diagrama de colaboração também pode conter objetos ativos, que executam paralelamente com outros.



9.7. Diagrama de Atividade

Diagramas de atividade capturam ações e seus resultados. Eles focam o trabalho executado na implementação de uma operação (método), e suas atividades numa instância de um objeto. O diagrama de atividade é uma variação do diagrama de estado e possui um propósito um pouco diferente do diagrama de estado, que é o de capturar ações (trabalho e atividades que serão executados) e seus resultados em termos das mudanças de estados dos objetos.

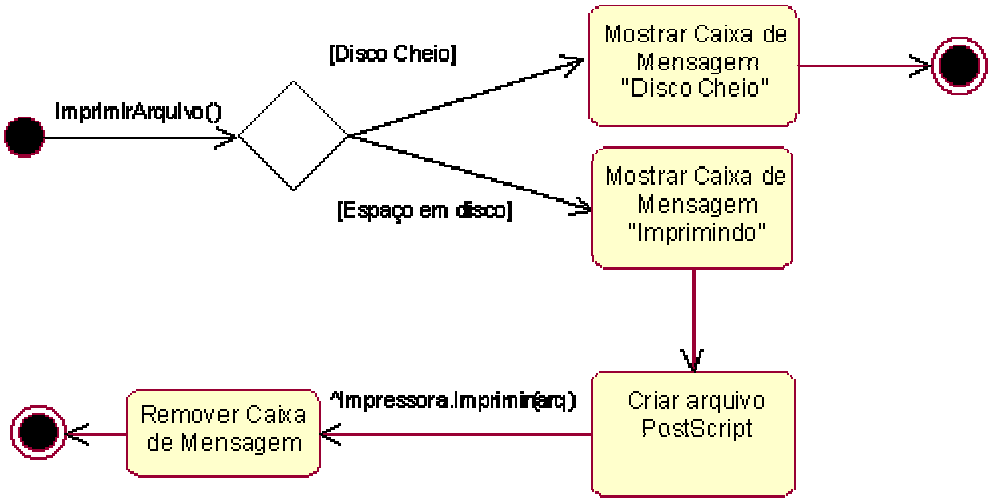
Os estados no diagrama de atividade mudam para um próximo estágio quando uma ação é executada (sem ser necessário especificar nenhum evento como no diagrama de estado). Outra diferença entre o diagrama de atividade e o de estado é que podem ser colocadas como "swimlanes". Uma swimlane agrupa atividades, com respeito a quem é responsável e onde estas atividades residem na organização, e é representada por retângulos que englobam todos os objetos que estão ligados a ela (swimlane).

Um diagrama de atividade é uma maneira alternativa de se mostrar interações, com a possibilidade de expressar como as ações são executadas, o que elas fazem (mudanças dos estados dos objetos), quando elas são executadas (sequência das ações), e onde elas acontecem (swimlanes).

Um diagrama de atividade pode ser usado com diferentes propósitos inclusive:

- Para capturar os trabalhos que serão executados quando uma operação é disparada (ações). Este é o uso mais comum para o diagrama de atividade.
- Para capturar o trabalho interno em um objeto.
- Para mostrar como um grupo de ações relacionadas podem ser executadas, e como elas vão afetar os objetos em torno delas.
- Para mostrar como uma instância pode ser executada em termos de ações e objetos.
- Para mostrar como um negócio funciona em termos de trabalhadores (atores), fluxos de trabalho, organização, e objetos (fatores físicos e intelectuais usados no negócio).

O diagrama de atividade mostra o fluxo sequencial das atividades, é normalmente utilizado para demonstrar as atividades executadas por uma operação específica do sistema. Consistem em estados de ação, que contém a especificação de uma atividade a ser desempenhada por uma operação do sistema. Decisões e condições, como execução paralela, também podem ser mostrados na diagrama de atividade. O diagrama também pode conter especificações de mensagens enviadas e recebidas como partes de ações executadas.



9.8. Diagrama de Componente

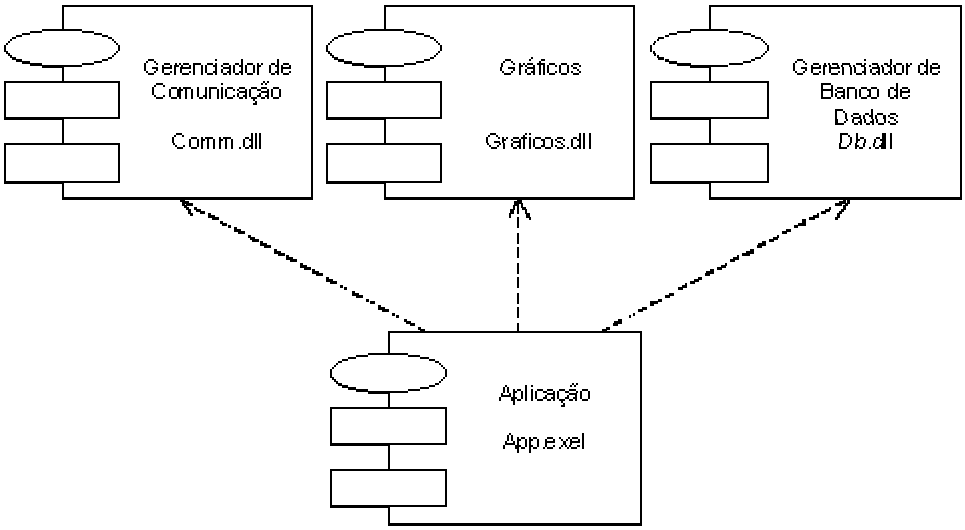
O diagrama de componente e o de execução são diagramas que mostram o sistema por um lado funcional, expondo as relações entre seus componentes e a organização de seus módulos durante sua execução.

O diagrama de componente descreve os componentes de software e suas dependências entre si, representando a estrutura do código gerado. Os componentes são a implementação na arquitetura física dos conceitos e da funcionalidade definidos na arquitetura lógica (classes, objetos e seus relacionamentos). Eles são tipicamente os arquivos implementados no ambiente de desenvolvimento.

Um componente é mostrado em UML como um retângulo com uma elipse e dois retângulos menores do seu lado esquerdo. O nome do componente é escrito abaixo ou dentro de seu símbolo.

Componentes são tipos, mas apenas componentes executáveis podem ter instâncias. Um diagrama de componente mostra apenas componentes como tipos. Para mostrar instâncias de componentes, deve ser usado um diagrama de execução, onde as instâncias executáveis são alocadas em *nodes*.

A dependência entre componentes pode ser mostrada como uma linha tracejada com uma seta, simbolizando que um componente precisa do outro para possuir uma definição completa. Com o diagrama de componentes é facilmente visível detectar que arquivos *.dll* são necessários para executar a aplicação.



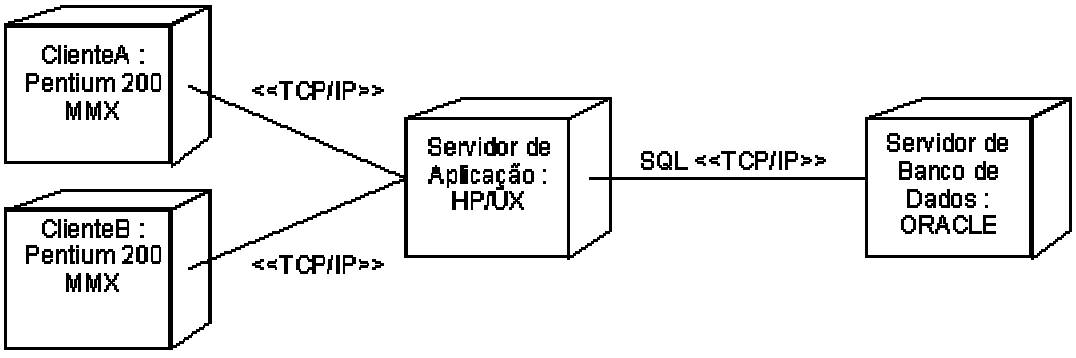
Componentes podem definir interfaces que são visíveis para outros componentes. As interfaces podem ser tanto definidas ao nível de codificação (como em Java) quanto em interfaces binárias usadas em run-time (como em OLE). Uma interface é mostrada como uma linha partindo do componente e com um círculo na outra extremidade. O nome é colocado junto do círculo no final da linha. Dependências entre componentes podem então apontar para a interface do componente que está sendo usada.

9.9. Diagrama de Execução

O diagrama de execução mostra a arquitetura física do hardware e do software no sistema. Pode mostrar os atuais computadores e periféricos, juntamente com as conexões que eles estabelecem entre si e pode mostrar também os tipos de conexões entre esses computadores e periféricos. Especifica-se também os componentes executáveis e objetos que são alocados para mostrar quais unidades de software são executados e em que destes computadores são executados.

O diagrama de execução demonstra a arquitetura run-time de processadores, componentes físicos (devices), e de software que rodam no ambiente onde o sistema desenvolvido será utilizado. É a última descrição física da topologia do sistema, descrevendo a estrutura de hardware e software que executam em cada unidade.

O diagrama de execução é composto por componentes, que possuem a mesma simbologia dos componentes do diagrama de componentes, *nodes*, que significam objetos físicos que fazem parte do sistema, podendo ser uma máquina cliente numa LAN, uma máquina servidora, uma impressora, um roteador, etc., e conexões entre estes *nodes* e componentes que juntos compõem toda a arquitetura física do sistema.



10. Um processo para utilizar a UML

A UML contém notações e regras que tornam possível expressar modelos orientados a objetos. Mas ela não prescreve como o trabalho tem que ser feito, ou seja, não possui um processo de como o trabalho tem que ser desenvolvido. Já que a UML foi desenvolvida para ser usada em diversos métodos de desenvolvimento.

Para usar a UML com sucesso é necessário adotar algum tipo de método de desenvolvimento, especialmente em sistema de grande porte onde a organização de tarefas é essencial. A utilização de um processo de desenvolvimento torna mais eficiente calcular o progresso do projeto, controlar e melhorar o trabalho.

Um processo de desenvolvimento descreve "o que fazer", "como fazer", "quando fazer", e "porque deve ser feito". Este também descreve um número de atividades que devem ser executadas em uma certa ordem. Quando são definidas e relacionadas às atividades de um processo, um objetivo específico é alcançado.

Em seu uso normal, a palavra "processo" significa uma relação de atividades que devem ser executadas em uma certa ordem sem importar o objetivo, regras ou material a ser usado. No processo de desenvolvimento da engenharia de software, é necessário saber o objetivo final do processo, definir regras a serem seguidas e adotar um método fixo de desenvolvimento.

Um método (processo) tradicional de desenvolvimento orientado a objetos é dividido em análise de requisitos, análise, design (projeto), implementação, e testes. A análise de requisitos captura as necessidades básicas funcionais e não-funcionais do sistema que deve ser desenvolvido. A análise modela o problema principal (classes, objetos) e cria um modelo ideal do sistema sem levar em conta requisitos técnicos do sistema. O design expande e adapta os modelos da análise para um ambiente técnico, onde as soluções técnicas são trabalhadas em detalhes. A implementação consiste em codificar em linguagem de programação e banco de dados os modelos criados. E as atividades de testes devem testar o sistema em diferentes níveis, verificando se o mesmo corresponde as expectativas do usuário.

Existe um processo desenvolvido pela Rational Inc., mesma empresa que desenvolveu a UML, que monta duas visões do desenvolvimento de um sistema: visão gerencial e técnica. A visão técnica utiliza as tradicionais atividades de análise, design e implementação, enquanto a visão gerencial utiliza as seguintes fases no desenvolvimento de cada geração do sistema.

- Início: Define o escopo e objetivo do projeto;
- Elaboração: Desenvolve o produto em detalhes através de uma série de interações. Isto envolve mais análise, design e programação;
- Transição: Gera o sistema para o usuário final, incluindo as atividades de marketing, suporte, documentação e treinamento.

Cada fase no ciclo é executada em séries de interações que podem sobrepor outras fases. Cada interação consiste tipicamente em atividades tradicionais como análise e design, mas em diferentes proporções dependendo da fase em que esteja a geração do sistema em desenvolvimento.

Ferramentas modernas devem dar suporte não apenas para linguagens de modelagem e programação, mas devem suportar um método de desenvolvimento de sistemas também. Isso inclui conhecimento das fases em um processo, ajuda online, e aconselhamentos do que fazer em cada fase do desenvolvimento, suporte a desenvolvimento interativo e fácil integração com outras ferramentas.

11. O Futuro da UML

Embora a UML defina uma linguagem precisa, ela não é uma barreira para futuros aperfeiçoamentos nos conceitos de modelagem. O desenvolvimento da UML foi baseado em técnicas antigas e marcantes da orientação a objetos, mas muitas outras influenciarão a linguagem em suas próximas versões. Muitas técnicas avançadas de modelagem podem ser definidas usando UML como base, podendo ser estendida sem se fazer necessário redefinir a sua estrutura interna.

A UML será a base para muitas ferramentas de desenvolvimento, incluindo modelagem visual, simulações e ambientes de desenvolvimento. Em breve ferramentas de integração e padrões de implementação baseados em UML estarão disponíveis para qualquer um.

A UML integrou muitas idéias adversas, e esta integração vai acelerar o uso do desenvolvimento de softwares orientados a objetos.

12. Um estudo de caso em UML

Diante do apresentado no decorrer do trabalho, aplicaremos aqui grande parte dos conceitos abordados diante de uma aplicação da UML num problema fictício que poderá ser de grande ajuda no melhor entendimento das potencialidades da linguagem de modelagem unificada.

O estudo de caso dará mais ênfase na fases de análise de requisitos, análise e design, já que as principais abstrações dos modelos do sistema se encontram nestas fases do desenvolvimento.

Desenvolveremos uma modelagem em UML para criarmos um sistema de manutenção e controle de contas correntes e aplicações financeiras de um banco fictício.

Antes de dar início à primeira fase da modelagem, faremos algumas considerações sobre o que o sistema se propõe a fazer e outras observações que consideramos de suma importância para o bom entendimento do problema.

- O sistema suportará um cadastro de clientes, onde cada cliente cadastrado poderá ter várias contas correntes, vários dependentes ligados a ele, e várias contas de poupança.

- Cada dependente poderá possuir várias contas de poupança, mas não poderão ter uma conta corrente própria.

- Entendemos poupança como uma conta que possui um valor, um prazo de aplicação a uma taxa de juros (definida no vencimento da poupança).

- Entendemos Aplicações Pré-fixadas como uma aplicação de um valor, em um prazo pré-determinado a uma taxa de juros previamente definida.

- Tanto a conta corrente quanto a poupança deverão manter um histórico de todas as movimentações de crédito, débito, transferências e aplicações de pré-fixados (pré-fixados apenas para conta corrente).

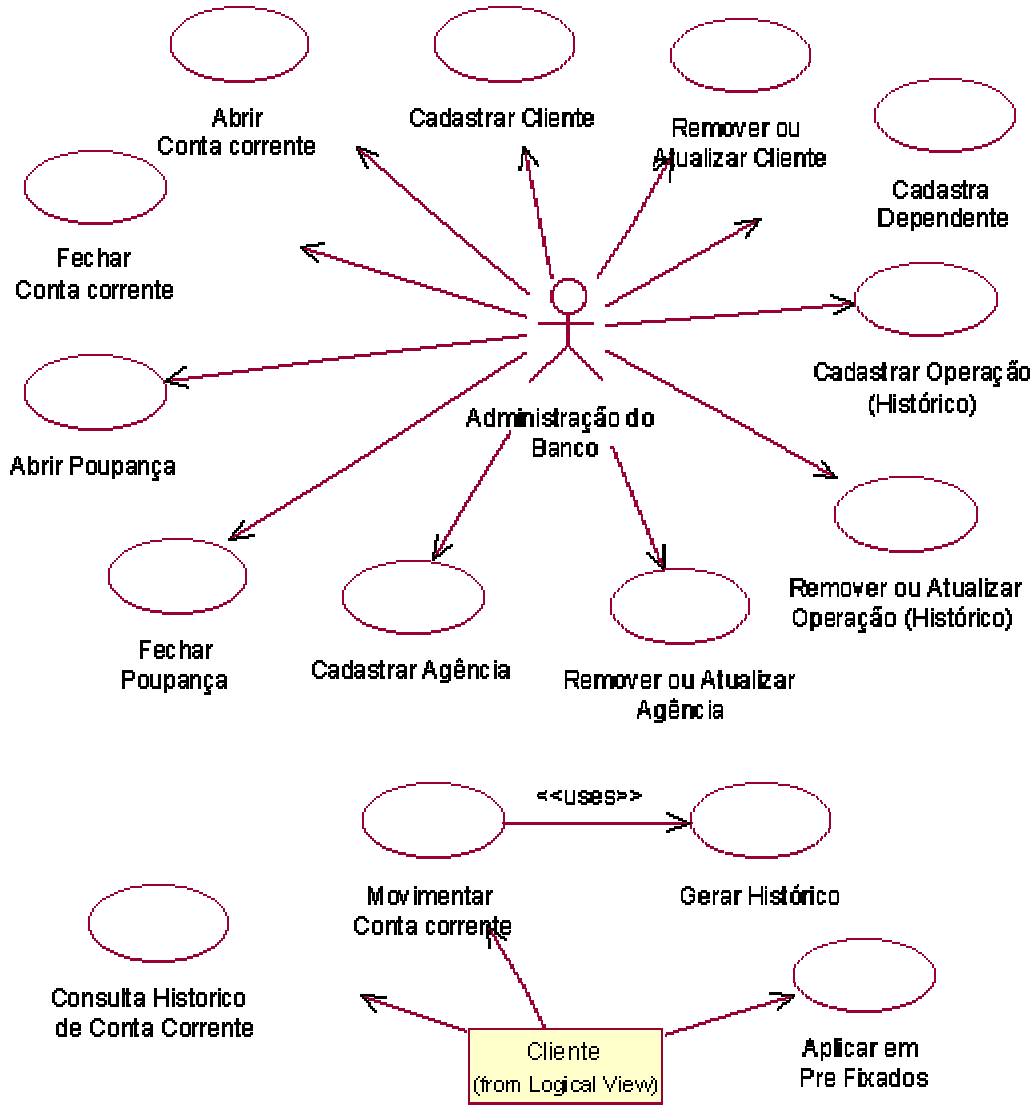
- Uma conta corrente poderá ter várias aplicações pré-fixadas ligadas a ela.

12.1. Análise de Requisitos

De acordo com nossa proposta o sistema implementará funções básicas que serão desempenhadas pela Administração do banco e pelos seus clientes. As principais funções do sistema são:

- Cadastrar novo cliente
- Excluir ou editar cliente
- Cadastrar dependente
- Excluir ou editar dependente
- Abrir conta corrente
- Fechar conta corrente
- Abrir poupança
- Fechar poupança
- Movimentar conta corrente
- Aplicar em pré-fixados
- Consultar histórico de conta corrente ou poupança
- Cadastrar Agência
- Excluir ou Editar Agência

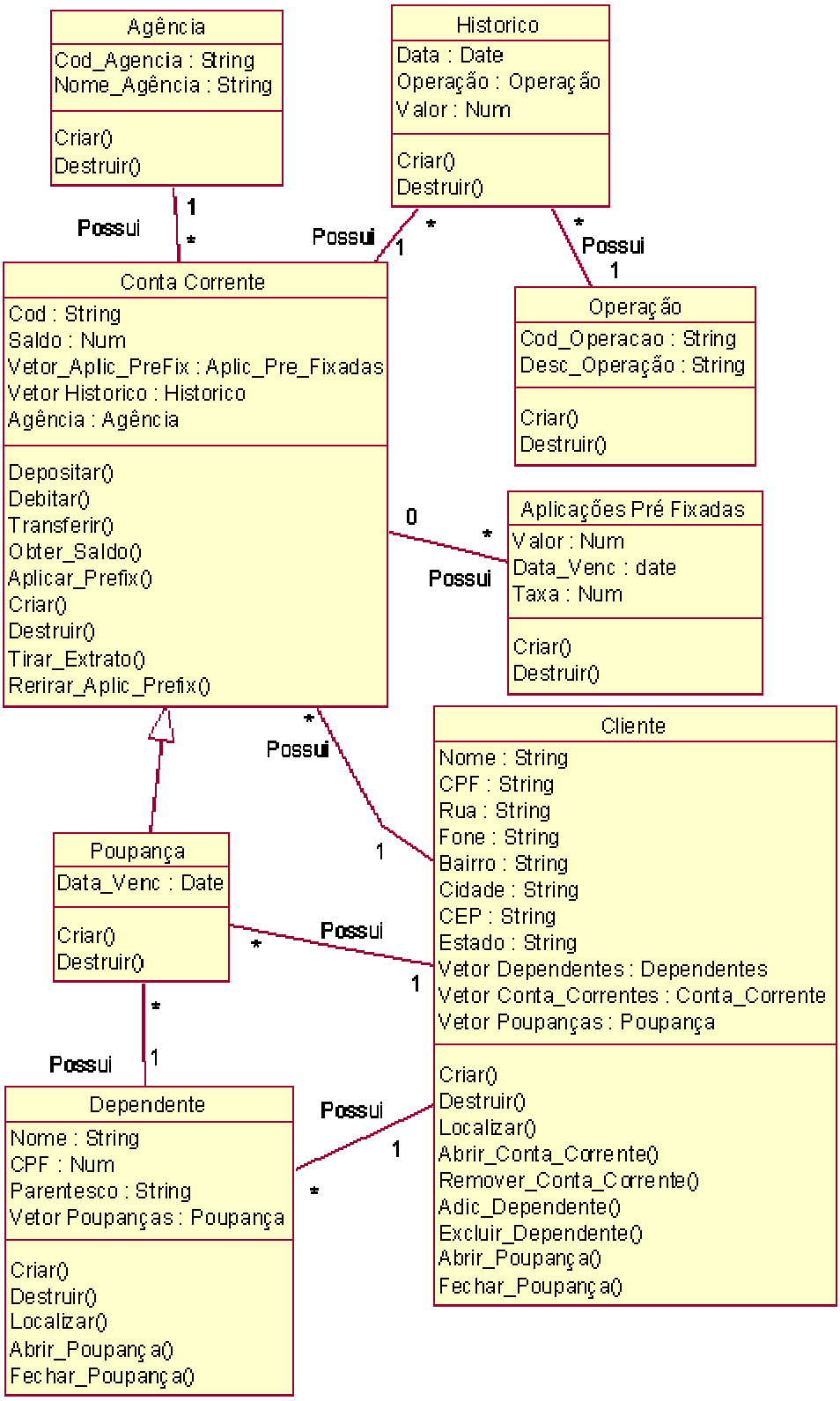
Tendo em mãos esta relação de atividades, já podemos modelar o diagrama de use-case do sistema.



12.2. Análise

Na fase de análise, tendo em mãos o diagrama de use-case, podemos definir o diagrama de classes do sistema. Este primeiro diagrama da fase de análise deverá ser totalmente despreocupado de qualquer tipo de técnica relacionada a implementação do sistema, ou seja, métodos e atributos de acesso a banco de dados, estrutura de mensagens entre objetos, etc. não deverão aparecer nesse primeiro diagrama, apenas os tipos de objetos básicos do sistema.

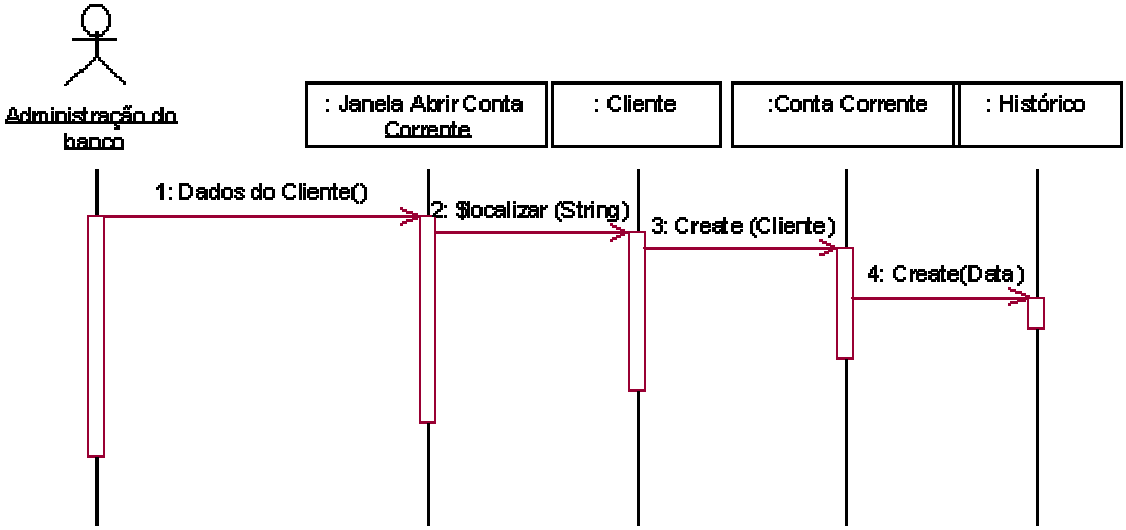
Analisamos e percebemos que existirão 8 classes no sistema e que se relacionarão segundo o diagrama de classes a seguir.



Já temos em mãos as funções primordiais do sistema (diagrama de use-cases) e o diagrama de classes da análise do domínio do problema, partiremos agora para traçar como estas classes irão interagir para realizar as funções do sistema. Lembramos que ainda nesta fase nenhum tipo de técnica de implementação deve ser considerada.

Para modelarmos como os objetos do sistema irão interagir entre si, utilizamos o diagrama de sequência ou o de colaboração. E modelaremos um diagrama para cada função (use-case)

definida no diagrama de use-cases. Escolhemos o diagrama de sequência para dar mais ênfase a ordem cronológica das interações entre os objetos. Já se faz necessário utilizar idéias básicas da modelagem da interface do sistema como as janelas. Mas esses objetos de interface serão totalmente detalhados na fase de design.



Nesta fase modela-se também o diagrama de estado das classes. Mas este se enquadra em situações onde o comportamento dos objetos é importante para aplicação. Em casos de modelagens de sistemas para equipamentos mecânicos.

12.3. Design

Nesta fase começaremos a implementar em nossos modelos os melhoramentos e técnicas de como realmente cada função do sistema será concebida. Serão modelos mais detalhados com ênfase nas soluções para armazenamento dos dados, funções primordiais do sistema e interface com o usuário.

A fase de design pode ser dividida em outras duas fases:

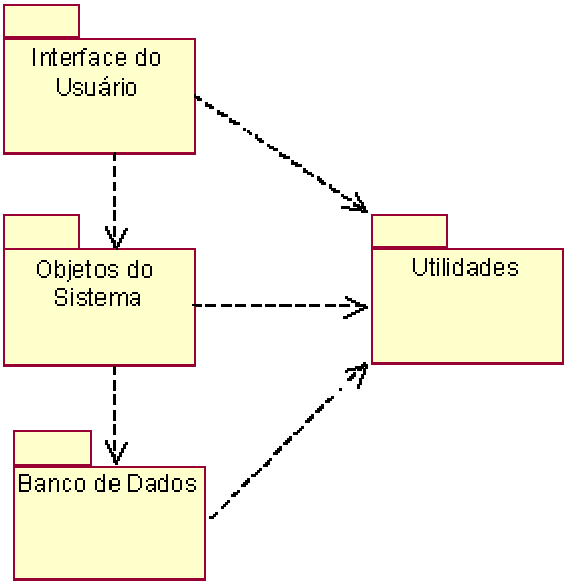
- Design da arquitetura: Este é o design de alto nível onde os pacotes (subsistemas) são definidos, incluindo as dependências e mecanismos de comunicação entre eles. Naturalmente, o objetivo é criar uma arquitetura simples e clara, onde as dependências sejam poucas e que possam ser bidirecionais sempre que possível.

- Design detalhado: Esta parte detalha o conteúdo dos pacotes, então todas classes serão totalmente descritas para mostrar especificações claras para o programador que irá gerar o código da classe. Modelos dinâmicos do UML são usados para demonstrar como os objetos se comportam em diferentes situações.

Design da arquitetura

Uma arquitetura bem projetada é a base para futuras expansões e modificações no sistema. Os pacotes podem ser responsáveis por funções lógicas ou técnicas do sistema. É de vital importância separar a lógica da aplicação da lógica técnica. Isso facilitará muito futuras mudanças no sistema.

Em nosso caso de estudo, identificamos 4 pacotes (subsistemas):



- Pacote da Interface do Usuário: Estarão contidas as classes para a criação da interface do usuário, para possibilitar que estes acessem e entrem com novos dados no sistema. Estas classes são baseadas no pacote Java AWT, que é o padrão Java para criação de interfaces. Este pacote coopera com o pacote de objetos do sistema, que contém as classes onde os dados estão guardados. O pacote de interface chama operações no pacote de objetos do sistema para acessar e inserir novos dados.

- Pacote de Objetos do Sistema: Este pacote inclui classes básicas, ou seja, classes que foram desenvolvidas exatamente para tornar o sistema em desenvolvimento funcional. Estas classes são detalhadas no design, então são incluídos operações e métodos em sua estrutura e o suporte à Persistência é adicionado. O pacote de objetos deve interagir com o de banco de dados e todas as suas classes devem herdar da classe Persistente do pacote de banco de dados

- Pacote de Banco de Dados: Este pacote disponibiliza serviços para as classes do pacote de objetos fazendo com que os dados armazenados no sistema sejam gravados em disco.

- Pacote de Utilidades: Este contém serviços que são usados por todos os outros pacotes do sistema. Atualmente a classe ObjId é a única no pacote, e é usada para referenciar os objetos persistentes em todo o sistema.

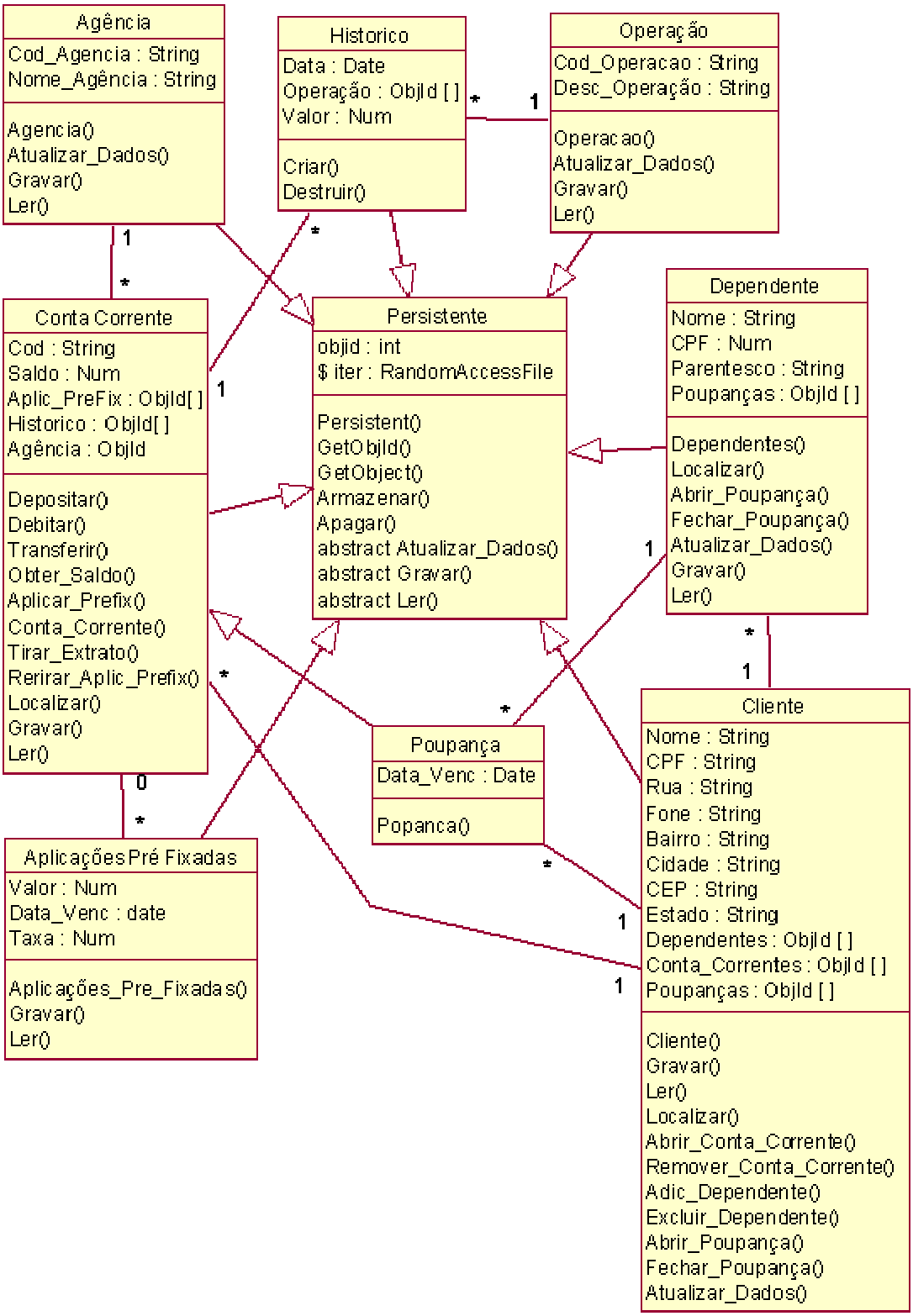
Design detalhado

O propósito do design detalhado é descrever as novas classes técnicas do sistema, como classes de criação da interface, de banco de dados e para expandir e detalhar a descrição das classes de objetos, que já foram definidas na fase de análise.

Tudo isto é feito com a criação de novos diagramas de classes, de estado, e dinâmicos. Serão os mesmos diagramas criados na fase de análise, mas é um nível de detalhamento técnico mais elevado.

As descrições de use-cases provenientes da fase de análise são usados para verificar se estes estão sendo suportados pelos diagramas gerados na fase de design, e diagramas de sequência são usados para ilustrar como cada use-case é tecnicamente implementada no sistema.

Chegamos a um diagrama de classes mais evoluído com a inclusão de persistência.



Criamos os diagramas de sequência para funções do sistema, descritas no diagrama de use-cases, já possuindo os parâmetros para cada mensagem entre os objetos.

O layout das janelas deve ser criado com alguma ferramenta visual de acordo com a preferência do desenvolvedor. Ferramentas visuais já geram o código necessário para a criação de janelas. Algumas ferramentas já suportam a adição de controladores de eventos para eventos disparados por usuários como cliques em botões. O ambiente gera um método ‘okbutton_Clicked’ que será chamado quando o botão "OK" for pressionado.

A aplicação resultante da interface de usuário é uma janela principal com um menu de opções. Cada opção escolhida do menu mostrará uma janela nova que juntas serão responsáveis por receber as informações do usuário e executar a função a qual se propõem a fazer.

12.4. Implementação

A fase de construção ou implementação é quando as classes são codificadas. Os requisitos especificam que o sistema deve ser capaz de rodar em diversos tipo de processadores e sistemas operacionais, então a linguagem escolhida foi Java.

Pelo fato de em Java cada arquivo poder conter uma e somente uma classe, podemos facilmente escrever um diagrama de componentes contendo um mapeamento das classes provenientes da visão lógica.

Agora codificamos cada classe do pacote de objetos do sistema, a interface, o banco de dados e o pacote de utilidades. A codificação deve ser baseada nos modelos desenvolvidos nas fases de análise de requisitos, análise e design, mais precisamente nas especificações de classes, diagramas de classes, de estado, dinâmicos, de use-cases e especificação.

Existirão algumas deficiências durante a fase de codificação. A necessidade da criação de novas operações e modificações em operações já existentes serão identificadas, significando que o desenvolvedor terá que mudar seus modelos da fase de design. Isto ocorre em todos os projetos. O que é mais importante é que sejam sincronizados a modelagem de design com a codificação, desta forma os modelos poderão ser usados como documentação final do sistema.

12.5. Testes

A aplicação deverá ser testada. Deve-se verificar se o programa suporta toda a funcionalidade que lhe foi especificada na fase de análise de requisitos com o diagrama de use-cases. A aplicação deve ser também testada da forma mais informal colocando-se o sistema nas mãos dos usuários.

13. Conclusão

O criação de uma linguagem para a comunidade de desenvolvedores em orientação a objetos era uma necessidade antiga. A UML realmente incorporou muitos recursos com dão a linguagem uma extensibilidade muito grande.

Os organização da modelagem em visões e a divisão dos diagramas especificando características estáticas e dinâmicas do sistema tornou a UML fácil de ser utilizada e fez com que qualquer tipo de comportamento possa ser visualizado em diagramas.

A modelagem visual orientada a objetos agora possui um padrão, e esse padrão é extremamente simples de ser escrito a mão, sendo robusto para especificar e descrever a grande maioria das funções, relacionamentos e técnicas de desenvolvimento orientado a objetos que hoje são utilizados. Novas técnicas irão surgir e a UML também estará preparada já que tudo estará baseado nas idéias elementares da orientação a objetos.

Sem dúvida alguma a UML facilitará às grandes empresas de desenvolvimento de software uma maior comunicação e aproveitamento dos modelos desenvolvidos pelos seus vários analistas envolvidos no processo de produção de software já que a linguagem que será

utilizada por todos será a mesma, acabando assim com qualquer problema de interpretação e mal-entendimento de modelos criados por outros desenvolvedores. Os modelos criados hoje poderão ser facilmente analisados por futuras gerações de desenvolvedores acabando com a diversidade de tipos de nomenclaturas de modelos, o grande empecilho do desenvolvimento de softwares orientados a objetos.

Os fabricantes de ferramentas CASE agora suportarão a UML em seus softwares e a fase de codificação será cada vez mais substituída pela geração de código automático desempenhada pelas ferramentas CASE.