UNIFEI – Universidade Federal de Itajubá
IEE – Instituto de Engenharia Elétrica
DON – Departamento de Eletrônica

MODELO PRÁTICO PARA DESENVOLVIMENTO DE SISTEMAS ORIENTADO A OBJETO BASEADO NA UML

Dissertação de Mestrado apresentada à Universidade Federal de Itajubá para obtenção do título de Mestre em Engenharia Elétrica.

Orientadora: Profª Drª Lúcia R. Horta Rodrigues Franco

Autor: Carlos Henrique Rodrigues Cardoso

ITAJUBÁ - MG
Novembro/2002

UNIFEI – Universidade Federal de Itajubá
IEE – Instituto de Engenharia Elétrica
DON – Departamento de Eletrônica

MODELO PRÁTICO PARA DESENVOLVIMENTO DE SISTEMAS ORIENTADO A OBJETO BASEADO NA UML

Dissertação de Mestrado apresentada à Universidade Federal de Itajubá para obtenção do título de Mestre em Engenharia Elétrica.

Orientadora: Profª Drª Lúcia R. Horta Rodrigues Franco

Participantes da Banca Examinadora:
Prof. Dr. Jorge Luis Risco Becerra (USP)
Prof. Dr. Leonardo de Mello Honório (UNIFEI)
Profª Drª Lúcia Regina Horta Rodrigues Franco (UNIFEI)

Autor: Carlos Henrique Rodrigues Cardoso

DEDICATÓRIA

Eu dedico este trabalho ao Grande Arquiteto do Universo que tornou a nossa existência possível, minha esposa Rita pela incansável parceria, compreensão e amor, a meus filhos Bruno e Carla que justificam qualquer esforço e a meus pais Jair e Marilda pelo “investimento” que fizeram durante toda a minha vida.

AGRADECIMENTOS

Agradeço a toda a minha família pela compreensão durante o desenvolvimento deste trabalho. Agradeço ainda à minha orientadora Lúcia pelas valiosas orientações durante a pesquisa e montagem deste trabalho. Agradeço ao Afonso Soares pelas longas horas de discussão (e teimosia) sobre engenharia de software, ao Prof. Dr. Mauro Spinola pelas valiosas orientações e a toda equipe do Inatel Competence Center - SW que permitiu o desenvolvimento de ferramentas e a aplicação e validação prática do modelo.

ÍNDICE

ABSTRACT

This paper shows a process, or model, of software development based on UML, focusing the practical use of UML diagrams. This process generates diagrams step by step so it can change the development activities interconnected. The process's name is Practical Model to Software Development oriented to objects based on UML, or MPDS.

The MPDS is a modeling process presented through activities and diagrams (i.e., the UML diagrams) and oriented to Use Case[40]. It allows in a clear and consistent way to develop a system of medium duration from 6 to 12 months of continuous work, developed by a team with no or little experience in software modeling.

The MPDS's purpose is to allow a system that can be developed and maintained in a consistent way and that the performed work can, as always as possible be used in new projects. The models' use as MPDS[3][22] can increase considerably the quality associated with the speed in the systems development, making the small companies obtain a high competitive level.

RESUMO

Este trabalho apresenta um processo, ou modelo, de desenvolvimento de software baseado na UML focalizando o uso prático dos diagramas da UML. Este processo gera os diagramas passo-a-passo de modo a tornar as atividades de desenvolvimento interligadas. O nome deste processo é Modelo Prático para Desenvolvimento de Software orientado a Objetos baseado na UML, ou MPDS.

O MPDS é um processo de modelagem apresentado através de atividades e diagramas (no caso, os diagramas da UML) e orientado ao caso de uso[40]. Ele permite de forma clara e consistente desenvolver um sistema de duração média de 6 a 12 meses de trabalho contínuo, desenvolvido por uma equipe com nenhuma ou pouca experiência em modelagem.

A finalidade do MPDS é permitir que um sistema possa ser desenvolvido e mantido de forma consistente e que o trabalho realizado possa, sempre que possível, ser aproveitado em novos projetos. O uso de modelos como o MPDS [3][22] pode aumentar consideravelmente a qualidade associada à rapidez no desenvolvimento de sistemas fazendo com que pequenas empresas tenham um alto grau de competitividade.

LISTA DE FIGURAS

SÍMBOLOS, SIGLAS E ABREVIATURAS

- UML – *Unified Modeling Laguage* (Linguagem de Modelagem Unificada)
- OO – *Object Orientation* (Orientação a Objetos)
- MPDS – Modelo Prático para Desenvolvimento de Sistema
- OMG – Object Management Group – Grupo responsável pela definição da UML
- Sistemas Embarcados – Programas desenvolvidos para executar em hardware dedicado como por exemplo: telefones celulares, elevadores, ou seja, equipamentos eletro-eletrônicos em geral que não sejam microcomputadores.
- Clientes – ou Clientes do Sistema são as pessoas responsáveis pelo fornecimento dos requisitos do sistema, ou seja, as pessoas que conhecem as necessidades do usuário. Pode eventualmente ser os próprios usuários.
- Usuários – ou Usuários do Sistema são as pessoas que irão utilizar o sistema após o desenvolvimento, ou seja, as pessoas que terão de fato interação com o sistema, que utilizarão o sistema. Pode eventualmente ser os próprios clientes.
- *Framework* – Representa um conjunto de componentes (classes ou subsistemas) que pode e deve ser utilizado no desenvolvimento de novos sistemas. Os componentes de um *framework* devem ser testados e documentados para facilitar a reutilização.
- Desenvolvimento Paralelo – É a capacidade de se distribuir adequadamente as atividades entre os componentes da equipe para agilizar o processo de desenvolvimento.
- Incremento – Um incremento representa uma diferença de tempo entre duas versões do sistema.

- Iteração – Uma iteração representa uma seqüência de atividades distinta e planejada e com uma forma de avaliação resultando em uma versão.

1. INTRODUÇÃO

1.1 O Início da Engenharia de Software

Durante muito tempo o desenvolvimento de software foi realizado de forma empírica, sem um processo definido. Por muitas vezes os autores e usuários do sistema eram as mesmas pessoas ou faziam parte da mesma equipe. O que implicava que o uso e o conhecimento do sistema ficasse restrito a um grupo. Com a popularização dos computadores, o número de usuários aumentou drasticamente e foi necessário o desenvolvimento de interfaces amigáveis que permitissem que não especialistas em computação pudessem utilizá-los. Porém o desenvolvimento dos sistemas continuava caótico. Algumas técnicas surgiram como orientação a objetos que, a principio, prometiam melhorar o desenvolvimento e a reutilização de código. Porém a popularização das linguagens orientadas a objeto, que permitiriam que o processo de desenvolvimento fosse alterado, não ocorreu de forma substancial.

Na década de 90 cada pesquisador imaginava que o seu modelo era o melhor. Houve uma época conhecida como a “guerra dos modelos” até que os três principais pesquisadores (Jacobson, Rumbaugh e Booch) decidiram unir os trabalhos. Eles criaram um novo modelo utilizando o que cada um tinha de melhor. Esta união gerou a Linguagem de Modelagem Unificada - UML (*Unified Modeling Language*). Isto foi decisivo para a padronização. Submeteram este nova linguagem ao OMG (*Object Management Group*) que a transformou em um padrão aberto. A UML tem a cada dia se estabelecido como um padrão para o desenvolvimento de software e mais e mais pessoas estão utilizando-a.
Normalmente o desenvolvimento de projetos realizado por pequenas empresas (ou grupos) não utiliza nenhum tipo de modelagem em função do desconhecimento e da complexidade dos diagramas. Os sistemas são desenvolvidos de forma ad hoc e as promessas da Orientação a Objetos, como reutilização de código ou facilidade de manutenção não tem acontecido na maioria dos projetos.

Este trabalho cria um processo, ou modelo, de desenvolvimento de software baseado na UML focalizando o uso prático dos diagramas da UML. Tal processo gera os diagramas passo-a-passo de modo a tornar as atividades de desenvolvimento interligadas. O nome deste processo é Modelo Prático para Desenvolvimento de Software Orientado a Objetos baseado na UML, ou MPDS. O Prático do nome do processo esta relacionado a principal fonte de informações para a sua criação, ou seja, projetos reais.

O MPDS é um processo de modelagem apresentado através de atividades e diagramas (no caso, os diagramas da UML) e orientado ao caso de uso[40]. Este processo permite de forma clara e consistente desenvolver um sistema de duração média de 6 a 12 meses de trabalho contínuo, desenvolvido por uma equipe com nenhuma ou pouca experiência em modelagem.

A finalidade do MPDS é permitir que um sistema possa ser desenvolvido e mantido de forma consistente e que o trabalho realizado possa, sempre que possível, ser aproveitado em novos projetos. O uso de modelos como o MPDS [3][22] pode aumentar consideravelmente a qualidade associada à rapidez no desenvolvimento de sistemas fazendo com que empresas inexperientes (empresas incubadas por exemplo) tenham um alto grau de competitividade. O aumento da qualidade e rapidez esta relacionado com a possibilidade de descoberta e diminuição dos riscos associados a um projeto. Além disso a padronização irá facilitar o aproveitamento de conhecimento adquirido em projetos anteriores.

Na maioria das vezes a documentação de projeto é considerada um peso extra e normalmente é feita à parte do projeto. A vantagem de se utilizar um processo como MPDS é que a documentação é gerada como base para o desenvolvimento, para negociar com o cliente, para o desenvolvimento e para testar, faz parte do processo. O MPDS utiliza um único documento para todo o projeto. Deste modo serve de auxílio e troca de informações entre os membros da equipe. Mesmo porque se for necessário utilizar a documentação, quando é feita à parte, não será aderente ao produto desenvolvido. Vale a pena ressaltar que a finalidade primordial do MPDS não é gerar documentação. A documentação é um produto natural do trabalho durante o desenvolvimento do projeto.

1.2 Os problemas

O desenvolvimento de software é uma atividade extremamente complexa e subjetiva por envolver como principal "matéria prima" pessoas (a principal preocupação do CMM - *Capabity Maturity Model*[35] no nível 2 é a gerência intimamente ligado a pessoas) e por envolver tecnologia muito nova [17][25]. Além disto, o relacionamento entre clientes e desenvolvedores sem um procedimento definido gera dificuldades e não raro leva a atritos que terminam em quebra de contrato e prejuízo para ambas as partes.

Uma tentativa de organizar o desenvolvimento foi a criação da técnica de programação estruturada utilizada em linguagens como Cobol, Fortran, C entre outras. O desenvolvimento utilizando a técnica estruturada é baseado no fluxo de execução do sistema. Nesta técnia é difícil a mudança após o início da implementação. O uso de um procedimento estruturado é mais artificial do que outras técnicas, o que dificulta muito a mudança. Via de regra o cliente, muitas vezes, muda de opinião depois do início do desenvolvimento. E isto acontece principalmente por falta de conhecimento das necessidades reais do sistema ou da dificuldade de priorizar [41][51].

Tempos mais tarde e tentando fazer com que o desenvolvimento fosse mais próximo do mundo real foi criada a técnica de desenvolvimento orientado a objetos. Os dados e operações são armazenados em classes que possuem um comportamento claro. As principais linguagens são Ada, Java e C++[45][54]. As técnicas de desenvolvimento vem evoluindo muito nos últimos anos. Porém a técnica de Orientação a Objetos não consegue resolver sozinha os problemas ligados a desenvolvimento de software. A orientação a objetos permitiu a criação de uma linguagem gráfica chamada UML (*Unified Modeling Language*) que serve de base para o MPDS. Muitas vezes programas orientados a objetos não foram desenvolvidos utilizando realmente as técnicas de orientação a objetos. As definições como encapsulamentos, isolamento, atribuição de responsabilidades, entre outras nem sempre são aplicadas ao desenvolvimento.

É impossível desenvolver um sistema que satisfaça a todas as necessidades de uma só vez, principalmente se estas necessidades vão surgindo enquanto se desenvolve.

É importante estar preparado para administrar e gerenciar as necessidades do cliente[27][29]. É mais fácil ter um modelo que preveja as mudanças e inclusão de requisitos, do que tentar congelar as necessidades em um determinado momento, não dá para escolher e definir o que o cliente precisa.

Além do cliente tem-se um problema na outra ponta, a equipe de desenvolvimento. Existe uma frase que diz que "o desenvolvedor de software é discípulo de **Ivonsaf**", ou seja é discípulo da "**I**rresistível **von**tade de **sa**ir **f**azendo". Isto talvez seja uma característica normal à engenharia, principalmente às novas engenharias, como a engenharia de software, em função da facilidade de se iniciar um projeto. Neste campo as ferramentas e novas linguagens "visuais" fizeram um desserviço a tecnologia de desenvolvimento, principalmente para pequenos grupos. A frase que mais se ouve é "Não há tempo a perder com projeto, precisamos iniciar o desenvolvimento". Começar o desenvolvimento de imediato é como fazer uma casa sem planta baixa. A chance de dar errado é muito grande.

O desenvolvimento de software nos últimos anos passou a ser realizado em países do terceiro mundo em função do barateamento dos computadores e facilidade no acesso a informações através da internet (ver exemplo da Índia). A cada dia o hardware esta mais barato e potente. A intenção é facilitar o desenvolvimento rápido de novos programas com qualidade, escopo, prazos e custos adequados e atendendo a um maior número de clientes com necessidades diferentes.

O desenvolvimento de software é uma atividade que demanda um planejamento extremamente eficiente ao qual os projetistas não estão acostumadas a fazer. Não é muito difícil de se aprender uma nova linguagem de programação. Porém cada desenvolvedor possui um método de trabalho próprio que foi desenvolvido durante a carreira e que lhe é totalmente particular. Isto é um dos maiores problemas no desenvolvimento em equipe, unem-se vários "artistas" que acham que "meu jeito é o melhor" e não se consegue padronizar o desenvolvimento. Alguém sai da equipe e ninguém sabe como substituí-lo.

É importante que exista um estudo acadêmico no sentido de gerar tecnologia de modelagem e processo de desenvolvimento dedicado a empresas inexperientes em

engenharia de software. Esta é a motivação desta dissertação: apresentar um processo baseado em um modelo conhecido e unificado como a UML. Porém apresentando: os caminhos, as atividades que devem ser feitas, o porquê deve ser feito, como uma atividade está ligada a próxima, como um diagrama está ligado ao próximo.

1.3 A Engenharia de Software e o surgimento da UML

Até o início da década de noventa pouco se dizia a respeito de engenharia de software. Aliás nem este termo era muito utilizado, dando-se preferência para o termo programação. Alguns como Roger S. Pressman foram pioneiros em encarar o desenvolvimento de sistemas como sendo uma engenharia. Porém com características muito particulares. Software é considerado como sendo "uma moeda com duas faces: é ao mesmo tempo um produto e um veículo que entrega um produto"[39]. Este é o caso dos programas que estão em aparelhos celulares por exemplo.

No início da década de 90 viu-se o que ficou conhecido como a guerra das metodologias [9], ou seja, vários pesquisadores de engenharia de software diziam que a sua metodologia era o melhor e iria resolver todos os problemas. Infelizmente as coisas não são tão simples assim e em meados da década decidiu-se que era mais prudente reunir o que houvesse de melhor em cada proposta e criar um modelo único. Foi assim que Jacobson, Booch e Rumbaugh criaram a UML (*Unified Model Language*) onde pretendia-se apresentar uma ferramenta unificada que servisse de sustentação ao desenvolvimento.

A UML inicialmente foi apresentada como um padrão e um dos fatores mais importantes para o uso da UML foi o fato de existir uma ferramenta chamada *Rose*. Esta ferramenta foi desenvolvida pela Rational, uma empresa composta pelos três principais criadores da linguagem.

Aderiram ao modelo empresas como Digital, HP, I-Logix, Intelicorp, IBM, ICON, MCI, Microsoft, Oracle, Rational, Texas e Unisys. Isto permitiu a criação da versão 1.0 da UML que foi padronizado pela OMG em janeiro de 1997. Entre janeiro e junho de 97 outras empresas, como Ericsson e Andersen, passaram a fazer parte do grupo e em

novembro de 97 foram aprovadas as revisões na UML gerando a versão 1.1. Em junho de 98 foi lançada a versão 1.2 e em dezembro de 98 foi lançada a versão 1.3 que é a versão mais utilizada[56].

A UML, apesar de baseada em estudos com mais de 10 anos, é ainda muito recente. Isto faz com que muitos cursos e profissionais ainda não tenham tido tempo de adquirir o conhecimento e aplicá-lo no dia a dia do desenvolvimento. A UML é uma ferramenta. É apresentada como um conjunto de diagramas com suas respectivas finalidades, porém não é responsável em apresentar um processo de engenharia. Por isto não orienta o processo de desenvolvimento[57].

1.4 O contexto desta dissertação

Esta dissertação apresenta o MPDS, cuja finalidade é servir de metodologia de desenvolvimento de sistemas para empresas inexperientes (empresas incubadas) que estejam envolvidos em projetos com duração de 6 a 12 meses. Estes fatores, considerados como condição de contorno, fazem com que o uso de um processo adequado e bem sustentado possa servir de guia ao desenvolvimento. A UML é muito abrangente e complexa para ser utilizada sem um processo, principalmente por grupos inexperientes. O MPDS foi desenvolvido para atender estes grupos.

1.5 Os resultados alcançados

Apresenta-se com este trabalho um método chamado de Modelo Prático para Desenvolvimento de Sistemas Orientados a Objetos (MPDS) baseado em uma seqüência de diagramas da UML associados a atividades constituindo um processo de modelagem.

O MPDS foi aplicado em projetos acadêmicos no Curso de Computação III do Instituto Nacional de Telecomunicações – Inatel, além disto foi utilizado em projetos comerciais do Inatel Competence Center tão diferentes quanto ferramenta CASE, Sistema de Monitoramento de Chão de Fábrica via Web ou ainda Sistema de Tempo Real para Gerenciamento de Chamadas Telefônicas. Espera-se que com o MPDS os profissionais

da área de desenvolvimento de software assim como professores e estudantes possam utilizá-lo como uma ferramenta prática de desenvolvimento de sistemas.

O Processo Unificado UP(*Unified Process* que deu origem ao *Rational Unified Process* RUP)[9] foi desenvolvido pelos três amigos, criadores da UML, além do Philip Krutchen. A Intenção do UP é apresentar um processo que atende-se a qualquer tipo de projeto. Porém a complexidade deste processo é muito grande e existem vários papéis a serem desempenhados. Empresas incubadas (inexperientes e normalmente pequenas) não conseguem utilizar o UP e acabam não adotando processo algum.

1.6 O Modelo Prático para Desenvolvimento de Sistemas Orientados a Objetos baseada na UML (MPDS)

O desenvolvimento de software pode ser considerado uma atividade de raciocínio humano única e sem precedentes na história. É uma mistura de criatividade com tecnologia o que faz com que se necessite de um processo que atenda as expectativas tecnológicas sem restringir a criatividade no desenvolvimento do projeto. A criatividade é uma característica humana difícil de ser aprendida, quase impossível de ser ensinada e muito menos determinada por um processo. Já o conhecimento e a aplicação do conhecimento tecnológico pode e deve ser modelado, uniformizado e transmitido a fim de aumentar a produtividade e a qualidade do produto gerado[30][31].

Um ponto muito importante no desenvolvimento de um sistema (software) é a definição da arquitetura adequada. Isto não é um trabalho trivial e depende de experiência em projetos. É muito difícil para uma empresa incubada (e muitas vezes inexperiente) ter um arquiteto. Por isso empresas inexperientes devem definir uma área específica de produtos ligados a um único tipo de arquitetura (Sistemas embarcados, Sistemas para E-Comerce, Sistemas para Controle de Processo, etc). Isto até que a experiência e a estabilidade da empresa permitam experimentar áreas que exijam novos conhecimentos de arquitetura.

O MPDS foi baseado em uma linguagem já conhecida e abrangente, a UML, e propõe uma forma de desenvolvimento criado principalmente do prisma do desenvolvedor.

Apresenta de forma clara a seqüência do trabalho a ser desenvolvida, ou seja, onde começa e onde termina o trabalho e quais são as etapas a serem seguidas.
Vale salientar que o MPDS é um processo de engenharia de software. Está focado no desenvolvimento em si e não na gerência do desenvolvimento.
O MPDS segue a linha do "Problema ao Sistema" que permite identificar o(s) problema(s) que suscitou(aram) a necessidade de uma solução e desenvolver um sistema com prazo, custo e qualidade adequados.
Os passos do MPDS são:

- **Problema**: O problema na realidade é composto pelo conjunto de requisitos que o cliente tem que um sistema possa resolver. Existem inúmeras formas de resolver o mesmo problema, ou seja, existirão vários tipos de sistemas que podem ser desenvolvidos a partir destas necessidades. O que se quer garantir utilizando um modelo de engenharia de software é que o resultado atenda as necessidades com qualidade e que as melhorias possam ser feitas.

- **Levantamento e Análise de Requisitos**: Uma vez sendo apresentado aos problemas é preciso formatá-los e analisá-los de forma a levantar os possíveis riscos da não implementação do sistema nas condições apresentadas pelo cliente. Esta fase é fundamental porque representa um compromisso do que se está contratando e o que será entregue no final do projeto.

- **Modelagem de caso de uso**: Com os requisitos em mãos definem-se os atores e os casos de uso que irão atender a estes requisitos. Deve-se colher nos requisitos funcionais do sistema (requisitos com os quais o usuário irá interagir) os casos de uso que permitirão atendê-los.

- **Escolha do(s) caso(s) de uso a ser(em) trabalhada(s)**: A escolha do ou dos caos de uso a serem trabalhadas deve utilizar as avaliações de riscos e prioridades já

levantadas. Deve-se atacar os problemas mais difíceis primeiro e deixar o que já se tem domínio da solução para depois. Esta escolha também deve levar em conta quais os casos de uso são básicas para o funcionamento do sistema para que as outras a serem desenvolvidas possam ser incorporadas no incremento seguinte. Nesta fase criam-se protótipos de partes do sistema a fim de diminuir os riscos dos casos de uso o mais cedo possível.

- **Modelagem de classes de análise**: A partir dos casos de uso a serem trabalhadas determinam-se as classes de análise que as compõem. Este trabalho representa o primeiro passo de fragmentação do caso de uso no sentido do sistema.

- **Modelagem dinâmica utilizando o diagrama de seqüência**: A modelagem dinâmica é formada principalmente por duas entradas:
    - Para cada caso de uso os objetos referentes as classes de análise e os atores são colocados no diagrama de seqüência e,
    - O fluxo de eventos do caso de uso é utilizado para determinar as mensagens e sua seqüência no diagrama.

  As classes de análise serão fundidas ou desmembradas durante o processo de análise dinâmica dos objetos que representam estas classes. Este trabalho representa a **ponte** entre a análise e o projeto. É difícil definir neste ponto onde termina a análise e onde começa o projeto. O trabalho na modelagem dinâmica, principalmente no diagrama de seqüência deverá gerar a maioria das classes de projeto. Isto facilitará a busca das operações (e por conseqüência os métodos) e dos atributos, que as classes de projeto terão, além de permitir identificar os subsistemas que farão parte do sistema. Isto também facilita a reutilização de código já desenvolvido. Deve-se desenvolver os diagramas seqüência de análise com as mensagens do fluxo de eventos do caso de uso e as classes (objetos) de análise. Isto permite inclusive a validação dos fluxos de eventos. E após a

compreensão do modelo, permite desenvolver os diagramas de seqüência de projeto com operações no lugar das mensagens.

- **Modelagem de classes de projeto:** Após a modelagem dinâmica passa-se a modelagem estática das classes de projeto detalhando os relacionamentos, atributos e métodos, comportamento definido, subsistemas, pacotes, etc. As classes de projeto representam a última fase do projeto. A partir daí passa-se a geração de código que representa a implementação do projeto.

- **Geração de código:** A geração de códigos executáveis representam o início do final do incremento do MPDS que começou com a análise do problema. Um novo código será gerado e testado ao final de cada ciclo de processo, construindo assim as partes do sistema.

- **Testes:** Os testes dos códigos gerados a cada incremento facilitam a integração e o funcionamento do sistema. Todos os testes devem ser documentados para que possam ser repetidos a qualquer momento durante o desenvolvimento do projeto.

- **Sistema:** O sistema é a conclusão da montagem de cada parte de código gerado e testado em cada ciclo de desenvolvimento ou incremento. O ciclo será responsável por uma parte de código que é adicionado ao código já existente. Todas as partes do código devem ser testadas. O sistema está concluído quando todos os caos de uso forem desenvolvidas ou todos os requisitos forem atendidos.

- **Modelagem de Sistemas de Tempo Real:** Caso o sistema necessite se comportar de modo a responder a mensagens quase que instantaneamente onde “dado atrasado é um dado errado” [12], caracterizando um sistema de tempo real, deve-se utilizar o diagrama de estado para modelar o comportamento dinâmico relativo

aos estados. Os estados geralmente são implementados em métodos de estado de uma classe de controle.

- **Modelagem de Pacotes/Componentes**: O desenvolvimento de sistemas complexos ou de sistemas para internet irão necessitar da divisão em pacotes e componentes. Eles determinarão a forma de desenvolvimento e da montagem final. Os componentes são normalmente utilizados para sistemas de 3 ou mais camadas desenvolvidos, por exemplo com: COM+ (Microsoft) JavaBeans, J2EE (Sun). Existem sistemas que utilizam componentes e possuem somente uma camada.

A figura 1.1 a seguir mostra um diagrama de atividades a serem executadas e *labels*, representando os diagramas da UML utilizado durante o desenvolvimento. Este diagrama é o MPDS. A explicação dos tópicos anteriores deve ser acompanhada da visualização é análise desta figura.

Os capítulos 2 a seguir irá fazer uma apresentação introdutória da orientação a objetos e os capítulos 3 a 10 irão apresentar em detalhes cada passo do MPDS mostrado na figura 1.1. Os capítulos 11 e 12 apresentam as conclusões do trabalho.

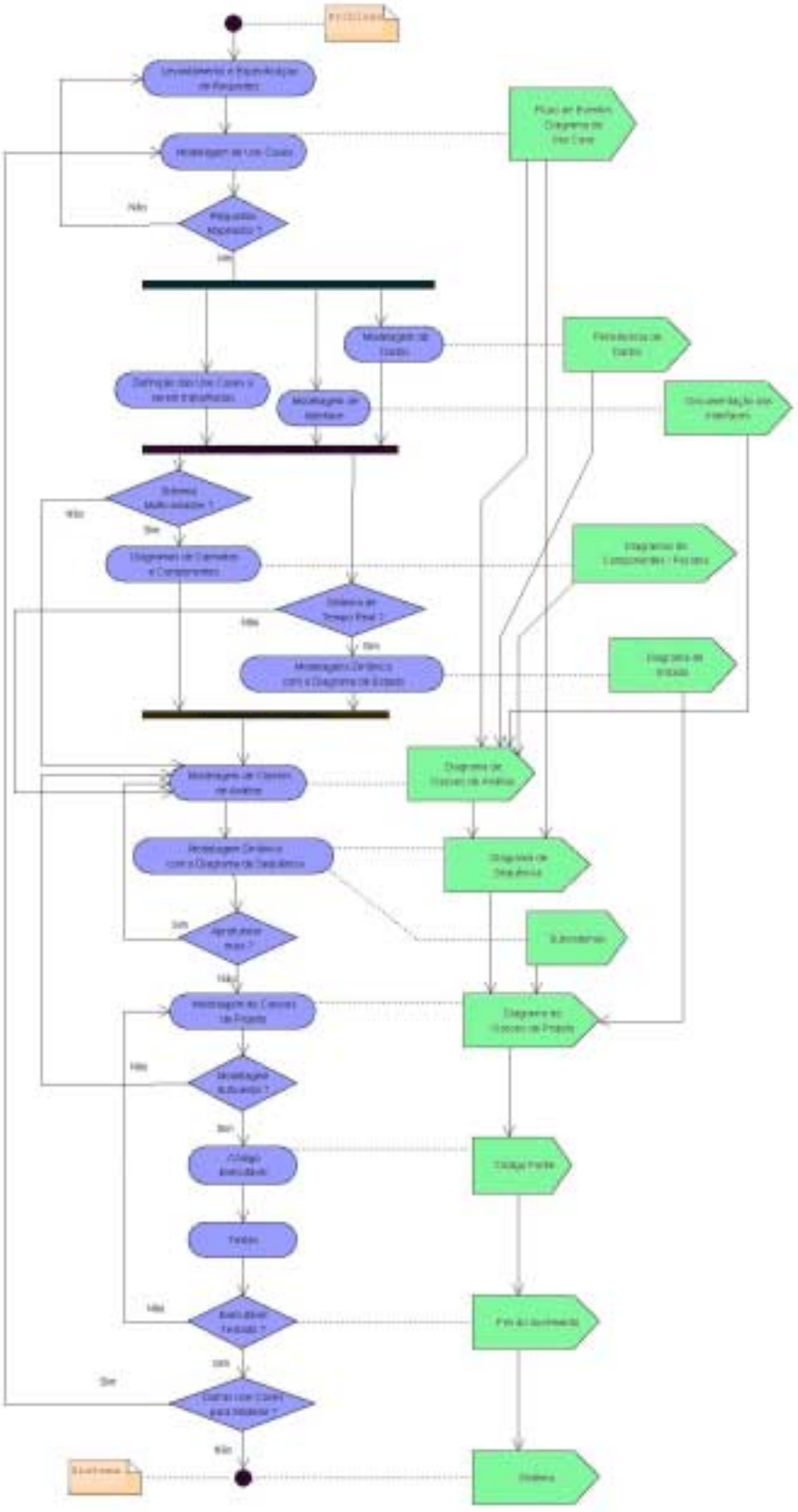


Figura 1.3 .MPDS - Atividades e Diagramas da UML

2. ORIENTAÇÃO À OBJETOS

2.1 O que é ?

O desenvolvimento de software sofreu evoluções que marcaram os rumos e geraram correntes como o desenvolvimento estruturado, que ainda hoje é utilizado, principalmente para sistemas embarcados. Apesar de ser muito utilizado, e com sucesso, o desenvolvimento estruturado possui uma característica negativa em muitos projetos quando o sistema cresce fica cada vez mais difícil a manutenção e continuidade do sistema.

Uma das correntes e que tem evoluído muito é o desenvolvimento orientado a objetos[54] onde as responsabilidades são divididas entre objetos que se relacionam e se comunicam[4][13]. Para exemplificar podemos imaginar o desenvolvimento de um software que simulasse uma agência bancária simples. Em um projeto estruturado definiríamos algumas estruturas como por exemplo, cliente, caixa, conta que iriam armazenar os dados e desenvolveríamos métodos que manipulariam estes dados para executar funções como pagamento, retirada, depósito, etc. O sistema se comporta em um *looping* onde os métodos são chamados em uma determinada seqüência para executar as operações necessárias.

Esta não é a forma como uma agência bancária funciona, as operações não são seqüenciais, as estruturas como cliente, caixa ou conta além de dados podem e devem executar operações.

Em um sistema orientados a objetos o cliente teria além dos dados(nome, cpf, endereço, etc) operações como por exemplo: retirarDinheiro, pagarConta, depositarDinheiro, etc, que são típicas de todos os clientes. Já o caixa realizaria operações como: fazerPagamento, verificarSaldo, transferirSaldo, etc. E o cliente teria um relacionamento com o caixa da agência para executar as operações. É desta forma que uma agência bancária funciona e é desta forma que a modelagem orientada a objetos encara o sistema a ser desenvolvido. Caso quiséssemos criar, por exemplo, um cliente especial,

deveríamos atualizar somente o cliente e não seria necessário alterar todo o programa, ou seja, os objetos estão isolados. A figura 2.1 mostra o diagrama de relacionamento entre cliente e caixa.

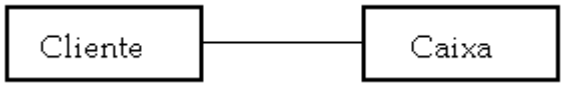


Figura 2.1 – Relacionamento entre cliente e caixa

2.2 Classes, Objetos e Instâncias

Objeto é um conceito que existe no mundo real (existem objetos no mundo real). Objeto é tudo aquilo com o qual nos podemos relacionar ou que relacionam entre si, de uma agulha a um vaso, de uma nuvem a uma vaca, o mundo real é repleto de objetos diferentes e com diferentes formas de relacionamentos. O fato de utilizar uma técnica de desenvolvimento que se aproxime do mundo real facilita a manutenção e evolução de sistemas. O desenvolvimento de software orientado a objetos utiliza os mesmos conceitos do mundo real para **simular** o que ocorre no dia-a-dia quando um sistema esta em execução.

Outra característica do desenvolvimento orientado a objetos, e que também pertence ao mundo real, é a definição do conceito de classe que representa um conjunto de objetos com características similares, quando foi dito que uma vaca é um objeto, na verdade refere-se a classe vaca, porque não foi identificada nenhuma vaca em particular, podemos então dizer que a Mimosa, a Fartura e a Malhada são objetos que pertencem a classe vaca. Em modelagem orientada a objetos diz-se que um objeto é uma instanciação de uma classe, ou seja, um objeto é o que existe de fato enquanto a classe é um conceito abstrato. E podemos dizer que apesar das vacas não serem semelhantes elas possuem uma série de características que permite associá-las a classe Vaca. A figura 2.2 mostra a relação de classe e objetos.

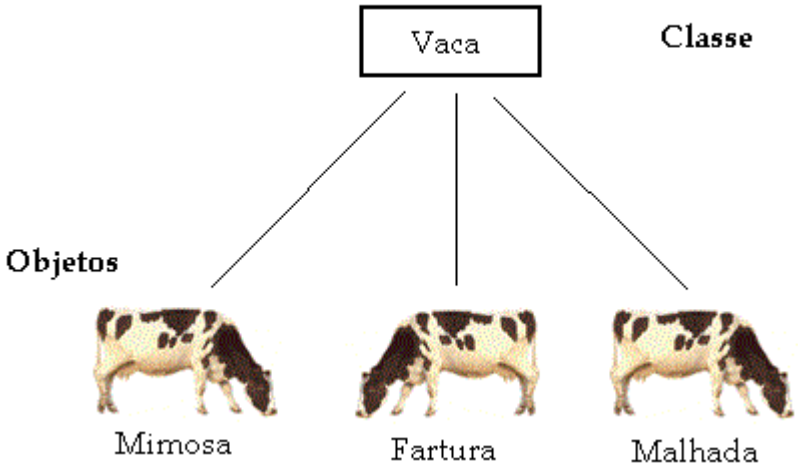


Figura 2.2 – Exemplo de Classe e Objeto (Instanciação da Classe)

2.3 Atributos e Operações

Um conceito associado a uma classe ou a um objeto é a propriedade ou um conjunto de propriedades que uma classe possui, por exemplo podemos associar a propriedade "fazer mooo" a classe Vaca. A propriedade de uma classe pode ser dividida em duas partes:

- Atributos: Onde é armazenado os dados do objeto, como no caso da classe Vaca podemos exemplificar o nome, idade, produção média de leite, entre outras.
- Operações: É o comportamento do objeto. Normalmente o termo operação é usado, porém algumas vezes pode ser chamado de serviço ou método ou ainda atividade. Na classe vaca temos, por exemplo, as seguinte operações: dar leite, pastar, ruminar, e outras.

A UML criou uma notação específica para representar uma classe[8][30] e podemos ver o exemplo na figura 2.3 a seguir:

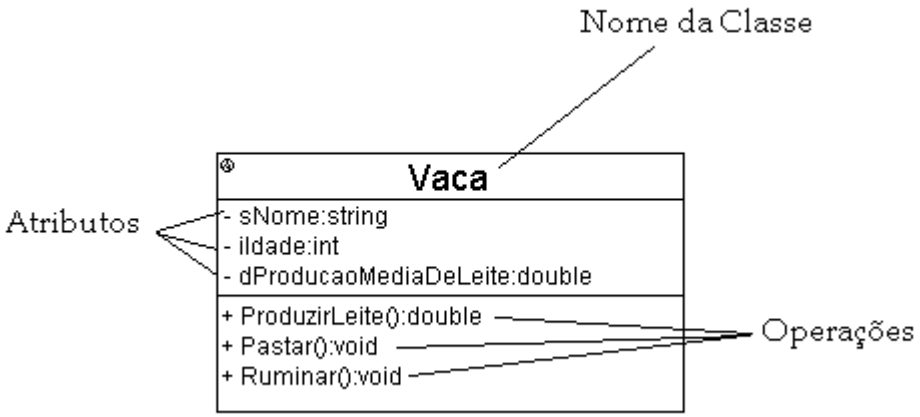


Figura 2.3 – Modelagem da Classe Vaca

É claro que o animal Vaca real é bem mais complexa que o nosso modelo(classe), porém o modelo deve ser suficiente para a representação da classe do ponto de vista de desenvolvimento de software ou do que se pretende que o sistema faça.

2.4 Relacionamentos

Como foi dito anteriormente os objetos relacionam entre si e como o que se faz é modelar os objetos é muito importante que os objetos possuam operações que permitam o relacionamento. Imagine um objeto da classe Fazendeiro que possui um operação chamada RetirarLeite que é executado quando o fazendeiro se relaciona com o objeto da classe Vaca, ou seja, é importante que o modelo contemple a operação produzirLeite para a classe Vaca, caso contrário este relacionamento não será possível. A seguir são apresentados os principais relacionamentos entre classes.

2.4.1 Herança

Uma classe pode possuir um relacionamento de herança com outra classe. Nestes casos dizemos que a classe “mãe” é menos especializada, ou seja, mais genérica e a classe “filha” é mais especializada. Este relacionamento é conhecido como “é um tipo”, ou “é do tipo”, por exemplo, “vaca é um tipo de mamífero”, portanto podemos dizer que a classe vaca possui um relacionamento de herança com a classe mamífero, ou também, que a classe vaca é derivada da classe mamífero. A representação de um relacionamento de herança é feito através de uma seta que vai da classe filha para a classe mãe[18], como mostrado na figura a seguir.

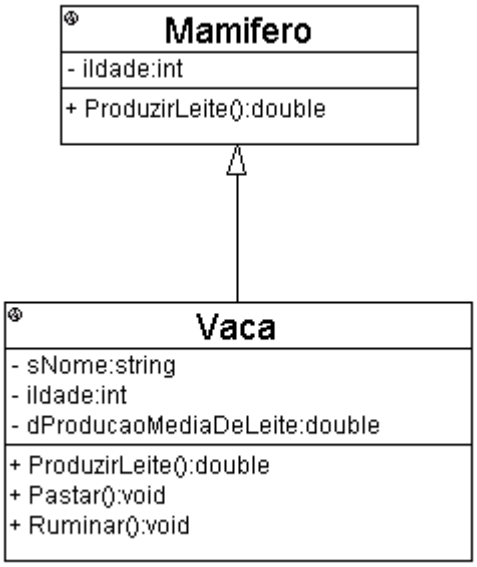


Figura 2.4 – Exemplo de Herança entre Classes

2.4.2 Agregação, Composição e Dependência

Outras formas de relacionamento são agregação, composição ou dependência[14][33]. Estes relacionamentos são semelhantes diferenciando em relação a intensidade. Um relacionamento de composição existe quando uma classe é composta por outra de tal

forma que a parte que compõe não existe se não existir o todo, ou seja, “a parte não vive sem o todo” é o caso por exemplo de um pedido de compras e os itens do pedido, se destruirmos o pedido de compras todos os itens serão destruídos, obrigatoriamente, porque não haverá mais compra.

A agregação é semelhante, porém a parte vive sem o todo, ou seja, se não existir o todo a parte continuará existindo, é o caso da vaca e o fazendeiro, onde existe um relacionamento de agregação. A figura 2.5 mostra o diagrama de classe com os relacionamentos de herança e agregação.

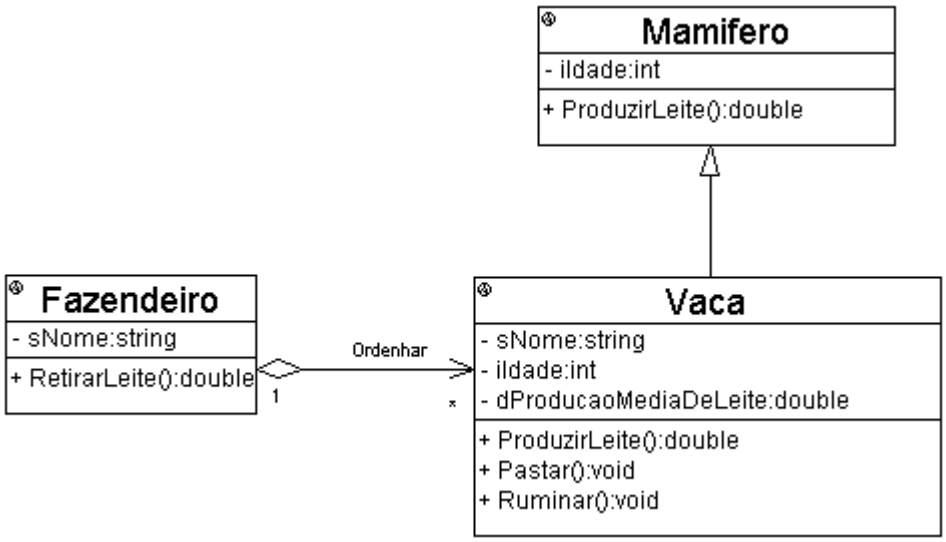


Figura 2.5 – Exemplo de Agregação entre Classes

Já uma dependência é quando uma classe depende de outra porém isto ocorre esporadicamente durante a vida dos objetos, é o exemplo entre uma fábrica e uma consultoria de marketing.

Estes conceitos facilitam o desenvolvimento da modelagem, porém na maioria dos casos práticos a geração do código, que é o que em última análise estamos buscando, pode mudar pouco e isto varia de linguagem para linguagem mas a compreensão do modelo é

fundamental para os grupos de desenvolvedores. São os relacionamentos que irão determinar a princípio as operações que uma classe deverá possuir para satisfazer ao relacionamento; esta é uma das formas de se determinar que operações deveremos ter.

2.4.3 Multiplicidade

No relacionamento entre duas classes existe uma determinada multiplicidade, por exemplo um fazendeiro pode possuir várias vacas, portanto o relacionamento chamando ordenhar que esta exemplificado na figura a seguir é do tipo “1 para muitos”. A princípio não se pode definir de antemão quantas vacas um fazendeiro terá que ordenhar, então deixamos o relacionamento em relação a vaca com o símbolo “*” (que é a notação de UML para muitos[8][14]) e do lado do fazendeiro “1” que esta em destaque na figura a seguir. Existem outros relacionamentos possíveis neste exemplo(todo fazendeiro também é mamífero, por exemplo) mas que não são relevantes para o modelo, devemos nos concentrar no comportamento que cada objeto deve ter para atender as necessidades do sistema. A figura 2.6 ressalta a multiplicidade entre as classes fazendeiro e vaca.

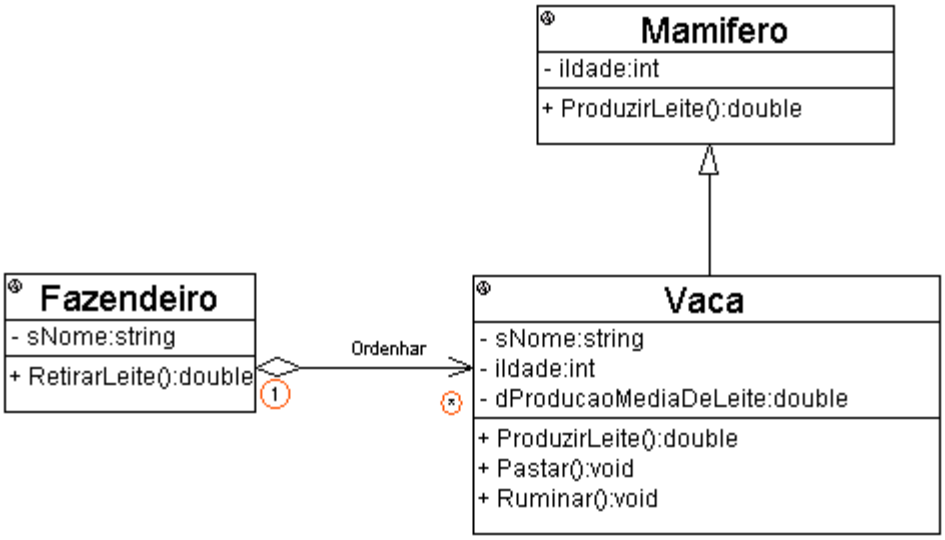


Figura 2.6 – Exemplo de multiplicidade de relacionamento entre classes

2.5 Polimorfismo

Polimorfismo é uma das características importantes no desenvolvimento orientado a objetos, polimorfismo é a propriedade que indica que uma operação pode, apesar de ter o mesmo nome, executar ações diferentes[54]. Existem dois tipos de polimorfismo, o estático e o dinâmico.

2.5.1 Polimorfismo Estático

Este conceito não é de simples compreensão e mais uma utiliza-se um exemplo para explicar. Imaginemos uma classe chamada Data que armazena os dados de dia, mês e ano além de hora, minuto e segundo, como mostrado, na figura 2.7, a seguir.

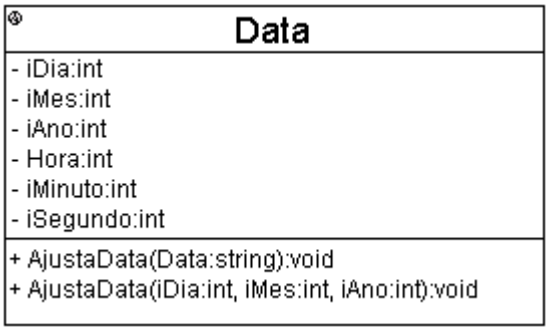


Figura 2.7 – Exemplo de polimorfismo estático

Ao chamar a operação AjustaData a operação exata que irá ser executada dependerá dos parâmetros que são passados, ou seja:

- Se utilizarmos AjustaData("15/09/2001") será executada a primeira operação, porém
- Se utilizarmos AjustaData(15,9,2001) será executada a segunda operação

A classe que se relaciona com a classe Data não necessita saber os detalhes da operação AjustaData, deve simplesmente usar a que melhor lhe convier.

A vantagem disto é que a classe pode atender a vários usuários da classe e cada um "enxergá-la" de maneira diferente. Os objetos do mundo real também executam as mesmas atividades porém atendendo de forma diferente; dependendo de como e quais os "parâmetros" são passados.

2.5.2 Polimorfismo Dinâmico

Uma outra forma de polimorfismo é o dinâmico, vamos mais uma vez utilizar um exemplo, imaginemos a classe FormaGeométrica que esta apresentada na figura 2.8 a seguir.

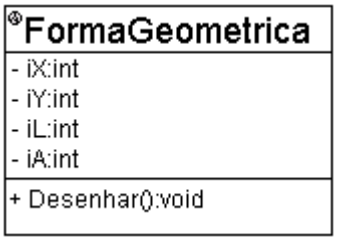


Figura 2.8 – Exemplo de Classe Abstrata

Esta classe possui uma operação chamada Desenhar, porém como desenhar uma forma geométrica se não sabemos exatamente como ela é ? Esta operação é chamada operação abstrata (o que faz com que a classe FormaGeométrica seja chamada de classe abstrata), ou seja, serve como referência, porém não é implementada, a implementação fica a cargo da classe filha(classe derivada) mostrada na figura 2.9 a seguir.

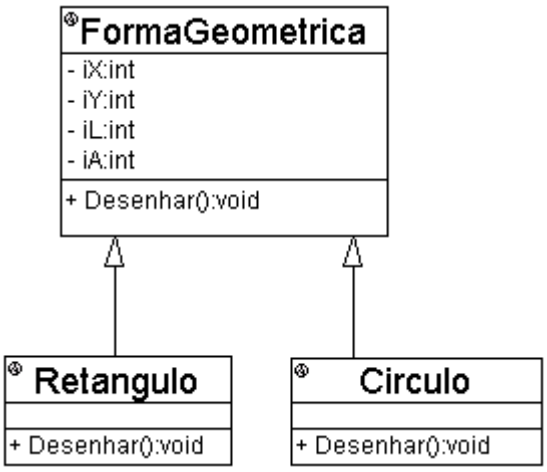


Figura 2.9 – Exemplo de polimorfismo dinâmico

Imagine que temos dois objetos (ou instanciação da classe) Retangulo e Circulo e o ponteiro para FormaGeometrica:

```
Retangulo R;
Circulo C;
FormaGeometrica *pFG;
```

Tem-se o seguinte código:

```
void Desenhar()
{
        pFG = &R; //O ponteiro aponta para o objeto retângulo
        pFG->Desenhar(); //É desenhado um retângulo
        pFG = &C; //O ponteiro aponta para o objeto círculo
        pFG->Desenhar(); //É desenhado um círculo
}
```

Podemos observar que simplesmente chamando a operação Desenhar de um objeto abstrato FormaGeometrica que aponta para o objeto não abstrato, e o objeto executa a operação Desenhar correta.

3. CAPTURANDO REQUISITOS

3.1 Requisitos do Cliente X Classes de Projeto

O objetivo final da modelagem é desenvolver o diagrama de classes de projeto, ou seja, desenvolver de classes onde será implementado o código que executará as funções do sistema. O diagrama de classes de projeto é composto pelas classes com seus atributos e operações e pelo relacionamento entre elas. O processo de desenvolvimento deve ser feito por incrementos[9][35]. O que será descrito deste ponto em diante é como adquirir os requisitos (ou necessidades do cliente, "o problema") e como caminhar até o diagrama de classes de projeto ("o sistema"). Como o processo é incremental, ao final do primeiro incremento, gera-se parte do código e inicia-se novamente o processo para os requisitos que não foram trabalhados, caminhando-se novamente pela seqüência de atividades e diagramas que serão apresentados, e assim repetindo para cada incremento.

3.2 Como obter os Requisitos

Requisito é o nome dado a todos os tipos de necessidades que se identificam para um sistema[26]. Normalmente é obtido através de entrevistas com os clientes ou alguém que conheça a necessidade dos usuários. Estas entrevistas devem ser conduzidas utilizando-se *checklists* para orientar as perguntas.

A obtenção dos requisitos do sistema não é uma atividade trivial[39], uma vez que quem os fornecerá muitas vezes não sabe de antemão tudo que deva ser implementado, ou seja, sabe qual é o problema mas não tem idéia de como é a solução. É por estas e outras razões que se deve dedicar uma atenção muito especial para obtê-los. É, normalmente, durante a obtenção dos requisitos que os responsáveis pelo desenvolvimento interagem com os problemas e necessidades do cliente e acabam conhecendo e aprendendo sobre o negócio[58] no qual o sistema estará envolvido. Portanto esta fase é fundamental para o sucesso do sistema.

O responsável pelo desenvolvimento deve ser capaz e estar disposto a aprofundar-se nos problemas do cliente. Não se deve deixar nenhum tipo de requisito de fora do levantamento[59] por mais que pareça impossível de realizá-lo no momento. Esta avaliação deve ser feita em um outro momento. Nestes casos deve ser negociado para que estes requisitos se transformem em requisitos futuros. É óbvio que isto só será possível caso estes os mesmos não sejam prioritários. Deve-se deixar claro que este requisito terá um impacto muito grande no desenvolvimento do projeto, devendo ter grande influência no prazo de entrega do sistema[47].

Uma forma de facilitar a especificação dos requisitos é dividi-los nos seguintes tipos:

- Requisitos Funcionais(F) – Especifica uma ação que o sistema deverá ser capaz de realizar, sem levar em consideração restrições físicas como por exemplo, um requisito que especifique o comportamento das entradas e saídas do sistema.
- Requisitos de Dado(D) – Referem-se à estrutura estática do software. Estes requisitos podem ser atributos simples ou aqueles que irão formar uma base de dados lógica. Para este último tipo deverão possuir especificações tais como: tipo de informações usadas por várias funções, freqüência de uso, capacidade de acesso, entidades e relacionamentos, restrições de integridade e requisitos de retenção de dados.
- Requisito de Interface(I) – Refere-se a interfaces de cliente e também com outros sistemas de software ou de hardware ou ainda de comunicação. Deve-se detalhar as interfaces do cliente (formato de tela, formato de relatórios, estruturas de menus, funções de teclas, etc.), interfaces do software e hardware (outros produtos de software requeridos tais como, DBMS, Browser Web, sistema operacionais e interfaces com outros sistemas de aplicação, indicando nome, versão e fabricante, incluindo definições do comportamento de cada uma delas) e ainda como exemplo, interfaces de comunicação tais como redes LAN (TCP/IP, Sockets, RPC, etc.).
- Requisito não funcional(N) – Requisito não funcional ou requisito especial pode incluir; requisitos legais ou referente a algum regulamento, a aplicação de padrões e também os atributos de qualidade do sistema a ser construído. Adicionalmente, outros

requisitos tais como; o sistema operacional a ser utilizado, requisitos de compatibilidade e restrições de projeto, também deverão ser capturados neste item.

Uma forma interessante para identificar os requisitos é utilizar a seguinte definição:

ER[f|a][F|D|I|N][XXXX].N onde
ER significa Especificação de Requisitos
[f|a] significa requisito futuro ou atual
F|D|I|N apresenta em qual tipo de requisito esta associado
XXXX é o número ou sigla do projeto que o requisito esta associado
N é o número seqüencial de requisitos do projeto

3.3 Os Riscos e Prioridades associados a Requisitos

Uma atividade muito importante na Especificação de Requisitos é determinar qual a prioridade o cliente associa a cada um dos requisitos levantados. Em geral o cliente tem uma tendência de dizer que todos são de altíssima prioridade, porém isto nem sempre é verdade. Esta negociação é muito importante e é uma atividade muito subjetiva e difícil de ser conduzida por pessoas extremamente técnicas. É nesse momento que o papel de gerência de desenvolvimento de software entra em cena, não se pode dizer sim a tudo que o cliente pede, deve-se ser bastante claro ao mostrar as dificuldades de se desenvolver um sistema e o impacto em custo e prazo. As prioridades irão facilitar na condução do projeto e em quais atividades serão realizadas e a que tempo. Os níveis de prioridade podem ser divididos da seguinte forma:

. Altíssima
. Alta
. Média
. Baixa
. Baixíssima

Outro fator importante é determinar o nível de risco associado a um requisito. O que determina o risco de desenvolvimento de um sistema ? O risco de um requisito é medido relativo a:

- Qual o risco do requisito não ser entregue no prazo pedido pelo cliente
- Qual o conhecimento técnico para implementação
- Qual a dificuldade de obtenção de informações técnicas.

Enfim a avaliação deve refletir o conhecimento técnico da equipe e das necessidades do projeto. Os níveis de risco podem ser divididos da seguinte forma:

. Altíssimo

. Alto

. Médio

. Baixo

. Baixíssimo

A determinação do conjunto Prioridade/Risco (P/R) irá orientar o desenvolvimento, ou seja, assim que possível deve-se **baixar o nível dos riscos associados aos requisitos de maior prioridade**. Faça a análise de riscos, mesmo que se erre na avaliação, isto irá auxiliar em projetos futuros. Uma das maneiras é o desenvolvimento de protótipos que possam ou permitir que o risco seja reduzido ou negociar com o cliente uma nova forma de atender o requisito, e é importante que isto ocorra durante a fase inicial do projeto de modo que todos os riscos estejam no nível baixo ou baixíssimo antes da fase de implementação[21].

Deve-se ter em mente que o sucesso do projeto se baseia em levantar e especificar da forma mais precisa possível os requisitos e em seguida determinar uma estratégia que permita no prazo mais curto possível reduzir o nível dos riscos associados aos requisitos[31].

3.4 Gerenciamento de Requisitos

O Gerenciamento de Requisitos é uma atividade que permite monitorar as necessidades apresentadas pelo cliente e o andamento da solução destas do ponto de vista de projeto. É impossível congelar os requisitos do cliente, uma vez que a cada passo e conforme o sistema vai sendo construído vai também se tornando mais claro como o sistema deve funcionar, que características tecnológicas o sistema deve ter. Assim o cliente vai sentindo a necessidade de novos recursos a serem implementados. É importante salientar a necessidade de se documentar e controlar todos os requisitos associados ao sistema, sejam eles oriundos do cliente ou determinados pela análise do sistema. É de fundamental importância que se possa rastreá-los de modo a medir o quanto um determinado requisito (que pode surgir a qualquer momento) terá impacto no prazo final do projeto. A documentação, identificação e análise de risco associado aos requisitos servem de base para gerenciá-los a fim de minorar os problemas quando a mudança. Um erro neste ponto irá se propagar durante o projeto e gerar um série de problemas que na maioria das vezes leva ao cancelamento do projeto e/ou ao reprojeto em novas bases o que é sempre desastroso, caro e desgastante tanto para o cliente quanto para a equipe.

3.5 Exemplo de Requisitos

Para facilitar a compreensão de como devem ser especificados os requisitos de um sistema vamos exemplificar utilizando alguns necessários para desenvolvimento de um Caixa Eletrônico de Banco. A seguir são apresentados alguns requisitos para demonstrar como são avaliados e descritos.

3.5.1 ERaF0001.1 – Sacar Dinheiro

O sistema deverá permitir que usuário saque dinheiro do caixa eletrônico. Esta atividade é responsável pelo ciclo que se inicia quando o usuário solicita o saque de dinheiro e se encerra, no fluxo ótimo, quando o usuário recebe em dinheiro o valor solicitado e o documento de comprovação de saque.

Tabela 3-1- Risco do Requisito Sacar Dinheiro

Descrição do risco	Risco	Prioridade
A implementação depende da definição da forma de comunicação entre o caixa e o sistema servidor que fornecerá as informações a respeito das contas do usuário	Altíssimo	Altíssima

3.5.2 ERaF0001.2 – Movimentar valores entre contas

O sistema deverá permitir movimentar valores entre contas do mesmo usuário.

Tabela 3-3 Risco do Requisito Movimentar valores entre contas

Descrição do risco	Risco	Prioridade
A implementação depende da definição da forma de comunicação entre o caixa e o sistema servidor que fornecerá as informações a respeito das contas do usuário	Altíssimo	Altíssima

3.5.3 ERaF0001.3 – Emitir extrato com saldo e completo por período

O sistema deverá permitir emitir extrato com o saldo da conta selecionado pela usuário ou emitir extrato com dados completos de crédito e débito por período não maior que 3 meses anteriores a data atual configurado pelo usuário.

Tabela 3-5 - Risco do Requisito Emitir extrato com saldo da contra e extrato completo por período

Descrição do risco	Risco	Prioridade
A implementação depende da definição da forma de comunicação entre o caixa e o sistema servidor que fornecerá as informações a respeito das contas do usuário	Altíssimo	Altíssima

3.5.4 ERaI0001.4 – Propaganda dos produtos do Banco

O sistema deverá permitir apresentar propaganda dos produtos do banco quando o sistema de caixa eletrônico não estiver em uso.

Tabela 3-7- Risco do Requisito Propaganda dos produtos do Banco

Descrição do risco	Risco	Prioridade
A implementação é conhecida, existe pessoal disponível para o desenvolvimento e o prazo para desenvolvimento é razoável.	Baixo	Média

3.5.5 ERaF0001.5 – Instrução fácil para uso de qualquer opção do Caixa Eletrônico

O sistema deverá permitir, que a qualquer momento do uso de uma opção (serviço) do caixa eletrônico, o usuário possa ser auxiliado no uso e nas informações que estão sendo solicitadas.

Tabela 3-9- Risco do Requisito Instrução fácil para o uso de qualquer opção do Caixa Eletrônico

Descrição do risco	Risco	Prioridade
A implementação não é totalmente conhecida, existe pessoal disponível para o desenvolvimento e o prazo para desenvolvimento é razoável.	Médio	Alta

3.5.6 ERaF0001.6 – Informação ao usuário do não funcionamento do Caixa Eletrônico

O sistema deverá permitir em caso de falta de dinheiro, falta de comunicação com o sistema central ou qualquer outra falha o usuário seja informado.

Tabela 3-11 - Risco do Requisito Informação ao usuário do não funcionamento do Caixa Eletrônico

Descrição do risco	Risco	Prioridade
A implementação é totalmente conhecida, existe pessoal disponível para o desenvolvimento e o prazo para desenvolvimento é razoável.	Baixo	Alta

3.6 Avaliação

Os requisitos apresentados são alguns exemplos para o desenvolvimento de um sistema de Caixa Eletrônico, e neste caso estão colocados somente os funcionais. Os requisitos apresentados tem níveis diferentes de risco. Devemos avaliar os de mais alto risco e eliminar o risco associado o mais rápido possível, por exemplo para o requisito ERaF0001.1 – Sacar Dinheiro deve-se desenvolver protótipos ainda na fase de levantamento e especificação e negociar com o cliente qual a melhor solução encontrada para o atendê-lo antes de iniciar a fase de implementação do sistema (Note que o risco deste requisito esta associado a outros). Além dos requisitos de alto risco deve-se desenvolver protótipos de interface, ou seja, janelas de interface com o usuário para que possa ser avaliada pelo cliente como uma prévia do produto final.

4. COMO TRANSFORMAR REQUISITOS EM CASOS DE USO

Uma vez especificados os requisitos (vale salientar que não é necessário obter e especificar todos os requisitos do sistema, pode-se iniciar a determinação dos casos de uso logo com os primeiros requisitos a mão)inicia-se o processo de especificação do caso de uso [15][40][47] que foi introduzido por Ivar Jacobson[8][9] e que veremos com mais detalhes neste capítulo.

4.1 Atores

Um ator [8] é qualquer pessoa ou sistema externo (SE) que tenha interação com o sistema que esta em desenvolvimento. O nome ideal seria **papel** e não ator. Isto tem confundido alguns projetistas que acabam identificando somente as pessoas que acessam o sistema e não levam em consideração que outros SEs podem e devem ser representados como ator. Além do nome foi definido para a UML um símbolo que ajuda nesta associação com pessoas interagindo com o sistema como mostrado na figura a seguir.

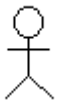


Figura 4.1 – Símbolo associado a Ator na UML

A identificação dos atores é o primeiro passo para a criação do caso de uso, um ator representa uma classe fora do sistema que se envolve de alguma forma com o mesmo. O uso do conceito classe é importante porque o ator não é um objeto, ou seja, não é uma pessoa (ou SE) em particular que irá utilizar o sistema é sim o papel que esta pessoa (ou SE)específica ou conjunto de pessoas representam para o sistema.

Um ator pode ser representado através de um diagrama de especialização (ou generalização) como utilizamos com classes. A figura a seguir representa um diagrama

de atores com relacionamento de especialização. Neste exemplo estamos identificando alguns dos possíveis atores do caixa eletrônico de banco. Um ator é o cliente do banco, porém temos neste caso três tipos diferentes de ator que derivam do ator genérico cliente de banco, são eles:

- Cliente VIP
- Cliente Especial
- Cliente Padrão

Estes clientes possuem papéis comuns, porém em função da forma de atendimento existem características que são específicas para cada tipo de cliente do banco e por conseqüência do caixa eletrônico.

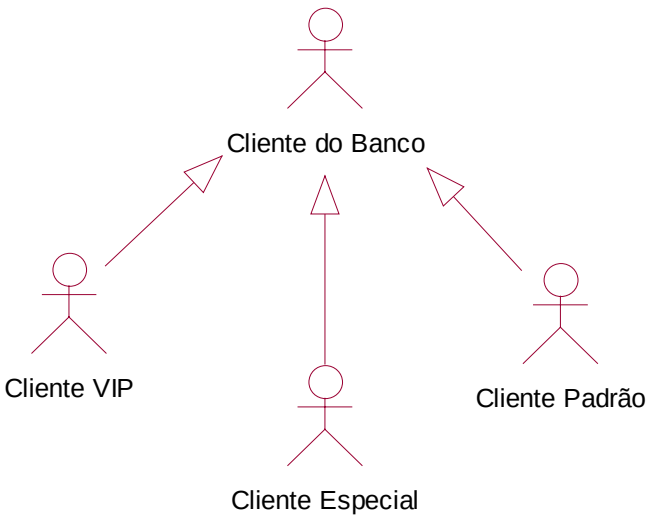


Figura 4.2 – Diagrama de Especialização (Generalização) de Atores.

O primeiro passo para identificar os atores e realizado entre o analista juntamente com o cliente (sempre que possível) que os identificam e os organizam em categorias (classes) de atores [8]. Dois critérios são importantes para identificar um ator:

1. Deve ser possível de identificar pelo menos um usuário que representa o ator candidato, isto ajuda a encontrar somente os atores relevantes e ignorar os atores fictícios.
2. Deve existir um mínimo de sobreposição de papéis entre os diferentes atores que relacionam com o sistema, não deve existir dois atores com os papéis semelhantes em relação ao sistema, se isto ocorrer devemos combiná-los em um ator ou tentar utilizar a generalização.

O analista após identificar os atores deve dar um nome e fazer uma breve descrição dos papéis de cada ator no sistema. É muito importante definir nomes que representem o maior número de papéis de um determinado ator, isto irá facilitar a identificação.

4.2 Caso de Uso

Um caso de uso [8][14] pode ser definida como:

Um conjunto de funcionalidades de um sistema, representado por fluxos de eventos iniciados por um ator e apresentando um resultado de valor a um ator.

Após identificar os atores do sistema deve-se iniciar a identificação dos casos de uso de modo a transformar os requisitos do sistema passado pelo cliente em algo que possa ser entendido pela equipe de desenvolvimento. Esta passagem representa o elo de ligação entre o processo de desenvolvimento e as necessidades do cliente. A identificação dos casos de uso baseado nos requisitos não é uma atividade trivial[10][44], uma vez que cada sistema é necessariamente diferente do outro. Porém representa um avanço do ponto de vista do processo de desenvolvimento, pois qualquer identificação por mais precária que seja é melhor do que nenhuma. O processo de desenvolvimento é incremental, ou seja, o analista terá oportunidade de passar outras vezes por este trabalho. Existe sempre a possibilidade de se corrigir e aprofundar o detalhamento dos casos de uso.
Para identificar um caso de uso devemos seguir os seguintes passos[8][49]:

1. Analisar e agrupar todos os requisitos do ponto de vista das funcionalidades(requisitos funcionais) do sistema a ser desenvolvido, ou seja, imagine um ciclo completo de uso dos sistema e determine quais os requisitos estão associados a este ciclo.
2. Uma vez agrupados determine quais os atores interagem com este ciclo.
3. Descreva o fluxo ótimo para este ciclo, ou seja, o fluxo onde nada de errado acontece e a entrada do ator levará ao resultado no final sem erros ou problemas.
4. Descreva os fluxos alternativos ou de exceção para este ciclo, ou seja, quando e onde algo pode der errado.
5. Caso o fluxo seja complexo será necessário desenvolver um diagrama de atividades para este ciclo. Sempre que possível é interessante gerar um diagrama de atividades para demonstrar o fluxo de eventos graficamente.
6. Partes de um caso de uso pode aparecer em outros casos de uso, por exemplo o caso de uso para identificar o *login* do usuário analisando nome e senha, desta forma é interessante dividir o caso de uso em partes para não repetir a descrição. Existem três tipos de relacionamento entre casos de uso, que são:
    1. Herança : É equivalente a herança entre classes ou atores, os casos de uso podem herdar o comportamento do caso de uso da qual deriva.
    2. Inclusão : O caso de uso que é incluída é sempre executada, ou seja, o fluxo de eventos deste caso de uso é sempre acessado, porém pode ser representada por um caso de uso independente.
    3. Extensão : O caso de uso estende o comportamento do caso de uso, ou seja, o fluxo de eventos do caso de uso pode ou não ser acessado.

O desenvolvimento dos seis itens apresentados representa a criação de um caso de uso. É importante ao final da criação de cada caso de uso reunir-se com o cliente no sentido de discutir se o que foi analisado esta de acordo com as necessidades. Os casos de uso são

uma tradução das necessidades do cliente, peça que o cliente analise o que foi criado. Os casos de uso e os protótipos, como as janelas de interface com o usuário representam a primeira visão do cliente do produto final e tem importância fundamental no desenvolvimento do projeto. Não se deve tentar detalhar todos os casos de uso ao mesmo tempo. Da mesma forma que foi feito para os requisitos deve-se determinar qual(is) casos de uso(s) é(são) fundamental(is) para o início do desenvolvimento do sistema e detalhá-la ao máximo na primeira fase. A determinação de quais casos de uso serão trabalhadas é de responsabilidade do arquiteto do sistema que deve possuir uma visão completa do sistema que a ser desenvolvido, o arquiteto irá basear-se no levantamento de prioridades e riscos dos requisitos que estão associados a um caso de uso.

Considerando os atores do caixa eletrônico um caso de uso é sacar dinheiro, portanto podemos descrevê-la da seguinte forma:

- **Caso de Uso Sacar Dinheiro**

Breve Descrição: Esta caso de uso é responsável pelo ciclo que se inicia quando o usuário solicita ao Caixa Eletrônico o saque de dinheiro e se encerra, no fluxo ótimo, quando o usuário recebe o dinheiro e o documento de saque.

Pré condição:

O usuário solicitar o saque de dinheiro.

Início do caso de uso

Este caso de uso inicia quando o usuário solicita o saque de dinheiro do caixa eletrônico.

Tabela 4-1 – Fluxo Ótimo

Ações Recebidas	Ações Realizadas
1. O usuário solicita o saque de dinheiro	2. O terminal pede que ele passe o cartão
3. O usuário passa o cartão solicitado pelo terminal	4. O terminal lê os dados do cartão 5. Verifica se o cartão é válido 6. Solicita a senha
7. A senha é digitada pelo usuário	8. Avalia a senha

	9. Solicita que o usuário entre com a quantia a ser sacada
10. A quantia é digitada pelo usuário	11. É verificada a quantia de recursos na conta do usuário
	12. É feita a impressão do recibo de saque 13. É liberado o valor solicitado 14. O sistema volta ao estado inicial para a execução de outros serviços para o mesmo ou para outros usuários.

Fluxos Alternativo:

Tabela 4-2 - Cartão inválido ou passado de forma incorreta pelo leitor

Ações Recebidas	Ações Realizadas
1. O usuário passa o cartão solicitado pelo terminal	2. O terminal lê os dados do cartão 3. Verifica se o cartão é válido 4. Determina que o cartão é invalido 5. Pede ao usuário que passe o cartão novamente e mostra isto graficamente 6. Volta ao ponto 3 do fluxo ótimo

Tabela 4-3 – Senha Incorreta

Ações Recebidas	Ações Realizadas
1. A senha é digitada pelo usuário	2. Avalia a senha 3. Verifica que a senha não é válida e solicita que o usuário

	digite novamente a senha 4. Volta ao ponto 7 do fluxo ótimo e após a terceira tentativa o sistema informa que não poderá mais executar qualquer operação com este cartão por um prazo de 24 horas

Tabela 4-4 – Saldo Insuficiente

Ações Recebidas	Ações Realizadas
1. A quantia é digitada pelo usuário	2. É verificada a quantia de recursos na conta do usuário 3. Não há recursos suficientes para a finalização do saque 4. É solicitado um novo valor de acordo com o saldo do usuário 5. Volta ao ponto 10 do fluxo ótimo

A figura 4.3 a seguir apresenta o mesmo fluxo de eventos, porém na forma de um diagrama de atividades, onde todos os fluxos são mostrados de uma forma única. O diagrama de atividades permite apresentar as atividades executadas e as mudanças de fluxo, ou questões.

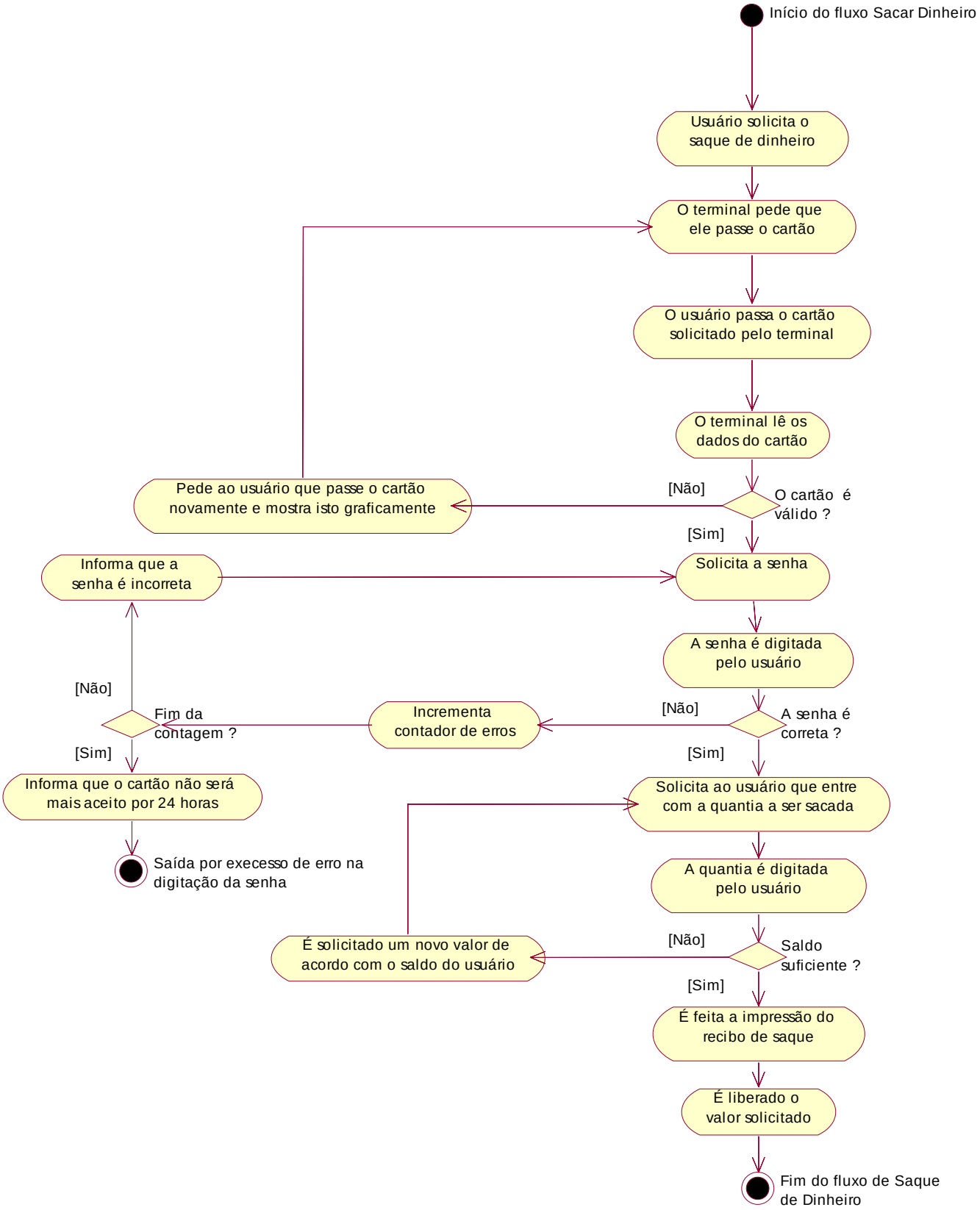


Figura 4.3 – Exemplo do uso de Diagrama de Atividades para Caso de Uso

4.3 Diagrama de Caso de Uso

A fim de complementar o caso de uso a UML definiu um diagrama que a apresenta como uma elipse com o nome e o relacionamento com os atores do sistema. O diagrama de caso de uso não substitui a descrição dos fluxos de eventos e do próprio caso de uso, porém permite determinar o relacionamento entre atores e casos de uso e ainda permite dividi-la.

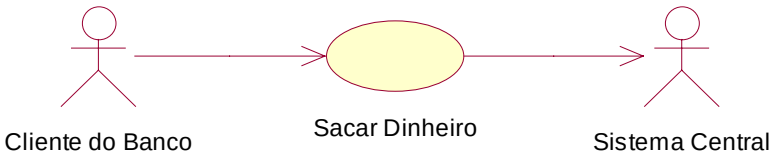


Figura 4.4 – Exemplo de Diagrama de Casos de Uso

Observações:

Para o caso de uso “Sacar Dinheiro” existem mais dois atores:

- Impressora, que é responsável por imprimir o recibo de retirada de dinheiro
- Equipamento de Liberação de Dinheiro, que é responsável, como o próprio nome diz, pela liberação das notas correspondente ao valor solicitado pelo usuário.

Estes atores não serão modelados para facilitar didaticamente a compreensão do modelo.

A criação do caso de uso de derivação ou inclusão[8] permite fragmentar um caso de uso e deve ser feito em duas situações:

- O fluxo de eventos é utilizado por outros casos de uso ou
- O fluxo de eventos é muito complexo e irá dificultar o detalhamento em um único caso de uso.

Como já foi dito anteriormente, além da derivação os casos de uso podem ser divididas por inclusão e extensão que serão apresentados a seguir.

4.3.1 Inclusão

A inclusão representa um fluxo de eventos que se repete em outros casos de uso, desta forma é possível criar o caso de uso uma única vez e incluí-la nas outras que tenham o mesmo fluxo de eventos. Na tabela de fluxo de eventos quando for necessário citar o fluxo deve-se colocar o texto: Incluir o caso de uso [Nome do caso de uso]. Um exemplo é o fluxo de eventos que avalia a senha do usuário do caixa eletrônico, este caso de uso será utilizada por outros casos de uso do sistema e sempre será necessário executá-la.

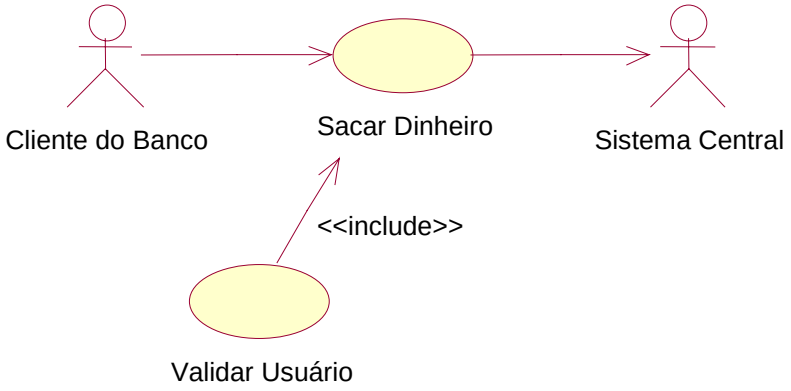


Figura 4.5 – Caso de Uso Incluído

Desta forma o fluxo de eventos de "Verificar Senha" será independente do caso de uso "Sacar Dinheiro" porém incluído nesta e se chama "Validar Usuário".

- Caso de Uso Validar Usuário

Ações Recebidas	Ações Realizadas
1. O usuário inicia um serviço do sistema	2. A senha é solicitada
3. A senha é digitada pelo usuário	4. Avalia a senha, se válida encerra com resposta afirmativa 5. Verifica que a senha não é válida e solicita que o usuário digite novamente a senha 6. Volta ao ponto 2 deste fluxo e após a terceira tentativa o sistema informa que não poderá

	mais executar qualquer operação com este cartão por um prazo de 24 horas

4.3.2 Extensão

Quando um fluxo de eventos faz parte de um caso de uso mas nem sempre este fluxo é executado pode-se retirá-lo do caso de uso e colocá-lo como uma extensão do caso de uso. Como para inclusão, deve-se quebrar o caso de uso caso o fluxo de eventos se repita em outro(s) caso(s). Um exemplo seria o caso de uso Saldo Insuficiente que não é executada para alguns clientes do banco(se for cliente VIP não é necessário consultar a conta).

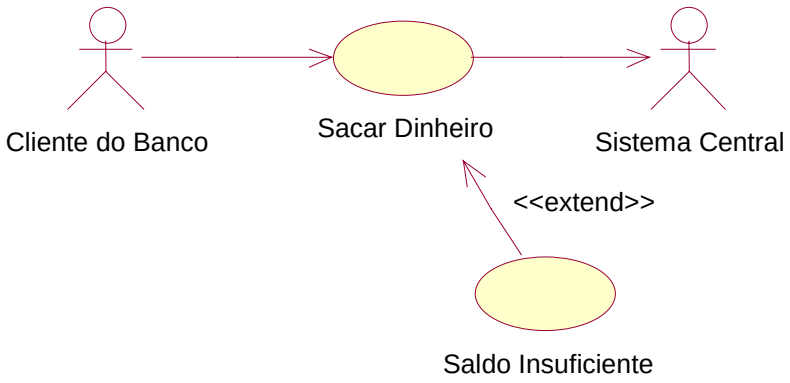


Figura 4.6 – Caso de Uso Estendida

- Caso de Uso Saldo Insuficiente

Ações Recebidas	Ações Realizadas
1. Não há recursos suficientes para a finalização do saque	2. É solicitado ao usuário um valor inferior ao digitado de acordo com o saldo do usuário
3. A nova quantia é digitada pelo usuário	4. É verificada a quantia de recursos na conta do usuário 5. Se valor correto retorna com resposta positiva, senão 6. Volta ao passo 2 do fluxo

4.3.3 Derivação

A derivação entre caso de uso é semelhante a derivação em classes, ou seja, o caso de uso filho herda as propriedades do caso de uso pai. Para exemplificar imagine que tenhamos dois tipos de validação do usuário, uma seria através da senha e outro através de identificação das impressões digitais do usuário, neste caso temos o caso de uso "Validar Usuário" dividida em duas como mostrado na figura a seguir.

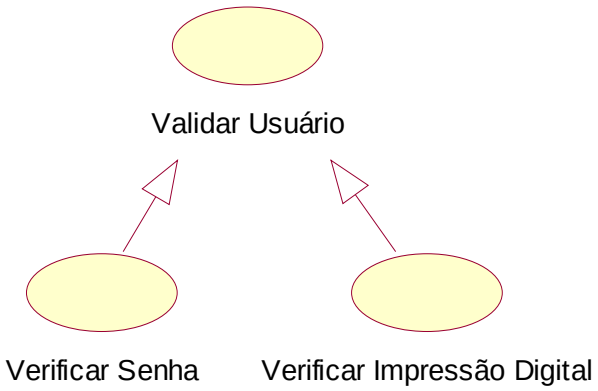


Figura 4.7 – Derivação entre Casos de Uso

4.4 Avaliação

A análise dos requisitos e transformação em caso de uso é um passo muito importante[7][55] e deve ser feito com muita atenção. Por isto deve ser discutido com o cliente e revisitado sempre que alguma dúvida surgir[10][28]. Lembre-se que o fluxo de eventos é que representa, na forma de tabela, um conjunto de requisitos do cliente, peça que seja avaliado com o máximo critério possível. A subdivisão do caso de uso depende do nível de complexidade e deve ser avaliado se deve ou não ser feito, a figura a seguir mostra o diagrama completo da caso de uso "Sacar Dinheiro".

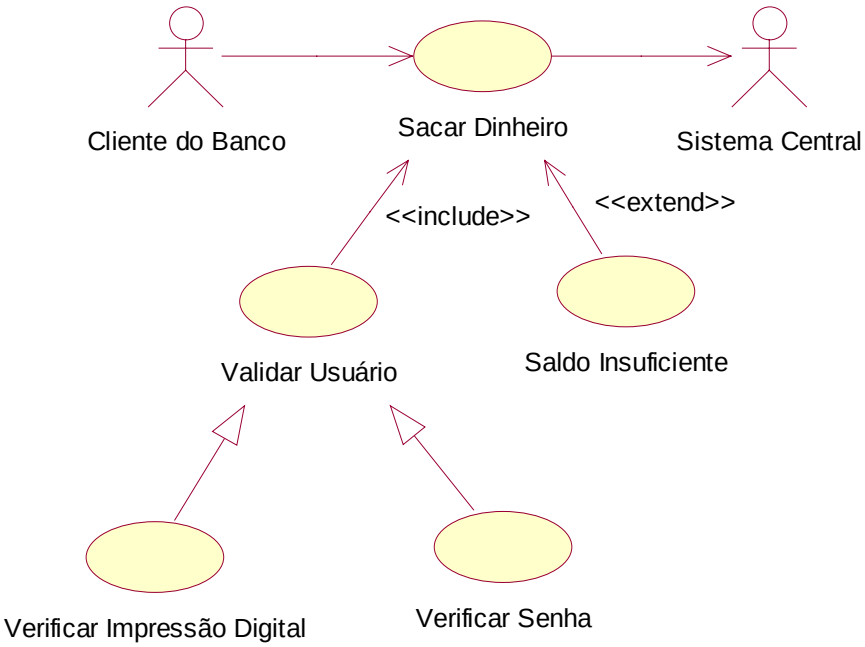


Figura 4.8 – Exemplo de fragmentação do Caso de Uso

5. CLASSES DE ANÁLISE *(ROBUSTNESS MODELING)*

5.1 Introdução

Após a definição dos casos de uso que atendem os requisitos do cliente o passo seguinte é transformar o caso de uso em classes de análise[43]. A transformação consiste em determinar classes de fronteira, controle e entidade que são mostrados em detalhes neste capítulo. Classes de Análise representam uma abstração de uma ou mais classes de projeto e/ou subsistemas[9].As classes de análise possuem as seguintes características preliminares:

- As classes de análise representam o comportamento relacionado a requisitos funcionais (caso de uso) no início da análise e vão adquirindo novas funcionalidades com a seqüência do projeto.
- As classes de análise raramente possuem operações ou argumentos (que são característicos das classes de projeto). O comportamento da classe é definido sem se preocupar com os detalhes, ou seja, a responsabilidade e o comportamento da classe deve ser descrita na forma de texto.
- Um comportamento muito importante das classes de análise é o relacionamento com as outras classes. Estes relacionamentos irão gerar no futuro as operações que definem o comportamento da classe.
- As classes de análise podem definir atributos, porém estes atributos são sempre conceituais, ou seja, não é feito o detalhamento.

As características das classes de análise são mutantes, ou seja, conforme se evolui na análise a caminho do projeto serão agregadas mais informações até que em um determinado ponto a classe de análise se torna uma classe de projeto[42].

Com as classes de análise inicia-se o processo de distribuição do comportamento do caso de uso em classes. Na primeira avaliação do caso de uso provavelmente irá se chegar a um número de classes de análise pequeno que representam de forma macro o

comportamento do caso de uso. Porém com o aprofundamento da análise novas classes surgirão e outras serão fundidas de modo que no final do processo as classes de análise estarão muito próximas das classes de projeto.

Existem três tipos de classes de análise: Classes de Fronteira, Classes de Controle e Classes de Entidade.

5.2 Classes de Fronteira

As classes de fronteira são responsáveis pela comunicação entre o caso de uso e o Ator[37], sendo assim existe uma regra básica que determina que cada relação entre Ator e o caso de uso é responsável por uma classe de fronteira e este relacionamento envolve receber ou apresentar informações do sistema ao ator.

Os comportamentos destas classes estão intimamente ligados com o ator que mantém relacionamento com o caso de uso. É através da classe de fronteira que o ator terá os seus requisitos atendidos. Uma interface com o usuário ou na interface de comunicação deve ser isolada em uma ou mais classes de fronteira.

Classes de fronteira muitas vezes estão associadas a janelas, *forms*, interfaces de comunicação, interface de impressora, sensores, terminais e APIs(*Aplication Program Interfaces*).

A figura 5.1 a seguir mostra como é o símbolo para classes de fronteira.

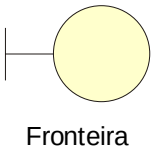


Figura 5.1 - Símbolo para Classe de Fronteira

Para o exemplo de caso de uso “Sacar Dinheiro” identificamos na primeira análise que existem duas classes de fronteira que são:

- Janela de Interface com o usuário, que irá ser responsável por obter e enviar informações para o usuário
- Interface de Comunicação com o sistema central, que será responsável por buscar informações do cliente no sistema central onde se encontram os dados dos correntistas

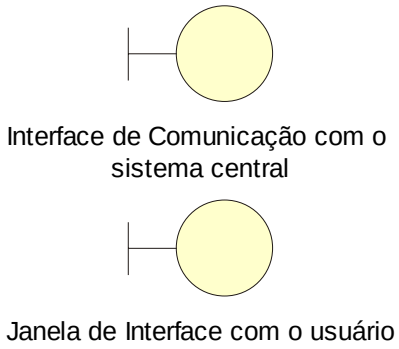


Figura 5.2 - Exemplos Classes de Fronteira

Observações:

Existem mais duas classes de fronteira, uma que interage com a impressora que imprime o recibo de liberação de dinheiro e outra responsável por enviar as informações para o equipamento que libera o dinheiro, mas para facilitar a compreensão e simplicidade do modelo, estas duas classes não serão modeladas.

5.3 Classes de Entidade

As classes de entidade são utilizadas para modelar informações que são duradouras e persistem durante a execução do sistema. São informações e comportamentos associados a conceitos como objetos ou eventos da vida real. Um objeto de entidade (instancia de uma classe de entidade) não é necessariamente passivo; pode e deve possuir um comportamento mais complexo relacionado as informações que representam e isolam as mudanças no sistema relativo a estas informações. Não existe uma regra básica para determinar as classes de entidade, deve-se analisar as informações relativas ao caso de

uso, normalmente em uma primeira análise são determinadas de 1 a 3 classes de entidade por caso de uso.
As classes de entidade são representadas como mostrado na figura 5.3 a seguir.

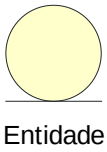


Figura 5.3 - Símbolo para Classe de Entidade

Avaliando o caso de uso “Sacar Dinheiro” identificamos as seguintes classes de entidade:

- Cliente, que é responsável por armazenar as informações do usuário durante o processo de saque de dinheiro no caixa eletrônico
- Cartão/Conta, que é responsável por armazenar as informações do cartão e conta do usuário durante o processo de saque de dinheiro no caixa eletrônico

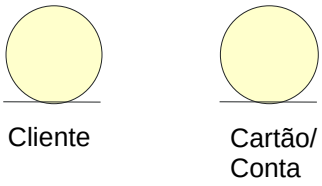


Figura 5.4 - Exemplos Classes de Entidade

5.4 Classes de Controle

As classes de controle representam a coordenação, sequenciamento, transações e controle entre os objetos do caso de uso, isto é, todo o controle do caso de uso será encapsulado nas classes de controle. As classes de controle são também utilizadas para representar ações complexas como lógica do negócio que represente o comportamento do caso de uso e que não representam uma informação duradoura como no caso das informações das classes de entidade.

A parte dinâmica do sistema será modelada em classes de controle uma vez que elas manipulam e coordenam as principais ações e controle de fluxos e distribuem o trabalho a ser executado pelas classes de entidade ou de fronteira.

A regra para a primeira análise é uma classe de controle para cada caso de uso, normalmente a classe de controle recebe mensagens das classes de fronteira e distribuem as tarefas e informações para outras classes de entidade ou de fronteira e normalmente retornam informações para a classe de fronteira que enviou a primeira mensagem. Com a evolução da análise outras classes de controle poderão surgir.

As classes de controle são representadas como mostrado na figura 5.5 a seguir.

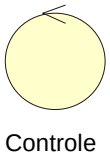


Figura 5.5 - Símbolo para Classe de Controle

Para o caso de uso “Sacar Dinheiro” e seguindo a regra identificamos a classe controlar informações cuja imagem é mostrada a seguir.

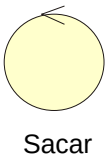


Figura 5.6 - Exemplo de Classes de Controle

Um sistema pode não ter classes de controle. Neste caso o controle pode ser uma operação de controle que esteja em uma classe de entidade ou de fronteira.

5.5 Avaliação

É chamado de realização do caso de uso a determinação das classes de análise relacionados a este caso de uso e o relacionamento entre estas classes. Este passo permite

passar a fase seguinte determinando os comportamentos do relacionamento entre as classes.

A figura 5.7 a seguir mostra a realização do caso de uso "Sacar Dinheiro".

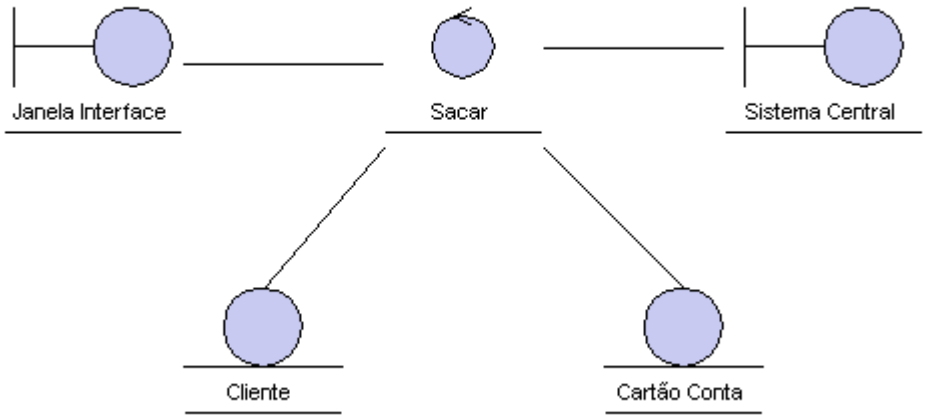


Figura 5.7 - Exemplo de Realização de Caso de Uso ou Classes de Análise

6. DIAGRAMAS DE INTERAÇÃO

6.1 As relações entre as Classes de Análise

Uma vez determinadas as classes de análise devemos, tendo em mãos os fluxos de evento dos casos de uso (e/ou um diagrama de atividades), trabalhar no relacionamento entre estas classes ou melhor entre os objetos de análise uma vez que o relacionamento existe entre objetos. Na definição de classes e objetos vimos que a classe vaca tem a propriedade de "produzir leite" porém quem produz mesmo é a mimosa ou malhada por exemplo. Vaca é uma classe e portanto é abstrata, produto do raciocínio humano, não existe no mundo real e portanto não pode produzir leite. Neste momento estamos interessados nas mensagens que são enviadas de um objeto para outro e nos serviços realizados que estão descritos no fluxo de eventos dos casos de uso. Estas mensagens e serviços estão, de alguma forma, descritas no fluxo de eventos do caso de uso. Esta fase é, inclusive, importante para validar o fluxo de eventos e verificar se nenhum dos requisitos deixou de ser atendido pelo caso de uso. A UML define dois diagramas de Interação[8]:

- Diagrama de Seqüência e o
- Diagrama de Colaboração

Estes dois diagramas são duas formas diferentes de apresentar os relacionamentos entre os objetos de análise, será utilizado o Diagrama de Seqüência uma vez que este diagrama esta mais ligado ao fluxo de eventos que foi desenvolvido para o caso de uso[42][50], desta forma aumenta-se a consistência na evolução do processo de desenvolvimento.

6.2 Diagrama de Seqüência

O Diagrama de Seqüência, como o nome diz, apresenta uma seqüência de eventos que determinam o comportamento dinâmico do caso de uso e que estão são descritos no fluxo de eventos. Neste diagrama, para esta fase, são apresentados os atores e as classes de

análise na parte superior do diagrama e as linhas que se seguem abaixo de cada um deles representa a linha de vida do objeto ou da ação do ator sobre o objeto. A figura 6.1 a seguir mostra um exemplo onde um Ator 1 inicia o processo enviando a mensagem para uma classe de fronteira, esta passa a mensagem para a classe de controle; estas mensagens e serviços devem ser levantadas a partir do fluxo de eventos. Um exemplo de instanciação (construção) do objeto é mostrado no relacionamento entre o objeto controlar e o objeto entidade, que “constrói” e “destrói” o objeto onde é colocado um sinal “X”. A figura a seguir mostra o diagrama de seqüência com este exemplo de instanciação. Quando o objeto envia uma mensagem a si próprio, isto é representado por uma linha de ida e volta como no caso do exemplo da mensagem chamada “Auto Mensagem” no objeto controlar. O diagrama de seqüência deve ser depurado até o momento onde todos os comportamentos necessários e conhecidos e representados no diagrama e é nesse momento que iniciamos o processo de desenvolvimento das classes de projeto.

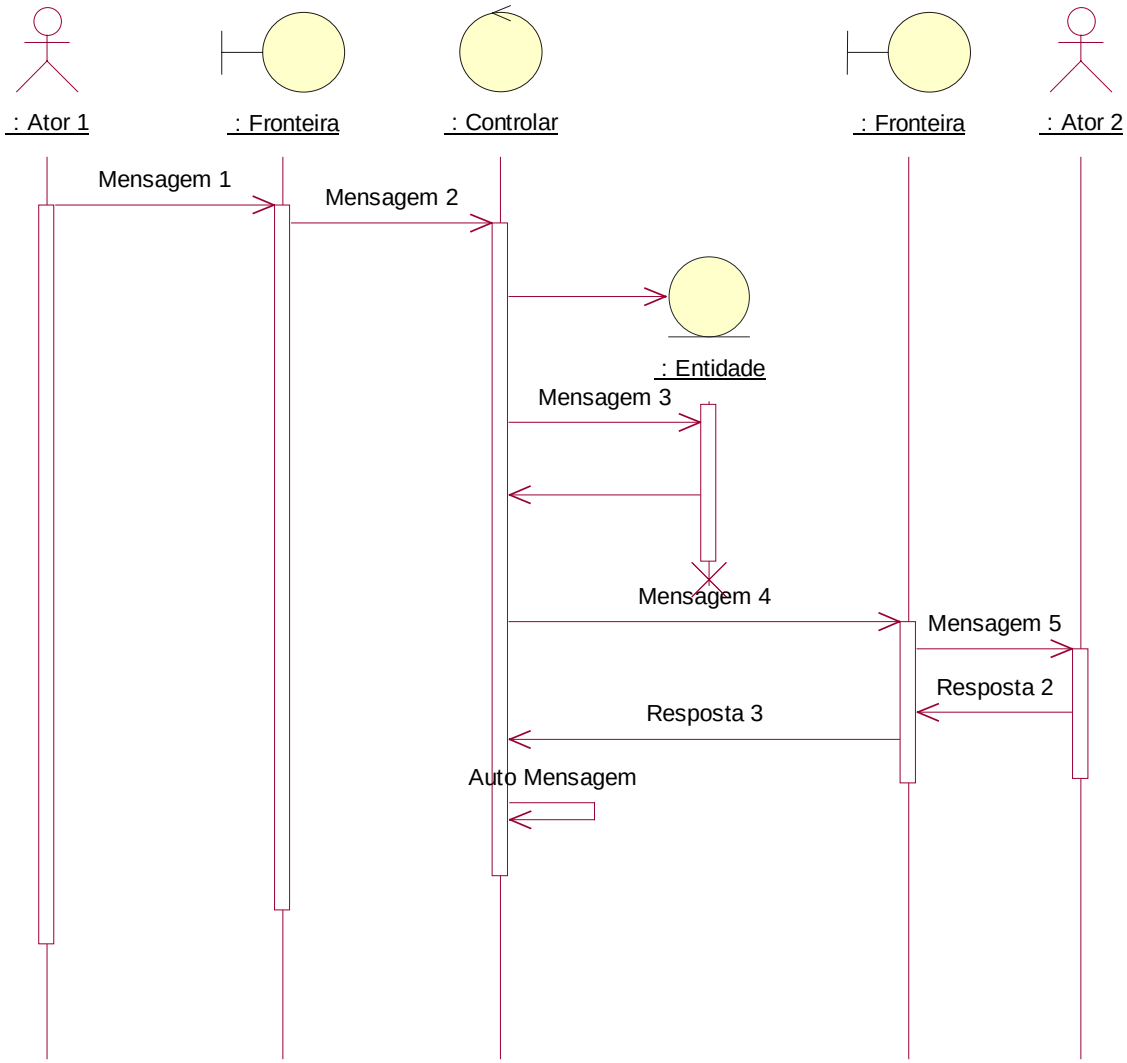


Figura 6.1 - Diagrama de Seqüência

6.3 Diagrama de Colaboração

O diagrama de colaboração é muito semelhante ao diagrama de seqüência, porém mostra o relacionamento das classes no que diz respeito ao sentido das mensagens e o número de mensagens dando ênfase à organização dos objetos que participam de uma interação[8]. A figura a seguir mostra um exemplo de diagrama de colaboração baseado no mesmo exemplo do diagrama de seqüência da figura anterior.

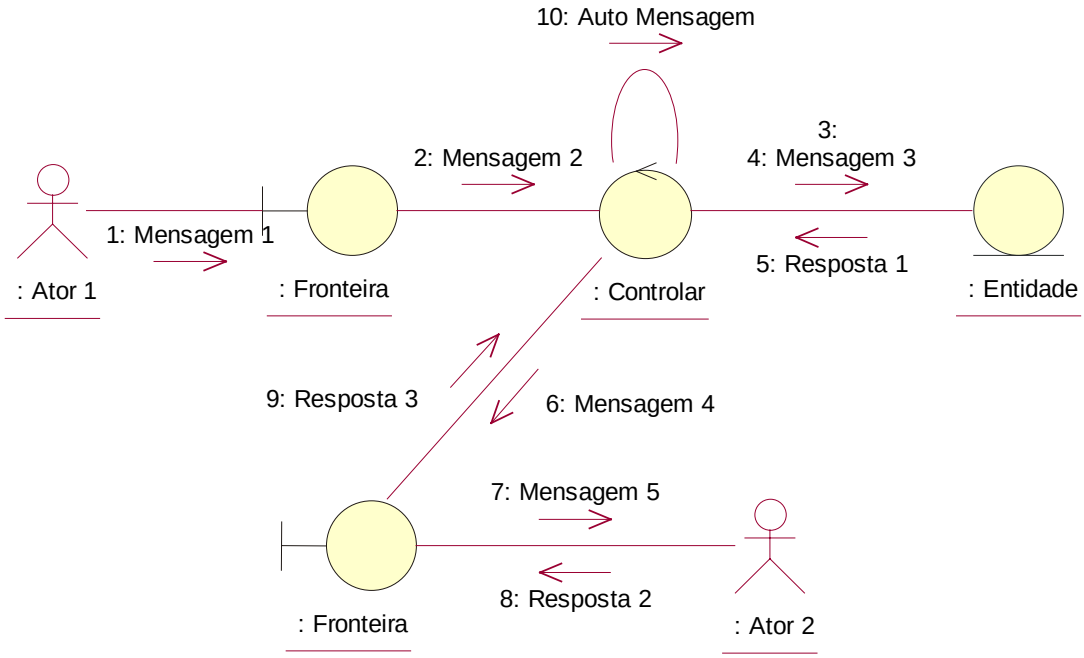


Figura 6.2 - Diagrama de Colaboração

6.4 Exemplo de Diagramas de Seqüência

Considerando os atores, as classes de análise e o fluxo de eventos determinados para o caso de uso “Sacar Dinheiro” visto anteriormente foi desenvolvido o diagrama de seqüência apresentado na 6.3 seguir para o fluxo ótimo. Neste diagrama os objetos foram representados pelo nome das classes sem utilizar os símbolos das classes de análise.

Estão preparados para iniciar o processo de determinação das primeiras operações das classes de projeto. As mensagens do diagrama serão as primeiras mensagens que cada objeto (e por conseqüência a classe) terá para atender para cumprir os requisitos do sistema.

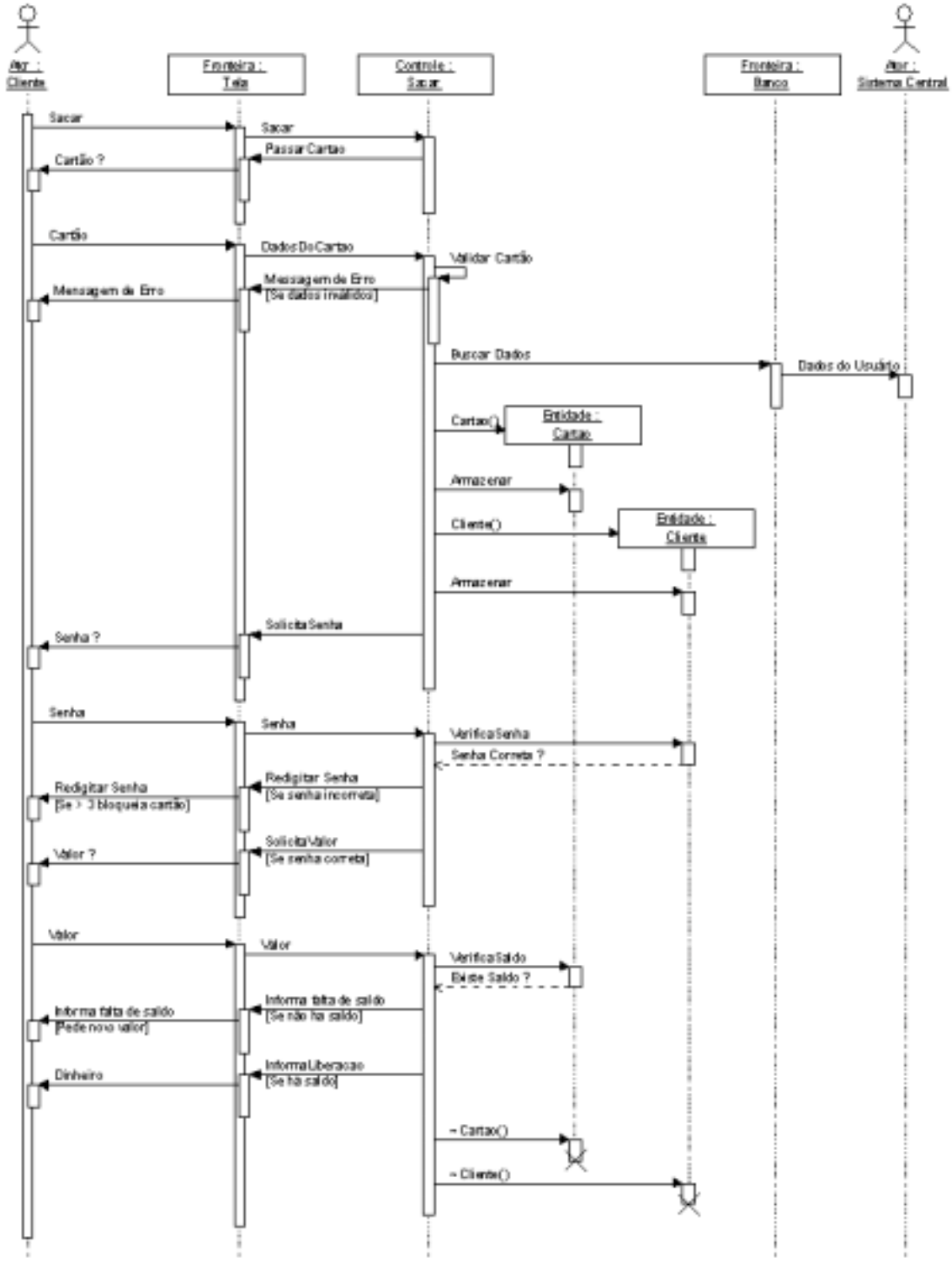


Figura 6.3 - Exemplo do uso do Diagrama de Seqüência

7. SUBSISTEMAS

7.1 Desenvolvimento Paralelo

Com a criação do diagrama de seqüência é possível iniciar a modelagem de classes de projeto. Porém nem sempre é possível implementar todo o projeto simultaneamente, é necessário avaliar o diagrama no sentido de dividi-lo em partes que podem ser implementados por grupos ou individualmente. Para isto o sistema deve ser dividido em pacotes ou subsistemas[8][14]. Esta divisão não possui uma regra definitiva, é importante avaliar os recursos humanos e técnicos (como subsistemas já existentes) disponíveis para dar início a implementação do projeto[20].

O desenvolvimento paralelo permite realizar um conjunto de tarefas simultâneas, sem a perda de controle do trabalho. A utilização de subsistemas força a definição de fronteiras entre os implementações. Isto irá garantir a não interferência entre os projetos, ou seja, define-se um contrato de comunicação entre pacotes ou subsistemas que garanta um desenvolvimento adequado. Caso seja necessário mudar a interface negocia-se novamente, porém mantendo sincronismo entre as diferentes partes do sistema.

7.2 Subsistemas

O subsistema é um tipo especial de pacote. A grande diferença é que um subsistema possui um comportamento definido, ou seja, tem responsabilidades claras e uma forma conhecida de ser acessado, a esta forma dá-se o nome de interface. Do ponto de vista conceitual um subsistema esta a meio caminho entre um pacote e uma classe. A figura 7.1 a seguir mostra a representação para subsistemas na UML.

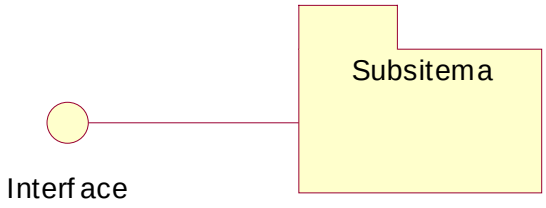


Figura 7.1 - Símbolo de Subsistema para UML

A identificação de subsistemas é um passo importante no projeto de software. Permite a criação de uma biblioteca de classes que compõem um subsistema e que irá formar um *framework* de desenvolvimento que, uma vez bem modelado e documentado, pode ser utilizado em outros projetos. Pode-se implementá-los utilizando-se padrões de projeto[20].

Quando se decide desenvolver um subsistema esta se buscando diminuir os acoplamentos entre as partes do projeto e consequentemente aumentar a portabilidade entre as partes. O isolamento de funcionalidades em subsistemas (que também ocorre na definição dos pacotes mas de uma forma menos definida) facilita as mudanças no sistema. Em particular nas mudanças de requisitos durante o desenvolvimento. Isto permite uma evolução independente entre as partes do projeto.

A identificação de subsistemas é feita a partir da análise dos diagramas de seqüência. No início não se deve preocupar com detalhes internos do subsistema, concentrando as atenções na interface (o contrato entre os subsistemas e sistema). Deve-se focar no encapsulamento do comportamento e responsabilidades do subsistema e abstrair-se do desenvolvimento das classes.

A seguir são apresentados o passos para desenvolver um subsistema:

- Distribuir o comportamento nos elementos (classes) pertencentes ao subsistema.
- Documentar os elementos.
- Descrever os relacionamentos (contratos de interface) entre os elementos.

É importante definir uma lista de avaliação para verificar se todos os comportamentos necessários foram atendidos.

As responsabilidades de um subsistema são representadas pelas operações de interface que devem ser claras e bem definidas para facilitar a reutilização. É interessante que esta documentação seja focada nas responsabilidades e na interface, detalhando e exemplificando. É desejável desenvolver programas-exemplo que facilitem a compreensão do uso prático do subsistema.

Os subsistemas deverão ter ao final do desenvolvimento os diagramas de caso de uso e classes de projeto com relacionamento entre as classes.

Um exemplo de subsistema para o projeto de caixa eletrônico, mostrado na figura 7.2, e para o caso de uso Sacar Dinheiro é o que faz a interface com o sistema central do banco para buscar informações do cliente do banco.

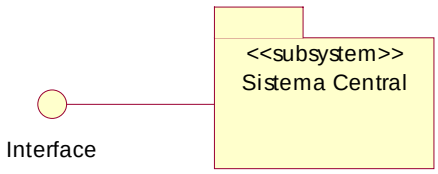


Figura 7.2 - Exemplo de Subsistema

Isto isolaria o acesso ao sistema central do banco o que provavelmente será utilizado por outros casos de uso.

8. CLASSES DE PROJETO

8.1 Classes de Projeto

Uma vez trabalhado o diagrama de seqüência, transformando as mensagens em métodos, no sentido de levantar todas as possíveis classes e subsistemas que irão compor o caso de uso inicia-se a montagem das classes de projeto[52]. Elas representam o último passo antes da geração de código. Vale lembrar que o aprofundamento do trabalho no diagrama de seqüência vai depender da complexidade e do conhecimento técnico. As classes de projeto foram introduzidas no capítulo Orientação a Objetos, porém neste capítulo iremos analisar as classes do ponto de vista de projeto, ou seja, como desenvolve-las a partir dos caos de uso, classes de análise e do diagrama de seqüência.

Para identificarmos as classes de projeto deve-se:

- Codificar as classes de projeto inicias que são retiradas diretamente do diagrama de seqüência
- Definir as operações de relacionamento que aparecem nos diagramas de seqüência
- Definir atributos e outras operações
- Definir dependências, associações e generalizações
- Verificar a consistência da classe quanto aos requisitos não funcionais
- Criar, se necessário, novas classes de projeto

Isto será apresentado ao longo deste capítulo. A quantidade de classes que devem existir em um projeto depende da complexidade do projeto e em especial do caso de uso que se esta desenvolvendo, podemos dizer que:

- Muitas classes simples significam que:
    - Encapsulam um pouco de toda a inteligência do sistema
    - São fortes candidatas a serem reutilizadas
    - São mais fáceis de serem implementadas

- Poucas classes complexas significam que:
  - Encapsulam boa parte da inteligência do sistema
  - São mais difíceis de serem reutilizadas
  - São mais difíceis de serem implementadas

Uma classe deve possuir um propósito bem definido e deve ser responsável por fazer uma coisa e fazê-la bem.

Sabe-se que o aprofundamento do estudo do diagrama de seqüência irá gerar em última análise um conjunto de classes dos tipos fronteira, controle ou entidade. É importante estabelecer estratégias que facilitem a implementação destas classes. As estratégias para esta implementação estão apresentadas a seguir.

8.1.1 Técnicas para implementar classes de fronteira

Quando inicia-se a implementação de classes de fronteira deve-se primeiro identificá-las nos dois tipos principais, que são:

- Classes de Fronteira que executam interface com o usuário. Neste caso devemos analisar se existe alguma ferramenta que irá gerar automaticamente esta interface e o quanto desta interface será gerado pela ferramenta. A escolha certa irá implicar na facilidade ou não do desenvolvimento destas classes o que pode, em alguns casos, implicar em atrasos na entrega do sistema. O fato destas classes estarem diretamente ligadas ao usuário faz com que aos olhos do cliente represente o próprio sistema. É importante o desenvolvimento de protótipos(o quanto antes) com estas classes (interfaces) de modo a deixar a interface o mais próximo possível da necessidade do cliente.
- Classes de Fronteira que fazem interface com sistemas externos. Neste caso normalmente existe um protocolo, aberto ou proprietário, de comunicação entre os sistemas. É muito importante isolar este comportamento do resto do sistema de modo a não influenciar o desenvolvimento em caso de mudança por qualquer razão do protocolo. Estas classes são muito importantes e na maioria das vezes implicam em um risco alto (as vezes altíssimo) em função do não conhecimento do protocolo ou

uma documentação insuficiente do sistema. Por isto na maioria dos casos deve-se ter muita atenção com este tipo de classe de fronteira.

8.1.2 Técnicas para implementar classes de entidade

As classes de entidade representam o conhecimento e/ou dados do sistema e portanto devem conter os métodos e atributos que representem este comportamento as informações definidas em casos de uso. Estas classes executam a maioria dos comportamentos do caso de uso. É importante ressaltar que as classes de entidade são responsáveis pelos dados e por qualquer outra informação relativa a estes dados. Estes dados não estão, necessariamente, relacionados a banco de dados que representam uma forma de dar persistência aos dados. Em alguns casos os requisitos de performance podem influenciar na reavaliação destas classes, em particular os métodos que estão diretamente ligados a estes requisitos. Em alguns casos de uso talvez seja mais prudente que os comportamentos das classes de entidade sejam incorporados as classes de controle.

8.1.3 Técnicas para implementar classes de controle

Antes da implementação de uma classe de controle deve-se primeiro avaliar se a(s) classe(s) de controle é(são) realmente necessária(s), uma vez que a distribuição do controle pode não ser complexo o suficiente para determinar a existência de uma classe de controle específica. Em outros casos a distribuição do controle pode ser muito complexa e seria necessário dividi-la em duas ou mais classes. Como foi colocado no tópico anterior talvez as responsabilidades de algumas (ou todas) classes de entidade podem ser transferidas às classes de controle.

A decisão do que fazer esta relacionado com a complexidade, probabilidade de mudanças, performance, distribuição do comportamento ou gerenciamento da transação.

8.2 Operações

Muitas operações virão diretamente do diagrama de seqüência que quanto mais trabalhado mais irá fornecer informações que determinarão as operações. Não é necessário obter todas as operações diretamente do diagrama de seqüência uma vez que outras operações necessariamente irão surgir da modelagem do diagrama de classes de projeto. E esta modelagem das classes de projeto é importante porque outras operações serão descobertas através do estudo do comportamento da classe, como:

- Operações de gerenciamento
- Cálculos necessários com os atributos da classe
- Operações de teste
- Operações internas que facilitem a compreensão e a distribuição das funcionalidades

Uma característica muito importante de uma operação é o nome apropriado, indicando a finalidade, deve levar em conta a perspectiva do cliente da classe, ser consistente e relativo a responsabilidade da operação. Deve-se também definir claramente a assinatura da operação, por assinatura da operação entende-se os tipos dos parâmetros (inteiro, *string*, etc) e os próprios parâmetros que são passados a operação, o retorno da operação e o nome da operação. Deve-se definir se os parâmetros são opcionais ou não, se forem deverão possuir um valor *default* (que caso não se envie nenhum valor será utilizado este valor *default*, por exemplo int iValue=10, onde 10 é este valor). Deve-se definir se os parâmetros serão passados por valor, por referência ou por ponteiro:

- Por valor: o valor é o próprio parâmetro, deve ser utilizando quando se deseja simplesmente utilizá-lo.
- Por referência: o valor passado é o próprio parâmetro que poderá ser alterado dentro da operação, este uso tem vantagens para parâmetros (ou objetos) pequenos, caso contrário é interessante a passagem de ponteiro.

- Por ponteiro: o valor passado é o endereço de onde esta armazenado o parâmetro, deve ser utilizado quando se necessita alterar o valor do parâmetro dentro da operação.

No caso de muitos parâmetros a serem passados para a operação deve-se utilizar um objeto com todos os dados e operações de manipulação destes dados que os representem e que permitam uma assinatura mais clara e limpa.

As operações possuem visibilidade que permitem reforçar o encapsulamento na classe e podem ser de três tipos:

+ pública, são operações que podem ser acessadas por operações de outras classes e que representam a interface da classe com as classes cliente(+ é o símbolo da UML para operações públicas)

 # protegida, são operações que podem ser acessados pelas operações da própria classe e pelas classes derivadas (especialização) (# é o símbolo da UML para operações protegidas).

- privada, são operações que podem ser acessados somente pelas operações da própria classe, sendo totalmente encapsuladas (- é o símbolo da UML para operações privadas).

A figura a seguir mostra uma classe com um exemplo de cada tipo de operação. Neste exemplo ao invés do caractere determinado pela UML é utilizado um ícone que também é possível.

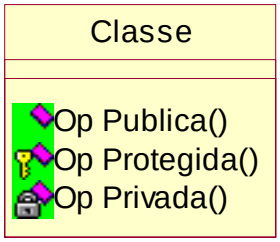


Figura 8.1 - Exemplo de classe com 3 tipos de operações

Uma vez determinados as operações deve-se definir os métodos, ou seja, deve-se descrever e implementar o corpo da operação. A palavra método é muitas vezes utilizada no mesmo sentido de operação (o que a princípio não faz muita diferença), mas método é a construção da operação. Nesta construção deve ser avaliado algoritmos especiais, que outros objetos ou operações serão utilizadas, como os atributos serão utilizados, como será implementado o relacionamento entre as classes e como isto reflete no método.

8.3 Atributos

Inicialmente determinamos os atributos da classe que se originou de classes de entidade. Estas classes possuem atributos que fazem parte da própria modelagem. Para identificar outros atributos devemos estudar as descrições dos métodos e determinar quais informações devem ser mantidas.

Os atributos possuem uma representação determinada por um nome, um tipo e um valor padrão (*default*) que é opcional. É importante na representação dos atributos seguir normas que facilitem a identificação.

O atributos, assim como as operações, possuem visibilidade que são:

+ público, são atributos que podem ser acessadas por operações de outras classes. **Não é recomendado o uso de atributos públicos** porque isto irá degradar consideravelmente o encapsulamento da classe. Se for necessário a informação ou alterar o valor de um atributo deve-se desenvolver uma operação pública que permita acessar o atributo(+ é o símbolo da UML para atributos públicos)

 # protegido, são atributos que podem ser acessados pelas operações da própria classe e pelas classes derivadas (especialização) (# é o símbolo da UML para atributos protegidos).

- privado, são atributos que podem ser acessados somente pelas operações da própria classe, sendo totalmente encapsulados (- é o símbolo da UML para atributos privados).

A figura a seguir mostra uma classe com um exemplo de cada tipo de atributo, neste exemplo ao invés do caractere determinado pela UML é utilizado um ícone que também é previsto.

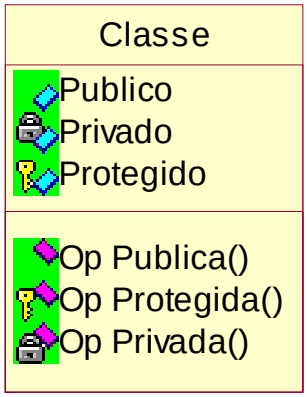


Figura 8.2 - Exemplo de atributos de classe

8.4 Operações e Atributos de escopo

Normalmente os atributos e as operações de uma classe só irão existir quando a classe for instanciada, ou seja, se transforme em um objeto. Porém existe a possibilidade de se ter um atributo ou operação que seja relacionado com a classe. Um exemplo de atributo de escopo é o valor máximo de um determinado valor. No sistema para Sacar Dinheiro poderia ser o valor máximo retirado do caixa por vez ou o tempo mínimo entre um saque e outro. Nestes exemplos estes valores não estão relacionados ao objeto e sim ao sistema. O método que permitisse ler ou alterar um valor de escopo teria que ser realizado por um método de escopo.

A figura a seguir mostra um exemplo de variável de escopo que na UML é representado por um atributo sublinhado. Como a variável de escopo esta associado com todo o sistema ela pode ser pública.

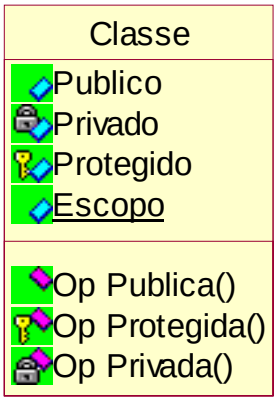


Figura 8.3 - Exemplo de classe com atributo de escopo

8.5 Relacionamento entre as classes

Toda classe possui um relacionamento, não faz sentido desenvolver uma classe que não possua um cliente, seria desenvolver uma classe sem finalidade. Os relacionamento sempre são feitos através da parte pública da classe, para ser mais exato, pelas operações (métodos) públicas uma vez que não deve existir atributos públicos.

Considerando o relacionamento de herança (especialização/generalização) podemos demostrar o relacionamento como na figura 8.4 a seguir.

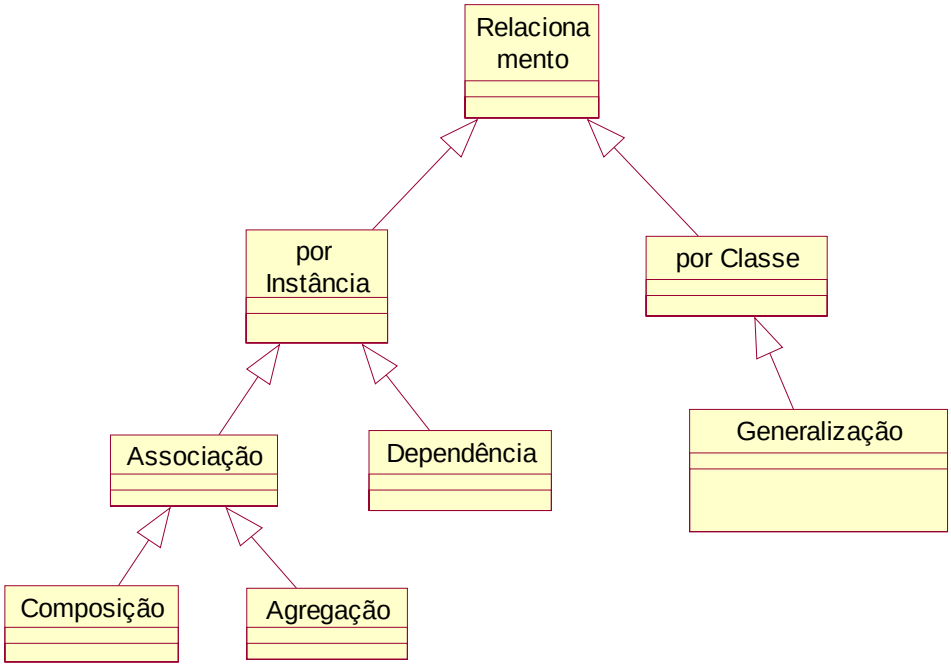


Figura 8.4 - Árvore de herança de relacionamentos

O relacionamento é dividido em dois tipos:

- por Instância: Neste tipo existe um objeto ou mais objetos da outra classe que realizam o relacionamento
- por Classe: Neste tipo existe um relacionamento de herança (generalização \ especialização) entre as classes e neste caso o relacionamento não é por objeto.

8.5.1 Dependência

A dependência se caracteriza por uma relação leve entre dois objetos, ou seja, a relação não é estrutural. Nos relacionamentos por instância existe sempre a figura do objeto fornecedor e do objeto cliente, ou seja, um objeto irá fornecer um determinado serviço que será consumido pelo cliente. Para determinar o tipo de relacionamento é importante

avaliar como cliente e fornecedor se relacionam e como o cliente é visível ao fornecedor. A figura 8.5 é o exemplo de diagrama de classe com dependência.

Figura 8.5 - Relacionamento entre Cliente e Fornecedor

No tópico seguinte iremos detalhar a associação, porém é importante fazer uma comparação no sentido de avaliar se um relacionamento é uma dependência ou uma associação. Associações são relacionamentos estruturais, ou seja, são relacionamento mais duradouros e normalmente o cliente está o tempo todo necessitando de serviços do fornecedor.

Um relacionamento de dependência pode ser implementado como:

- Uma variável local a uma operação
- Um parâmetro por referência
- Uma classe utilitária

Um relacionamento de associação pode ser implementado como:

- Um atributo da classe

Na figura 8.6 a seguir Cliente tem um relacionamento de dependência com o Fornecedor1 uma associação com o Fornecedor2, e esta é a forma de representação na UML.

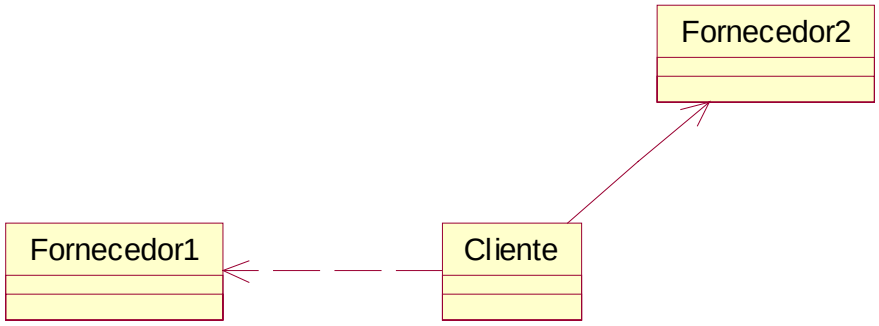


Figura 8.6 - Representação na UML de dependência e associação

8.5.1.1 Dependência com variável local

A implementação de uma variável local a uma operação é exemplificado na figura a seguir, a operação op1() contém um objeto instanciado do objeto fornecedor 1.

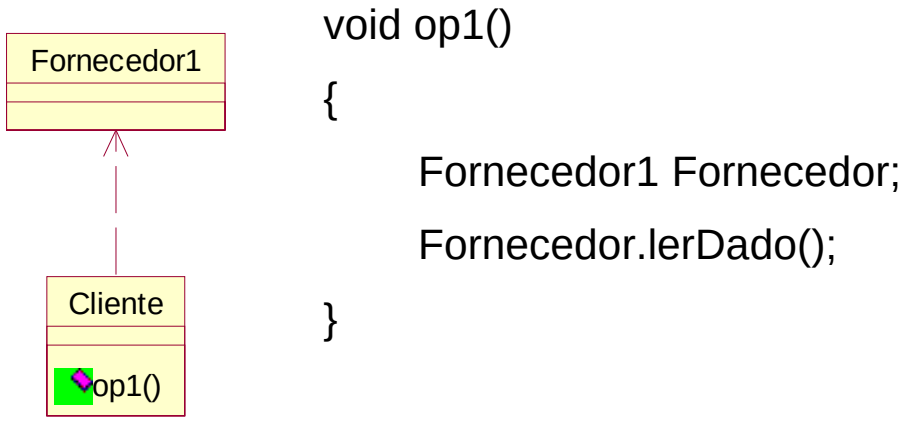
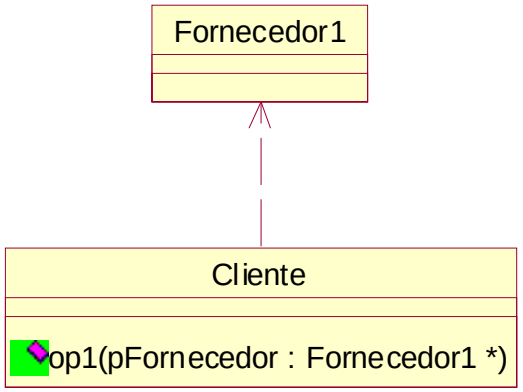


Figura 8.7 - Exemplo de implementação de dependência com variável local

A dependência nem sempre deve ser modelada e isto deve ser avaliado para que não poluir o diagrama de classes de projeto.

8.5.1.2 Dependência com parâmetro por referência

A implementação de dependência é exemplificada na figura a seguir onde a instância (um objeto) da classe Forncedor1 é passado para a classe cliente que irá utilizar os serviços.



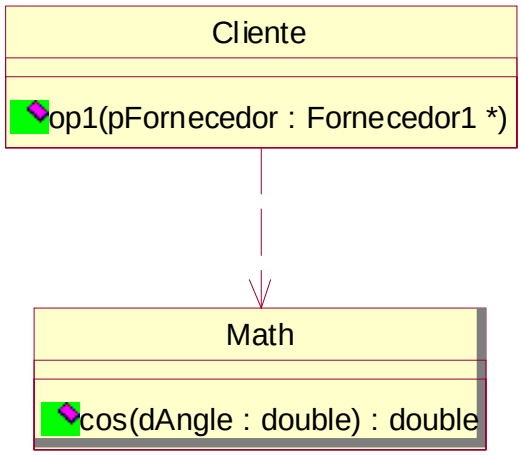

```
void op1(Fornecedor1* pFornecedor)
{
        pFornecedor->lerDado();
}
```

Figura 8.8 - Exemplo de implementação de dependência por passagem de parâmetro

Esta modelagem deve ser feita dependendo da necessidade de compreensão do relacionamento.

8.5.1.3 Dependência com classe utilitária

Uma instância de uma classe utilitária (*Utility Class*) é visível para todas as classes, é considerada uma classe global.




```
void op1(pFornecedor* pFornecedor)
{
        double dValue;
        dValue = Math.cos(30);
}
```

Figura 8.9 - Exemplo de uma dependência por Classe Utilitária

Uma classe utilitária é uma classe de auxílio e normalmente não é modelado, ou seja, não aparece no modelo, este relacionamento poderá aparecer em várias classes que irá poluir

o diagrama. As classes utilitárias são representadas na UML com o lado direito e inferior com uma faixa cinza.

8.5.2 Associação

Como já foi dito anteriormente uma associação é um relacionamento duradouro e pode ser dividido em composição e agregação. Em alguns casos é implementado como um atributo da classe e não é modelado, em outros casos é modelado no diagrama (apesar que na geração de código o efeito será o mesmo). É importante determinar qual a navegabilidade, ou seja, que classe conhece a outra ou se as duas classes se conhecem, ou seja, podem acessar os serviços uma da outra. A multiplicidade que determina quantos objetos de uma classe esta associado a outra classe. Outra característica importante é determinar se é necessário o desenvolvimento de uma classe de associação que será responsável pelo relacionamento.

8.5.2.1 Composição

Composição é uma associação muito forte e representa que o tempo de vida entre os objetos coincide, uma frase representa este relacionamento:

"**A parte não vive sem o todo**"

Um exemplo é a classe árvore que é **composta** de galhos, então valeria dizer que um galho não vive sem a árvore e isto é verdade.

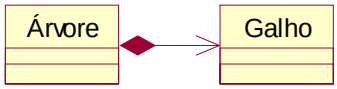


Figura 8.10 - Exemplo de Composição

A representação de agregação na UML é um losango colorido(normalmente preto) no lado da classe que possui a parte (o todo), a seta representa a navegabilidade, ou seja, quem utiliza os serviços de quem.

Uma pergunta que se deve fazer é se devemos utilizar atributos ou composição e devemos seguir as seguintes regras:

- Usamos composição quando:
  - As classes necessitam de modelagem independente
  - O relacionamento implica no comportamento do sistema
  - Existe necessidade de mostrar a visibilidade entre as classes relacionadas
- Caso contrário utilizamos atributos e o relacionamento não aparece na modelagem

8.5.2.2 Agregação

Uma agregação é uma associação de força média em que o tempo de vida não coincide e os objetos podem existir independente um do outro. Como exemplo podemos dizer que os pássaros estão **agregados** a floresta assim como as árvores (em tese árvores e pássaros podem existem sem florestas).

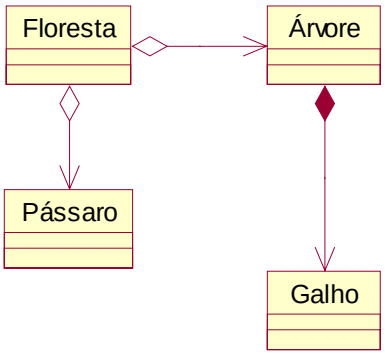


Figura 8.11 - Exemplos de Agregação

A agregação é representado na UML como um losango branco como mostrado na figura anterior. É importante ter em mente que o modelo representa um mundo real mas sem

expressar todos os detalhes do mundo real seria impossível. Portanto devemos modelar o que é realmente importante para que possamos entender completamente o que se deseja do sistema. Use o bom senso sempre que estiver modelando, não exagere nos relacionamentos, porém não deixe de fora relacionamentos importantes para a compreensão, evolução ou manutenção do sistema.

8.5.2.3 Classes de Associação

Uma classe de associação contém as propriedades do relacionamento existindo uma instância por relacionamento. A classe de associação é sugerida quando se tem uma associação múltipla (um para muitos 1 -> *) para composição ou agregação. As classes de associação para este tipo de relacionamento normalmente são listas de objetos e são responsáveis pela passagem de parâmetros entre os objetos e pela manutenção dos objetos nesta lista. Uma forma de implementar uma lista é utilizar classes parametrizadas (*template*) que são classes que definem outras classes e que são implementas em tempo de compilação, representam classes conhecidas como *container*, por exemplo listas, pilhas, filas, etc. Uma classe *template* é apresentada na UML como a seguir:

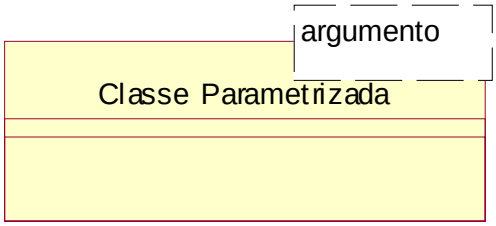


Figura 8.12 - Representação de *templates* na UML

Existem basicamente três tipos de multiplicidade em agregação ou composição:

- 1 para 1 (1 -> 1) – Um objeto de uma classe possui relacionamento com um objeto da outra classe.
- 1 para muitos (1 -> *) ou (1 -> n) – Um objeto de uma classe possui relacionamento com vários objetos da outra classe.

- Muitos para muitos (* -> *) ou (n -> n) – Muitos objetos de uma classe possuem relacionamento com vários objetos da outra classe. Esta opção deve ser evitada, optando-se por uma classe de associação entre as classes de tal forma que temos:

Figura 8.13 - Relacionamento Muitos para Muitos

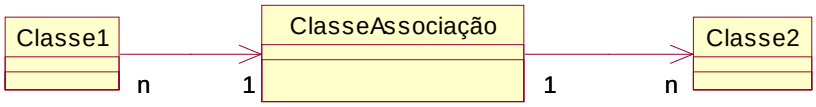


Figura 8.14 - Implementação com uma Classe de Associação

As outras formas de multiplicidade são derivadas destas três.

8.5.2.4 Navegabilidade

A navegabilidade indica que a classe de origem tem visibilidade da classe destino e se não houver direção identificada as classes podem visualizar uma a outra. Um exemplo de composição bidirecional de 1 para 1 pode ser implementada em C++ da seguinte forma:

```
class Fornecedor
{
        private:
        Cliente *m_pCliente;
        public:
        Fornecedor(Cliente *pCliente){m_pCliente = pCliente}; // Construtor com ligação com o  cliente
        ~Fornecedor();
}

class Cliente
{
        private:
        Fornecedor *m_pFornecedor;
        public:
        Cliente(){
                m_pFornecedor = new Fornecedor(this); //Construtor inicializando objeto do fornecedor
                };
        ~Cliente();
}
```

O que indica que é uma composição é que o Fornecedor não existe sem o Cliente e é bidirecional porque ambos os objetos podem chamar métodos de um para o outro. O relacionamento é um para um, ou seja, um objeto de uma classe terá um objeto da outra classe.

8.5.3 Generalização

Uma classe pode compartilhar a estrutura e/ou o comportamento de outra classe e este compartilhamento é chamado de generalização que também é conhecido como “é um tipo de” ou “é um”, ou seja, quando ao descrever a classe se utilizar esta expressões provavelmente existirá um relacionamento de generalização entre as classes. Se utilizarmos o exemplo de florestas e árvores, sabemos que florestas são compostas por Jatobás, Cedros, Anjico, Peróba, e por ai vai. Estes exemplos são todos **tipos de** árvore e Jatobá **é uma** árvore.

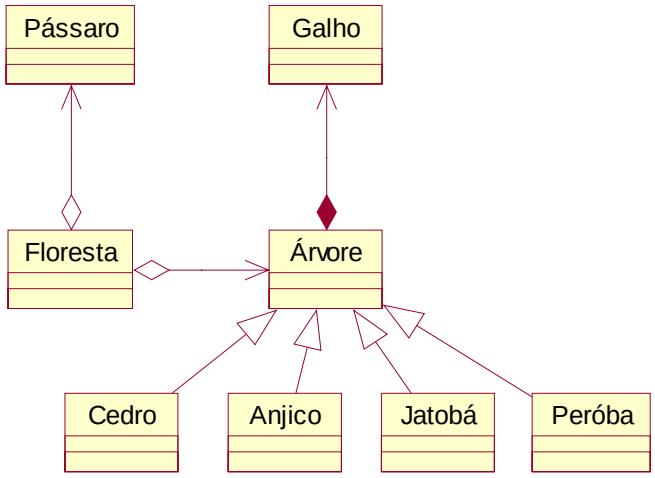


Figura 8.15 - Exemplo de Generalização (Herança) e outros relacionamentos

Uma classe muito utilizada em generalização é a classe abstrata. Classes abstratas representam comportamentos genéricos mas se instanciarmos estas classes elas não tem como implementar o comportamento. Para implementar será necessário que uma classe concreta faça o trabalho. Por exemplo, sabemos que uma árvore tem o comportamento de gerar frutos, porém se formos instanciar a classe árvore não saberemos como implementar este comportamento mas podemos implementar a operação caso a classe seja Jatobá este comportamento é conhecido.

Em C++ não existe o conceito de interface, por isto podemos utilizar classes abstratas para implementar uma interface.

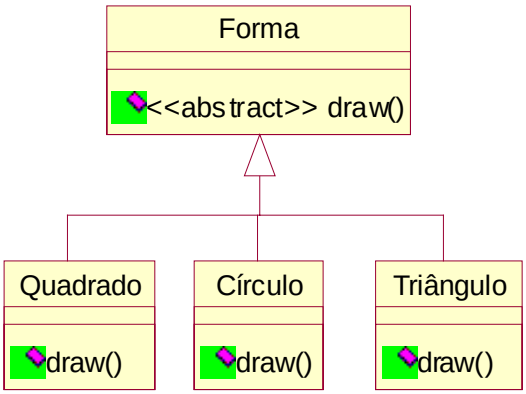


Figura 8.16 - Exemplo de relacionamento de generalização com classe abstrata

Em desenvolvimento não se deve utilizar múltipla herança, ou seja, uma classe que herda o comportamento de outras duas. Isto pode causar problemas de execução. A seguir é mostrado um exemplo de múltipla herança, onde não é possível determinar qual operação da classe será executada quando for solicitado o serviço Draw.

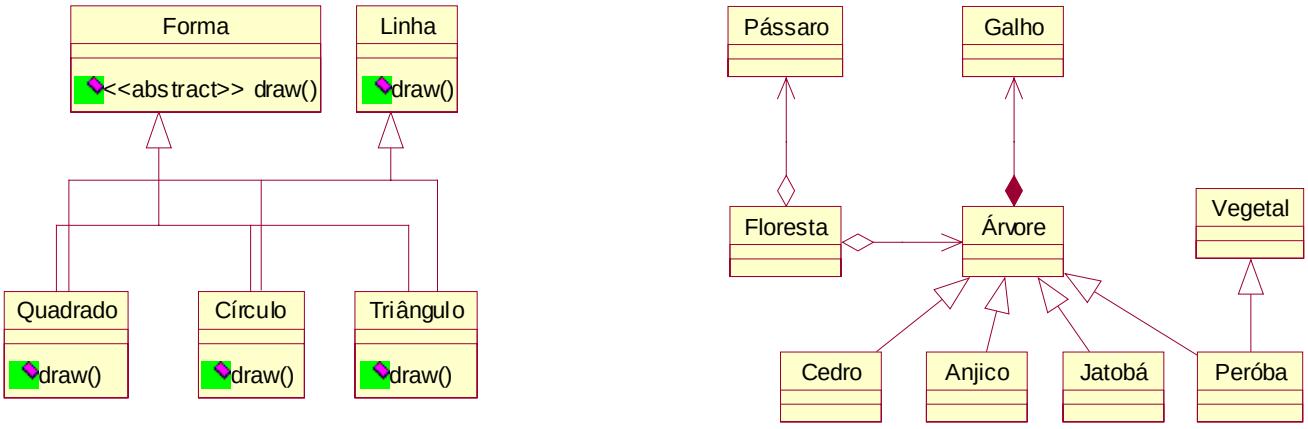


Figura 8.17 - Exemplos de múltipla herança

Mas como resolver o problema quando se necessita de um relacionamento mais duradouro? Muitas vezes resolvemos o problema substituindo uma das generalizações por composição. Uma outra forma é verificar se na realidade o que temos não é herança

em seqüência, por exemplo, sabemos que Peróba é realmente uma árvore e um vegetal, porém todos as outras árvores são vegetais então podemos mudar o modelo como é mostrado a seguir.

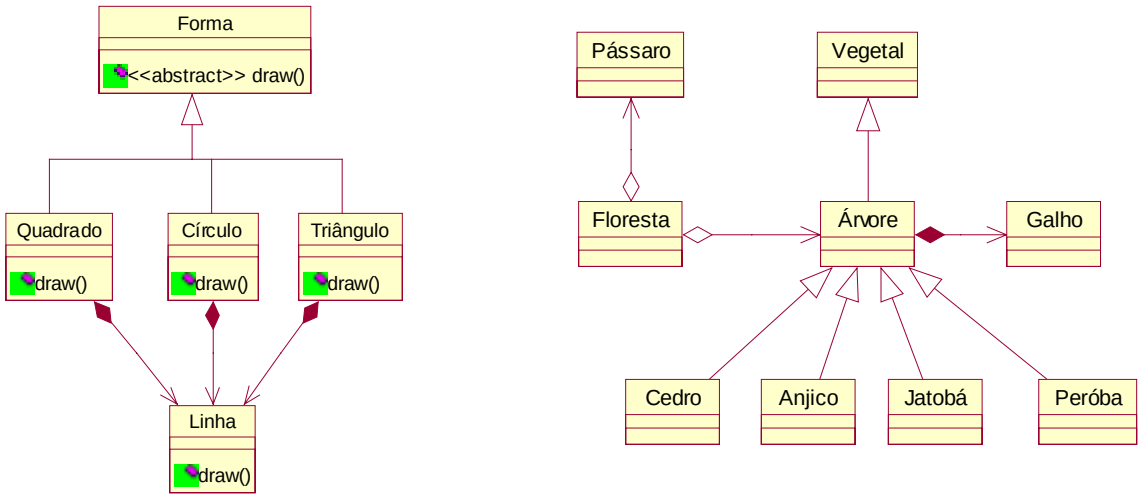


Figura 8.18 - Como solucionar o problema de múltipla herança

8.6 Diagrama de Classes de Projeto

O diagrama de classes de projeto representa o ponto onde o MPDS inicia o final do ciclo. Neste ponto o desenvolvimento do modelo toma a forma do projeto e as classes desenvolvidas serão apresentadas com os relacionamentos, operações e atributos. É possível iniciar o trabalho de transformação de operações em métodos que é o preenchimento das operações com a seqüência de código que irá prover o serviço. No diagrama de classes de projeto é possível avaliar a modelagem do ou dos casos de uso que foram trabalhadas e analisar do ponto de vista da arquitetura escolhida para o sistema se esta tudo de acordo ou se é necessário que se altere alguma coisa na modelagem.

As ferramentas de geração de código automático são capazes de gerar o esqueleto do projeto que será implementado, gerando então um código executável. A montagem do modelo de classes de projeto depende da experiência do projetista com a linguagem e

com orientação a objetos. Um projeto será tanto melhor quanto mais simples e fácil de manter ele for, além é claro de corresponder aos requisitos de funcionalidade, robustez e performance.

8.7 Geração de Código

A geração de código é o penúltimo passo do MPDS e está intimamente ligado a linguagem que foi escolhida para o desenvolvimento. A cada geração e teste (que veremos mais no próximo capítulo) de código considera-se o final de um incremento. Então a seqüência do projeto deve ser avaliada e os caminhos orientados a partir da avaliação dos riscos. Determina-se que outras partes do projeto serão trabalhadas.

O processo é cíclico e vai-se a cada ciclo gerando-se mais codificação que permitirá ao final do último ciclo a integração final do sistema.

As classes de análise de várias casos de uso são mapeadas em classes de projeto que permitam executar as funcionalidades.

9. TESTES

9.1 Introdução

Existem vários estudos e ferramentas para facilitar a execução de testes que detectem o quanto antes os famosos *"bugs"* de software que podem comprometer todo o desenvolvimento e consequentemente todo o produto[5][11]. Existe um consenso que diz que todo software tem algum *bug* que pode nunca ser descoberto, porém a função do teste é permitir que um sistema seja consistente e confiável o bastante para ser utilizado[16]. No MPDS o teste representa o final do ciclo quando o executável é gerado. Os testes de sistema dependem fundamentalmente do tipo de sistema que esta em desenvolvimento, existem 3 tipos de teste que devem ser utilizados, dependendo do tipo de sistema. Para todos os três tipos de teste deve-se preencher a seguinte tabela[24].

<table>
<tr><td colspan="2">Responsável:
Inclua o nome da pessoa responsável pela execução do teste</td><td>Data:
Inclua a data de execução do teste no formato dd/mm/aa</td></tr>
<tr><td colspan="3">Recursos necessários:
Inclua a especificação de hardware e software da(s) máquina(s) envolvida(s) no teste.</td></tr>
<tr><td>Hardware</td><td>Configuração</td><td>Software</td></tr>
<tr><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td></tr>
<tr><td colspan="3">Procedimentos:
Descreva os procedimentos para a execução do teste.</td></tr>
<tr><td colspan="3">Resultados:
Descreva os resultados obtidos ao final do teste.</td></tr>
</table>

9.2 Teste de Classe

A classe representa, em orientação a objetos, a menor parte de um projeto. Quanto melhor for desenvolvido o sistema, mais específica e simples devem ser as classes. Desta forma é importante garantir que todos os métodos da classes executem de acordo com o que foi determinado, inclusive, em casos de exceção que devem ser tratados pelos métodos. Sendo assim, o foco é testar a classe, ou seja, confirmar se a classe atende as responsabilidades atribuídas. Para este fim são desenvolvidos métodos específicos para teste que devem ser diferenciados dos demais com, por exemplo, a inclusão da palavra *test* antes do nome do método. A criação de métodos específicos para teste facilita que após cada incremento os testes sejam repetidos para verificar se o comportamento da classe não foi alterado de forma incorreta.

9.3 Teste de *Stress*

O teste de *stress* é utilizado para aplicar ao sistema um conjunto muito grande de dados e analisar o comportamento do sistema. Ao final do incremento o executável gerado é submetido a um programa especificamente desenvolvido para se comunicar com o sistema em desenvolvimento e intensificar o uso. Normalmente este teste é aplicado a sistemas que manipulem dados, ou sistemas de tempo real.

9.4 Teste de Funcionalidade

O teste de funcionalidade é baseado nos casos de uso desenvolvidas para o sistema. Os fluxos de evento ótimos e alternativos, são aplicados ao sistema a fim de garantir o funcionamento do sistema de acordo com os casos de uso (ou requisitos funcionais) trabalhadas no incremento. A utilização de caso de uso para testes normalmente é chamada de *test case*[32], ou caso de teste, o que na prática representa utilizar os fluxos de eventos para verificar se o sistema os atende.

10. SISTEMAS MULTICAMADAS E SISTEMAS DE TEMPO REAL

10.1 Introdução

A complexidade do sistemas pode variar de um simples programa de consulta de dados até sistemas complexos utilizando a internet como meio de comunicação e com dados e usuários espalhados por todo o mundo. Como desenvolver um sistema tão complexo ? A complexidade e a responsabilidade pode é deve ser dividida em pequenas partes especializadas em determinadas atividades. Esta técnica é conhecida como **Objeto Distribuído**[19], e está relacionada com as decisões de arquitetura do projeto que dependem de muito de fatores como: experiência da equipe nas tecnologias, tempo hábil para aquisição de novos conhecimentos, necessidades do sistema, etc. Neste ponto do desenvolvimento deve-se iniciar o projeto do sistema do ponto de vista de pacotes, que representam os agrupamentos de partes do sistema que facilite o desenvolvimento e a manutenção do sistema; de camadas, que representam os níveis de negócio representado na construção do sistema; e de componentes que representam os códigos gerados e como se comunicam.

10.2 Componentes

Os componentes são as partes reais que compõem o produto como por exemplo arquivos executáveis, bibliotecas como DLL, *ActiveX*, componentes COM, COM+ ou DCOM, ou ainda componentes java como por exemplo *JavaBeans* ou *Java Enterprise Beans*. Estes componentes são responsáveis pelo funcionamento do sistema principalmente quando se trata de um sistema que funciona em camadas. Em especial sistemas para internet e como o desenvolvimento de sistemas distribuídos para este ambiente esta crescendo a cada dia (principalmente com o surgimento de tecnologias como .NET e J2EE). É fundamental a modelagem dos componentes para facilitar a distribuição do comportamento dos casos de

uso. O momento de fazer esta modelagem vai depender do conhecimento das necessidades do sistema, o quanto antes melhor porém talvez seja necessário aprofundar o desenvolvimento para definir o modelo de componentes, as vezes é necessário realizar o primeiro incremento para definir corretamente este modelo.

A figura a seguir mostra um exemplo de quatro componentes para um sistema que, por exemplo, funcionaria na internet.

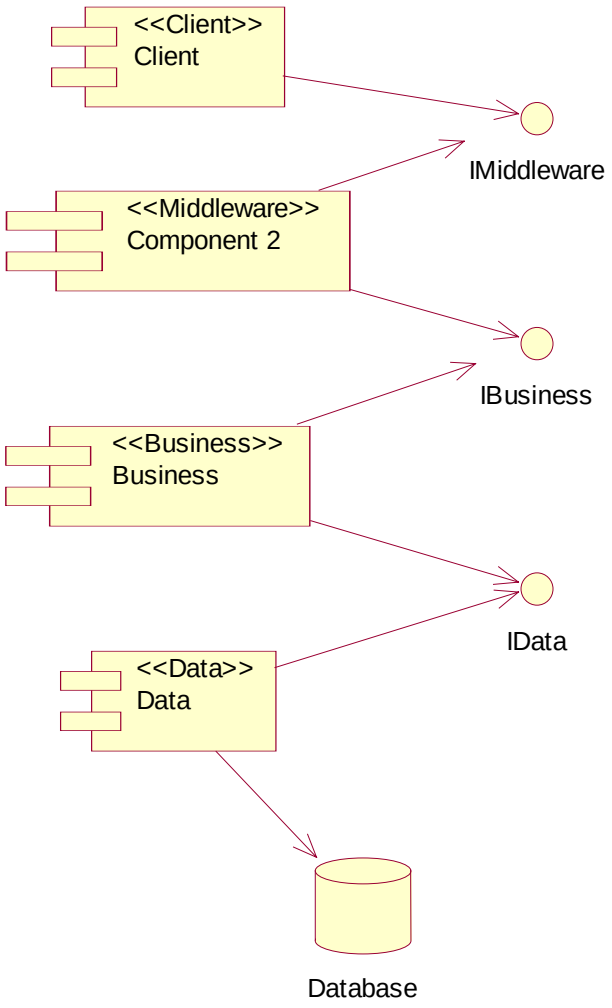


Figura 10.1 - Componentes em sistemas distribuídos e camadas

A técnica do uso de componentes irá cada vez mais fazer parte do desenvolvimento de projetos e portanto a modelagem dos componentes é fundamental para o desenvolvimento do projeto sistema. A divisão do sistemas em componentes força a definição de interfaces claras que permitam a comunicação entre eles, esta interface é um **contrato** entre as partes do projeto.

10.3 Pacotes

Os pacotes são unidades menores e que facilitam a melhor compreensão do sistema, porém na maioria dos casos o pacote não possui um comportamento único e determinado, ou seja, um pacote existe em função da subdivisão de um projeto. Os pacotes irão possuir relacionamento entre si e que esta relacionado com o contrato estabelecido no uso de cada parte do sistema. Os pacotes permitem um detalhamento mais apurado e o isolamento de comportamentos do sistema. Estudando as classes de análise deve-se verificar que grupos de classes poderão fazer parte de um pacote e sempre que possível faça a divisão em pacotes, isto irá facilitar a continuação do desenvolvimento.

As classes internas a um pacote podem ser públicas ou privadas:

- Classes públicas podem ser acessadas de fora do pacote e representam a "interface" do pacote, porém estas classes não necessariamente executam algum tipo de ação de conformidade.
- Classes privadas só podem ser acessadas por classes internas ao pacote e representam as classes que executam funções que são necessárias a uma ou mais classe do pacote

As classes públicas representam o contrato de desenvolvimento que permitirá o desenvolvimento em paralelo e qualquer alteração destas classes devem ser feitas de comum acordo entre as equipes de desenvolvimento.

Na UML o pacote é apresentado como mostrado na figura a seguir.

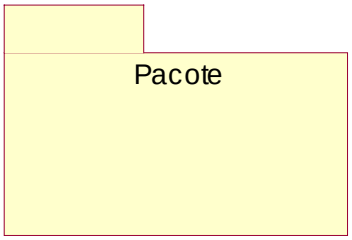


Figura 10.2 - Símbolo de pacote na UML

10.4 Camadas

No início os sistemas eram desenvolvidos em uma camada, ou seja, era construído um único sistema que era responsável pelos dados e pela interface com o cliente. Em função da complexidade dos sistemas, principalmente em sistemas desenvolvidos para a internet, foi necessário criar meios que permitissem a comunicação entre estas partes e surgiu (como mostrado na figura a seguir) o sistema de 3 camadas:

- uma camada cliente (ou *client*), onde é feita a interface de dados com o usuário do sistema,
- uma camada servidor (ou *server*), onde é feita o fornecimento dos dados e
- uma camada meio (ou *middleware*) que é responsável pela comunicação e troca de dados entre cliente e servidor.

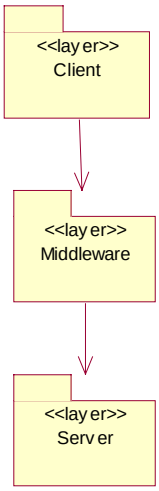


Figura 10.3 - Sistema de três camadas

Em função dos sistemas se tornarem mais complexos e da necessidade de distribuição dos dados (que originalmente estavam na camada servidor) que pode estar em qualquer lugar(fisicamente falando), foi necessário a divisão da camada *server* (como mostra a figura a seguir)em duas partes:

- camada de negócio, que é responsável pelo tratamento dos dados que são enviados ou recebidos do cliente e
- camada de dados, que é responsável pela interface com o repositório de dados.

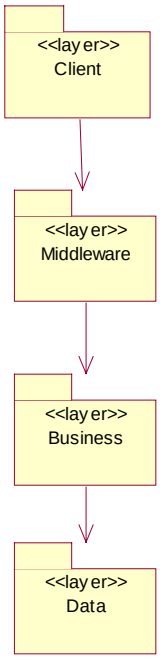


Figura 10.4 - Exemplo de sistema com 4 camadas

A decisão da divisão do sistema em camadas vai depender da complexidade e necessidades do sistema. Esta decisão irá influenciar diretamente na tecnologia a ser adotada e no conhecimento técnico necessário da equipe para o desenvolvimento do projeto. O uso de padrões de componentes irá facilitar a escolha da melhor forma de dimensionar o projeto em função da tecnologia existente.

As camadas normalmente são consideradas pacotes e vão naturalmente influenciar na distribuição da atividade de desenvolvimento em paralelo.

10.5 Sistemas de Tempo Real

Um sistema de tempo real leva em consideração o tempo de resposta a um estimulo externo, normalmente esta associado a um equipamento o sistema externo ao sistema em desenvolvimento[12][53]. Alguns sistemas operacionais (como QNX por exemplo) permitem a execução de processos de modo a atender a estas necessidades. Os processos

são executadas obedecendo uma determinada política e prioridade. Nestes casos os processos compartilham um mesmo processador no sentido de atender a reposta a evento tão rápida quanto possível. Porém não é necessário que o sistema operacional seja especificamente para tempo real, pois esta avaliação é feita em termos do estímulo e da resposta a este estímulo, ou seja, o sistema operacional deve ser capaz de executar tarefas múltiplas (*multi task*) “pseudo” simultâneas. Isto a maioria dos sistemas operacionais atuais permite. O sistemas de automação industrial via de regra são de tempo real, uma vez que as respostas devem atender a uma necessidade real do processo ou linha de produção[19][23].

Uma vez identificado que o sistema é de tempo real deve-se utilizar o diagrama de estado da UML para demostrar o comportamento dinâmico do objeto(ou classe) responsável pelos estados, que não é possível de ser analisado no diagrama de seqüência. A implementação do diagrama de estado não é complexa e muitas vezes um método de controle analisa o estado atual e o evento recebido e promove a mudança de estado ou não em função do evento recebido. Os métodos da classe responsáveis por cada estado irão atualizar os valores de estado da classe.

10.6 Exemplo de Sistema de Tempo Real: Reconhecimento de Chamada Telefônica

Um sistema de reconhecimento de chamada esta ligado a linha telefônica e ao receber as chamadas recebe também o número do telefone que esta chamando, estes números são lidos, armazenados na memória para uma futura consulta e apresentados em um *display*. A figura a seguir mostra o diagrama de estado que representa estes estados.

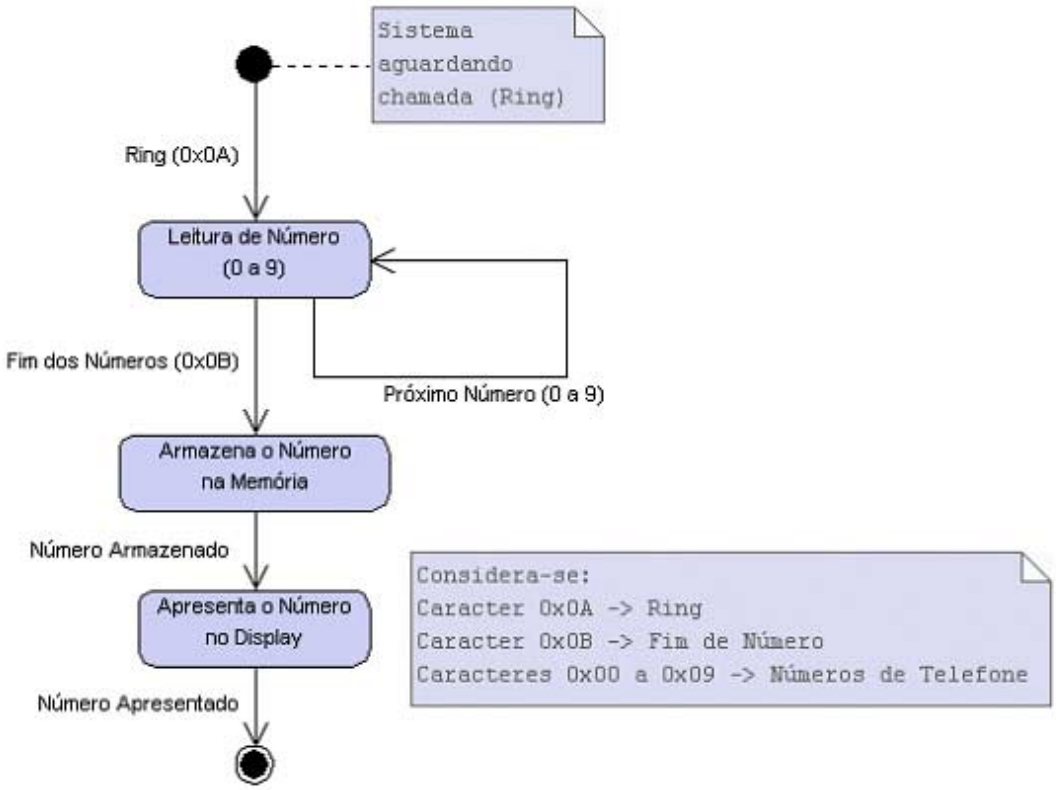


Figura 10.5 - Exemplo de Diagrama de Estado

11. ANÁLISE DOS RESULTADOS

11.1 Aplicando o Modelo

Para a utilização do MPDS é necessário o conhecimento dos diagramas da UML e de treinamento na forma de utilização do modelo. É importante também que se disponha de uma ferramenta de apoio ao desenvolvimento que facilite a geração dos diagramas.

As fases que são abrangidas pelo MPDS estão ligadas a Analistas, Projetistas e Arquitetos de Sistemas que normalmente são as pessoas com mais experiência da equipe, o que facilitará a utilização. Porém como o MPDS está baseado em uma linguagem conhecida e aberta, a UML, é possível que estudantes possam se utilizar das técnicas apresentadas aqui no desenvolvimento de projetos.

O MPDS foi desenvolvido para pequenos projetos e pequenas equipes(equipes inexperientes em engenharia de software como empresas incubadas por exemplo), porém tanto grandes projetos como pequenos podem utilizá-lo. Neste caso o que poderá diferenciar será a quantidade de diagramas gerados, o número de casos de uso encontrados e o número de interações necessárias para o desenvolvimento completo do sistema. Normalmente projetos mais complexos necessitam de um número maior de pessoas/equipes e isto influenciará o gerenciamento do projeto.

Este trabalho mostrou a necessidade de criação de um livro e um curso que pudessem ser utilizados como base para o treinamento de desenvolvedores e pessoas envolvidas no processo de desenvolvimento de software.

Além do material didático esta pesquisa propiciou a base tecnológica necessária para a implementação de uma ferramenta de auxilio ao desenvolvimento que facilitasse a automação do processo de engenharia. Esta ferramenta permite a ligação entre as atividades e produz os diagramas e documentos relativos ao desenvolvimento. Esta ferramenta foi chamada de Cattleya (uma orquídea brasileira) e será apresentada em mais detalhes no capítulo de conclusões do trabalho.

11.2 Os prós e contra do Modelo Proposto

Neste tópico abordam-se as vantagens e desvantagens do modelo. Principalmente no que se refere a aplicação prática, uma vez que a intenção do modelo é facilitar e dar consistência ao processo de desenvolvimento de software.

11.2.1 As vantagens do MPDS

11.2.1.1 É baseado na UML

O MPDS não é um novo modelo e sim uma abordagem prática do uso da UML no processo de engenharia para desenvolvimento de software. Isto faz do MPDS um modelo baseado em um conceito padrão de mercado e utilizado por várias empresas ao redor do mundo.

11.2.1.2 Um Modelo Prático

O MPDS é um modelo com uma visão totalmente prática. Transformando uma técnica conhecida e padronizada em um processo prático levando em consideração a visão do desenvolvedor e do grupo de desenvolvedores. E ainda levando em consideração as novas tecnologias relacionadas com o desenvolvimento de software, como objetos distribuídos e *extreme programming* por exemplo.

11.2.1.3 O uso de uma ferramenta de desenvolvimento

Para o desenvolvimento de software é importante o uso de uma ferramenta CASE (*Computer-Aided Software Engineer*) que automatize a modelagem do sistema como apresentado pelo MPDS. O uso de uma ferramenta dá agilidade durante o desenvolvimento do sistema. O MPDS sendo baseado na UML facilita o uso de ferramentas uma vez que existe um número muito grande de opções no mercado.

11.2.1.4 Qualidade, Produtividade e Manutenção

O foco principal do MPDS é permitir um processo de engenharia claro e consistente. Isto poderá levar ao desenvolvimento em um nível crescente de **produtividade**, que produza um produto final com **qualidade** que facilite a **manutenção** e evolução. A maior

eficiência no uso do modelo será conseguida quanto mais projetos forem desenvolvidos utilizando-o e aprimorando o processo com adaptações pontuais.

11.2.2 As desvantagens do MPDS

11.2.2.1 Implantação

Como todo novo processo o MPDS necessita de uma fase de implantação onde projetos serão escolhidos para facilitar a compreensão das equipes envolvidas em desenvolvimento de software. Esta fase pode causar atrasos e num primeiro momento parecer que o processo mais atrapalha do que ajuda porque a produtividade irá cair até que o modelo esteja implementado.

11.2.2.2 Treinamento

Na fase de implantação será necessário o treinamento da equipe e sempre que novos pessoas forem contratadas também deverão ser treinadas. Isto irá causar o investimento de tempo e recursos financeiros que devem ser levados em conta na utilização do modelo.

11.2.2.3 Necessita de uma ferramenta de desenvolvimento

Uma das grandes vantagens de se utilizar um modelo é a utilização de ferramentas que facilitem o processo de desenvolvimento. Em alguns casos isto pode inviabilizar o uso do modelo. Este trabalho de pesquisa serviu de base para o desenvolvimento de uma ferramenta, que na sua versão mais avançada, irá permitir desenvolver o sistema utilizando todo o MPDS. Gerando toda a documentação e código (para as principais linguagens orientadas a objeto) necessários ao desenvolvimento do sistema.

11.3 Comparações com outros Modelos e/ou Processos

O MPDS possui características de alguns modelos e/ou processos já existentes focalizando a necessidade de pequenas empresas/grupos de desenvolvimento de software. Um modelo, que apresenta soluções de engenharia de software e que está sendo seguido por várias empresas é o RUP (*Rational Unified Process*)[46] da *Rational Corporation*. O RUP possui o inconveniente de ser um produto comercial, o que não é interessante para pequenas empresas. O RUP é extremamente poderoso porém complexo e difícil de ser

adaptado a uma pequena empresa/grupo. Segundo Martin Fowler o RUP é um *framework* de processos e como tal pode acomodar uma grande variedade de processos, e esta é sua crítica a respeito do RUP: “Como o RUP pode ser qualquer coisa, acaba não sendo nada”[17]. O processo de engenharia de software no RUP está associado à gerência de projeto e gerência de configuração e mudança. O MPDS é focado no processo de engenharia para pequenas empresas/grupos de desenvolvimento de software permitindo agilidade no desenvolvimento do projeto.

Recentemente surgiu um movimento chamado *Extreme Programming* (XP)[3]. O XP sugere que o grupo de desenvolvimento deve adaptar a necessidade de modelagem e documentação para a engenharia de software às necessidade do projeto. Porém em pequenas empresas/grupos que não sejam experientes e que não conheçam algum processo é difícil realizar a adaptação. Algumas práticas estabelecidas para XP estão presentes no MPDS ou podem ser utilizadas sem nenhuma alteração, como por exemplo: planejamento de versão, planejamento de iteração, versões pequenas em cada iteração, programação aos pares, desenvolvimento orientado a testes, integração contínua, padronização de código, entre outras. Em relação ao projeto e documentação do sistema o MPDS complementa o XP, fornecendo uma solução de engenharia baseado nos diagramas da UML e seguindo uma seqüência de atividades que muito se próxima do XP.

Uma linha de trabalho que se baseou no XP é o *Agile Modeling* (AM)[6][1]. O AM é definido por Scott Ambler como sendo uma coleção de princípios que quando aplicados no desenvolvimento de software representam uma coleção de práticas de modelagem. Muitas práticas do AM são semelhantes ao XP(como por exemplo: “use o diagrama correto”), e podem ser utilizadas em conjunto com o MPDS. Com relação a modelagem a sugestão do AM é “modelar com um propósito” e “utilizar múltiplos diagramas” o que corresponde ao MPDS. No entanto o MPDS vai além, propondo um *template* (este *template* pode ser adquirido na página de documentos do site www.umlnapratica.com.br) único que apresente toda a documentação do projeto. Além disso o MPDS apresenta o **onde** e o **como** utilizar os diagramas, dando propósito a modelagem, através da apresentação da seqüência de atividades a serem desenvolvidas durante o

desenvolvimento. É importante ressaltar que nem XP e nem AM utilizam-se especificamente de diagramas da UML para a modelagem do sistemas. Outros diagramas podem ser utilizados(principalmente no AM) para a modelagem do sistema.

Existe um outro processo chamado de "*ICONIX process*"[40] da empresa *ICONIX Software Engineering, Inc* que, assim como o MPDS e o RUP, é orientado a casos de uso. Este processo tem a desvantagem, assim como o RUP, de ser comercial. Além de orientado a caso de uso o processo *ICONIX* é também orientado a protótipo. Diferente do MPDS, o processo ICONIX sugere que seja feita uma "modelagem de domínio" que represente uma primeira versão do diagrama de classes de alto nível. Uma importante contribuição deste processo para o MPDS foi a definição e uso das classes de análise [43] que não faz parte da UML. Os autores, Doug Rosenberg e Kendall Scott, consideram que este modelo situa-se entre o RUP e o XP.

O processo *ICONIX* considera o desenvolvimento das classes de análise como a ponte entre os casos de uso e o diagrama de seqüência. Este raciocínio é aproveitado no MPDS. Diferente do processo ICONIX, o MPDS utiliza o diagrama de seqüência como o elo de ligação entre o final da análise e início do projeto. Além disto o MPDS mostra e dá ênfase a seqüência das atividades de desenvolvimento e não somente aos diagramas a serem utilizados.

O MPDS utiliza conceitos encontradas no RUP, no XP, no AM e no processo ICONIX. Estas metodologias foram incorporadas a um processo incremental e interativo, simples e que fornece orientações claras no processo de desenvolvimento de software. Para facilitar a utilização do processo um *template* simples foi criado para ser utilizado como documentação do projeto. O MPDS ainda se baseia fortemente nos diagramas da UML para facilitar a compreensão e troca de informação dos desenvolvedores entre si e com o cliente.

11.4 Avaliação em projetos reais

Com a intenção de avaliar a aplicação do MPDS em projetos reais o modelo foi aplicado a 17 grupos de alunos do curso de Computação III do Inatel, sendo cada grupo formado

por 6 ou 7 alunos com a intenção de desenvolver um produto de software completo utilizando o MPDS. Vale salientar que os alunos não possuíam qualquer conhecimento de orientação a objetos, possuindo um conhecimento básico da linguagem C. A avaliação dos projetos foi a seguinte:

10 grupos (59%) atingiram os objetivos acadêmicos porém os projetos não foram satisfatórios.

3 grupos (18%) atingiram os objetivos acadêmicos e os projetos foram modelados e implementados corretamente.

4 grupos (23%) atingiram os objetivos acadêmicos, os projetos foram modelados e implementados corretamente e foram tratados como produtos, sendo que os alunos continuaram o desenvolvimento do projeto.

Apesar de mais da metade dos grupos ter atingido somente os objetos acadêmicos (que em última análise é o foco do curso) 41 % dos grupos conseguiu utilizar as técnicas do MPDS em um projeto real. O uso do modelo no curso permitiu a assimilação muito rápida do conceito de engenharia de software em grupos inexperientes em desenvolvimento orientado a objetos.

11.5 Aplicabilidade em pequenas empresas/equipes de desenvolvimento de software

O MPDS foi desenvolvido com foco em pequenas empresas/equipes de software com a intenção de definir um modelo de engenharia de software que pudesse ser adotado sem representar um peso para estas empresas. A motivação para o desenvolvimento e difusão do MPDS é que do conjunto de empresas brasileiras de desenvolvimento de software, 77,4 % dele é composto por micro ou pequenas, sendo que as micro (que representam 35,7 % do total) possuem de 1 a 9 pessoas[36]. Em virtude do número muito pequeno de pessoas a maioria esta envolvida na produção e comercialização dos produtos de software.

12. CONCLUSÕES

12.1 Contribuições Práticas da Pesquisa

12.1.1 A Ferramenta Cattleya

Este trabalho de pesquisa iniciou-se com o estudo da UML e a possibilidade de aplicação no desenvolvimento de projetos práticos. Junto a este estudo iniciou-se a avaliação de ferramentas de apoio ao desenvolvimento, ferramentas CASE. Chegou-se a conclusão que as ferramentas ou são muito caras ou são simples e não atendem a todo o processo de desenvolvimento. Para dar suporte ao desenvolvimento iniciou-se o desenvolvimento da ferramenta Cattleya que na sua primeira versão teria as seguintes características:

- Diagrama de classes de forma a facilitar o desenvolvimento das classes de projeto e seu relacionamento
- Geração de código em C++

Esta primeira fase do sistema foi encerrada em dezembro de 2001 com o produto sendo disponibilizado para uso a partir de março de 2002. O sistema Cattleya pode ser adquirido gratuitamente através de *download* pela internet.

12.2 Contribuições Didáticas da Pesquisa

12.2.1 Curso de Engenharia de Software

Esta pesquisa proporcionou o desenvolvimento de um curso de engenharia de software a ser aplicado em cursos de engenharia elétrica e em engenharia de computação. A parte teórica do curso é composta dos seguintes módulos:

- Sistemas Orientados a Objetos
- Especificação de Requisitos
- Modelagem de caso de uso

- Classes de Análise
- Diagrama de Seqüência
- Pacotes e Subsistemas
- Classes de Projeto
- Projeto de Sistemas

A ferramenta Cattleya esta sendo usada no laboratório para o desenvolvimento de projetos de curso.
Este curso foi iniciado em fevereiro de 2002 para o terceiro período de Engenharia Elétrica do Inatel.

12.2.2 Livro “UML na prática – Do Problema ao Sistema”

Em função da necessidade de um processo de engenharia que facilite o uso da UML esta pesquisa proporcionou a criação de um livro. Este livro servirá de base para o treinamento, inclusive o curso de engenharia de software, de profissionais que participem de projetos de desenvolvimento. Servirá também de um guia prático para o processo de engenharia de software.

12.3 Trabalhos Futuros

12.3.1 Melhoria no modelo e atualização ferramentaCattleya

O modelo, fruto desta pesquisa, é resultado de um estudo de técnicas desenvolvidas ao longo dos últimos anos para o desenvolvimento de software e em especial da UML. É fruto ainda de anos de experiência no desenvolvimento de sistemas e dos problemas associados a esta atividade. A intenção do MPDS é representar a visão prática e ser aderente às necessidades das equipes de desenvolvimento. Por esta razão o aprimoramento do modelo deve ser contínuo e busca sempre encontrar soluções que dêem esta tão esperada aderência. A ferramenta Cattleya deverá suportar todas as atividades para que as equipes possam ter uma ferramenta de alto valor agregado que

aumente a competitividade das empresas desenvolvedoras. Isto permitirá que os produtos possam ter sempre o mais alto nível de qualidade que se exige dos sistemas cada vez mais complexos.

12.3.2 Sistemas Embarcados

Para o processo de desenvolvimento de software já existe um volume de estudo bastante grande realizado durante os últimos anos. Vários modelos e processos foram criados. Os sistemas embarcados (também conhecidos como *firmware*), são responsáveis por um percentual muito maior do software desenvolvido no mundo e possuem peculiaridades e características muito diferentes. Várias características, como dependência direta do projeto de *hardware*, dificultam muito o trabalho e apesar disto muito menos estudo foi e tem sido feito no sentido de desenvolver modelos que possam ser utilizados para o desenvolvimento de *firmware*. O MPDS apresenta soluções ainda incompletas para sistemas embarcados ou de tempo real.

12.3.3 Modelagem de sistema com componentes

Os novos desafios para o desenvolvimento de software estão intimamente ligados a sistemas que utilizarão a internet como base operacional . Isso implica no aprimoramento das técnicas de engenharia de software[2], uma vez que os sistemas passarão a ser compostos de partes, os chamados **componentes**. Estes poderão estar em qualquer lugar do mundo e poderão se comunicar. Este campo da engenharia de software deverá requisitar muitas pesquisas nos próximos anos de modo a permitir que os produtos finais, que na verdade serão uma coletânea de vários produtos associados, possam funcionar adequadamente. Este trabalho faz uma introdução, nos capítulos Modelagem de Componentes e Modelagem de Pacotes e Camadas, ao desenvolvimento de sistemas baseados em componentes.

13. REFERÊNCIAS BIBLIOGRÁFICAS

[1] Ambler, S. ,“Architectural Modeling”, Software Development Magazine, 1999, On-line de http://www.sdmagazine.com

[2] Ambler, S., “Distributed Object Transactions”, Software Development Magazine, 2000, On-line de http://www.sdmagazine.com

[3] Ambler, S. ,“Extreme Modeling”, Software Development Magazine, 2000, On-line de http://www.sdmagazine.com

[4] Ambler, S. ,“Object-Oriented Business Rules”, Software Development Magazine, 2000, On-line de http://www.sdmagazine.com

[5] Ambler, S. ,“Object Testing Patterns”, Software Development Magazine, 1999, On-line de http://www.sdmagazine.com

[6] Ambler, S., “The Principles of Agile Modeling”, The Official Agile Modeling(AM) Site, 2002, On-line de http://www.agilemodeling.com/principles.htm

[7] Berard, E. ,“Be Careful With Use Cases”, The Object Agency, Inc, 2000, On-line de http://www.toa.com/pub/use_cases.htm

[8] Booch, G., Rumbaugh J., Jacobson, I., “The Unified Modeling Language User Guide”, Addison-Wesley, 1999.

[9] Booch, G., Rumbaugh J., Jacobson, I., “The Unified Software Development Process” , Addison-Wesley, 1999.

[10] Cockburn, A. ,“Writing Effective Use Case”, Addison-Wesley, 2001.

[11] Dimaggio, L. ,“A Roadmap for Software Test Engineering”, Software Development Magazine, 2001, On-line de http://www.sdmagazine.com

[12] Douglass, B., “Real-Time UML”, Addison-Wesley Longman, 1998

[13] Evans, G., “Object Think: It Really is Different”, 1999, Evanetics, Inc., On-line de http://www.evanetics.com

[14] Eriksson, H., Penker, M., “UML Toolkit”, Wiley Computer Publishing, 1998.

[15] Firesmith, D., “Use Case Modeling Guidelines”, Lante Corporation, IEEE Transaction Software Engineer , 1999, pp. 184-193

[16] Fowler, M. ,“A UML Testing Framework”, Software Development Magazine, 1999, On-line de http://www.sdmagazine.com

[17] Fowler, M. ,“The New Methodology”, Martin Fowler web site, 2001, On-line de http://www.martinfowler.com

[18] Fowler, M., Scott, K., “UML Distilled”, Addison-Wesley, 1998

[19] Fleisch, W., “Applying Use Cases for the Requirements Validation of Component-Based Real-Time Software”, IEEE Transaction Software Engineer , 1999, pp. 75-84

[20] Gamma, E., Helm, R., Johnson, R. Vlissides, J., “Design Patterns”, Addison-Wesley, 1997

[21] Hantos, P., “A Systems Engineering View of Requirements Management for Software-intensive Systems ”, IEEE Transaction Software Engineer 1999, pp. 620-621

[22] Henderson-Sellers, B., Collins, G., Graham, I. “UML-Compatible Process”, IEEE Transaction Software Engineer, 34th Hawaii International Conference on System Sciences, 2001.

[23] Heverhagen, T. Tracht, R., “Implementing Function Block Adapters”, Rational Corporation, 2001, On-line de http://www.rational.com

[24] Jeffries, R., “RecordMap, Test First”, X Programming, XP Magazine, 2002, On-line de http://www.xprogramming.com

[25] Jeffries, R., “XP and Reliability”, X Programming, XP Magazine, 2001, On-line de http://www.xprogramming.com

[26] Kiedaisch, F., Pohl, M., Bauer, S. Ortmann, S., “Requirements Archaeology: From Unstructured Information to High Quality Specifications”, IEEE Transaction Software Engineer , 2001, pp. 304-305

[27] Keuffel, W. ,“Best Practices Actually Applied”, Software Development Magazine, 2000, On-line de http://www.sdmagazine.com

[28] Korson, M. ,“Constructing Useful Use Case”, McGregor Korson web site, 2000, On-line de http://www.korson-mcgregor.com

[29] Kruchten, P., “The Rational Unified Process, an Introduction”, Addison-Wesley, 2000.

[30] Larman, C., “Utilizando UML e Padrões”, Bookman, 2000

[31] McConnell, S., “Rapid Development”, Microsoft Press, 1996

[32] McGregor, J., Major, M., “Selecting Test Case Based on User Priorities”, Software Development Magazine, 2000, On-line de http://www.sdmagazine.com

[33] Oestereich, B. ,“Developing Software with UML”, , Addison-Wesley, 1999

[34] Paulk, M. ,“Extreme Programming from a CMM Perspective”, Software Engineering Institute, 2001

[35] Paulk, M., Weber, C. Curtis, B., Chrissis, M., “The Capability Maturity Model”, Addison-Wesley, 1997.

[36] Pesquisa Censo SW, MCT/SEPIN/DSI, 2001.

[37] Phillips, C., Kemp, E., Sai, K., “Extending UML Use Case Modeling to Support Graphical User Interface Design ”, IEEE Transaction Software Engineer , 2001, pp.48-52

[38] Pressman, R., “Software Engineering”, McGraw-Hill, 2001

[39] Rosemberg, D., Scott, K., “Driving Design: The Problem Domain”, Software Development Magazine, 2001, On-line de http://www.sdmagazine.com

[40] Rosemberg, D., Scott, K., “Driving Design with Use Cases”, Software Development Magazine, 2000, On-line de http://www.sdmagazine.com

[41] Rosemberg, D., Scott, K. ,“Give Them What They Want”, Software Development Magazine, 2001, On-line de http://www.sdmagazine.com

[42] Rosemberg, D., Scott, K., “Sequence Diagram: One Step at a Time”, Software Development Magazine, 2001, On-line de http://www.sdmagazine.com

[43] Rosemberg, D., Scott, K., “Successful Robustness Analysis”, Software Development Magazine, 2001, On-line de http://www.sdmagazine.com

[44] Rosemberg, D., Scott, K. ,“Top Ten Use Case Mistakes”, Software Development Magazine, 2001, On-line de http://www.sdmagazine.com

[45] Rumbaugh, J., Blaha, M. Premerlani, W., Eddy, F., Lorensen, W., “Object-Oriented Modeling and Design”, Prentice Hall International Editions, 1991.

[46] Rumbaugh, J., “Rational Unified Process”, Rational Corporation, 2001 .

[47] Saeki, M, “Reusing Use Case Descriptions for Requirements Specification: Towards Use Case Patterns”, IEEE Transaction Software Engineer , 1999, pp. 309-316

[48] Sawyer, P., Sommerville, I., Viller, S., “Capturing The Benefits of Requirements Engineering”, IEEE Transaction Software Engineer , 1999, pp. 78-85

[49] Schneider, G., Winters, J., “Applying Use Cases”, Addison-Wesley, 2001.

[50] Selonen, P., “Generating Structured Implementation Schemes from UML Sequence Diagrams”, IEEE Transaction Software Engineer , 2001, pp. 317-328

[51] Selonen, P., Koskimies, K, Sakkinem, M. “How to Make Apples from Oranges in UML”, IEEE Transaction Software Engineer, 34th Hawaii International Conference on System Sciences, 2001.

[52] Smith, R. ,“Defining the UML Kernel”, Software Development Magazine, 2000, On-line de http://www.sdmagazine.com

[53] Spionla, M., “Diretrizes para desenvolvimento de software de sistemas embutidos”, Tese de Doutorado, Universidade de São Paulo – USP, 1999

[54] Stroustrup, B., “The C++ Programming Language”, Addison-Wesley, 1997.

[55] Susan, L., “Use Case Pitfalls: Top 10 Problems from Real Projects Using Use Cases”, IEEE Transaction Software Engineer , 1999, pp. 174-183

[56] Unified Modeling Language Specification Version 1.3 , Object Management Group (OMG), 1999

[57] What Is OMG-UML And Why Is It Important, OMG, 2001, On-line de http://www.omg.org

[58] Wiegers, K., “Karl Wiegers Describes 10 Requirements Traps to Avoid”, Process Impact, Software Testing & Quality Engineering, 2000, On-line de http://www.processimpact.com

[59] Wiegers, K., “Writing Quality Requirements”, Process Impact, 1999, On-line de http://www.processimpact.com