UNIVERSIDADE FEDERAL DE SANTA CATARINA
PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

ALEXANDRE PERIN DE SOUZA

ENGENHARIA DE SISTEMAS COMPUTACIONAIS: UMA PROPOSTA DE MAPEAMENTO DE MODELOS UML PARA A LINGUAGEM JAVA

Dissertação submetida à Universidade Federal de Santa Catarina como parte dos requisitos para a obtenção do grau de Mestre em Ciência da Computação.

Prof.: Vitório Bruno Mazzola, Dr.
Orientador

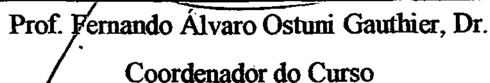
Florianópolis, setembro de 2000.

ENGENHARIA DE SISTEMAS COMPUTACIONAIS: UMA PROPOSTA DE MAPEAMENTO DE MODELOS UML PARA A LINGUAGEM JAVA

ALEXANDRE PERIN DE SOUZA

Esta dissertação foi julgada adequada para a obtenção do título de **MESTRE EM CIÊNCIA DA COMPUTAÇÃO** Área de concentração Sistemas de Computação e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação.

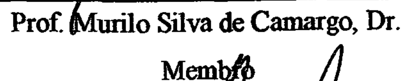
Prof. Vitório Bruno Mazzola, Dr.
Orientador



Prof. Fernando Álvaro Ostuni Gauthier, Dr.
Coordenador do Curso

Banca Examinadora:

Prof. Roberto Willrich, Dr.
Membro



Prof. Murilo Silva de Camargo, Dr.
Membro

Prof. Vitório Bruno Mazzola, Dr.
Membro

Agradecimentos

Ao professor Vitório Bruno Mazzola pela oportunidade, pelo trabalho de orientação e pelas imensas horas de dedicação.

Ao Senac de Lages na pessoa do diretor geral Aldo Spessatto pelo suporte, compreensão e apoio.

A Uniplac pelo incentivo e confiança.

Às amigas, Verinha e Valdete, que não mediram esforços para me ajudar neste desafio.

Ao professor Roberto Willrich pela valiosa participação e contribuição.

Aos alunos da Curso de Graduação em Informática, pela compreensão e apoio.

A todos os Colegas da Uniplac, em especial o Corpo Docente do Curso de Informática, pela constante amizade, convivência e disponibilidade de ajuda.

Aos meus Pais, pela vida, pelo freqüente apoio.

A minha família pela confiança.

A todos que me ajudaram . . .

A quem por tudo Foi, É e sempre Será possível: Deus.

Sumário

LISTA DE FIGURAS .... **VI**

LISTA DE TABELAS .... **IX**

LISTA DE ABREVIATURAS .... **X**

RESUMO .... **XI**

ABSTRACT .... **XII**

CAPÍTULO 1. INTRODUÇÃO **1**

1.1 APRESENTAÇÃO 1

1.2 JUSTIFICATIVA 4

1.3 OBJETIVOS 5

1.3.1 Geral *5*

1.3.2 Específicos *5*

1.4 ESCOPO DO TRABALHO 5

1.5 TRABALHOS RELACIONADOS 7

1.6 VISÃO HISTÓRICA DA PESQUISA 7

CAPÍTULO 2. LINGUAGEM JAVA **11**

2.1 APRESENTAÇÃO 11

2.2 CARACTERÍSTICAS 12

2.2.1 Simples *14*

2.2.2 Orientada a Objetos *14*

2.2.3 Interpretada *15*

2.2.4 Distribuída *17*

2.2.5 Robusta *18*

2.2.6 Segura *18*

2.2.7 Arquitetura Neutra *20*

2.2.8 Portável *20*

2.2.9 Alta Performance *21*

2.2.10 Encadeamentos Múltiplos *21*

2.2.11 Dinâmica *22*

2.3 CONCEITOS DE OO NO CONTEXTO DE JAVA 23

2.3.1 Introdução às Classes e aos Objetos *23*

2.3.2 Encapsulamento e Ocultamento de dados *23*

2.3.2.1 Modificadores 26

2.3.2.2 Interface 28

2.3.2.3 Criação de uma Instância de Objeto 29

2.3.3 Métodos *30*

2.3.3.1 Sobrecarga de Métodos (Overloading) 31

2.3.3.2 Métodos Estáticos 32

2.3.4 Mensagens *33*

2.3.5 Herança *34*

2.3.6 Polimorfismo *36*

CAPÍTULO 3. LINGUAGEM DE MODELAGEM UNIFICADA 40

3.1 INTRODUÇÃO 40

3.2 A NOTAÇÃO DA UML 42

3.2.1 Visões *42*

3.2.1.1 Visão Lógica 42

3.2.1.2 Visão de casos de uso "Use-case" 43

3.2.1.3 Visão de Componentes 43

3.2.1.4 Visão de Concorrência 43

3.2.1.5 Visão de Organização 43

3.2.2 Elementos Essenciais *44*

3.2.2.1 Classes 44

3.2.2.2 Objeto 45

3.2.2.3 Estado 45

3.2.2.4 Pacote 46

3.2.2.5 Componentes 47

3.2.2.6 Relacionamentos 47

3.2.3 Elementos Complementares *54*

3.2.4 Diagramas da UML *55*

3.2.4.1 Diagrama de Use-cases 56
3.2.4.2 Diagrama de Classe 57
3.2.4.3 Diagrama de Objeto 59
3.2.4.4 Diagramas de Seqüência 60
3.2.4.5 Diagramas de Colaboração 62
3.2.4.6 Diagramas de Estado 63
3.2.4.7 Diagramas de Atividade 64
3.2.4.8 Diagramas de Componentes 66
3.2.4.9 Diagramas de Implantação 67

CAPÍTULO 4. PROCESSO DE TRADUÇÃO UML - JAVA 69

4.1 INTRODUÇÃO 69
4.2 ANÁLISE DE CORRESPONDÊNCIAS ENTRE UML - JAVA 69
4.2.1 Limitações UML - Java 75
4.2.2 Processo de Mapeamento 76
4.3 ILUSTRAÇÃO DO PROCESSO DE MAPEAMENTO DE MODELOS UML EM JAVA 78
4.3.1 Estudo de Caso 1 – Aplicação "Hello World!" 78
4.3.2 Estudo de Caso 2 – Aplicação "Secretaria Eletrônica Hipotética" 83
4.3.3 Estudo de Caso 3 – "Sistema de Contas Bancárias Hipotético" 88
4.3.4 Estudo de Caso 4 – "Sistema de Informações Escolares Hipotético" 96
4.3.5 Estudo de Caso 5 – "Jogo de Dados - Craps" 100
4.4 ENGENHARIA REVERSA JAVA - UML 105

CAPÍTULO 5. CONCLUSÃO 107

BIBLIOGRAFIA 112

ANEXO 1 – COMPARAÇÃO ENTRE FERRAMENTAS CASE 111

ANEXO 2 – EXEMPLO DA IDE DO EDITOR DE MODELOS 113

ANEXO 3 – CÓDIGO JAVA EXEMPLO: INTERFACE E FINAL 114

LISTA DE FIGURAS

Figura 1 - Representação da geração de bytecodes em Java [SUN95] 15
Figura 2 - Representação do Processo de Compilação/Interpretação Java [SUN95] 16
Figura 3 - O Ciclo de Vida do Java e o Relacionamento com suas Camadas de Segurança[HOP97] 19
Figura 4 – Exemplo de definição da classe-Exemplo em Java 25
Figura 5 - Exemplo ilustrativo da classe funcionário em Java 25
Figura 6 – Fragmento de código Java utilizando o modificador final 27
Figura 7 - Código Java utilizando o modificador abstract 27
Figura 8 - Exemplo de utilização de Interfaces 29
Figura 9 – Fragmento de código Java de criação de objeto da classe Exemplo 29
Figura 10 - Exemplo de codificação de métodos em Java 30
Figura 11 - Exemplo de sobrecarga em Java 31
Figura 12 - Exemplo de código Java para membros de classe (variáveis e métodos).... 33
Figura 13 - Representação do Mecanismo de Passagem de Mensagens entre Objetos.. 33
Figura 14 - Exemplo de definição da classe-Carro em Java 35
Figura 15 - Exemplo de definição da sub-classe-carroCorrida em Java 35
Figura 16 - Exemplo de polimorfismo em Java 38
Figura 17 - Representação de uma classe em UML 44
Figura 18 - Representação de um objeto em UML 45
Figura 19 - Representação de um estado em UML 46
Figura 20 - Representação de um pacote em UML 47
Figura 21 - Notação de relacionamentos em UML 48
Figura 22 – Representação de Associação por Agregação 49
Figura 23 – Representação de Associação por Composição 50
Figura 24 - Generalização 52
Figura 25 - Generalização Restrita 53
Figura 26 - Generalização Completa 53
Figura 27 - Associação por Dependência 54
Figura 28 - Diagrama de Use Case 57
Figura 29 - Diagrama de Classes 59

Figura 30 - Diagrama de Objetos 60
Figura 31 - Diagrama de Seqüência 61
Figura 32 - Diagrama de Colaboração 63
Figura 33 - Diagrama de Estado 64
Figura 34 - Diagrama de Atividade [SUR99] 65
Figura 35 - Diagrama de Componentes 66
Figura 36 - Diagrama de Implantação 68
Figura 37 - Meta-modelo UML-Java, baseado em [GCA00] 73
Figura 38 - Esquema de tradução UML - Java 75
Figura 39 – Diagrama de Classe para o Exemplo Hello World 79
Figura 40 - O mecanismo de operação de paint 80
Figura 41 – Diagrama de componentes de HelloWorld 80
Figura 42 – Fragmento de código Java para Aplicação HelloWorld 81
Figura 43 – Fragmento de código Java para Relacionamentos de Dependência de Pacotes e Classes 82
Figura 44 – Fragmento de código Java para Relacionamento de Dependência 82
Figura 45 - Código Java completo para aplicação HelloWorld 83
Figura 46 - Diagrama de classe para uma Secretaria Eletrônica Hipotética 84
Figura 47 - Diagrama de Estados para uma Secretaria Eletrônica Hipotética 85
Figura 48 - Diagrama de Use-case para Secretaria Eletrônica Hipotética 85
Figura 49 - Código Java para a classe Secretaria Eletrônica Hipotética 87
Figura 50 - Diagrama de Use-case - Funções do funcionário do banco 89
Figura 51 - Diagrama Use-case - Funções do Cliente 89
Figura 52 - Diagrama de Classes para Sistema de Contas 90
Figura 53 - Diagrama de seqüência para criação de conta corrente – funcionário 91
Figura 54 - Diagrama de Seqüência para obtenção de saldo em conta corrente – cliente 91
Figura 55 - Fragmento de código Java para associação em cliente 92
Figura 56 - Fragmento de código Java para associação em conta 93
Figura 57 - Fragmento de código para especialização de conta 93
Figura 58 - Fragmento de código para especialização de Conta 93
Figura 59 - Código para as classes conta, corrente e poupança da Aplicação Contas .. 95

Figura 60 - Diagrama de classes para Sistema de Escola hipotético 97
Figura 61 - Diagrama Use-case para o Sistema de Informações Escolares Hipotético .. 98
Figura 62 - Fragmento de código Java para relacionamento do tipo agregação 99
Figura 63 - Tela de execução do aplicativo dados 100
Figura 64 - Diagrama de classes para o exemplo do jogo de dados 101
Figura 65 - Fragmento de código Java com inserção de pacotes e classes para o Jogo de dados 101
Figura 66 - Fragmento de código para atributos final 102
Figura 67 - Fragmento de código Java para método init( ) 103
Figura 68 - Fragmento de código para método play( ) 104
Figura 69 - Fragmento de código de implementação método actionPerformed( ) 105
Figura 70 - Exemplo da IDE do Editor de Modelos – Diagrama de classe 113
Figura 71 - Exemplo da IDE do Editor de Modelos - Geração de código 113

LISTA DE TABELAS

Tabela 1 - Correspondências de UML-Java, baseada em [FUR98] 72

Tabela 2 - Definição de templates para conversão UML-Java 74

LISTA DE ABREVIATURAS

UML	*Unified Modeling Language*
OOAD	*Object Oriented Analysis and Design*
CASE	*Computer Aided Software Engineering*
OO	*Oriented Object*
WEB	*World Wide Web*
TCP/IP	*Transfer Control Protocol/Internet Protocol*
HTTP	*Hypertext Transfer Protocol*
FTP	*File Transfer Protocol*
URL	*Universal Resource Location*
OMT	*Object Modeling Technique*
OOSE	*Object Oriented Software Engineering*
OMG	*Object Management Group*
UML	*Unified Modeling Language*
LISP	*List Processing - Linguagem de programação de computadores de alto nível utilizada para processamento de listas de instruções ou dados*
C++	*Linguagem de programação de computadores de alto nível orientada a objetos*
Smalltalk	*Linguagem de programação de computadores de alto nível orientada a objetos*
VM	*Virtual Machine*
JVM	*Java Virtual Machine*
Windows NT	*Sistema Operacional de rede*
JDK	*Java Developer Kit*
Applets	*Pequenos aplicativos, para internet, escritos em Java*
Browser	*Aplicativo utilizado para navegação, entre sites, na Internet*
POO	*Programação Orientada a Objetos*
IDE	*Integrate Development Environment*

RESUMO

O fortalecimento da UML (*Unified Language Modeling*) como linguagem padrão para modelagem, tem resultado no aparecimento e desenvolvimento de inúmeras ferramentas CASE de geração automática de código. Estas ferramentas permitem modelar sistemas observando aspectos funcionais, comportamentais, estruturais e/ou organizacionais. O interessante, é perceber quais modelos podem ser codificados em uma linguagem de programação e de que forma esses modelos são traduzidos. Esta problemática facilita o entendimento do mecanismo de funcionamento das ferramentas de geração automática de código, além de ser um estudo preliminar para o desenvolvimento de uma futura ferramenta CASE.

A idéia deste trabalho, está em realizar um estudo que possa determinar a forma pela qual são realizados mapeamentos de modelos, representados na notação UML para a linguagem Java.

PALAVRAS-CHAVES: Orientação a Objetos, Linguagem de Modelagem Unificada, Linguagem de Programação Java, Ferramentas CASE.

ABSTRACT

The growth of UML (Unified Language Modeling) as a modeling tool, resulted in the emergence and development of a lot of CASE environments. These tools allow to model systems observing a system from different points of view: functional, behavioral and structural. The interesting is to notice which models can be mapped into a programming language and the manner those models are translated. This problem helps the understanding of the mapping process implemented into automatic code generation tools, besides being a preliminary study for the development of a future CASE tool.

This work deals with the definition of a mapping technique of models written in UML to source code in Java, which has revealed interesting features for the development of distributed applications.

KEYWORDS: Oriented Objects, Unified Modeling Language, Java, Software Engineering.

CAPÍTULO 1. INTRODUÇÃO

1.1 APRESENTAÇÃO

Atualmente o advento da modelagem de sistemas, tanto no meio acadêmico como fora dele, vêm sofrendo avanços significativos principalmente nesta década. Este crescimento tem impulsionado o aparecimento de ferramentas CASE, de geração automática de código, e modificado o perfil dos desenvolvedores de sistemas.

A adoção da modelagem e o crescimento das ferramentas CASE se justificam pela mudança de conjuntura, seja de mercado – onde, atualmente, existe facilidade para aquisição de um computador, ou de complexidade de sistemas - onde tem-se novas necessidades, usuários diferentes e novos requisitos a partir de novas aplicações.

Desta forma, ganha espaço a idéia de que a complexidade do software e do hardware vêm diminuindo. Por isso, os esforços devem ser focalizados em métodos que busquem a simplificação de atividades relacionadas às pessoas.

No entanto, verifica-se um conjunto de razões muito sérias que devem ser levadas em consideração para que se possa trabalhar o ser humano numa perspectiva que torne o trabalho de desenvolvimento de sistemas menos complexo. Seguindo essa idéia, podemos destacar:

- a grande dificuldade, que é intrínseca ao ser humano, em relação a mudanças;
- a incerteza do sucesso de novas tecnologias;
- a falta de pessoal especializado para difusão de novas tecnologias;
- a utilização de ferramentas inadequadas; e
- sistemas antigos que, geralmente, consomem vasta quantidade de recursos humanos e financeiros para mantê-los vivos.

Contudo, existe a necessidade de mudança urgente para a sobrevivência da empresa, para a conquista de novos mercados e para adequação a um mundo cercado de mudanças.

Neste sentido, não é o suficiente a adoção de novas tecnologias de engenharia de software, mas a mudança de enfoque de modelagem, pois as necessidades e expectativas dos usuários estão sendo ampliadas e modificadas a todo momento, seja pelo advento de novos computadores ou pela enorme gama de aplicações disponíveis no mercado, como por exemplo: acesso à Internet e a facilidade de aquisição de um computador.

Sendo assim, a mudança de enfoque de modelagem tradicional, baseado em um conjunto de programas que executam processos sobre dados, para o enfoque por objetos se justifica pela naturalidade com que os objetos existem na natureza e pela sua característica intrínseca de analisar o mundo como ele é (*back to reality concepts*), o que permite organizar sistemas de maneira mais realística, fácil e natural.

Nesta perspectiva, justifica-se a importância da criação de modelos que sejam uma simplificação da realidade. Esta é uma técnica da engenharia já aprovada, pois construímos plantas/modelos de arquitetura de casas e grandes prédios para auxiliar a equipe de engenharia na construção do produto final **[BOO00]**.

Esta técnica, de modelagem de sistemas, não está presente somente na engenharia mas, também, em outras áreas como a indústria automotiva, aeroespacial e outras. Seria inconcebível, por exemplo, projetarmos um novo automóvel sem primeiro construirmos modelos para entendimento do seu funcionamento, da sua estrutura, dos seus componentes e do seu comportamento. Além do mais, testes, cronogramas e custos seriam melhor dimensionados.

Sendo assim, abaixo, lista-se um conjunto de aspectos que reforçam a compreensão e o entendimento da construção de modelos:

- A modelagem ajuda a visualizar o sistema futuro;

- Através da modelagem (estrutural, dinâmica e funcional) pode-se perceber e especificar o que deve ser feito;
- A modelagem proporciona um guia para o desenvolvimento, do como deve ser feito; e
- Os modelos, construídos, ajudam a documentar as decisões tomadas.

Ainda, como justificativa para utilização de modelos, pode-se citar:

- o trabalho de modelagem deve ser realizado com a idéia da despreocupação com o **como deve ser feito**, ou seja, nos efeitos gerados na fase de modelagem. A modelagem deve ser fundamentada na essência e não nos efeitos; e
- outro aspecto a ser considerado, é a limitação da capacidade humana de compreender complexidades. Os modelos ajudam a diminuir a complexidade dos sistemas facilitando o seu entendimento como um todo.

Com esta perspectiva, de mudança de enfoque, agregada à necessidade de adoção de uma linguagem para representação de modelos surge a UML. A UML dá suporte à criação de modelos **[BOO00]**. Seguindo essa idéia, pode-se destacar também a linguagem Java que vem sendo utilizada para codificar/implementar modelos de sistemas.

Desta forma, este trabalho pretende oferecer aos engenheiros de software uma proposta de estudo de mapeamento de modelos representados na linguagem UML para a linguagem Java. A idéia, é mostrar de que forma modelos representados em UML são mapeados para Java.

Assim, a partir do perfeito entendimento do funcionamento do processo de mapeamento, pode-se chegar ao desenvolvimento de ferramentas CASE. Desta forma, não há dúvida que, o crescimento das CASE caminham na direção automática de geração de aplicativos a partir de uma especificação em nível de projeto.

Com esse fim, desenvolveu-se este trabalho que além deste capítulo introdutório, é constituído de três outros capítulos.

O capítulo 2 trata das características da linguagem Java, dos conceitos essenciais de orientação a objetos no contexto de Java.

O capítulo 3 apresenta características da linguagem de modelagem unificada.

O capítulo 4, corresponde ao Processo de Tradução UML - Java. Ele mostra uma análise comparativa entre modelos representados na notação UML e a sua correspondência na linguagem Java, apresentando também alguns estudos de caso.

Por fim, no capítulo 5, são tecidas algumas considerações acerca do trabalho desenvolvido.

1.2 JUSTIFICATIVA

Pode-se justificar este trabalho a partir das seguintes argumentações:

- ratificar a importância do conhecimento detalhado de processos de tradução de modelos para linguagens de programação;
- o pequeno número de referências a respeito de processos de tradução de modelos UML para a linguagem Java;
- a dificuldade em que se tem para estudo de processos de tradução de ferramentas CASE devido ao alto preço de aquisição das mesmas; e ao
- baixo número de ferramentas CASE que realizem mapeamentos UML para Java.

Em função do exposto acima, torna-se necessário um estudo que possa determinar detalhadamente a forma pela qual o processo de tradução é realizado.

1.3 OBJETIVOS

1.3.1 Geral

Esta dissertação tem por objetivo a realização de uma análise e elaboração de uma proposta de tradução de modelos de sistemas representados em UML para a linguagem de programação de computadores Java.

1.3.2 Específicos

São eles:

- estudo das características da linguagem de programação Java;
- refletir, discutir e propor processos de tradução de modelos em linguagens de programação;
- estudo da tecnologia de orientação a objetos no contexto de Java;
- aprendizado da notação padrão para modelagem de sistemas;
- análise de correspondências entre UML e Java;
- melhorar a qualidade da produção de sistemas computacionais;
- crescimento e aprimoramento de ferramentas CASE; e
- apresentação de ilustrações de aplicações do mapeamento UML – Java.

1.4 ESCOPO DO TRABALHO

Numa primeira fase será elaborado o processo de mapeamento para modelos que possuam as seguintes características:

- classes (métodos e atributos);
- relacionamentos; e
- seqüência de mensagens trocadas entre os objetos.

Não fazendo parte deste trabalho:

- a criação de mecanismos de tradução para aplicações que necessitem da utilização dos seguintes pacotes e classes Java: sql, security, rmi e net;
- o desenvolvimento do processo para diagramas de organização física de componentes; e
- desenvolvimento de uma ferramenta CASE completa.

Num segundo momento, como trabalho futuro, seria interessante a implementação de uma ferramenta CASE. Numa primeira análise, deve-se observar os requisitos e funcionalidades abaixo relacionadas:

- análise e escolha da metodologia de desenvolvimento;
- escolha do ambiente de programação;
- desenvolvimento de um editor de modelos e visões;
- escolha de linguagens alvo para mapeamento;
- análise e criação de tradutor(es) para as linguagens alvo de mapeamento;
- implementação de analisadores ou certificadores de modelos de acordo com as características das linguagens alvo escolhidas;
- implementação do processo de mapeamento;
- a equipe deve utilizar o desenvolvimento distribuído, o que traz vantagens econômicas de custo e de tempo;
- integração dos módulos; e
- testes e validação com os requisitos da ferramenta.

Como sugestão, para o desenvolvimento dessas etapas, adotar-se-ia o modelo em espiral **[PRE95]**. Devido as suas características dinâmicas de processo onde os riscos de insucesso são minimizados.

1.5 TRABALHOS RELACIONADOS

Analisando a bibliografia e *sites* na Internet, observa-se um número pequeno de trabalhos desta magnitude.

Pode-se citar algumas referências a artigos que tratam superficialmente do processo de engenharia de modelos para uma determinada linguagem de programação, tornando complicado a referência em relação a correspondências específicas UML para Java.

No entanto, em março deste ano, em Londres – Inglaterra, na SIGS – *Conference for Java Development*, apresentou-se um trabalho de mapeamento de UML para Java. O autor deste trabalho foi John Hunt e sinteticamente pode ser resumido com as seguintes palavras: considerações entre UML e Java, em particular a seguinte questão: "UML é uma linguagem independente?" e qual o seu envolvimento na conversão de projetos UML para código Java? o trabalho aborda características de mapeamento para Java, concentrando em particularidades em que não há suporte direto para Java.

Em nível de Brasil, procurando através da Internet, encontram-se referências sobre trabalhos de graduação sobre UML ou sobre Java. Não havendo, na Internet, referências de estudos de mapeamento entre UML e Java.

1.6 VISÃO HISTÓRICA DA PESQUISA

Nos primórdios da informática, na década de 50, programas de computador eram desenvolvidos de forma "ad hoc", ou seja, eram construídos ou criados de forma personalizada, geralmente, utilizando o intelecto de uma única pessoa. Estes sistemas desrespeitavam cronogramas, orçamentos e não havia a aplicação de conceitos de projeto formal, o que levava a sistemas de difícil manutenção e ampliação.

Por volta da década de 60, houve uma grande mudança na "filosofia" de desenvolvimento de software. O desenvolvimento "ad hoc" deu passagem a um método chamado de "cascata" ou ciclo tradicional de desenvolvimento de sistemas, pois era composto de várias fases e a passagem para uma fase seguinte dependia da conclusão da fase anterior **[PRE95]**. O marco de conclusão de uma fase era verificado após a criação de uma série de documentos que espelhavam o trabalho daquela fase. Contudo, apesar da adoção da abordagem mais formal, sistemas de software grandes e complexos ainda continuavam estourando cronogramas e aumentando os orçamentos.

A década de 70 foi marcada por outra mudança na "filosofia" no desenvolvimento de software. Tom DeMarco, em seu pioneiro livro *Structured Analysis and System Specification*, apresentou o conceito de *Engenharia de Software Baseada em Modelos*. A idéia, deste trabalho, era de percepção do funcionamento do sistema antes dele ser codificado, o que economizava esforços no momento da mesma, pois muitos dos erros conceituais eram já detectados no momento da construção do modelo.

A partir de então todos os modelos de engenharia de software seguiram esta corrente de pensamento, diferenciando apenas em aspectos de como devem ser construídos e quem deve construí-los.

No sentido de acompanhar a evolução natural do conhecimento, houve um aprimoramento constante das técnicas e metodologias para a engenharia de produção de software, incluindo a análise e o projeto estruturado, a decomposição funcional, a análise essencial, engenharia da informação, engenharia reversa, os métodos orientados a objetos, a análise e projeto orientada a objetos, entre outros.

Pode-se citar, com maior ênfase, os métodos orientados a objetos que surgiram em meados da década de 70 e início dos anos 80 e atualmente são inúmeros os existentes, o que traz, para muitos usuários, uma insatisfação e indecisão em relação aos enfoques disponíveis.

E, como complemento, cita-se, ainda três outros aspectos de mudanças ocorridas nos anos noventa que, de uma forma direta ou indireta são causas importantes de mudanças no enfoque, modelo e metodologia de construção de sistemas computacionais. São eles:

- os conceitos básicos do enfoque de orientação a objetos já estão maduros e a atenção mudou gradualmente da consideração em codificação para considerações sobre análise e projeto – Engenharia de Sistemas;
- anteriormente as nossas de idéias de concepção de sistemas eram influenciadas ou pré-concebidas a partir da linguagem de programação que estava disponível no mercado. Hoje, a tecnologia para o desenvolvimento de sistemas tornou-se mais poderosa no sentido de tornar o trabalho de desenvolvimento de sistemas focado no: **o que deve ser realizado**; e
- os sistemas que elaboramos hoje são diferentes, do que eram há alguns anos atrás, em todos os aspectos: interface homem-máquina, plataforma, segurança, complexidade, entre outros.

Surge então, na década de 90, com o objetivo de unificar e padronizar, a modelagem da informação, a *Unified Modeling Language* – UML, uma linguagem de modelagem de sistemas e a linguagem de programação Java com características necessárias para a produção de sistemas computacionais que atendam aos requisitos e padrões atuais de mercado **[BOO00]**.

Tendo em vista as constantes mudanças ocorridas nas décadas passadas e a partir do conhecimento e amadurecimento de novas tecnologias para o desenvolvimento de sistemas, vários desenvolvedores e pesquisadores foram influenciados e impulsionados a construírem ferramentas CASE de geração de código automática a partir de modelos, fato que pode ser observado no Anexo 1 – Comparação entre Ferramentas CASE.

No entanto, referências ou pesquisas que relatem e apresentem a forma ou método pelo qual se realiza correspondências entre modelos de representação para uma

determinada linguagem de programação de computadores, são mínimos e bastante restritos, pois, eles, representam e revelam o ponto crucial de qualquer mapeamento codificado numa ferramenta CASE de geração automática de código.

De fato, pesquisar, relatar, propor, validar e apresentar um processo de correspondência entre modelos de representação de sistemas e uma linguagem de programação, não é uma tarefa trivial e direta, mas concebível.

A idéia, num primeiro momento, consiste em apresentar uma proposta de mapeamento de modelos expressos em UML para a linguagem Java. Num segundo momento, como continuação do trabalho, sugere-se a construção de uma ferramenta CASE.

CAPÍTULO 2. LINGUAGEM JAVA

2.1 APRESENTAÇÃO

Java é uma linguagem de programação de computadores de alto nível de propósitos gerais orientada a objetos, desenvolvida, por volta dos anos de 1990, pela SUN *MicroSystem* e voltada a aplicações multimídia para internet. O nome Java é devido a uma Ilha situada na capital Jakarta na Indonésia.

A primeira versão da linguagem foi denominada Oak cujo o propósito era o desenvolvimento de pequenos aplicativos para controle de eletrodomésticos.

A proposta da *Sun* era o desenvolvimento de um sistema operacional que controlasse uma rede de eletrodomésticos. Com isso, seria possível, através de um computador, controlar uma residência, por exemplo.

Devido à popularização da Internet através da *World Wide Web*, a *Sun* decidiu postergar a idéia de controle de equipamentos domésticos de residências e direcionou a linguagem Oak para o desenvolvimento para a Internet, dando origem à linguagem Java.

O primeiro conjunto de ferramentas disponíveis foi ofertado pela empresa criadora do Java. Este conjunto foi composto por um Kit de desenvolvimento chamado de JDK (*Java Developer Kit*) que contém um interpretador Java, um *debbuger*, um visualizador de applets (*appletviewer*), bibliotecas e ajudas.

Grande parte das aplicações em Java são executadas sobre um outro programa (em geral um *browser*). O primeiro destes programas também foi lançado pela Sun, chamando-se *HotJava*, sendo um *browser* que consegue executar programas Java.

Atualmente temos algumas ferramentas visuais para desenvolvimento e auxílio ao desenvolvedor, entre elas pode-se citar o Café e o VisualCafe da Symantec, Visual J++ (da Microsoft) e Java WorkShop (da SunSoft).

Java, por possuir características da tecnologia de orientação a objetos, oferece um grande número de bibliotecas e *frameworks* de classes disponíveis gratuitamente para uso.

Java esta sendo amplamente utilizada na comunidade acadêmica e vem apresentando sinais de crescimento e conquista de mercado no meio comercial de desenvolvimento de software [SUN95].

2.2 CARACTERÍSTICAS

O desenvolvimento de programas é uma tarefa árdua, artesanal (em grande parte) e complexa. Os obstáculos que surgem no caminho do desenvolvedor de software são de naturezas diversas. Pode-se citar, como dificuldades:

- as particularidades que caracterizam cada área de desenvolvimento dos aplicativos atuais;
- incompatibilidade com uma arquitetura de hardware; e
- incompatibilidade com um determinado sistema operacional ou interface gráfica com usuário.

Além disso, não é difícil imaginar e de citar outros obstáculos que os programadores atuais enfrentam para poderem atender aos requisitos de um sistema de computação. Entre eles estão:

- Atendimento aos requisitos de validação do software (desempenho e grau de satisfação do usuário);

- utilização de ferramentas que proporcionem tornar zero o grau de ambigüidade nas relações usuário-desenvolvedor (diagramas, técnicas de entrevistas, protótipos);
- entendimento e aplicação de todos os recursos de uma linguagem de programação (manipulação da memória, especificidades e conceitos das linguagens);
- a utilização de técnicas que busquem diminuir o trabalho artesanal de programação;
- aquisições de bibliotecas de terceiros cuja probabilidade de compatibilidade com novas versões de hardware ou software é complexa (distribuição de software);
- a questão da portabilidade, segurança e robustez das linguagens de programação (software-produto gerado).

Para resolver vários dos problemas listados acima, surge a linguagem Java, que possui inúmeras características que atuam diretamente nos pontos que já citou-se. Java é uma linguagem de programação baseada num conjunto de habilidades que tornam a fase de codificação facilitada, pois ela é:

- Simples
- Orientada a Objetos
- Distribuída
- Interpretada
- Robusta
- Segura
- Arquitetura Neutra
- Portável
- Alta Performance
- Encadeamentos Múltiplos
- Dinâmica

A cada uma das habilidades foi reservado um tópico especial, neste texto, para uma análise mais detalhada, sendo utilizada a referência do *site* da *Sun Microsystem* [SUN95] como base e as referências: [NEW97], [DEI98] e [HOP97] para complemento do registro dos itens de 2.2.1 a 2.2.11.

2.2.1 Simples

A Linguagem Java foi projetada o mais parecido possível com a linguagem C e C++. Isso foi feito para tornar o sistema mais compreensível, e para encurtar o tempo necessário para aprender a nova linguagem, pois a maioria dos programadores atuais já possuem conhecimento nas linguagens C ou C++.

Java é simples e familiar, pois em muitos aspectos é similar ao C e ao C++ ; simples pelo fato de deixar de lado todos os artefatos históricos nunca utilizados e pouco entendidos dos recursos de C e C++.

À Linguagem Java, foi adicionado o conceito de *garbage collection* que simplifica a tarefa da programação em Java quando se trata da manipulação de memória pelo programador. Isto torna a linguagem menos complicado e mais eficiente no sentido de reduzir o número de problemas com manipulação de memória.

Outro aspecto que deve ser ressaltado é a habilidade de construir software que possa rodar em pequenas máquinas *standalone*. O tamanho do interpretador básico e o suporte a classes é aproximadamente de tamanho igual a 40 Kbytes; adicionando as bibliotecas básicas e o suporte a encadeamento são gastos mais 175 Kbytes.

2.2.2 Orientada a Objetos

Java é uma linguagem orientada a objetos. Isso significa que ela faz parte de uma família de linguagens que se concentram em definir os sistemas como coleções de classes/objetos, e nos métodos/operações que podem ser aplicados a esses objetos.

Ainda, Java possui todos os recursos de orientação a objetos presentes nas modernas linguagens de programação por isso ela se constitui numa ferramenta que proporciona uma melhora considerável na produtividade e confiabilidade no desenvolvimento na solução de problemas complexos através do computador.

Ao longo dos últimos anos a tecnologia de programação orientada a objetos foi considerada como sendo a melhor forma de solucionar problemas complexos. A abstração, o encapsulamento e a modularidade são mecanismos fundamentais na resolução de problemas através de modelos que se transformarão em projetos. O suporte a herança e o acoplamento dinâmico são aspectos de uma linguagem orientada a objetos que aumentam a produtividade e reusabilidade de objetos. Do ponto de vista do produto e da sobrevivência podemos dizer que estes fatores influem diretamente na competitividade no desenvolvimento de um produto (software).

2.2.3 Interpretada

Java é tanto Interpretada como compilada. Em primeiro lugar o compilador Java traduz o código Java em uma linguagem intermediária chamada de Java *bytecodes*. Com um interpretador, cada instrução *bytecode* é executada numa máquina. A compilação apenas acontece uma vez; a interpretação ocorre todo tempo em que o programa é executado. A Fig. 1 ilustra este processo.

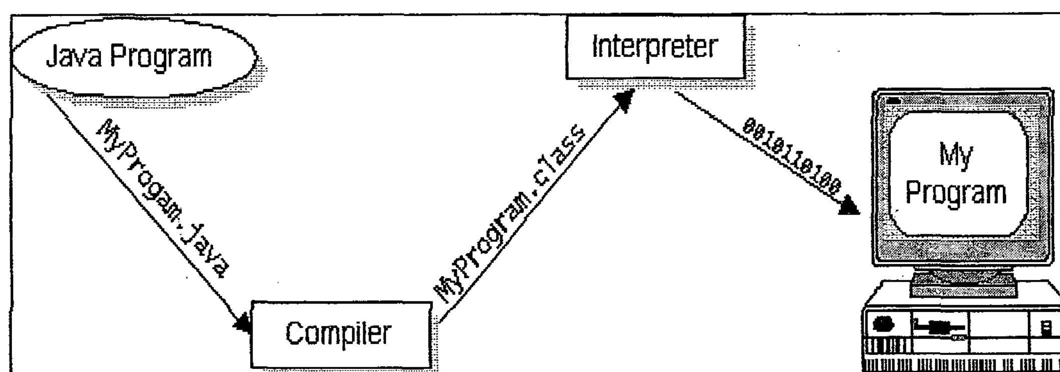


Figura 1 - Representação da geração de bytecodes em Java [SUN95]

Os *bytecodes* gerados pelo compilador se baseiam na especificação de uma *Java Virtual Machine* (JVM - Máquina Virtual do Java) que é uma máquina implementada em software. A máquina virtual é semelhante a uma Unidade Central de Processamento de Dados real, com seu próprio conjunto de instruções, seu formato de armazenamento e seus registradores. Contudo, por ser escrita em software (podendo ser também escrita em hardware), ela é portável. Tudo que é necessário fazer para o código Java ser compilado em uma plataforma e executada em outra é fornecer um interpretador e um ambiente *runtime* do Java.

Com o Java *bytecodes* é possível que um aplicativo seja executado em qualquer máquina. Pode-se compilar um programa para *bytecodes* em qualquer plataforma que tenha um compilador Java. Os *bytecodes* podem ser executados em qualquer implementação do Java VM. Por exemplo, o mesmo programa em Java pode rodar em Windows NT, Solaris e Macintosh. Em [SUN95] pode-se observar o processo de compilação/interpretação Java.

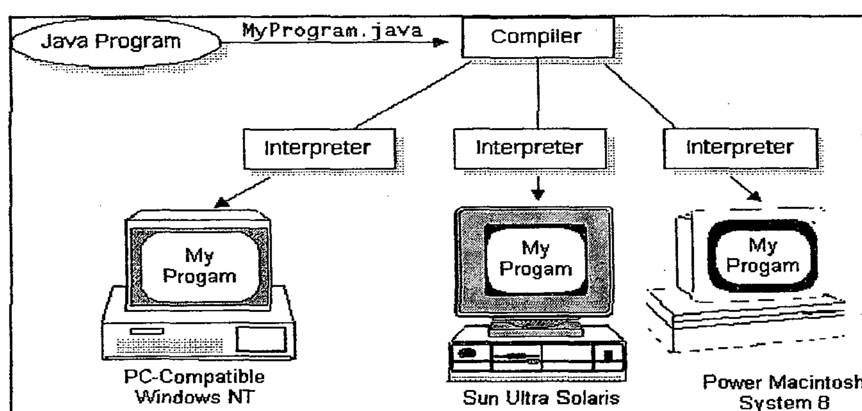


Figura 2 - Representação do Processo de Compilação/Interpretação Java [SUN95]

Usar o Java significa que apenas uma fonte de código Java precisa ser mantida para que o código de bytes possa rodar em diversas plataformas.

Esta combinação de compilação e interpretação é uma característica positiva, pois facilita a segurança e estabilidade. O ambiente Java contém um elemento denominado *linker*, que verifica a entrada de dados na sua máquina para assegurar que

ela não contém nenhum arquivo intencionalmente prejudicial (segurança) nem arquivos que possam danificar o funcionamento de seu computador (consistência/robustez). O mais importante nesta combinação de compilação e interpretação é a redução da preocupação com a gerência e incompatibilidade de versões.

Outro ponto positivo do interpretador Java é o fato de melhorar o desenvolvimento de software, pela eliminação da fase de linkedição do processo de desenvolvimento. É possível passar direto da compilação para execução, pois a linkedição ocorre durante a execução do aplicativo.

O fato de que a parte final da compilação é realizada por um dispositivo específico de plataforma, que é mantido pelo usuário final, libera o desenvolvedor da responsabilidade de manter várias fontes para diversas plataformas. A interpretação também permite que os dados sejam incorporados em *runtime*, que é fundamento do comportamento dinâmico do Java.

2.2.4 Distribuída

Java possui um biblioteca de rotinas extensivas para lidar com os protocolos TCP/IP, HTTP e FTP. Uma das grandes vantagens das *applets* (aplicativos pequenos para internet) e aplicativos em Java é que eles podem abrir e acessar objetos dentro da Web através de URLs com a mesma facilidade com que podem acessar um sistema de arquivo local.

Uma das características do sistema distribuído é que vários projetistas, em diversos locais, podem colaborar em um projeto comum. Por exemplo, usando o Java, pode-se criar um aplicativo, orientado a objetos, construtor de mecanismo de sinergia (em sites locais ou remotos). Usando o construtor de mecanismo de sinergia orientado a objetos, os colaboradores podem trabalhar juntos para desenvolver um mecanismo melhor (mais rápido e econômico).

2.2.5 Robusta

Quanto mais robusta (consistente) for a linguagem, menos provável será que os programas "travem". A linguagem Java, por ser fortemente tipada, permite uma ampla verificação em tempo de compilação, o que significa que qualquer problema (bug) pode ser encontrado sem muito esforço e logo, pois a maior parte da verificação de tipos é realizada em tempo de compilação.

Um programa Java não pode travar um sistema porque não possui permissão para acessar qualquer parte da memória do computador. Os programas escritos em outras linguagens tradicionalmente podem acessar, e portanto, podem modificar qualquer parte da memória do seu sistema, mas o Java tem uma limitação essencial. Os programas em Java são capazes de acessar apenas uma área restrita de memória, assim eles não podem alterar um valor que não deva ser mudado.

Como uma verificação final de segurança, o Java tem o linker. O linker é parte do ambiente de *runtime*. Ele entende o sistema de tipo e, durante a execução, repete muitas das verificações de tipo feitas pelo compilador para evitar problemas de incompatibilidade de versões.

2.2.6 Segura

A presença do *garbage collection* permite eliminar erros mais comuns que os programadores cometem quando são obrigados a gerenciar diretamente a memória (C, C++, Pascal). A eliminação do uso de ponteiros, em favor do uso de vetores, objetos e outras substitutivas trás benefícios em termos de segurança.

A característica de segurança em Java pode ser explicada através de um modelo de segurança de várias camadas.

A primeira refere-se à construção do compilador que impede a geração de *bytecodes* que não estão em conformidade com as regras de segurança.

A segunda refere-se a um verificador de *bytecodes* que inspeciona os *bytecodes* da classe, à medida que eles são carregados no sistema.

A terceira camada é responsável pela carga através de um *namespace runtime*. Este procedimento é realizado por um módulo chamado de *class loader* representado por uma classe de mesmo nome.

A camada final de segurança realiza o teste de código à medida em que ele (aplicativo) está sendo executado. O módulo responsável pela verificação de violação de qualquer restrição de segurança definida para o ambiente em uso, é chamado *de Security Manager* que corresponde a uma classe Java de nome similar. O funcionamento do *security manager* pode variar de acordo com o navegador em uso. Existem navegadores extremamente conservadores com a capacidade de empregar o Java, como o *Netscape Navigator*, que impedem por completo que as *applets* sejam lidos ou gravados no seu sistema de arquivos local. Por outro lado, o navegador *HotJava* pode ser configurado para autorizar operações sobre arquivos de maneira flexível.

A Fig. 3 apresenta uma visão geral do ciclo de vida do código do Java, à medida que ele passa pelas camadas de segurança.

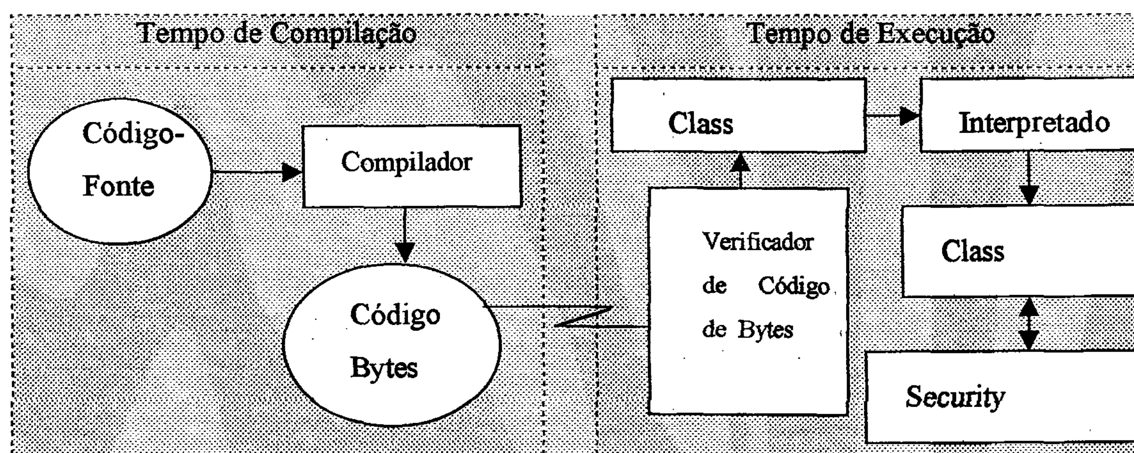


Figura 3 - O Ciclo de Vida do Java e o Relacionamento com suas Camadas de Segurança[HOP97]

2.2.7 Arquitetura Neutra

Java foi projetada para suportar aplicações em rede. Em geral, redes de computadores são compostas de uma variedade de sistemas com uma variedade de CPU's e arquiteturas de sistemas operacionais. Para que uma aplicação Java possa ser executada em qualquer componente da rede, o compilador gera um código que é executado devido à presença de um sistema Java de *runtime*.

Isto é possível não somente para redes de computadores, mas para computadores *standalone*. Para os computadores pessoais, as aplicações são escritas tendo que produzir várias versões da mesma aplicação que devem ser compatíveis com computador pessoal padrão IBM e com Apple Macintosh. Ainda eles devem levar em conta a diversificação de arquiteturas de CPU, isto faz com que a produção de software possa ser executada em qualquer arquitetura. Com Java, a mesma versão da aplicação executa/roda em todas as plataformas.

O compilador Java faz isto através da geração de instruções de *bytecodes* que não têm nada a ver com uma arquitetura específica de um computador. Alias, eles são projetados tanto para serem interpretados em qualquer máquina e facilmente transcritos ou traduzidos para código nativo de uma máquina a qualquer momento.

2.2.8 Portável

A portabilidade é um ponto crítico para o sucesso de um software. Por exemplo, na Internet, você não pode fazer suposições a respeito da arquitetura em que o seu software será executado. Nesta visão, você terá que desenvolver o seu produto na plataforma básica do seu cliente, seja ela Intel, PowerPC ou outra arquitetura, assegurando também que ele será apresentado segundo as características locais da interface nativa.

O Java resolve este problema com a abordagem de que o compilador Java gera *bytecodes* que são interpretados durante a execução do aplicativo. O fato da geração de *bytecodes* é importante pois não caracteriza dependência em relação a uma determinada plataforma. Por exemplo, até a implementação de um tipo inteiro é realizado de forma diferente em plataformas distintas (16 bits em algumas e 32 bits em outras). O Java define o seu próprio conjunto de tipos primitivos e realiza a construção de valores no momento da execução a partir de *bytecodes* gerados.

2.2.9 Alta Performance

O Java não tem um grande desempenho no sentido de ser o "mais rápido do que uma rotina em C++ comparável". De fato, é quase da mesma velocidade, segundo testes realizados pela Sun Microsystems.

O desempenho de código de bytes interpretados é adequado para a maioria das tarefas comuns, pois as verificações de tipos, registradores e outros que se fizerem necessários são realizados em tempo de geração dos *bytecodes*, tornando a aplicação mais "leve" quando ela for interpretada pela máquina final.

2.2.10 Encadeamentos Múltiplos

Um dos recursos fundamentais que o Java oferece para melhorar o desempenho é o encadeamento múltiplo (multithreading). A maioria dos aplicativos interativos se caracteriza por longos períodos de tempo durante os quais o usuário faz pausas entre as ações para decidir o que fazer em seguida. Nos ambientes tradicionais de um único *thread* (linhas de execução), o aplicativo pode permanecer ocioso durante períodos. Contudo, nos ambientes *multithreaded*, o aplicativo pode executar outras tarefas durante os períodos de ociosidade ou em qualquer outro momento. Essa característica é crítica para os aplicativos de rede, que podem utilizar longos períodos de tempo para fazerem o carregamento de arquivos.

Pode-se ainda utilizar o ambiente *multithreading* simples do Java para executar diversos tipos de ações de otimização no seu programa. Além disso, o Java emprega *threads* para aperfeiçoar seu próprio desempenho. O *garbage colletion* que cuida das tarefas de gerenciamento de memória é executado em segundo plano como um *thread* de baixa prioridade.

2.2.11 Dinâmica

O Java é uma linguagem mais dinâmica do que o C ou C++. Ao contrário de qualquer uma dessas linguagens, Java foi projetado para se adaptar a ambientes em evolução.

Um dos principais problemas no desenvolvimento de alguma aplicação que usa C++ é a dependência em relação a uma biblioteca de classes de terceiros. Qualquer atualização nesta biblioteca pode trazer como trabalho adicional ou a re-compilação e re-distribuição de seu próprio software.

Java, por outro lado, possui recursos através do paradigma de objetos que resolvem o problema de biblioteca de terceiros. Além disso, Java retarda a vinculação de módulos, as bibliotecas podem ser atualizadas e modificadas sempre que necessário. Os métodos e variáveis de instância podem ser acrescentados ou excluídos sem nenhum impacto negativo sobre os clientes das bibliotecas.

Assim como Lisp e Smalltalk, o Java é uma linguagem dinâmica pois possibilita a criação de protótipos devido a sua característica de flexibilidade.

2.3 CONCEITOS DE OO NO CONTEXTO DE JAVA

2.3.1 Introdução às Classes e aos Objetos

O desenvolvimento de aplicações, utilizando a linguagem de programação Java, prevê a necessidade do entendimento do modelo que fundamenta qualquer aplicação nessa linguagem. Esse modelo está baseado na criação de classes, ou seja, são as classes que fornecem a especificação do estado, do funcionamento e do comportamento de um objeto durante a sua execução. Ela por sua vez é a definição ou abstração de um ou mais objetos, que são reais.

É importante frisar que um objeto pode ter valores diferentes nos seus estados, mas, no entanto, possui características-atributos e serviços-métodos iguais.

2.3.2 Encapsulamento e Ocultamento de dados

Em Java não se pode dispensar a utilização de classes, pois uma característica importante da definição e utilização de classes é a capacidade de encapsular a sua funcionalidade.

A indivisibilidade de dados e procedimentos/funções permite que somente os métodos daquela classe possam alterar ou ter acesso aos atributos daquela mesma classe. Desta forma, ao contrário da tecnologia de programação estruturada, os dados estão protegidos sendo somente acessados por procedimentos do próprio objeto [AMB97].

As variáveis de classe, chamadas de atributos, compõem o "núcleo" do objeto. Os métodos, que "cercam" o objeto, implementam serviços e são utilizados para mudarem o estado do objeto. A proteção dos atributos de um objeto é chamado de encapsulamento.

O encapsulamento é usado quando se quer esconder detalhes de implementação não importantes para outros objetos [AMB97].

O uso do conceito de encapsulamento trás um benefício bastante considerável, pois existe a possibilidade de alteração da implementação de um método sem afetar a aplicação [AMB97].

Pode-se, por exemplo, alterar o mecanismo de troca de canais de uma televisão sem mudar a interface, o controle remoto. Esta características possibilita, quando necessário, alteração da implementação dos métodos sem a preocupação de modificação na aplicação.

Na definição da classe, no núcleo do objeto, temos os atributos, definidores de estado de um objeto, protegidos do acesso de outros objetos pelos métodos. Assim percebe-se que, a maneira mais coerente de alterarmos o estado de um objeto é seguindo a interface disponível.

O esquema clássico de representação de objetos (célula), diz respeito ao conceito de ocultamento de dados. Isto pode ser feito em Java, através da inserção de especificadores de acesso.

Através da inserção de modificadores tem-se a possibilidade de "esconder" os atributos de outros objetos. Isto dá a possibilidade de acessar objetos sem o receio de modificação ou interferência em outros objetos e, a certeza de que, o acesso é realizado somente pela interface disponível.

No que diz respeito à sintaxe existem duas partes a serem consideradas na definição de classes em Java [NEW97]:

a) a declaração; e
b) o corpo.

Por exemplo, a seguir, apresenta-se dois exemplos de codificação de classes em Java.

```
public class Exemplo {
        int variavel1 = 0;                          // atributo inteiro definido inicialmente
como 0

        public void setValor(int novoValor) {       // método atualiza valor da variável
variavel1
                variavel1 = novoValor;
        }

        public int getValor( ) {                    // método retoma valor atual da
variavel1
                return variavel1;
        }

}
```

Figura 4 – Exemplo de definição da classe-Exemplo em Java

A Fig. 4 mostra a estrutura interna da classe Exemplo. Ela está composta pelo atributo variavel1 que está sendo inicializada com o valor zero. Já no corpo da classe Exemplo, tem-se os métodos setValor, que atualiza o atributo variável1 e o método getValor que retoma o valor do atributo variavel1.

```
public class Funcionario {
        String nome;
        String cpf;
        private double salario;
        Funcionario (String x, String y, double s) {
                nome = x;
                cpf = y;
                salario = s;
        }
        double dobraSalario( )          {
                return (salario*2);
        }
}
```

Figura 5 - Exemplo ilustrativo da classe funcionário em Java

No exemplo acima, é importante citar dois pontos. O primeiro diz respeito ao método funcionário, que é um tipo de método especial, que inicializa o objeto. O segundo é a inserção do modificador de acesso private antes da declaração do atributo salário. Ele permite o acesso ao atributo salario, de métodos pertencentes à classe atual, não permitindo acesso de métodos de outras classes e nem de sub-classes, da classe atual.

Como foi visto acima, a declaração de uma classe é bastante simples, no entanto, existem quatro propriedades da classe que podem ser definidas na declaração:

- Modificadores
- Super Classes que faz referência ao nome da classe geradora
- Interfaces

2.3.2.1 Modificadores

Na criação de classes pode-se aplicar um dos seguintes modificadores:

- public
- final
- abstract

Por padrão, o nível de acesso "friendly" é atribuído a todas as classes. Isso quer dizer que, embora a classe possa ser estendida e usada por outras, apenas os objetos que estiverem no mesmo pacote poderão usar essa classe. Já o modificador *public* quer dizer que uma classe pode ser usada ou estendida por qualquer objeto, independentemente do pacote em que esteja.

O uso do modificador *final* antes da palavra *class* além de dar terminalidade para a classe, é importante no sentido de padronizar, pois há aplicações onde fica evidente a

não criação de classes que tratem funções de maneira diferentes, assegurando, desta forma, a consistência.

Quando existir a necessidade de definir um atributo constante, utiliza-se o modificador final antes do identificador de atributo. Este modificador ainda pode ser utilizado quando deseja-se que métodos não sejam sobrepostos por outras classes. A seguir tem-se um exemplo de utilização do modificador final.

```
...
public final esteMetodoNaoPodeSerSobreposto(int y, int x, String n);
...
```

Figura 6 – Fragmento de código Java utilizando o modificador final

O modificador *abstract* é usado quando pelo menos um método não é concluído, ou seja, quando a conclusão da classe seja realizada depois, através da redefinição de método(s) com o código real. A Fig. 7 ilustra a utilização do modificador *abstract*.

```
// arquivo fonte Objeto.java
abstract class Objeto {
        static int x=2; // variável de classe
        int y;          // variável de instância

        void trataValor (int y) {
                this.y = y;
        }
        abstract void apresentar( );
}

// arquivo fonte Reta.java
final class Reta extends Objeto {
        void apresenta( ) {
                System.out.println(" ... teste modificador abstract ... ");
        }
        public static void main(String args[]) {
                Reta r = new Reta( );
                r.apresenta( );
        }
}
```

Figura 7 - Código Java utilizando o modificador abstract

Na Fig. 7, observa-se que o método apresentar é abstrato e por isso deve ser definido em uma outra classe não abstrata. A codificação de métodos abstratos é realizado em classes mais específicas possibilitando a criação de templates em OO.

Ainda é importante ressaltar que, classes abstratas são classes não concluídas, pois seus métodos não estão implementados. Isto trás como conseqüência a impossibilidade de criação de instâncias de classes desse tipo.

2.3.2.2 Interface

Utiliza-se Interfaces quando se deseja proporcionar uma grande flexibilidade à aplicação. Esta flexibilidade pode ser entendida na perspectiva da utilização de herança múltipla, onde uma determinada classe é composta pela junção de várias classes diferentes.

A utilização de Interfaces possibilita a geração de classes a partir de outras classes através da declaração de métodos ou constantes que, por sua vez, são declaradas como *abstract*.

A inserção do modificador *abstract* é necessário para permitir que a nova classe possa implementar e definir constantes de acordo com a sua especificidade. Desta forma, melhorando a sua funcionalidade.

A utilização de Interfaces pode ser entendida através da Fig. 8:

```
// Arquivo fonte Conjunto.java
public interface Conjunto1 {
        public static final String mes1 = "janeiro";
}

// Arquivo fonte Conjunto2.java
public interface Conjunto2 {
        float y=1;
        public void apresenta( );
}

// Arquivo fonte Iexe.java
public class Iexe implements Conjunto1, Conjunto2 {

        public void  apresenta ( ); {  // implementação real
                System.out.println(".... teste interface ...");
        }
        public static void main(String args[]) {
        Iexe teste = new Iexe( );

                teste.apresenta( );
        System.out.println("primeiro mês "+ mes1+"... variável ..."+y);
        }
}
```

Figura 8 - Exemplo de utilização de Interfaces

2.3.2.3 Criação de uma Instância de Objeto

Após a definição de uma classe, cria-se ou instancia-se objeto(s). Para gerar uma instância, ou seja, uma ocorrência de um objeto, utiliza-se, em Java, o operador new.

Considerando a classe Exemplo, já devidamente definida em termos de atributos e métodos, instanciamos um objeto da classe Exemplo conforme é mostrado na Fig. 9, a seguir:

```
...
Exemplo primeiroObjeto = new Exemplo( );

...
```

Figura 9 – Fragmento de código Java de criação de objeto da classe Exemplo

O operador *new* executa duas tarefas. Primeiro aloca memória para o objeto e segundo chama um construtor da classe Exemplo. Um construtor é um método invocado quando um objeto é *instanciado*.

2.3.3 Métodos

Um método é composto de 2 partes: um corpo e uma definição. A definição por sua vez é composta por outros 3 componentes: um nome, uma lista de parâmetros e um valor de retorno. Estes três últimos componentes definem a assinatura de um método.

A seguir, é apresentado um exemplo de declaração de classe com seus métodos.

```
public class Exemplo {
        int variavel1 = 0;

        public void setValor(int novoValor) //  método  atualiza  valor  da  variável
variavel1
        {
                variavel1 = novoValor;
        }
        public int getValor()                   // método retorna valor atual da variavel1.
        {
                return variavel1;
        }
}
```

Figura 10 - Exemplo de codificação de métodos em Java

Na Fig. 10, temos a representação do valor de retorno de um método, a lista de parâmetros de um método (podendo ser vazia) e o nome do método. Características que determinam a assinatura de um método.

2.3.3.1 Sobrecarga de Métodos (Overloading)

A linguagem Java oferece um mecanismo de utilização de assinatura de um mesmo método, esse mecanismo é chamado de sobrecarga de métodos. A sobrecarga é definida a partir do nome do método e não implica em mesma assinatura de método **[HOP97]**. O mecanismo pode ser exemplificado através do exemplo abaixo.

```
// Arquivo fonte Sobre.java
public class Sobre {
        int variavel1 = 1;
        int variavel2 = 2;
        public void ajustaValor(int var1)  {
                variavel1 = var1;
        }
        public void ajustaValor(int var1, int var2)  {
                variavel1 = var1;
                variavel2 = var2;
        }
        public static void main(String args[]) {
        Sobre x = new Sobre( );
        x.ajustaValor(10);
        x.ajustaValor(1,1);
        System.out.println(x.variavel1);
        System.out.println(x.variavel2);
        }
}
```

Figura 11 - Exemplo de sobrecarga em Java

O método ajustaValor com parâmetro var1 somente atualiza o valor da variável de instância variavel1, já o método cuja assinatura é ajustaValor atualiza o valor de ambas as variáveis de instância, ou seja de variável1 e variável2.

Em Java existe um tipo especial de método cujo nome do método é o nome da classe. Esse método é chamado de Construtor. Ele é chamado automaticamente quando um objeto é instanciado. Um Construtor também pode sofrer sobrecarga de método.

Os Construtores quando não existem na sua classe de origem, são "procurados", através do mecanismo de herança do Java, na super classe imediata. Esta busca é continuada até o que se encontre o construtor procurado ou que se chegue a classe Object que é a super classe de todas as classes.

2.3.3.2 Métodos Estáticos

Quando são definidas instâncias de variáveis e/ou também métodos, estamos na verdade, criando para cada objeto uma cópia tanto dos métodos como das variáveis, por isso chamamos de variáveis e métodos de instância. O mecanismo de instância provê a característica da independência de dados, pois para cada objeto tem-se uma cópia de instância seja ela de variável ou de método. Desta forma pode-se modificar os atributos de um determinado objeto da mesma classe sem a preocupação da alteração nas variáveis de instância de outro objeto em execução.

Por outro lado, quando houver a necessidade de não realizar mais cópias de instância de variáveis e métodos podemos definir métodos e variáveis estáticas através da inclusão da palavra static antes da declaração de ambos. Isto implica em uma única cópia de método ou variável, sendo chamado de métodos de classe ou variáveis de classe pois a alteração dos mesmos afeta diretamente a classe. Eles são aplicados quando existe a necessidade de criamos uma cópia do método ou variável para mantermos um determinado estado ou comportamento à classe; embora possam haver várias instâncias da classe. A economia de memória é uma conseqüência intrínseca a definição de variáveis ou métodos de classe.

Ao declarar um método de uma classe como static, não será necessário a criação de uma instância para execução daquele método, sendo chamados de métodos de classe. Os aplicativos utilizam um método static em casos que não precisarão criar um objeto para chamar um método.

A Fig. 12 mostra a utilização de métodos *static*.

```
public class Teste {
        static int x;
        public static void main(String args[]) {
                System.out.println("... teste Java ... ");
        }
}
```

Figura 12 - Exemplo de código Java para membros de classe (variáveis e métodos)

2.3.4 Mensagens

Os objetos interagem e se comunicam com outros objetos através de mensagens que são enviadas de um objeto para outro. Por exemplo, quando um objeto A quer que um objeto B execute algum de seus métodos, o objeto A envia uma mensagem para o objeto B.

A seguir, pode-se observar um esquema que representa o processo de solicitação de mensagem entre objetos.

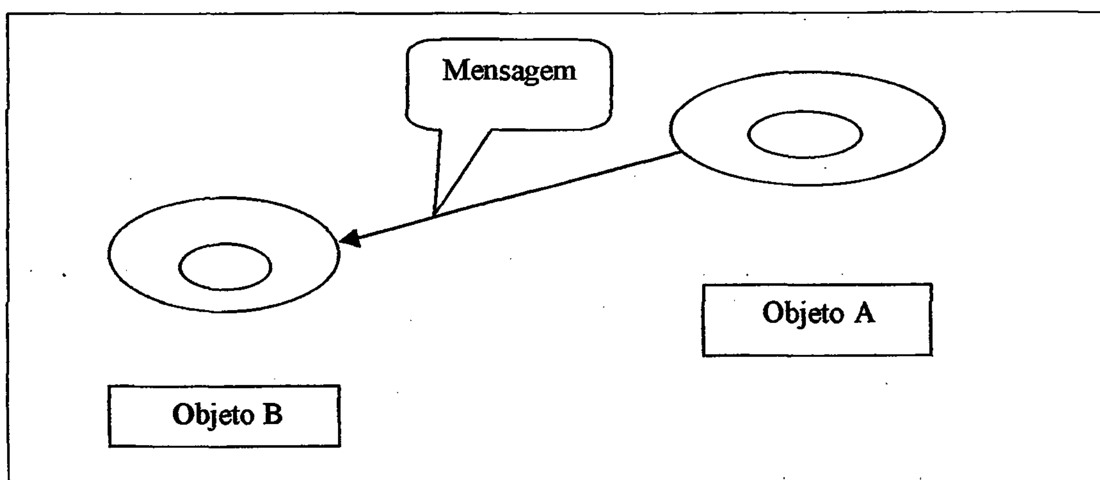


Figura 13 - Representação do Mecanismo de Passagem de Mensagens entre Objetos

Muitas vezes é necessário, além da mensagem para o objeto que nós queremos que execute um determinado método, um conjunto de outras informações que são

chamadas de parâmetros. Mesmo conhecendo o que um método faz, na maioria das vezes a utilização de parâmetros, que são passados juntamente com o nome do método, se faz necessária para que se possa definir exatamente o que o método deverá fazer, ou ainda qual método deverá executar tal operação.

As mensagens são compostas por três componentes:

- O objeto para o qual a mensagem é endereçada
- O nome do método a ser executado
- O conjunto de parâmetros necessários

Os três componentes acima são as informações suficientes para que um objeto, que receba uma mensagem, possa executar um método.

Assim, tem-se como benefícios das utilização de mensagens, o comportamento de um objeto é expresso pelos seus métodos, desta maneira o mecanismo de passagem de mensagens fornece todas as possíveis interações entre os objetos; e que os objetos não necessariamente precisam ser ou estar no mesmo processo ou igualmente na mesma máquina para enviar ou receber mensagens de outros objetos.

2.3.5 Herança

Uma das características da tecnologia da programação orientada a objetos é a de permitir que classes sejam definidas em termos de outras classes **[AMB97]**. Por exemplo, carros de passeio e carros de corrida são tipos diferentes de carros. Na terminologia da orientação a objetos, carros de passeio e carros de corrida são todos subclasses da classe carro. Similarmente, a classe carro é a super classe de carros de passeio e carros de corrida. Em Java utiliza-se a palavra reservada *extends* para fazer referência à definição de uma subclasse. Para a referência a uma super classe utiliza-se a palavra reservada *Super*.

A definição de uma subclasse carroCorrida a partir da super classe carro seria:

```
public class Carro {
        String marca;
        int velocidadeMax;
        Carro (String marca, int v) {
        this.marca = m;
                velocidadeMax = v;
        }
        public void parar( lista de parâmetros) { // codificação do método parar }
}
```

Figura 14 - Exemplo de definição da classe-Carro em Java

```
class carroCorrida extends Carro {
        int velocidadeBox;
        int numMarchas;
        carroCorrida (int vb, int nm) {
        super("Fiat",200);
        velocidadeMax = vb;
        numMarchas = num;
        }
}
```

Figura 15 - Exemplo de definição da sub-classe-carroCorrida em Java

É importante observar que a característica de herança permite que a classe nova herde todos os estados e os comportamentos da super classe, e ainda permite, a classe nova, a opção de acrescentar novos estados e/ou comportamentos que podem ser resumidos em melhor funcionalidade.

Todas as subclasses em Java são por padrão extensão da classe Objetcs.

Cada subclasse herda o estado (em forma de variáveis) e o comportamento (em forma de métodos) da super classe. Por exemplo, carros de corrida compartilham o mesmo estado: velocidade e marcha atual. Também, cada subclasse herda métodos da

super classe. Carros de corrida herdam da super classe carro trocar marcha, por exemplo.

Além das subclasses herdarem o estado e o comportamento da sua super classe, elas por sua vez podem estender, adicionar tanto métodos como variáveis. Subclasses também podem sobrescrever métodos herdados, desta forma implementando especialidades destes métodos. Por exemplo, vamos supor que todos os carros de corrida provêem a característica de possuírem 5 marchas (uma a mais que os carros normais). Assim é possível definir o método mudança de marcha sobrescrevendo-o, ou melhor, substituindo-o para que se possa utilizar este novo método. É importante destacar que a substituição de um ou mais métodos implica necessariamente em métodos que possuam a mesma assinatura.

A herança não está limitada a apenas um nível de extensão. Subclasses podem ser definidas de acordo com as necessidades.

Como benefícios da herança, pode-se ser citados os seguintes:

- Através do uso da herança, os desenvolvedores de programas podem reutilizar o código quantas vezes quiserem; e

- Os desenvolvedores de programas podem ainda implementar superclasses chamadas de abstratas que definem comportamento genérico. As superclasses definem e implementam parcialmente o comportamento, sendo que nas subclasses os desenvolvedores de programas terão a função de detalhar e especializar tudo aquilo que não estiver definido, e nem mesmo implementado.

2.3.6 Polimorfismo

Quando é definido um método que substitui um método de uma super classe está sendo provido este método de uma nova implementação; implementação particular a

esta nova classe. A referência a um método sem a preocupação a qual classe pertence mas sim possuindo a certeza do oferecimento do comportamento por parte do método caracteriza o polimorfismo [DEI98]. Isto significa dizer que para uma mesma classe tem-se implementações de mesmo nome e não de mesma assinatura de método, caracterizando portanto uma grande flexibilidade. Esta flexibilidade pode ser verificada quando um objeto envia uma mensagem e não se preocupa a qual classe pertence este objeto, ou seja, o objeto que recebe a mensagem é quem deve fornecer a implementação referente a mensagem.

É possível ilustrar o polimorfismo através do exemplo da definição da classe conta de um banco.

```
// Arquivo fonte Conta.java
public class Conta {
        String numero;
        double saldo;

        Conta( ) {
                numero = 000;
                saldo =  20;
        }
        Conta(String numero, double saldo) {
                this.numero = numero;
                this.saldo =  saldo;
        }
        public void calculaSaldo( ); {
                System.out.println(" ... objeto conta... ");
        }
        public static void main(String args[]) {
                Conta x = new Conta( );
                Poupanca cp = new Poupanca ( );
                Corrente cc = new Corrente( );
                x. calculaSaldo( );
                cc.calculaSaldo( );
                x = cp;
                x.calculaSaldo( );
        }
}
------------------------------------------------------------------------------------------------
-----
Dados  de saída:
objeto conta
objeto conta corrente 40.0
conta poupança 10.0
------------------------------------------------------------------------------------------------
-----
// Arquivo fonte Corrente.java
public class Corrente extends Conta {
        Corrente( ) { super( ); }
        public void calculaSaldo( ) { System.out.prinln(" objeto conta corrente " +
saldo*2); }
}
// Arquivo fonte Poupanca.java
public class Poupanca extends Conta {
        Poupanca( ) { super( ); }
        public void calculaSaldo( ) { System.out.prinln(" conta poupanca " + saldo/2); }
}
```

Figura 16 - Exemplo de polimorfismo em Java

A duplicação de nomes de construtores pode ser estranha, mas a linguagem Java permite, e isto é caracterizado como um polimorfismo do tipo "ad-hoc".

A palavra reservada "this" no método (construtor) conta é referência para o objeto atual ou corrente, evitando desta forma o conflito entre variáveis de instância e argumentos que são utilizados como variáveis locais em métodos.

CAPÍTULO 3. LINGUAGEM DE MODELAGEM UNIFICADA

3.1 INTRODUÇÃO

A UML nasceu da cooperação de três modelos mais utilizados no mercado para a modelagem de sistemas de software. O Object Modeling Technique (OMT) idealizado por James Rumbaugh, o Booch Method criado por Grady Booch e o Objectory (OOSE) Process de Ivar Jacobson. Esses três renomados pesquisadores da área de Engehnaria de Software, em meados de 1995, resolveram juntar seus esforços no sentido da padronização de uma notação que fosse reconhecida como padrão mundial para análise e projeto de sistemas orientados a objetos (OOAD). Esta notação ou linguagem de modelagem deveria unificar todo e qualquer tipo de aplicação, seja ela complexa, tempo real, cliente/servidor ou outros tipos de sistemas de software **[FUR98]**.

Em novembro de 1997, a Unified Modeling Language (UML) foi adotada pelo Object Management Group - OMG como padrão industrial de linguagem para especificação, visualização, construção, comunicação e documentação de sistemas de software. Ela foi projetada para ser utilizada em **[ALH98a]** e **[ALH98b]**:

- Estabelecer o domínio de um sistema através do diagrama de uso de caso;
- Através dos diagramas de interação perceber a dinâmica/funcionalidade de um sistema;
- Através do diagrama de classe representar estaticamente o estado de um sistema;
- Através do diagrama de transição representar o comportamento dos objetos;
- Apresentar a arquitetura através dos diagramas de componentes e de implementação; e

- Através de elementos chaves (estereótipos) estender a sua funcionalidade de acordo com as necessidades dos usuários e do sistema a ser desenvolvido.

A UML foi uma evolução natural e gradativa dos conceitos e técnicas aplicadas ao desenvolvimento de sistemas **[FUR98]**. Ela, por sua vez, não é uma linguagem de programação visual e sim uma linguagem de modelagem que permite a visualização, o gerenciamento e a apresentação de um sistema. Ela ainda apresenta como característica inerente ao seu desenvolvimento a facilidade do oferecimento da representação de conceitos fundamentais de orientação a objeto, tais como herança, polimorfismo, classes e objetos; ser independente de linguagem de programação, facilitando desta forma e deixando ao programador a flexibilidade em adaptar o seu modelo na linguagem que mais se enquadre a sua necessidade e, ainda, possui a integração das melhores práticas para o desenvolvimento de sistemas. Através do metamodelo de classe, a UML oferece uma base formal para definição estática da representação de um sistema de software.

O objetivo da adoção de uma linguagem de modelagem unificada é a de facilitar, de melhor compreender, de atualizar rapidamente e de padronizar o desenvolvimento de sistemas orientado a objetos, seja qual for o seu tipo, fazendo com que os métodos conceituais sejam também executáveis.

Assim, a notação da Linguagem Unificada de Modelagem - UML está baseada em cinco tipos de visões, nove tipos de diagramas e num conjunto de elementos que combinados juntamente com as visões e diagramas especificam e apresentam o sistema, a ser desenvolvido, no que diz respeito a sua funcionalidade tanto a nível conceitual como a nível funcional (estático e dinâmico).

Para um melhor entendimento da notação da linguagem de modelagem é necessário um estudo pormenorizado de cada um dos componentes que compõem a sua notação.

3.2 A NOTAÇÃO DA UML

3.2.1 Visões

Um sistema, geralmente, é composto de um conjunto muito particular de especificações (tempo, armazenamento, funcionalidade, confiabilidade, organização, planejamento, entre outras), o que torna, praticamente, impossível a representação de um sistema através de uma simples abstração gráfica.

As visões são abstrações que dão enfoque a níveis ou a partes específicas do sistema a ser modelado. Elas são compostas de uma série de gráficos que em conjunto mostram aspectos particulares do sistemas a ser construído.

A UML possui cinco visões que dão a equipe de análise a oportunidade de visualizar e de observar aspectos importantes, ou vitais, analisando para tanto uma das visões sem se preocupar com os detalhes ou enfoque de uma outra. Ainda, o desenvolvedor ou a equipe de desenvolvimento pode analisar duas ou até todas as visões de acordo com a sua necessidade ou característica do sistema. As visões contempladas pela notação UML são:

- Visão lógica;
- Visão de casos de uso;
- Visão de componentes;
- Visão de organização; e
- Visão de concorrência.

3.2.1.1 Visão Lógica

Esta visão é composta geralmente pêlos diagramas de classes, de objetos, seqüência, colaboração, atividade e de estado. Ela, por sua vez, trata da representação do sistema a nível de implementação, descrevendo como, e de que forma, os objetos

criados irão interagir mostrando o seu estado e o conjunto de atividades executadas internamente.

3.2.1.2 Visão de casos de uso "Use-case"

A Visão "Use-case" descreve a funcionalidade do sistema através de atores externos, os quais representam os usuários. Esta visão é montada sobre os diagramas de uso de caso e de atividade. Está visão tem um papel decisivo e central, pois a partir dela são definidas e construídas as outras visões do sistema.

3.2.1.3 Visão de Componentes

Esta visão é responsável pela descrição da implementação dos módulos e de suas dependências.

3.2.1.4 Visão de Concorrência

Esta visão trata da representação e da divisão do sistema em processos e processadores. É nesta visão que teremos idéia das execuções paralelas, se há ou não sincronismo e como se dará o processo de concorrência e execução de tarefas. Esta visão é composta pelos diagramas dinâmicos, que são os diagramas de estado, seqüência, colaboração e atividade e pelos diagramas de implementação, que são os diagramas de componentes e execução.

3.2.1.5 Visão de Organização

Por fim, a visão de organização apresenta a disposição física do sistema computacional e de como os componentes de sistema (periféricos, computadores, elementos de comunicação de dados, entre outros) se conectam entre si. Esta visão é representada pelo diagrama de execução.

3.2.2 Elementos Essenciais

A UML está definida a partir de seus elementos essenciais [FUR98]. Esses elementos são responsáveis por dar toda a sustentação formal e conceitual à linguagem de modelagem. Os elementos essenciais são divididos em:

- **Estruturais**: classes e casos de uso, são exemplos que representam partes estáticas de um modelo;
- **Comportamentais**: Interação e máquinas de estado, são exemplos que representam partes dinâmicas de um modelo;
- **Agrupamentos**: Pacotes, é um exemplo que representa parte(s) de agrupamento de um modelo;
- **Anotação**: Notas, é um exemplo que representa parte(s)s explicativa(s) de um modelo.

3.2.2.1 Classes

As classes são moldes ou modelos de objetos, de mesmo tipo, do mundo real, que se deseja mapear no mundo computacional. Elas são responsáveis pelas representações ou abstrações de objetos do mesmo tipo. Em UML as classes são representadas por um retângulo dividido em três partes. A primeira parte diz respeito ao nome da classe, a segunda parte está relacionado aos atributos ou características de uma classe. Por fim, temos a terceira parte que representa os métodos ou operações suportadas pela classe.

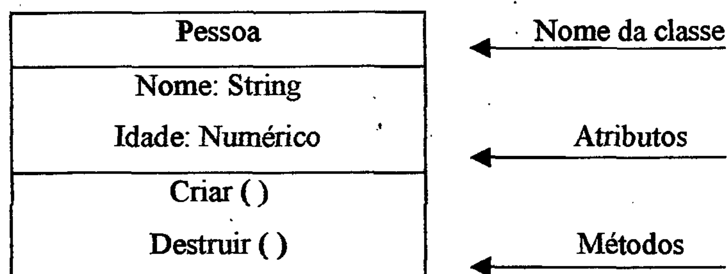


Figura 17 - Representação de uma classe em UML

3.2.2.2 Objeto

Um objeto é um elemento real que existe no nosso mundo, desta forma, podemos criar, destruir e acompanhar o seu estado e funcionamento. Em UML, um objeto possui a mesma representação do elemento essencial classe, exceto pelo nome do objeto ter que estar sublinhado e poder vir precedido do nome da classe separados entre dois pontos.

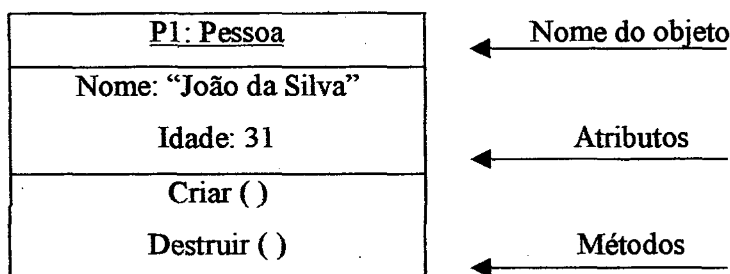


Figura 18 - Representação de um objeto em UML

3.2.2.3 Estado

Todos os objetos possuem seus atributos ou características que, de acordo com a interação com outros objetos, vão sofrendo modificações. Através dos seus atributos pode-se perceber o estado de um objeto.

Também, através da análise do estado de um objeto, pode-se prever todos os comportamentos que um determinado objeto poderá assumir ao longo da execução de um sistema.

Um estado pode conter três partes. A primeira representa o nome do estado. A segunda parte, opcional, mostra a variável de estado, onde os atributos do objeto podem ser listados e analisados. A última parte, também opcional, diz respeito às atividades, ou

seja, onde pode-se listar e analisar o conjunto de ações e eventos que são executadas antes que um objeto passe para outro estado.

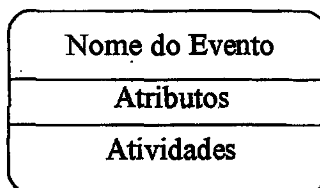


Figura 19 - Representação de um estado em UML

3.2.2.4 Pacote

Na UML são definidos elementos cujas notações podem ser empregadas ao longo dos diversos diagramas propostos. Dentre eles, podemos destacar o **Pacote** que é uma maneira de organização de elementos de modelo em grupos, podendo inclusive, estar aninhado dentro de outros pacotes. Esta técnica permite e ajuda o entendimento humano do elemento em estudo, principalmente em sistemas complexos. Os pacotes armazenam as classes dentro deles, sendo que cada classe definida num pacote possui escopo somente válido no pacote ao qual ela foi definida. Ainda, os pacotes podem ter visibilidade para indicar como outros pacotes terão acesso a seu conteúdo. São definidos quatro espécies de visibilidade de pacotes:

- **Privado**: acesso somente ao pacote que possui um elemento ou um pacote que importa o elemento de modelo, pode usá-lo.

- **Protegido**: a diferença entre a visibilidade privado e protegido está na herança, pois apenas pacotes com visibilidade protegida podem usar elementos de modelo de pacotes de generalização.

- **Público**: este tipo de visibilidade permite que outros elementos possam ver e usar o conteúdo do pacote.

- **Implementação**: as classes inseridas em pacotes podem necessitar de operações de classes em diferentes pacotes. Isto caracteriza dependência entre pacotes, portanto, existe dependência entre pacotes quando houver dependência entre classes de diferentes pacotes. Similar à visibilidade privada, mas elementos de modelo que tem uma dependência com um pacote não podem usar os elementos dentro daquele pacote.

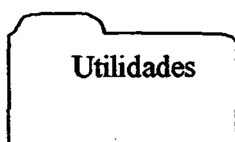


Figura 20 - Representação de um pacote em UML

3.2.2.5 Componentes

Os componentes representam detalhes físico de um sistema e são mostrados através do diagrama de componentes, eles podem ser arquivos ou o código fonte de um aplicativo.

3.2.2.6 Relacionamentos

Como poucas classes vivem sozinhas, elas, por si só, podem estar ligadas ou vinculadas através de relacionamentos. Os relacionamentos são responsáveis por ligações lógicas entre classes e objetos. A seguir são apresentados os relacionamentos suportados pela UML, baseados em **[BOO00]** e **[FUR98]**, bem como a suas notações.

- Associação
  - Agregação
  - Composição
- Realização
- Generalização
- Dependência e Refinamento

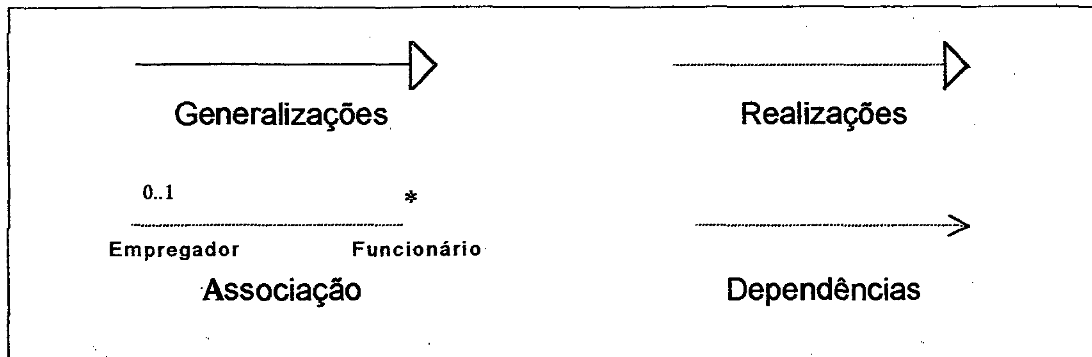


Figura 21 - Notação de relacionamentos em UML

3.2.2.6.1 Associação

As associações são utilizadas para representar dependência entre estruturas (classes, objetos) e geralmente possuem nome para identificar a sua função ou significado, sendo que a maioria dos detalhes importantes que esclarecem uma associação está em seus papéis que podem apresentar informações de:

a) Multiplicidade: também chamada de cardinalidade no enfoque entidade/relacionamento. A multiplicidade é um dos aspectos principais na UML, pois ela especifica a quantidade de correspondência que um objeto da classe A pode atingir em objetos equivalentes na classe B. Ainda a multiplicidade pode ser representada pela seguinte simbologia:
   - 1 = exatamente 1
   - * = muitos (zero ou mais)
   - 0..1 = opcional (zero ou um)
   - m..n = seqüência especificada

b) Ordenação: quando houver multiplicidade maior que um, o conjunto está ordenado ou não ordenado, sendo esse último, o padrão.

c) Qualificação: é um indicador ou um atributo que indica quais objetos farão parte do filtro do conjunto de objetos relacionados.

d) Navegabilidade: é representada por uma seta direcionada que indica a orientação do relacionamento.

e) Visibilidade: a visibilidade refere-se ao grau de acesso que pode ser inserido em atributos e operações de uma classe. Assim sendo, tem-se o operador de visibilidade público, identificado pelo operador (+), onde a permissão é completa para uso. O operador de visibilidade protegido (#), prevê que qualquer elemento descendente possa utilizar. Já para o operador de visibilidade privado (-), somente o próprio elemento poderá usar.

f) Agregação: é um tipo especial de associação para denotar como um objeto é composto, pelo menos em parte de outro, em uma relação de todo/parte. Neste tipo de associação diz-se que é "fraco" e do tipo é parte de, sendo que o todo é composto de partes. Neste tipo de relacionamento, não se impõe que a vida das partes esteja relacionada com a vida do "todo". Por exemplo, um computador possui relacionamento com o monitor, teclado e mouse.

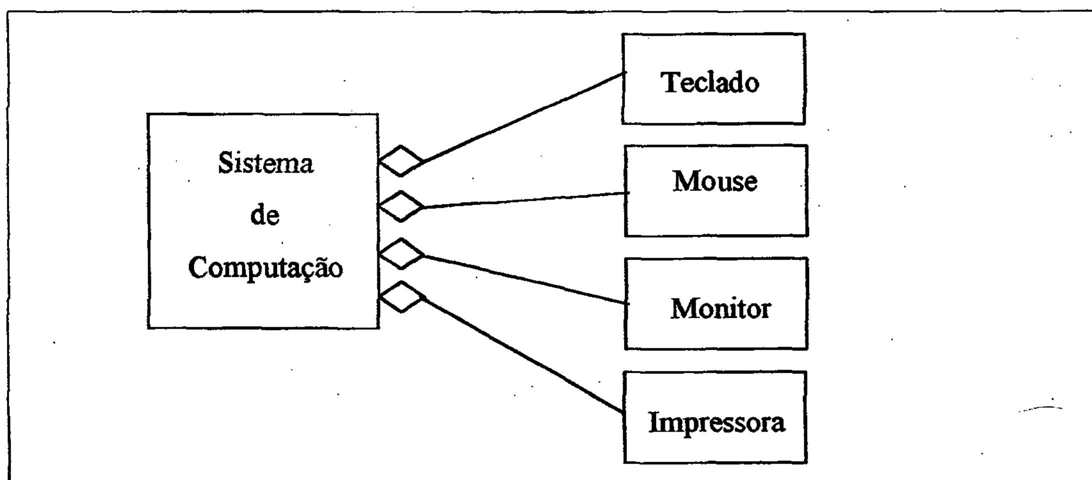


Figura 22 – Representação de Associação por Agregação

g) Composição: que é outro tipo de relacionamento. Diz-se desse relacionamento como "forte", pois a composição entre um elemento (o "todo") e outros elementos ("as partes") indicam que as partes só podem pertencer ao "todo" e

são criadas e destruídas como ele. Por exemplo um objeto janela possui correspondência com objetos de texto, botão e menu.

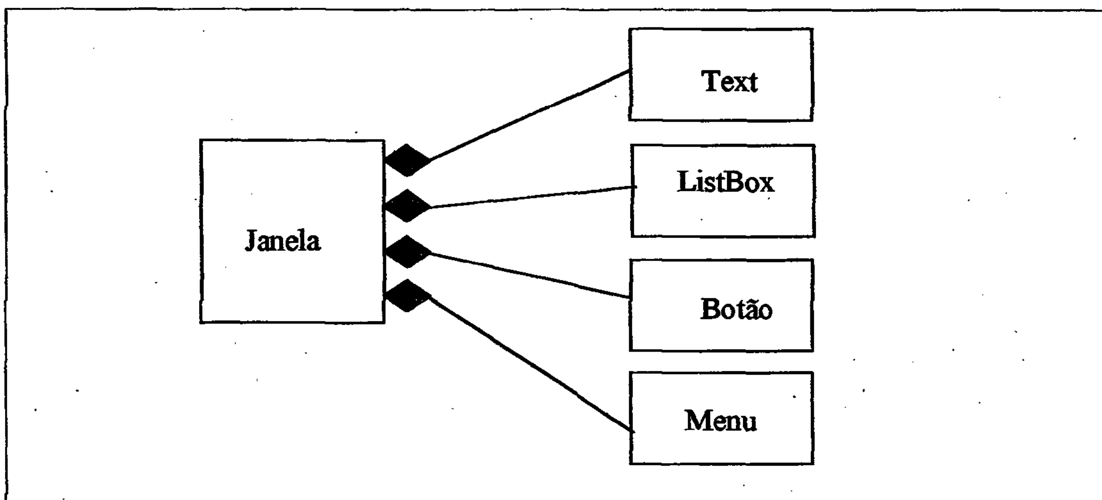


Figura 23 – Representação de Associação por Composição

Ainda, as associações podem ser ainda dos seguintes tipos **[FUR98]**:

- **Unária**: a existência de um relacionamento de uma classe para consigo própria conectando-se ambos os fins da associação a ela mesma (mas os dois fins são distintos) caracteriza associação unária. É também conhecida como associação reflexiva, cujo relacionamento pode conectar dois objetos diferentes de uma mesma classe ou um objeto a si próprio.

- **Binária/normal** - as associações normais, as mais comuns, contemplam apenas uma conexão entre classes. Elas são representadas por uma linha sólida entre as classes envolvidas, possuindo um nome, junto a linha que à representa, normalmente um verbo, mas substantivos também são usados. São também utilizadas setas que indicam a orientação das associações, ou seja, para onde a associação aponta. Mas as associações podem possuir dois nomes, significando um nome para cada sentido da associação.

- **Ternária** - A associação ternária associa três classes. Identificamos este tipo de associação através de um losango ligando as classes envolvidas.

- **n-ária**: é uma associação entre três ou mais classes, mas uma única classe pode aparecer mais de uma vez. Cada instância da associação representada em estruturas de classes respectivas. Associações ternárias ou superiores são mostradas como diamantes conectados ao símbolo de classe através de linhas com um caminho do diamante para cada classe participante.

- **Recursiva** - Este tipo de associação conecta uma classe a ela mesmo, ou seja, este tipo de associação representa semanticamente a conexão entre dois objetos da mesma classe.

- **Qualificada** - Este tipo de associação é utilizada com cardinalidade de 1 para vários ou vários para vários. Elas servem para especificar como o objeto no término da associação "n" é identificado, podendo ser visto como uma espécie de chave para identificação de todos os objetos na associação. Este identificador é representado no final da associação, junto a classe, onde a associação deve ser realizada.

- **Ordenada** - Utilizamos este tipo de associação quando desejarmos que uma ordem seja especificada através da associação. Ela pode ser escrita inserindo a palavra ordenada junto alinha da associação.

3.2.2.6.2 Generalização

No relacionamento de generalização (herança) existe uma relação entre um elemento mais geral e um elemento mais específico. O elemento específico contém todas as características do elemento geral e novas características que o tornem particular.

Os relacionamentos do tipo generalização podem ser de dois tipos: generalização normal ou generalização do tipo restrita. Esta última pode ainda ser dividida em *generalização restrita por sobreposição, disjuntiva, completa ou incompleta.*

a) Generalização Normal: neste tipo generalização, a subclasse herda tudo da super classe. Ela é representada por uma linha entre as duas classes que fazem o relacionamento, sendo colocado uma seta ao lado da subclasse indicando a generalização.

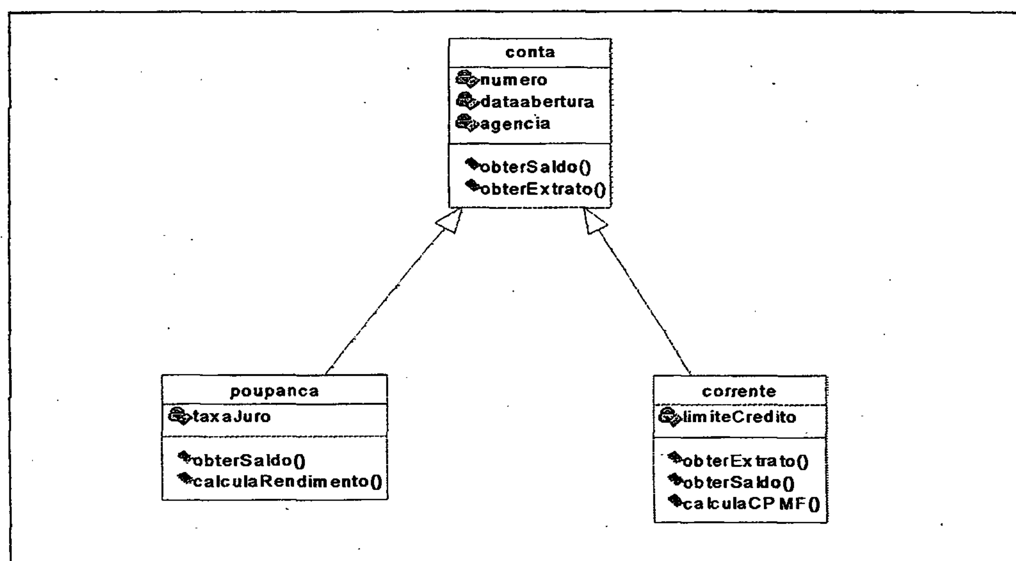


Figura 24 - Generalização

b) Generalização Restrita

- Por sobreposição e disjuntiva: significa que existe a herança de subclasses em relação a uma superclasse por sobreposição, novas subclasses destas podem herdar de mais de uma subclasse.

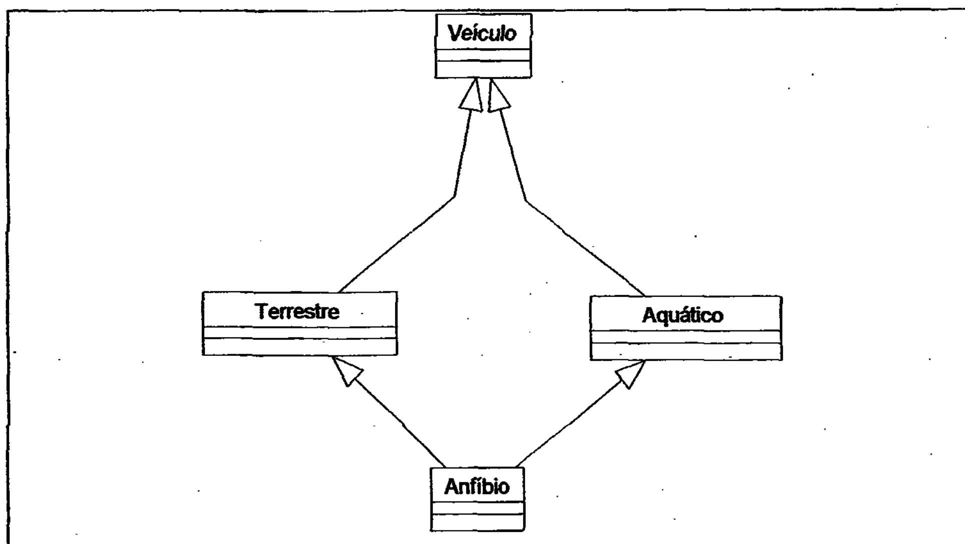


Figura 25 - Generalização Restrita

- Completa e incompleta: a não possibilidade de outra generalização é simbolizada por uma restrição, ou seja, significa que todas as subclasses já foram especificadas e não há mais possibilidade de generalização. A generalização padrão é a incompleta, sendo o contrário da completa.

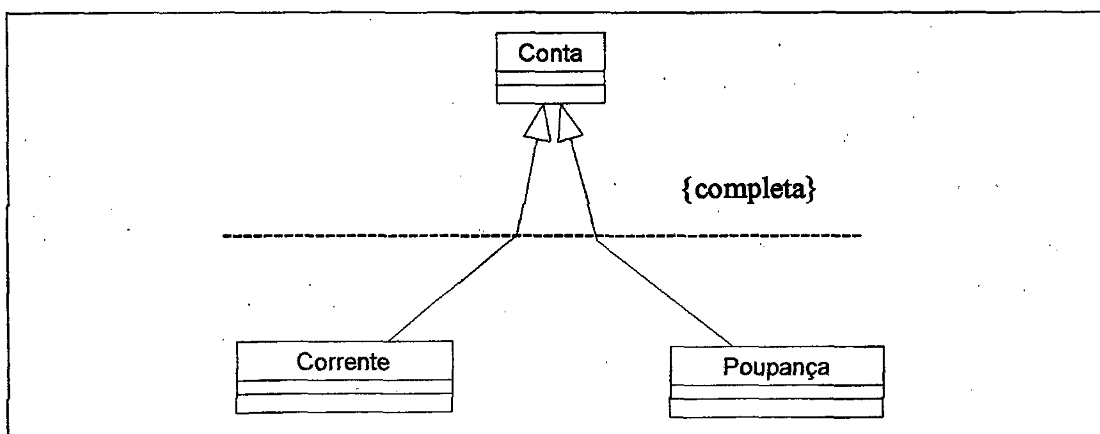


Figura 26 - Generalização Completa

3.2.2.6.3 Realização

Este tipo de relacionamento caracteriza-se por ser semântico entre classificadores, em que um classificador especifica um contrato que outro classificador garante executar.

3.2.2.6.4 Dependência e Refinamentos

Outro tipo de relacionamento é o de dependência onde qualquer mudança na especificação de um elemento pode alterar a especificação do elemento dependente. Ex. Cliente/Servidor.

Uma relação de dependência, em UML, é representada por uma linha tracejada com uma seta no final de um dos lados do relacionamento, e sobre esta linha a relação dependência existente entre as duas classes.

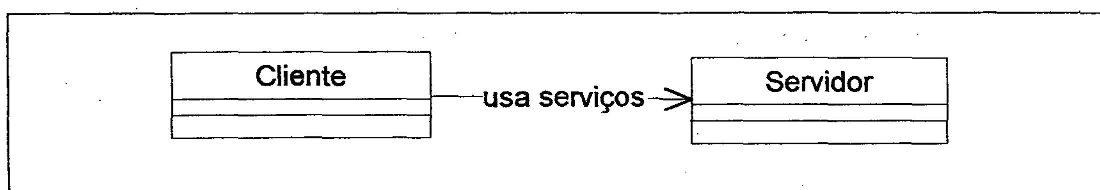


Figura 27 - Associação por Dependência

Já os refinamentos são utilizados, principalmente em grandes projetos, onde é visível a necessidade de coordenação de modelos. Os refinamentos são representados por uma linha tracejada entre as classes com um triângulo no final de um dos lados da associação. A coordenação de modelos pode ser comparada as etapas de análise de um projeto, onde temos que dividir e abstrair modelos para podermos ter uma noção de uma parte particular do projeto.

3.2.3 Elementos Complementares

São utilizados para tornar a notação do sistema a ser desenvolvido mais clara e fácil de ser entendida, ainda esses elementos são utilizados para, através dos seus mecanismos, estender ou adaptar a UML de acordo com a necessidade do usuário ou do método de análise a ser utilizado.

O **Estereótipo**, dá a possibilidade de extensão da UML pelo usuário, pois através dele podemos ter a liberdade de definir uma metaclassificação de elementos da UML, objetivando o ajuste da linguagem às necessidades do usuário ou do sistema a ser desenvolvido. Por exemplo, uma declaração de classe pode ser também um utilitário, isto quer dizer, um agrupamento de atributos e operações. A construção dessa classe - utilitários neste exemplo não se trata de uma construção fundamental e sim uma conveniência de programação.

A Nota, é outro elemento passível de utilização na modelagem de um sistema. Ela é um comentário colocado em um diagrama sem qualquer conteúdo semântico. A nota serve para tornar o modelo mais fácil, compreensivo e documentado para o usuário e para a equipe de desenvolvimento.

3.2.4 Diagramas da UML

A UML está baseada em diagramas que são representações gráficas de elementos de um sistema **[BOO00]**. São eles:

- Casos de Uso
- Classe
- Objeto
- Seqüência
- Colaboração
- Estados
- Atividades
- Componentes
- Implantação

Eles são representações gráficas que descrevem a abstração ou a representação de uma visão. A UML possui nove digramas que podem ser utilizados em mais de uma

visão de acordo com o sistema que está sendo modelado de acordo com as suas especificações.

3.2.4.1 Diagrama de Use-cases

Definem os requisitos funcionais do sistema.

São escritos em termos de seus atores que representam um entidade externa ao sistema como um usuário, um hardware, ou outro sistema que interage com o sistema modelado.

São utilizados para descreverem a visão externa do sistema e suas interações com o mundo exterior. Estes diagramas apresentam uma visão de alto nível de funcionalidade mediante entradas ou estímulos do usuário. Estes diagramas ainda descrevem a funcionalidade de um sistema através de atores interagindo com casos de uso.

Nos diagramas de uso de caso, o sistema é visto como uma caixa-preta que fornece situações de comportamento do sistema. São também utilizados para capturar as necessidades/expectativas do novo sistema, bem como para refinamento e aprimoramento de novas versões de um sistema. Nesse sentido podemos dizer que os diagramas de casos de uso são utilizados para:

1. obter os requisitos funcionais do sistema seja do desenvolvedor quanto do usuário;
2. proporcionar situações de aplicação para teste do sistema;
3. formar a base para o projeto do sistema e definir tarefas a serem desenvolvidas para a construção do sistema.

Nos diagramas de casos de uso são utilizados quatro componentes básicos:

a) Ator - entidade externa

b) Caso de uso - situação proposta entre usuário e sistema

c) Interação - mensagens trocadas entre ator-objeto e objeto-objeto

d) Sistema - aplicação a ser desenvolvida

A principal utilização de um diagrama de casos de uso está em descrever a funcionalidade do sistema poder ser percebida por atores externos.

Percebe-se que os diagramas de use case possuem as seguintes características:

- Definem os requisitos funcionais do sistemas;
- Atores são conectadas a use-cases através de associações;
- Use-cases em colaboração são importantes, pois descrevem e mostram classes/objetos; seus relacionamentos e sua interação exemplificando como as classes/objetos interagem para executarem uma atividade específica do sistema;
- São a base central para o desenvolvimento dos demais diagramas;
- A execução de um use-case é um cenário, pois mostra o caminho específico de cada ação.

Abaixo apresenta-se um exemplo de um diagrama de use case para atulização da situação de cliente.

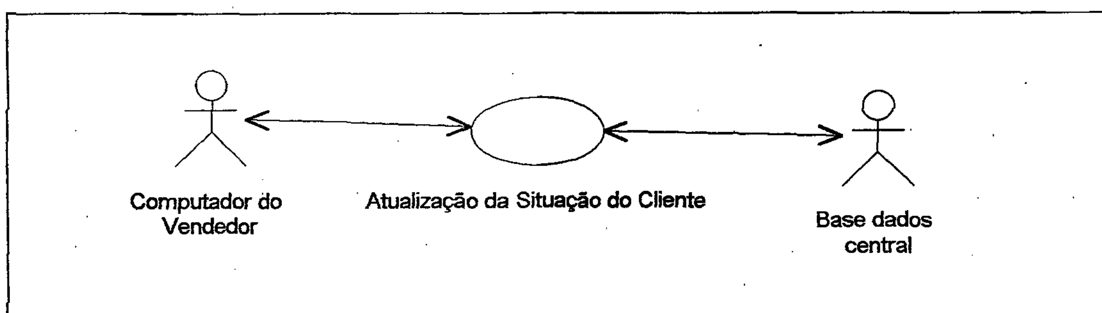


Figura 28 - Diagrama de Use Case

3.2.4.2 Diagrama de Classe

Este diagrama é o um dos mais importantes da UML, pois as classes são os elementos principais na representação de qualquer sistema orientado a objetos. Ele tem

por finalidade a representação de classes bem como o relacionamento existente entre elas.

As classes são representadas de acordo com a tecnologia de objetos, ou seja, através dos seus atributos (características), métodos (serviços), nome e de seus operadores de visibilidade (público, protegido e privado), que podem ser aplicados tanto para métodos como para atributos.

A visibilidade refere-se ao grau de acesso que pode ser inserido em atributos e métodos de uma classe. Assim sendo, tem-se o operador de visibilidade público, identificado pelo operador (+), onde a permissão é completa para uso. O operador de visibilidade protegido (#), que prevê que qualquer elemento descendente possa utilizar e o operador de visibilidade privado (-), onde somente o próprio elemento poderá usar.

Em síntese pode-se listar as seguintes importâncias de um digrama de classes:

- Apresenta a estrutura estática das classes que compõem um sistema;
- É considerado estático (utilização da palavra reservada ***static***) porque a estrutura de classes e relacionamentos descrita é sempre válida em qualquer parte do ciclo de vida do sistema;
- Mostras os relacionamentos, existentes, entre as classes;
- Permite a visualização da estrutura interna de uma classe (atributos e métodos);
- Ser um diagrama relativamente fácil de ser implementado em uma linguagem de programação que tenha suporte direto para construção de classes.

Abaixo segue um exemplo de diagrama de classes de um sistema que utiliza classes veículo, avião e carro. Pode-se notar a estrutura interna das classes, os seus relacionamentos (aqui representados através de uma especialização da classe conta).

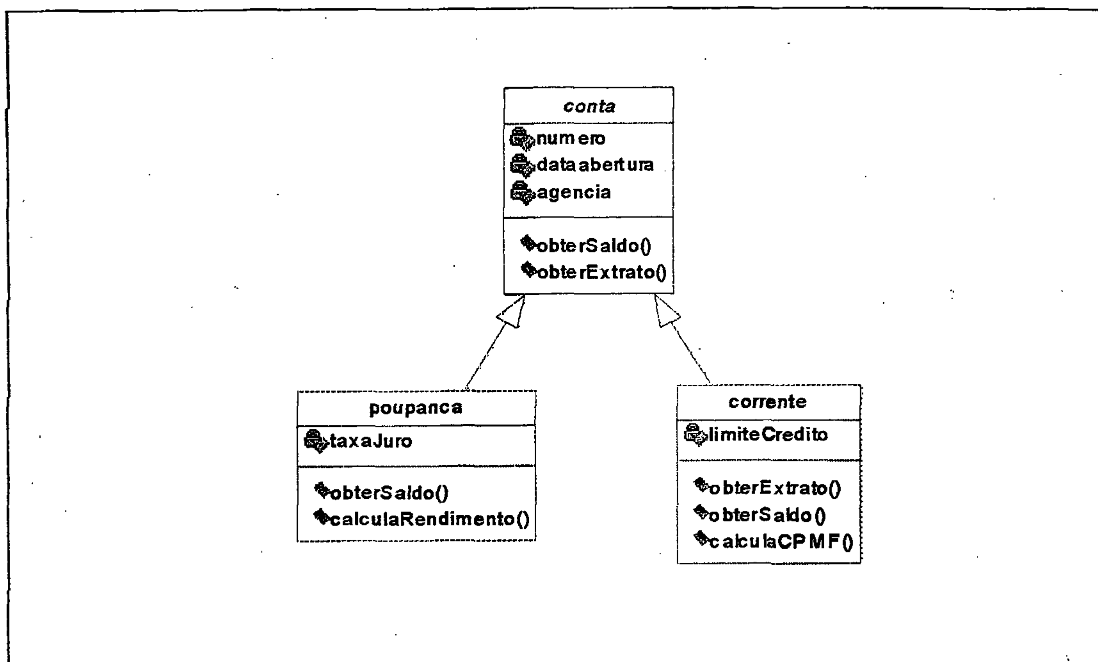


Figura 29 - Diagrama de Classes

3.2.4.3 Diagrama de Objeto

Os diagramas de objeto são estáticos (num determinado período de tempo) e por isso não mostram o comportamento dos objetos. Esses diagramas apresentam um instante de um sistema orientado a objetos em execução. Os objetos são mostrados neste diagrama com os valores de seus atributos e as ligações entre eles. Geralmente, a análise do diagrama de objetos é realizada em conjunto com o diagrama de classes, permitindo, assim, uma maior facilidade no entendimento, interpretação e identificação dos objetos.

Ainda neste tipo de diagrama temos os MULTIOBJECTS, que são conjuntos de objetos com um número indeterminado de elementos **[FUR98]**. Geralmente os MULTIOBJECTS são utilizados nos diagramas de colaboração, que serão vistos mais tarde, para representar o envio de mensagens entre vários objetos ao mesmo tempo.

A principal diferença entre um diagrama de classe e de um diagrama de objetos está no segundo que mostra um número de instâncias de classe, em vez de classes reais como *no primeiro.*

Como características principais desses diagramas pode-se citar:

- Mostra a ocorrência de objetos que foram instanciados das classes;
- Apresenta o perfil do sistema em certo momento de sua execução;
- A determinação de que operação é algo que é executado no objeto e método é o corpo ou definição da funcionalidade da ação a ser executada.

Desta forma, apresenta-se abaixo um exemplo de diagrama de objetos, onde tem-se que Pedro é um objeto derivado da classes engenheiro e que o objeto Projeto A é uma derivação da classe projeto. Ainda temos uma associação onde, neste instante, observa-se que Pedro(objeto) trabalha no Projeto A(objeto).

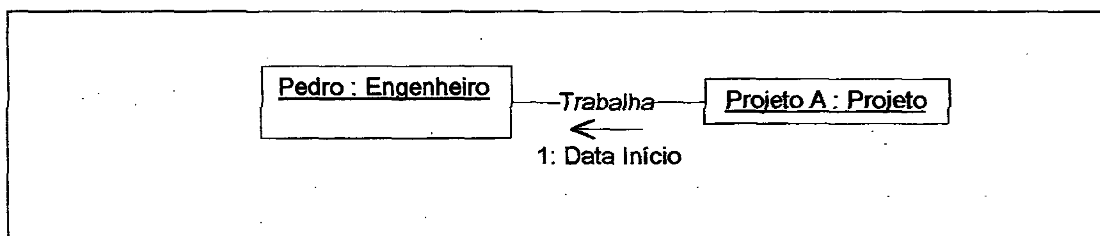


Figura 30 - Diagrama de Objetos

3.2.4.4 Diagramas de Seqüência

Os diagramas de seqüência possuem a finalidade de mostrar a seqüência de mensagens trocadas entre os objetos num determinado caso de uso ou função a ser provida pelo sistema. Estas seqüências de mensagens representam as interações entre objetos organizados em uma seqüência de tempo e de mensagens trocadas, onde são passadas informações para que ocorra a execução de alguma operação.

Os diagramas de seqüência são constituídos em duas dimensões. A dimensão vertical que simboliza o tempo e a dimensão horizontal simboliza a variação de objetos. As mensagens são simbolizadas por setas e têm números seqüenciais para que se torne mais explícito as seqüências no diagrama. O tempo de vida de um objeto é simbolizado por uma linha pontilhada.

Como características principais destes diagramas de interação, pode-se citar:

- Mostra a troca de mensagens entre os objetos para um cenário específico de uma certa atividade;
- Seu enfoque possui impacto no tempo, ou seja, por isso tem-se a sua estrutura dimensionada em dois eixos (tempo e objetos);
- Mostra a ativação/desativação de objetos;
- Faz parte dos diagramas de interação;
- A utilização desses diagramas não se restringe somente a atividades seqüenciais. Objetos concorrentes podem ser modeladas através desses diagramas, cada objeto com sua linha de execução (**Thread**).

A seguir, apresenta-se um exemplo de um diagrama de seqüência para um cenário específico de uma atividade de consulta a uma base de dados.

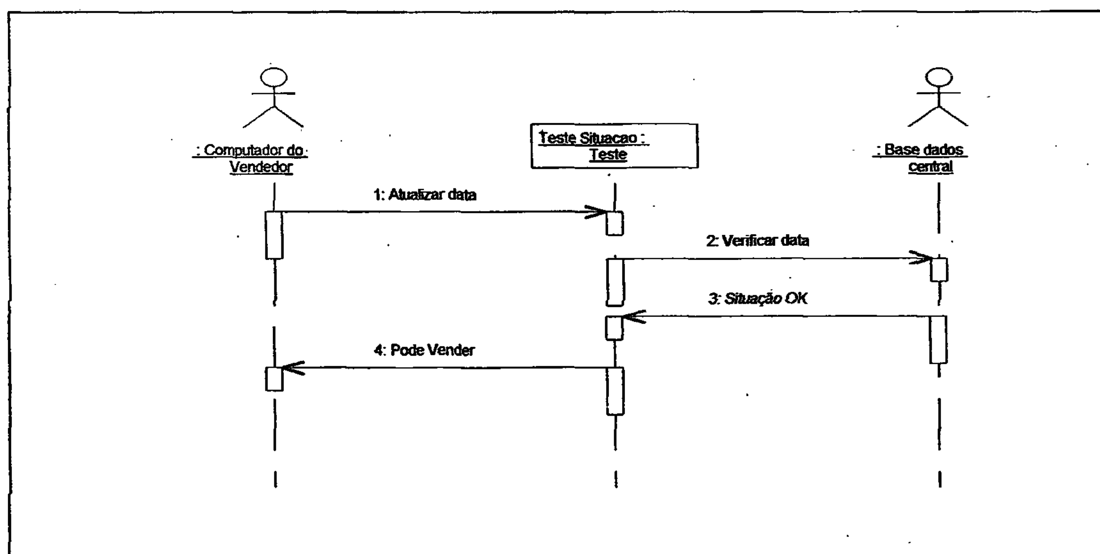


Figura 31 - Diagrama de Seqüência

3.2.4.5 Diagramas de Colaboração

Os diagramas de colaboração são responsáveis por mostrar os vínculos entre os objetos, ou seja, os relacionamentos entre os objetos, mas não trata o tempo como uma dimensão separada. A seqüência de tempo é indicada numerando-se as mensagens. Nos diagramas de colaboração temos os objetos desenhados como ícones e setas indicando as mensagens enviadas entre objetos para realização de um estudo de caso. Esses diagramas são uma extensão dos diagramas de objetos, pois além de mostrarem os objetos, apresentam os relacionamentos entre objetos de um estudo de caso organizado. Tanto objetos como as mensagens neste tipo de diagrama são identificados com um número, de maneira que estes são usados para evidenciar a seqüência de mensagens entre os objetos.

Pode-se diferenciar diagramas de seqüência de diagramas de colaboração pela enfoque de cada um e evidentemente pela sua notação. Esse último possui enfoque no contexto do sistema enquanto o primeiro enfatiza o tempo como base.

São características dos diagramas de colaboração:

- Mostra a colaboração dinâmica entre os objetos;
- Percebe-se, também, os objetos e seus relacionamentos;
- Apresenta a interação dinâmica de uma caso de uso organizado em torno de objetos e seus vínculos mútuos.

A seguir, tem-se um exemplo de diagrama de colaboração para um estudo de caso.

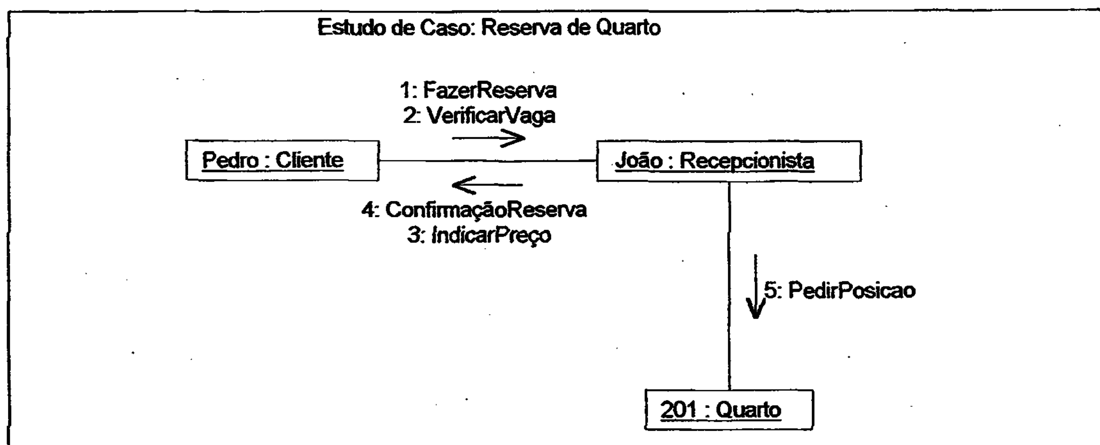


Figura 32 - Diagrama de Colaboração

3.2.4.6 Diagramas de Estado

São um complemento dos diagramas de classe, pois relacionam os possíveis estados que os objetos da classe podem ter e quais eventos podem causar a mudança de estado (transição). Os diagramas de estado são utilizados para modelar o estado em que um objeto pode estar e os eventos que fazem com que o objeto passe de um estado para outro. Esses diagramas são utilizados ainda para modelar o comportamento de objetos através de uma máquina contendo estados, transições, eventos e atividades.

São características dos diagramas de estados:

- Mostrar as seqüências de estados que um objeto ou uma interação assume a partir de estímulos, juntamente com suas respostas.
- Apresentam todos os estados possíveis que um objeto de uma certa classe pode se encontrar, mostrando também os eventos que provocaram as mudanças;
- São descritos somente para as classes que possuem um número de estados conhecidos e onde o comportamento das classes é afetado e modificado pelos diferentes estados. Geralmente os diagramas de estado modelam classes que possuem comportamento mecânico ou sistemas que possuem tal comportamento. Por exemplo: Sistema para controle de elevadores, Sistema para controle de centrais telefônicas.

Abaixo, apresenta-se um exemplo de diagrama de estado para um sistema de controle de acesso.

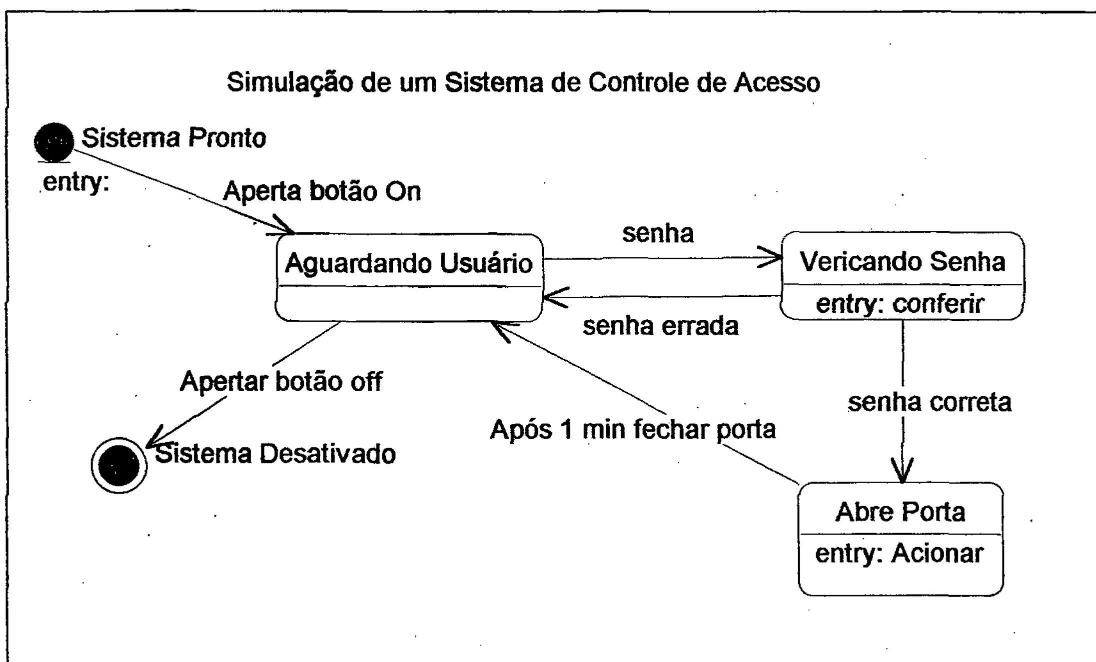


Figura 33 - Diagrama de Estado

3.2.4.7 Diagramas de Atividade

São um caso especial de diagramas de estado, pois são responsáveis por mostrar o processamento interno, descrevendo as atividades executadas em uma operação. Os diagramas de atividade são semelhantes aos antigos fluxogramas e possuem a finalidade de representar o fluxo entre atividades sendo elas concorrentes ou não. São um caso especial dos diagramas de estados, com a maioria das transições resultantes do término das atividades.

Esses diagramas são utilizados principalmente na modelagem de atividades concorrentes, para tanto são definidas barras de sincronização utilizadas para representar um único fluxo de controle em vários fluxos de controle concorrentes (FORK) e a sincronização de dois ou mais fluxos de controle concorrentes (JOIN). Ainda, nesse tipo de diagrama, são utilizadas SWIMLANES (rais) que definem quais

conjuntos de classes são responsáveis pela realização de cada atividade. Na maioria dos casos, as raias implicam concorrência, ou pelo menos independência das atividades.

Resumidamente pode-se observar que os diagramas de atividades servem para:

- Mostrar o fluxo seqüencial das atividades;
- Decisões e condições, como execução paralela, também podem ser mostradas;
- Apresentar a seqüência de ações a partir da execução de uma operação.

Em **[SUR99]** pode-se observar um exemplo de diagrama de atividades, como mostrado a seguir:

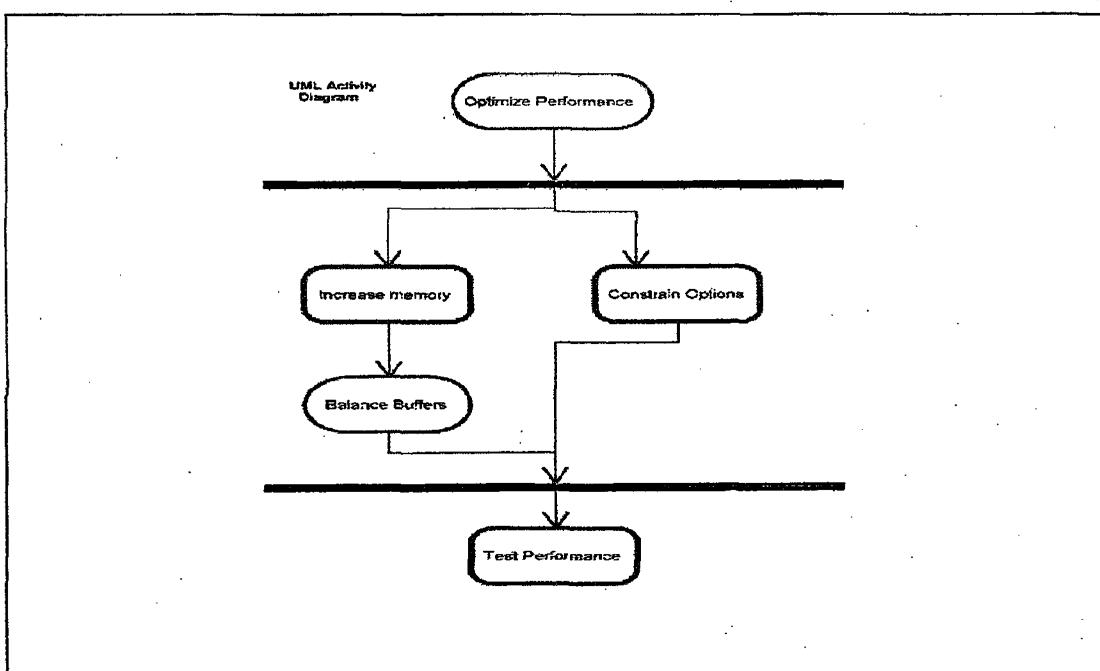


Figura 34 - Diagrama de Atividade [SUR99]

3.2.4.8 Diagramas de Componentes

Os diagramas de componentes são utilizados para representar os detalhes físicos de um sistema. Nesses diagramas são mostrados componentes e suas relações. Esses diagramas podem ser utilizados para modelar o código fonte de um aplicativo, dependências e vínculos entre componentes de software, sendo que, o que está sendo modelado deve obrigatoriamente ser físico, formado por bits.

Em resumo, os diagramas de componentes possuem as seguintes características:

- A possibilidade de descrição dos componentes de software e suas dependências entre si; e
- Que os componentes são a implementação na arquitetura física dos conceitos e da funcionalidade definidos na arquitetura lógica.

A seguir, é mostrado um exemplo de diagrama de componente.

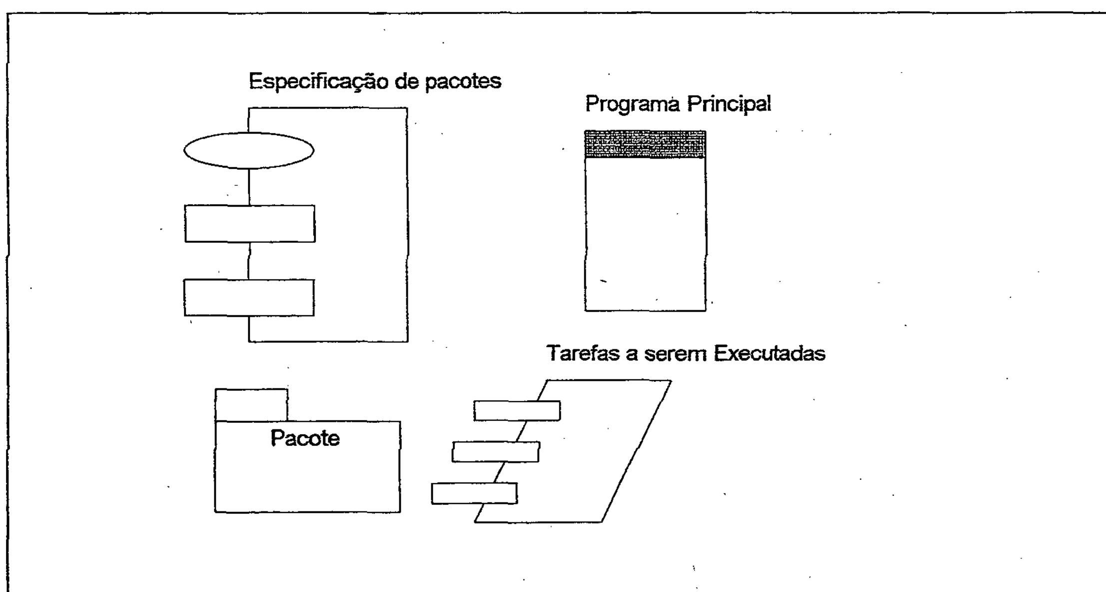


Figura 35 - Diagrama de Componentes

3.2.4.9 Diagramas de Implantação

Os diagramas de implantação são utilizados para modelar o ambiente em que o sistema será executado. Eles, por sua vez, só fazem sentido para sistemas que rodam em várias máquinas ou dispositivos. Já para os sistemas que serão executados em uma única máquina e escopo de comunicação se restringe ao teclado e monitor, geralmente os diagramas de implantação não são necessários.

Esses diagramas são compostos por nós e associações, ou seja, relacionamentos de comunicação. Por exemplo, um nó pode ser um computador, uma rede, um disco rígido, um sensor, etc. A associação entre dois nós representa uma conexão física entre eles, como uma conexão ethernet, uma linha serial ou um link de satélite.

Nos diagramas de implantação é possível utilizar o recurso de estereótipos que permite estender a linguagem UML com mais símbolos e nova semântica. Assim sendo, como os nós representam elementos físicos de um sistema, eles são os elementos mais estereotipados da UML.

Como características principais dos diagramas de implantação, tem-se:

- Mostrar a estrutura física do hardware e do software no sistema; e
- Apresentar a arquitetura run-time de processadores, componentes físicos (devices), e de software que rodam no ambiente do sistema.

A seguir, tem-se um exemplo de diagrama de implantação:

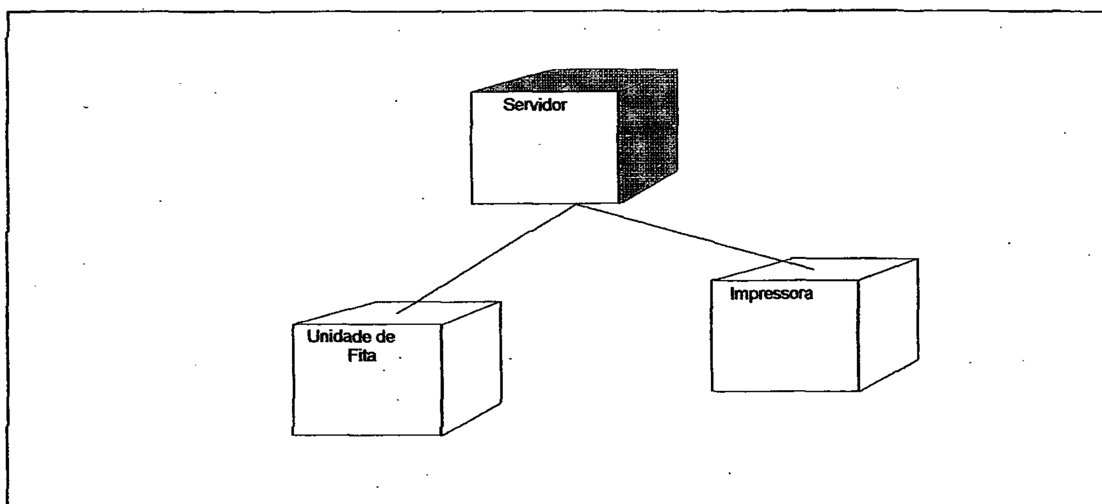


Figura 36 - Diagrama de Implantação

Os diagramas da UML procuram representar o sistema, a ser desenvolvido, de uma perspectiva diferente, assim a integração desses perspectivas ou diagramas de forma consistente, aliado a elementos de documentação, são elementos principais e iniciais para para uma boa modelagem. Por isso a UML procura prover uma notação padrão para o desenvolvimento de sistemas, utilizando para tanto de símbolos gráficos para representação do sistema em construção.

CAPÍTULO 4. PROCESSO DE TRADUÇÃO UML - JAVA

4.1 INTRODUÇÃO

A engenharia de sistemas computacionais corresponde ao processo de transformar ou traduzir um modelo lógico para código, em uma linguagem de programação. Nesta linha de pensamento, os seguintes tópicos serão abordados neste capítulo:

- análise de correspondência(s) entre modelos escritos em UML para Java;
- limitações UML – Java;
- a apresentação do processo de mapeamento através de estudos de casos; e
- engenharia reversa Java – UML.

4.2 ANÁLISE DE CORRESPONDÊNCIAS ENTRE UML - JAVA

Num primeiro momento, buscar-se-á reconhecer e registrar a forma com que as visões, oferecidas pela UML podem ser mapeadas em Java. Isto demandará tarefas como:

- determinar as visões a serem alvo de análise;
- a viabilidade de uma visão ser mapeada em Java; e
- a elaboração de uma tabela de correspondência entre UML e Java.

As visões apresentadas, anteriormente no capítulo 3, podem ser representadas a partir do estudo dos seus diagramas. Eles são o primeiro passo para a criação de um modelo.

Cada uma das visões enfoca aspectos particulares do sistema. Desta forma, será realizado uma análise de cada uma delas para então determinar a sua viabilidade no processo de mapeamento.

O passo inicial para o entendimento do mecanismo de tradução é o conhecimento dos modelos de representação de um sistema. São eles:

a) **Modelo estático ou estrutural**: mostra a estrutura de classes e relacionamentos mantidos entre elas. Pode-se observar, a estrutura estática do sistema, através da observação dos diagramas de classe, de objetos e da visão lógica fornecida pela UML;

b) **Modelo funcional**: apresenta os requisitos e a funcionalidade do sistema, podendo ser representado através de um conjunto de diagramas de use-cases, ou ainda, podendo ser observado através da análise da visão de use case;

c) **Modelo dinâmico**: mostra como o modelo modifica o seu estado e de que forma aconteceu esta transição. Podemos observar estes requisitos de estado através da análise do diagrama de estado, atividade, seqüência e colaboração;

d) **Modelo de organização**: este, por sua vez, apresenta como e de que forma estão organizados os componentes para implantação do sistema. Isto pode ser percebido através dos diagramas de componentes, implantação e também pela análise das visões de organização e de componentes.

A partir da apresentação e do estudo dos modelos acima, percebe-se que eles contemplam as visões oferecidas pela UML. São elas:

a) **Visão de Use-case**: ela é central, pois a sua construção é a base da construção das outras visões. Ela é montada sobre os diagramas de use-cases e eventualmente pelos diagramas de atividade e é definida a partir dos requisitos de **funcionalidade** do sistema. Desta forma, pode-se concluir que,

devido as suas características, ela não fará parte deste processo de mapeamento e sim poderá contribuir na definição do esquema ou do projeto de interface homem-máquina;

b) **Visão Lógica**: esta visão descreve como a funcionalidade do sistema será implementada e especifica a **estrutura** estática do sistema. Ela é descrita pelos diagramas de classes e objetos, sendo alvo de mapeamento UML - Java;

c) **Visão de Componentes**: ela descreve a implementação dos módulos que farão parte do sistema e suas dependências. Sua representação está descrita pelos diagramas de componentes. Os diagramas de componentes representam aspectos (ligação entre bibliotecas, arquivos de configuração, entre outros) que caracterizam o projeto do sistema computadorizado. Desta forma, pode-se justificar a ausência da visão de componentes no processo de mapeamento de UML – Java;

d) **Visão de Concorrência**: ela representa a divisão do sistema em processos e processadores, sendo suportada pelos diagramas de estado, seqüência, atividade, colaboração, implantação e de componentes. Esses dois últimos simbolizam características de projeto portanto não sendo alvo de tradução;

e) **Visão de Organização**: ela corresponde a representação da **organização** física do sistema, os computadores, periféricos e como eles se conectam entre si. Ela possui impacto no momento em que será realizado a implantação do sistema. Por isso a visão de organização física do sistema não será alvo de tradução.

Como as visões "escondem" a representação dos diagramas, pode-se concluir que o processo de mapeamento pode ser visto como um mecanismo de correspondências entre os componentes (diagramas) dos modelos (visões) de representação e as características presentes na linguagem Java.

A Tab. 1, baseada em [FUR98], apresenta uma relação de equivalências entre a notação UML e Java, imprescindível no processo de mapeamento.

Elemento UML	Java
Classe	Palavra reservada class
Relacionamento de generalização	Palavra reservada super
Relacionamento de especialização	Palavra reservada extends
Relacionamento de dependência classe	Palavra reservada import
Relacionamento de dependência método	Cria-se um método dependente. A derivação de objeto deve ser derivada da classe que armazena o método determinante (não dependente)
Relacionamento de associação	Inserir atributo chave na classe
Relacionamento de associação com navegabilidade	Inserção de atributo chave de acordo com a navegabilidade especificada
Relacionamento com cardinalidade	Inserção de atributo de acordo com a cardinalidade
Atributo constante	Final estático
Atributo de classe	Variável de classe – estático
Atributo de objeto	Variável de instância
Concorrência	Classe/objeto de concorrência com *thread*
Evento de envio	Chamada de método
Exceção	Classe/objeto de exceção com *throw/try/catch*
Mensagem	Invocação de método
Objeto	Objeto
Serviço/Operação de classe	Método de classe – estático
Serviço/Operação de objeto	Método de objeto – instância
Pacote	Package
Associação/Agregação	Variável de instância que une objetos
Interface	Implements
Classes abstratas	Abstract
Métodos abstratos	Abstract
Classe final	Final

Tabela 1 - Correspondências de UML-Java, baseada em [FUR98]

Para a efetivação do processo de transformação deve-se estabelecer um conjunto de regras que devem ser seguidas para o sucesso do processo em si. Pode-se expressar estas regras através de meta-modelos. Eles descrevem a forma pela qual os conceitos de

um modelo são transportados para Java. A Fig. 37 expressa o meta-modelo, baseado em [GCA00], para transformação UML – Java.

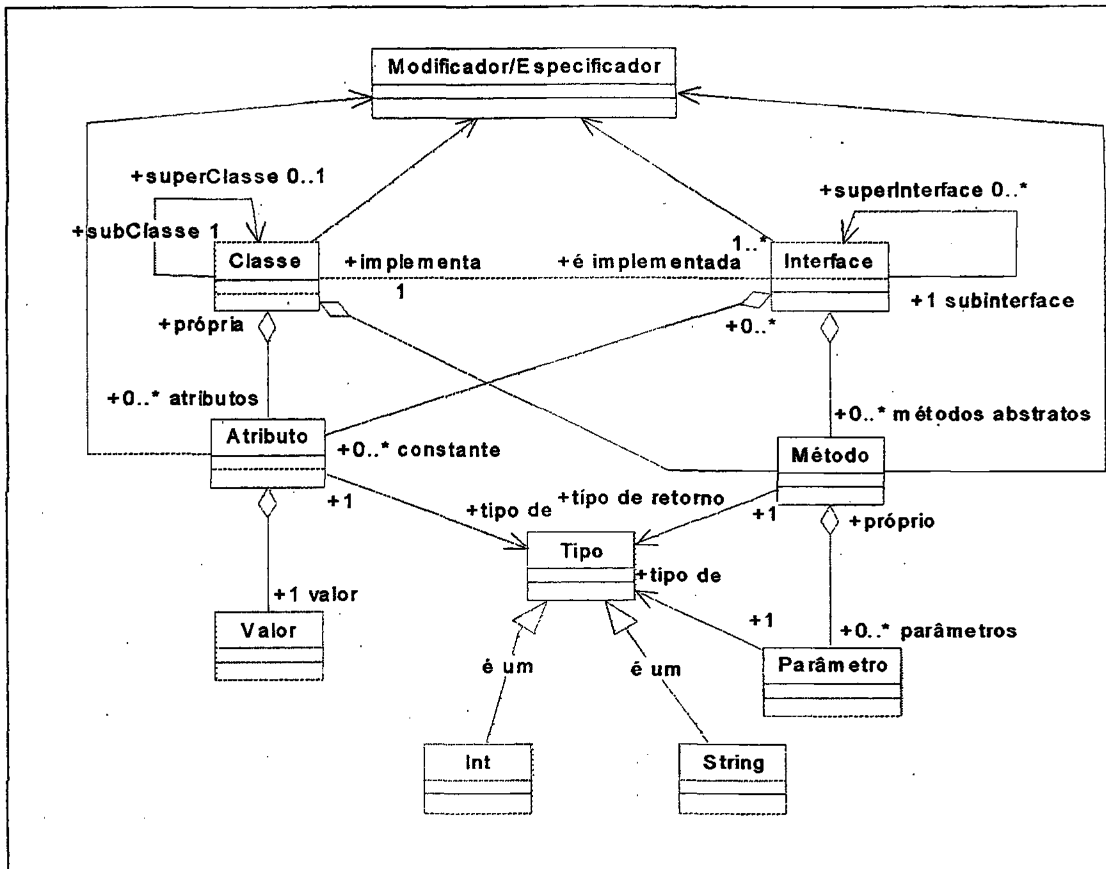


Figura 37 - Meta-modelo UML-Java, baseado em [GCA00]

A partir do meta-modelo definido acima, pode-se criar um conjunto de regras formais que transformam modelos em código Java. Ao conjunto de regras responsável pela transformação chamar-se-á de templates UML-Java.

Para cada componente do meta-modelo deve-se criar um template correspondente.

Elemento	Template
Classe	<UML:classe> id: string; modificador: <UML:Modificador>; lista de subClasse: <UML:classe>; superClasse: <UML:classe>; lista de interface: <UML:interface>; lista de métodos: <UML:método>; lista de atributos: <UML:atributo>; <classe:UML>
Interface	<UML:interface> id: string; superInterface: <UML:interface>; lista de métodos: <UML:método>; lista de atributos: <UML:atributo>; lista de classes: <UML:atributo>; modificador: <UML:modificador>; <interface:UML>
Método	<UML:método> classe: <UML:classe>/<UML:interface>; lista de parâmetros: <UML:parâmetro>; tipo: <UML:Tipo>; modificador: <UML:Modificador>; <método:UML>
Atributo	<UML:atributo> classe: <UML:classe>/<UML:interface>; tipo: <UML:Tipo>; valor: <UML:valor> <atributo:UML>
Valor	<UML:valor> int: conjunto dos inteiros; String: conjunto de caracteres; double: conjunto de inteiros sem sinal; float: conjunto dos reais; <valor:UML>
Tipo	<UML:Tipo> int: conjunto dos inteiros; String: conjunto de caracteres; double: conjunto de inteiros sem sinal; float: conjunto dos reais; void: void; <Tipo:UML>
Parâmetro	<UML: Parâmetro> nome: <UMLTipo> <Parâmetro:UML>
Modificador	<UML:Modificador> public/final/abstract/static <Modificador:UML>
Especificador	<UML:Especificador> public/private/protected <Especificador:UML>

Tabela 2 - Definição de templates para conversão UML-Java

A vantagem da definição de templates é a flexibilidade com que o processo de tradução pode ser realizado. Pode-se traduzir para qualquer linguagem que suporte a estrutura dos templates.

A Fig. 38 ilustra a criação do arquivo de templates.

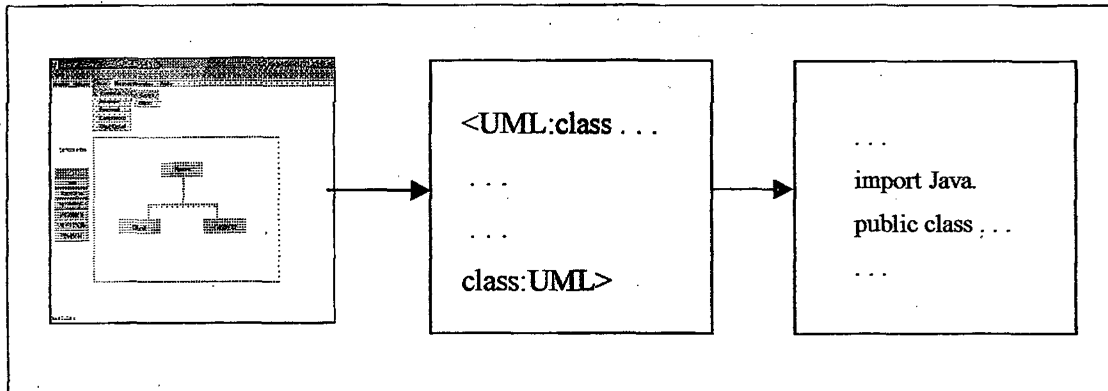


Figura 38 - Esquema de tradução UML - Java

Primeiramente constrói-se um modelo, utilizando um editor de modelos. Num segundo passo, este arquivo de modelos é transformado numa estrutura de representação, ou seja, em templates. Os templates são armazenados numa estrutura de dados intermediária que após é convertida em código Java.

É importante ressaltar que o conjunto de templates definidos não contemplam os relacionamentos. A idéia para transformação de relacionamentos é a utilização direta de correspondências UML-Java, presente na Tab. 1.

4.2.1 Limitações UML - Java

Nesta seção pretende-se detalhar as limitações do processo de tradução. Dentre as limitações pode-se citar:

- O mapeamento de aplicações distribuídas que utilizam de classes e/ou bibliotecas Java que permitem desenvolver aplicações com tais características não serão traduzidas pois, necessitaria de uma análise detalhada da possibilidade de utilização ou construção de ferramentas prontas para verificação de componentes existentes em cada nó. Geralmente indicados em diagramas de implantação;
- Os diagramas de componentes e de implantação não farão parte de mapeamento, pois correspondem a parte de projeto de sistemas computadorizados. Java não possui suporte ao nível de organização física e nem mesmo de detecção da interação entre componentes físicos; e
- vale a pena frisar a enorme quantidade de pacotes e classes Java disponíveis para o estabelecimento e desenvolvimento de aplicações, o que a torna rica. Assim, neste primeiro trabalho, concentrou-se na análise e estudo de modelos estáticos, o que sem dúvida pode ser expandido para a utilização de outras características de Java, tais como: distribuição, redes (implementação de agentes), multimídia, acesso a banco de dados, entre outras.

4.2.2 Processo de Mapeamento

A partir do conhecimento das características da linguagem Java, dos conceitos de orientação a objetos e da percepção das visões alvo de mapeamento, pode-se apresentar uma proposta de processo de transformação de modelos escritos em UML para Java.

Proposta de processo de mapeamento de UML para Java:

Passo 1) Desenvolver o modelo a ser construído através da utilização de uma ferramenta CASE que suporte a notação UML. Como sugestão tem-se em [SUR99] tabela de comparação entre ferramentas CASE.

Passo 2) Obter dados dos modelos criados no passo 1 e armazenar numa estrutura de dados;

Passo 3) A partir dos templates UML-Java, traduzir os dados armazenados na estrutura de dados do passo 2 para código Java, da seguinte forma;

3.1 Para a visão lógica faça:

a) Para componentes de diagramas de classes deve-se utilizar a Tab. 2 de Templates UML-Java para a realização da tradução:

- Classes (concretas, abstratas, estáticas ou finais)
- Interfaces
- Métodos final
- Métodos abstratos
- Relacionamentos (dependência, generalização, especialização, associação, associação com navegabilidade e/ou cardinalidade, agregação, composição);

b) para os diagramas de objetos a tradução deve ser realizada de forma direta em objetos Java.

3.2 Para os componentes presentes nos diagramas de seqüência, colaboração, atividade e estado onde haja a necessidade de implementação de concorrência, utiliza-se a Tab. 1 de correspondências UML – Java para realização da tradução:

a) para os elementos classe/objeto onde houver exceção; e

b) para os elementos classe/objeto onde houver concorrência ;

Passo 4) Os elementos de anotação devem ser codificados de acordo com a sintaxe dos comentários em Java.

Passo 5) Realizar uma análise léxica e sintática para obtenção da validação do processo de tradução;

Passo 6) Criar o arquivo **.class** correspondente aplicação.

4.3 ILUSTRAÇÃO DO PROCESSO DE MAPEAMENTO DE MODELOS UML EM JAVA

Este item faz referência a ilustrações do processo de mapeamento de modelos escritos em UML para Java. Neste sentido, adotou-se para a construção de modelos as seguintes ferramentas CASE: Rational Rose, versão 98, da Rational Corporation e o Argo/UML version 0.7 da University of Califórnia, versões demonstrativas educacionais. Sendo versões demonstrativas, alguns componentes não são instalados e por isso, dependendo da característica da aplicação, alguns diagramas ficam limitados na sua representação aos recursos disponíveis nas ferramentas.

Desta forma, através de estudos de casos, pode-se observar detalhadamente a tradução UML – Java.

4.3.1 Estudo de Caso 1 – Aplicação "Hello World!"

Hello World é uma aplicação trivial, mas interessante sob o ponto de vista de tradução. Ela apresenta uma *String* na tela de um computador, sob a forma de uma Applet, na coluna e linha especificada no método paint da classe Hello. Hello World trabalha com a importação de métodos, classes e pacotes caracterizando um relacionamento de dependência entre esses componentes.

A modelagem para Hello World parte do desenvolvimento do modelo estrutural. Ele é parte fundamental para o início da construção da aplicação. Este modelo está representado pelo diagrama de classes, construído através da utilização do Rational Rose 98.

A seguir, mostra-se o diagrama de classes para a aplicação Hello World.

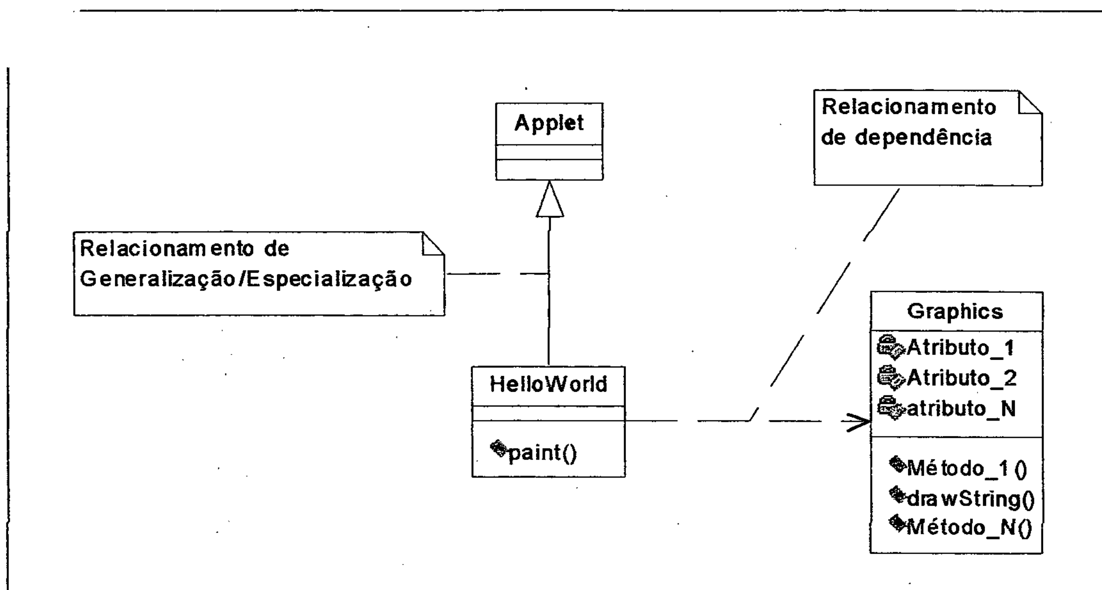


Figura 39 – Diagrama de Classe para o Exemplo Hello World

O diagrama de classes mostra o essencial para o funcionamento da aplicação "Hello World", deixando de fora vários itens particulares das classes Applet e Graphics. Mais referências sobre os pacotes e classes Java podem ser encontradas através de pesquisa na estrutura de bibliotecas de Java.

Analisando o diagrama de classes, percebe-se, claramente, que a classe Hello é uma classe derivada da classe Applet. Isto pode ser observado através do relacionamento de generalização/especialização entre as classes Hello e Applet. A classe Graphics é utilizada na assinatura e na implementação de um dos métodos (paint) da classe Hello. Fato que pode ser percebido através do relacionamento de dependências entre as classes.

Como o método *paint* é parte essencial para a apresentação da mensagem da applet, é interessante observar a seqüência de execução de mensagens e objetos envolvidos até a invocação da operação *paint*. Este detalhamento pode ser percebido e analisado através do diagrama de seqüência a seguir.

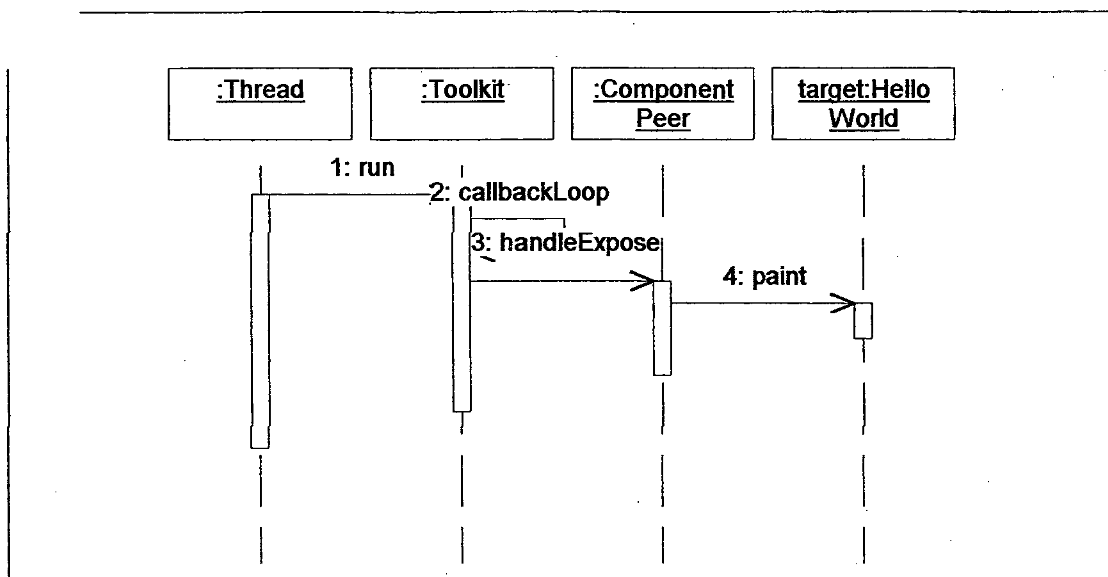


Figura 40 - O mecanismo de operação de *paint*

Para apresentação de uma Applet, é necessário um navegador que execute o objeto Thread da Applet. Entretanto, a classe Hello não é uma página da Web, e sim uma forma binária criada pelo compilador Java. Por isso, é importante a construção da visão de componentes. Neste sentido, abaixo, apresenta-se o diagrama de componentes que representa a visão física de componentes e as relações estabelecidas entre os mesmos.

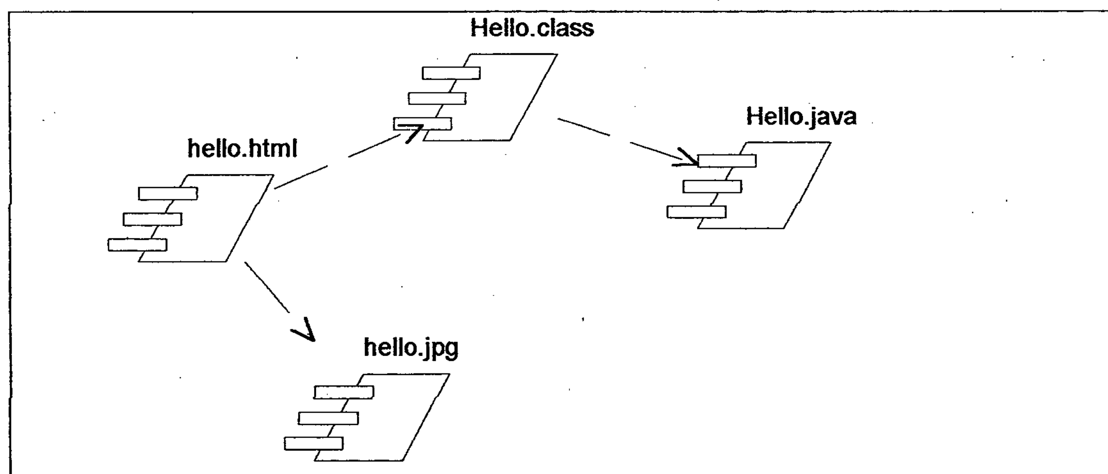


Figura 41 – Diagrama de componentes de HelloWorld

É importante ressaltar que o componente Hello.class, é um componente executável e os demais fazem parte da apresentação da Applet. Infelizmente, a representação é a mesma devido a limitação da ferramenta utilizada para construção do modelo de componentes.

A seguir tem-se a aplicação do processo de mapeamento para HelloWorld.

Passo 1) Desenvolvimento do modelo / montagem das visões lógicas de e concorrência, sendo que o exemplo Hello World não apresenta características de concorrência, não justificando a construção da visão de concorrência.

Passo 2) A execução deste passo deverá ser realiza no momento em que o processo de geração de código esteja automatizado;

Passo 3) Tradução dos modelos para Java

3.1 A estrutura de classes (atributos e métodos) é mapeada conforme estruturação sintática de classes presentes na linguagem Java. A seguir tem-se o código Java para a classe Hello World.

```
...
class Hello extends java.applet.Applet {
 public void paint (Graphics g) {
        g.drawString("HelloWorld!", 10, 10);
 }
...
```

Figura 42 – Fragmento de código Java para Aplicação HelloWorld

3.1 continuação, relacionamentos:

a. Generalização/Especialização existente é mapeado através da utilização do mecanismo de herança presente na linguagem Java. Pode-se observar tal fato através da palavra reservada *extends* que

caracteriza o relacionamento de especialização neste exemplo. Ainda, deve-se incluir a palavra impor java.Applet.*; para acesso ao pacote ao qual será derivada a classe Hello.

b. A Dependência entre as classes pode ser percebida através da observação do diagrama de classes. O relacionamento de dependência é mapeado através da inclusão do pacote ao qual existe a dependência de classes. Neste caso percebe-se a inclusão do pacote *Java.awt* após a palavra reservada *import* e ainda a inclusão do nome da classe (Graphics) que contém o método (drawString) a ser utilizado pelo método *paint* da classe Hello. A Fig. 43 apresenta o fragmento de código Java que representa a inclusão do pacote e da classe participante do relacionamento de dependência.

```
import Java.awt.Graphics;
...
```

Figura 43 – Fragmento de código Java para Relacionamentos de Dependência de Pacotes e Classes

Para completar o processo de mapeamento do relacionamento de dependência entre as classes Graphics e Hello, deve–se inserir na codificação do método *paint* a chamada ao método *drawString* da classe Graphics. A Fig. 44 seguir mostra o código Java para a invocação do método *drawString*.

```
...
 public void paint (Graphics g) {
  g.drawString("HelloWorld!", 10, 10);
 }
...
```

Figura 44 – Fragmento de código Java para Relacionamento de Dependência

A seguir tem-se o código Java completo para o exemplo HelloWorld.

```
import Java.awt.Graphics;
import java.applet.*;
class Hello extends Applet {
  public void paint (Graphics g) {
        g.drawString("HelloWorld!", 10, 10);
  }
}
```

Figura 45 - Código Java completo para aplicação HelloWorld

Passo 4) Os elementos de anotação devem ser codificados de acordo com a sintaxe dos comentários em Java.

Passo 5) Realizar uma análise léxica e sintática para obtenção da validação do processo de tradução;

Passo 6) Criação do arquivo **.class** correspondente à aplicação. Ver Fig. 45 - Código Java completo para aplicação HelloWorld.

4.3.2 Estudo de Caso 2 – Aplicação "Secretaria Eletrônica Hipotética"

O estudo de caso 2 diz respeito a modelagem para uma secretaria eletrônica hipotética. Neste exemplo é importante ressaltar o comportamento mecânico da secretaria que será modelado através de uma notação dinâmica.

Como características funcionais da secretaria tem-se:

- ligar aparelho;
- desligar aparelho;

- reproduzir mensagens gravadas;
- gravar mensagens; e
- rebobinar a fita.

A partir dos requisitos tem-se condições de iniciar a construção da modelagem da secretaria eletrônica hipotética. O primeiro passo é a construção da classe que, internamente, conterá atributos e métodos da secretaria.

Neste exemplo, utilizou-se o Rational Rose para a construção dos diagramas de classe e de estados.

Em conformidade com as funcionalidades apresentadas, a seguir mostra-se, a estrutura da classe e em seguida o diagrama de estados de uma secretaria eletrônica hipotética, juntamente com um diagrama de use-case que traduz a funcionalidade da secretaria eletrônica hipotética.

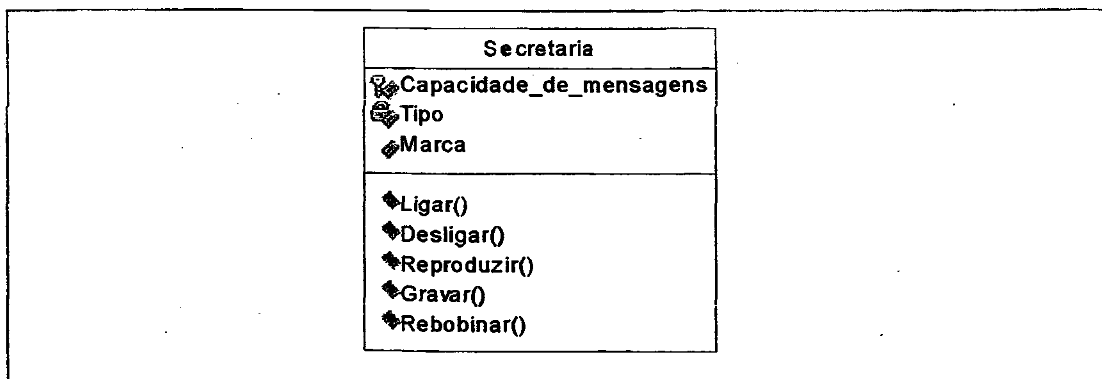


Figura 46 - Diagrama de classe para uma Secretaria Eletrônica Hipotética

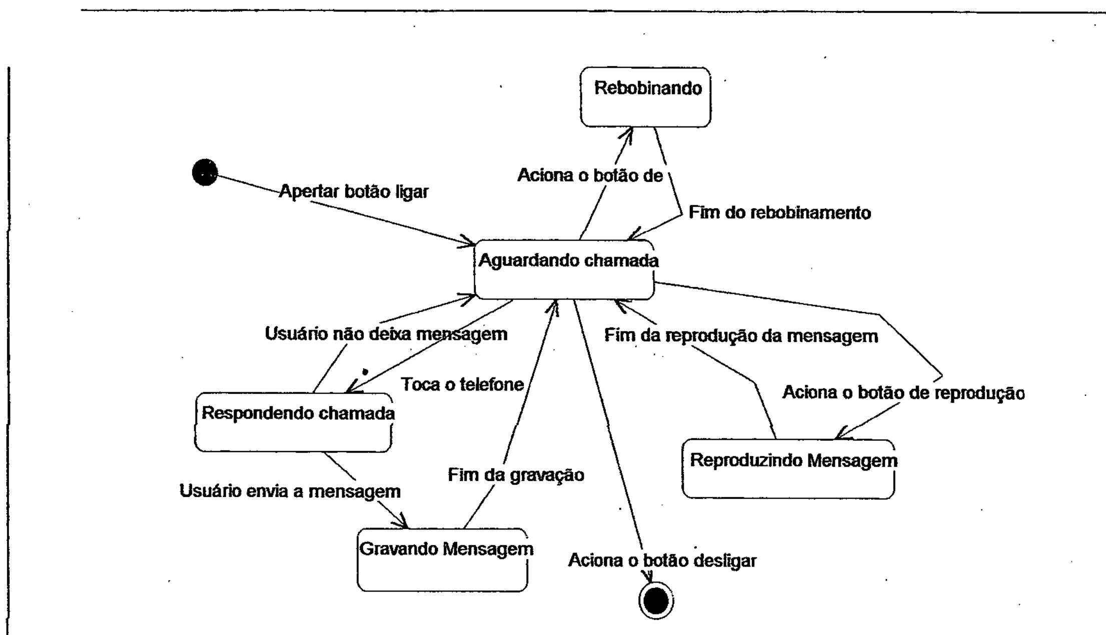


Figura 47 - Diagrama de Estados para uma Secretaria Eletrônica Hipotética

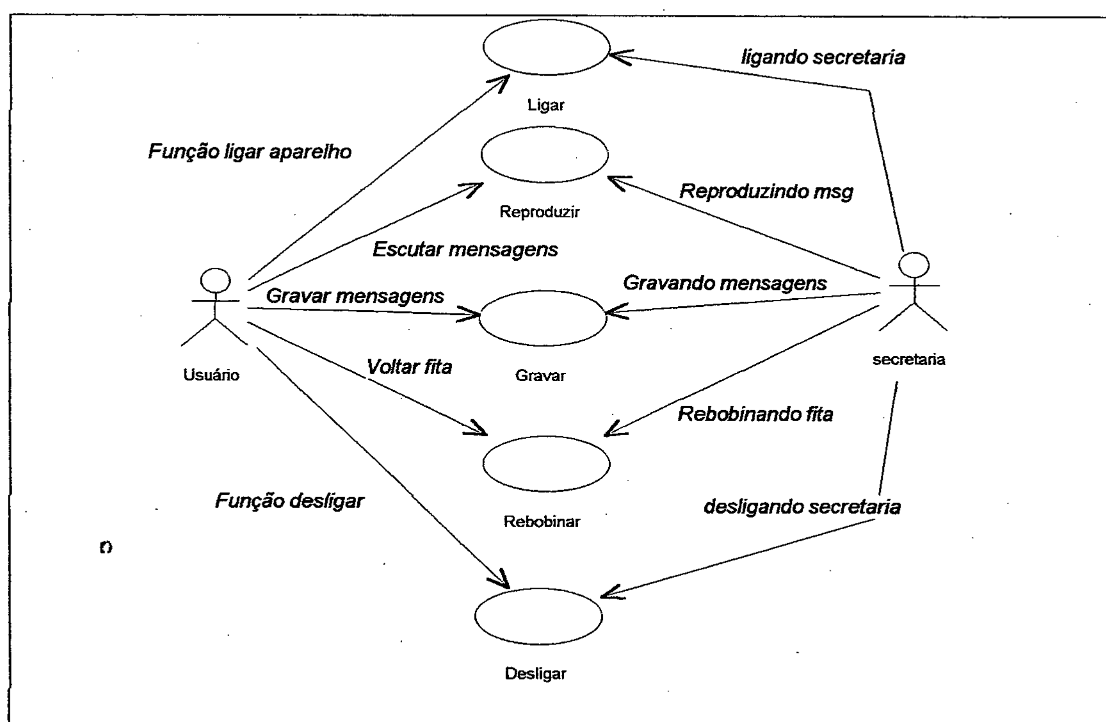


Figura 48 - Diagrama de Use-case para Secretaria Eletrônica Hipotética

Neste exemplo, percebe-se que o diagrama de estados fornece subsídios para o entendimento do comportamento da classe secretaria. Assim, o processo de

mapeamento ficaria restrito ao estabelecimento de correspondências da classe secretaria na codificação da linguagem Java. Processo equivalente e direto a mapear classes para a linguagem Java.

A seguir tem-se a aplicação do processo de mapeamento para Secretaria Eletrônica.

Passo 1) Desenvolver a modelagem da aplicação

Passo 2) A execução deste passo deverá ser realiza no momento em que o processo de geração de código esteja automatizado;

Passo 3) Tradução dos modelos para Java

3.1 para a visão lógica:

a) a Fig. 49 – código Java para a aplicação Secretaria Eletrônica, apresenta o resultado da tradução de classes para Java;

3.2 não aplicável devido a inexistência de concorrência

Passo 4) Não há elemento de anotação

Passo 5) A execução deste passo deverá ser realiza no momento em que o processo de geração de código esteja automatizado;

Passo 6) a seguir tem-se arquivo **secretaria.class**

```
// Arquivo fonte secretaria.class
public class secretaria {
   // atributos
   public int capacidade;
   public tipo string;
   public marca string;

   // métodos
   void secretaria ( ) {
         // código Java para implementação do método construtor - classe secretaria
   }
   public void desligar ( ) {
         // código Java para implementação deste método
   }
   public void produzir ( ) {
         // código Java para implementação deste método
   }
   public void rebobinar ( ) {
         // código Java para implementação deste método
   }
   public void gravar ( ) {
         // código Java para implementação deste método
   }
}
```

Figura 49 - Código Java para a classe Secretaria Eletrônica Hipotética

Já o diagrama de use-case poderá ser utilizado para a definição da interface homem-máquina, pois trata e representa as funcionalidades que a secretaria eletrônica deve prover.

Conclui-se que, o mapeamento de classes para Java é direto não interessando a natureza da entidade modelada, seja de comportamento dinâmico ou estático. Isto vem a reforçar a característica estrutural das classes e dinâmica dos objetos, pois são através deles que se percebe a mudança dos estados.

Percebe-se que, o mapeamento de classes que não estejam relacionadas com outras classes seja por associação, dependência ou outros é realizado de forma direta para Java.

4.3.3 Estudo de Caso 3 – "Sistema de Contas Bancárias Hipotético"

Este estudo de caso trata de um Sistema de Contas Bancárias Hipotético. O estudo é iniciado a partir do relato dos requisitos do sistema. São eles:

a) O sistema suportará cadastro de clientes, onde cada cliente poderá possuir várias contas;
b) As contas podem ser de 2 tipos: correntes ou poupanças. Entende-se por conta corrente aquela em que o cliente pode realizar operações de crédito e débito a qualquer tempo e conta poupança aquela que possui prazo determinado, a partir da data da abertura, para taxa de juros; desta forma, determinando o seu rendimento após um período de 30 dias;
c) Toda a conta, seja ela poupança ou corrente, deverá possuir um número e também um código de agência, bem como um histórico de movimentações (débito, crédito ou transferência).

De acordo com as funções do nosso sistema, tem-se 2 entidades que desempenharão as funções acima descritas. São elas: Cliente e o Funcionário do Banco. As funções que cada entidade poderá desempenhar são expressas através de diagramas use-cases, como é mostrado a seguir.

São atribuições do funcionário do banco:
a) Abrir conta corrente;
b) Abrir conta poupança;
c) Fechar conta corrente;
d) Fechar conta poupança;
e) Cadastrar cliente;
f) Cadastrar agência;
g) Cadastrar Operação

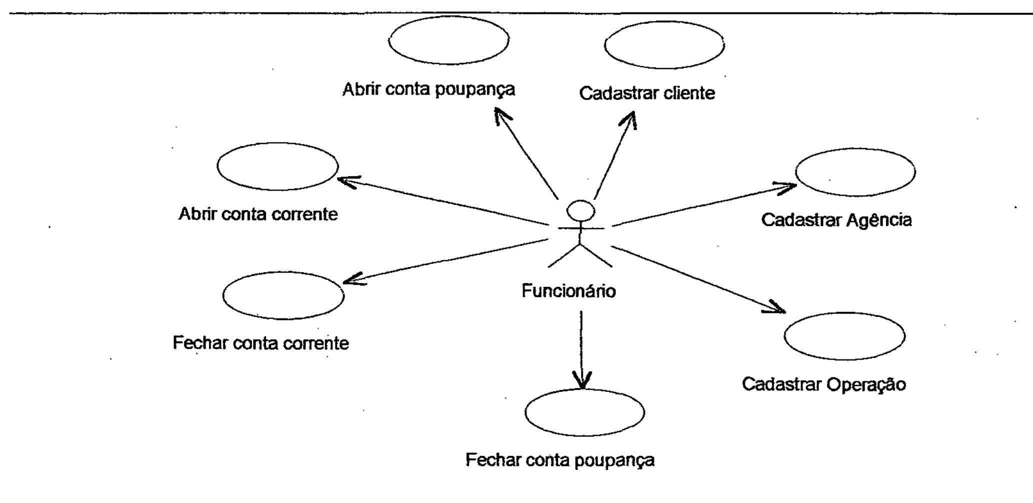


Figura 50 - Diagrama de Use-case - Funções do funcionário do banco

São funcionalidades exercidas pelos clientes:

a) Consultar saldo de conta corrente;
b) Consultar saldo de conta poupança;
c) Consultar extrato de conta corrente;
d) Consultar extrato de conta poupança;
e) Realizar transferências;
f) Movimentar a conta (crédito ou débito).

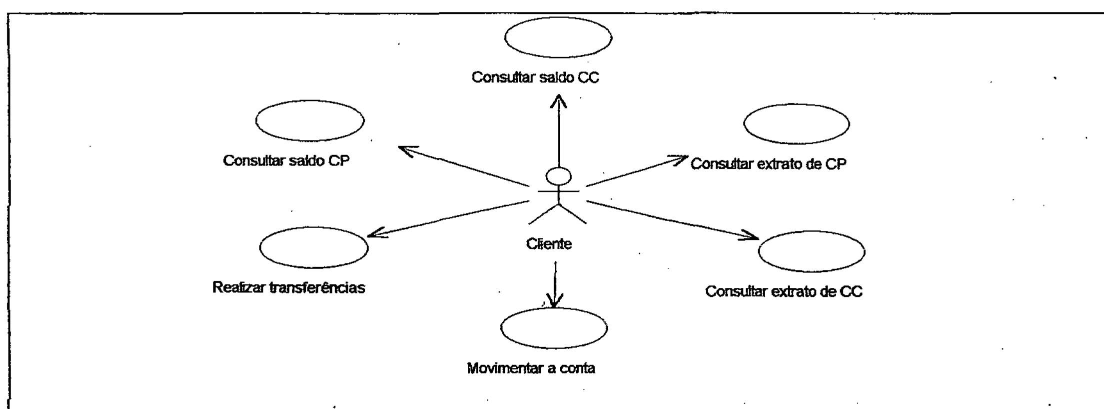


Figura 51 - Diagrama Use-case - Funções do Cliente

Analisando-se o conjunto de requisitos e de funcionalidade, percebe-se que haverão 6 (seis) classes envolvidas na modelagem do sistema. E estarão relacionadas como é mostrado no diagrama de classes a seguir:

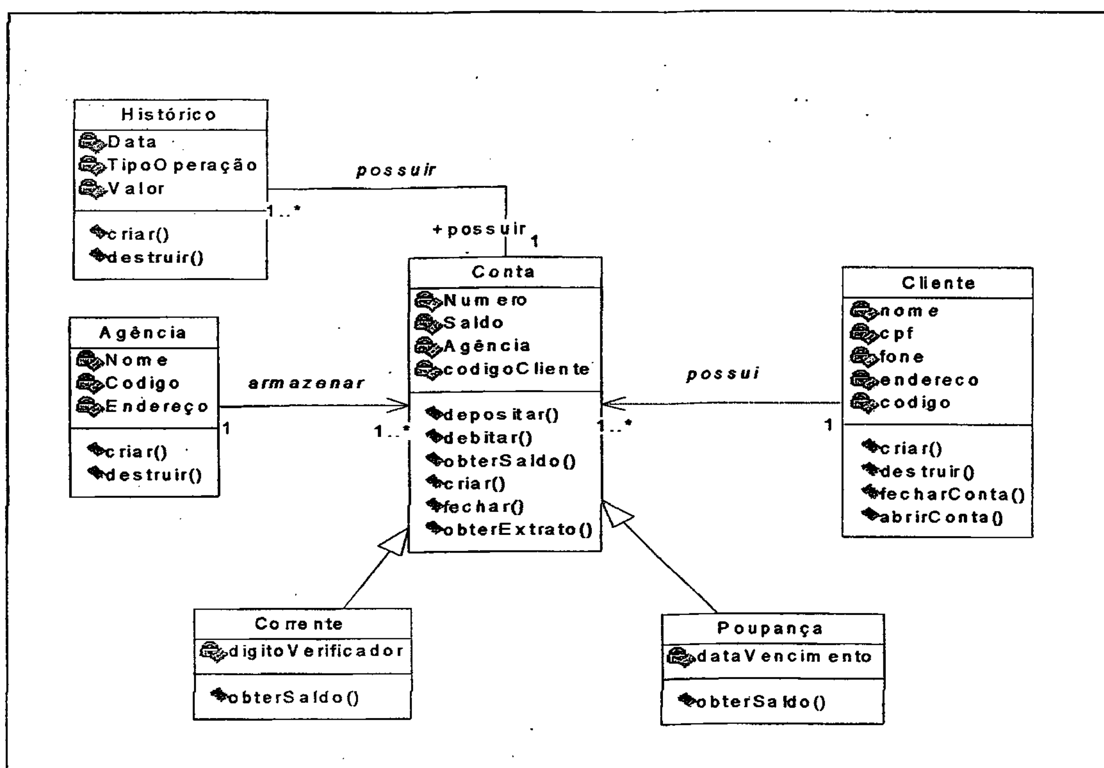


Figura 52 - Diagrama de Classes para Sistema de Contas

A interação de objetos no sistemas é realizada mediante a construção dos diagramas de seqüência ou de colaboração. Geralmente, utiliza-se o diagrama de seqüência para obtenção do tempo em relação a troca de mensagens entre os objetos. Por outro lado, utiliza-se o diagrama de colaboração quando se deseja entender a colaboração entre os objetos no contexto de um caso de uso. Optou-se, nesta aplicação, pela criação de diagramas de seqüência para cada use-case. Nos diagramas, abaixo, pode-se perceber a seqüência de mensagens trocadas ao longo do tempo para cada use-case do sistema.

Em virtude da limitação do número de use-cases pela ferramenta utilizada (Rational Rose demonstrativa), optou-se por apresentar 2 diagramas de seqüência, um para use–case envolvendo cliente e outro envolvendo funcionário.

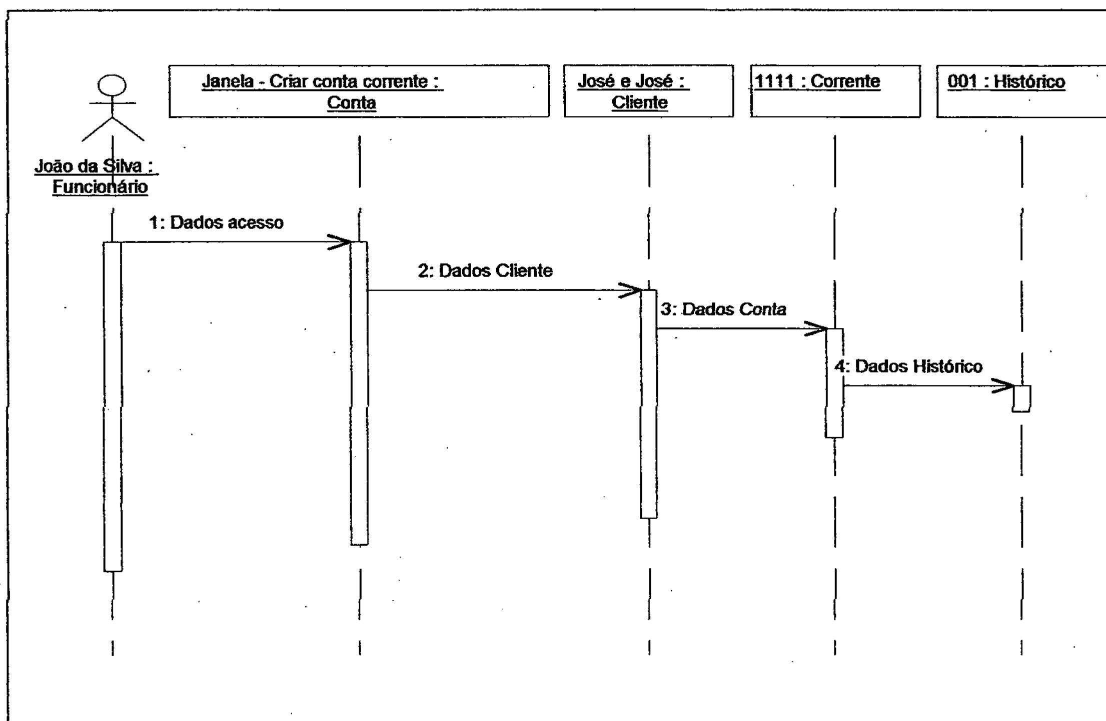


Figura 53 - Diagrama de seqüência para criação de conta corrente – funcionário

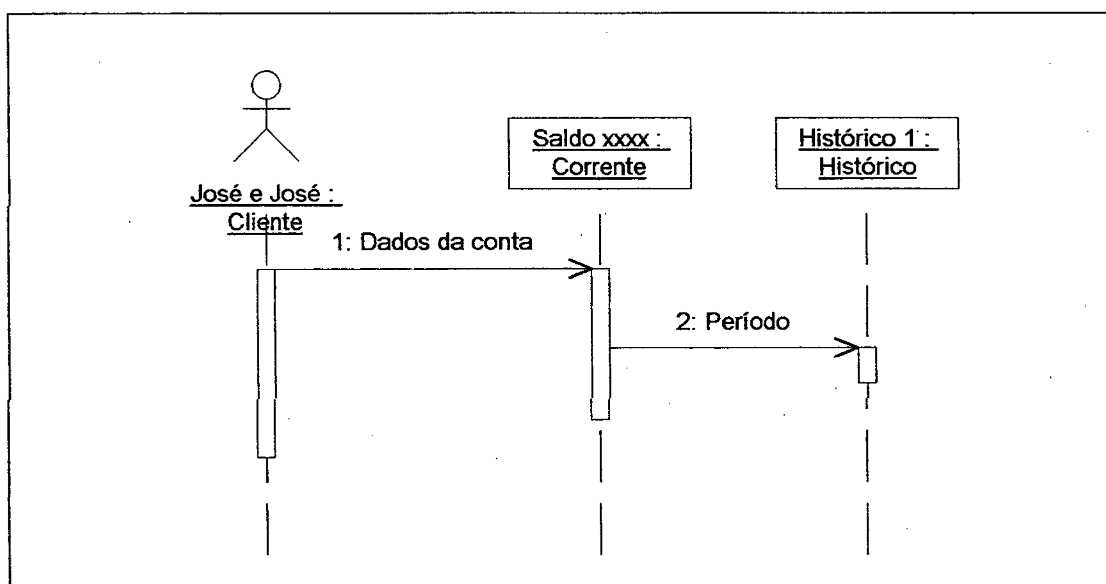


Figura 54 - Diagrama de Seqüência para obtenção de saldo em conta corrente – cliente

A seguir tem-se a aplicação do processo de mapeamento para Sistema de Contas de um Banco.

Passo 1) Construção do modelo

Passo 2) A execução deste passo deverá ser realiza no momento em que o processo de geração de código esteja automatizado;

Passo 3) Tradução

É importante ressaltar neste exemplo a ênfase no mapeamento de relacionamento do tipo associação com navegabilidade e/ou cardinalidade.

Os relacionamentos do tipo generalização/especialização são codificados a partir das inserção das palavras reservadas super e extends em java.

Por outro lado, os relacionamentos de associação são mapeados inserindo o atributo chave na classe de acordo com a semântica expressada pela navegabilidade e/ou pela cardinalidade da associação. Na aplicação em questão, para o relacionamento por associação entre as classes cliente e conta, o atributo chave que determina a semântica da associação é o código do cliente. Ele deverá estar presente na classe conta para realização da relação.

O fragmento de código Java para codificação de relacionamentos de associação seria o seguinte:

```
...
public class cliente {
        String nome;
        int codigo;
        String cpf;
        String fone;

        // a seguir os métodos da classe cliente
}
...
```

Figura 55 - Fragmento de código Java para associação em cliente

```
...
public class conta {
   String numero;
   int codigoCliente;   // atributo que estabelece a associação entre cliente e conta
   int saldo;
   String agencia;

   // a seguir os métodos da classe cliente
}
...
```

Figura 56 - Fragmento de código Java para associação em conta

A tradução para os relacionamento de generalização/especialização entre Conta e Corrente - Poupança são mapeados de acordo com a sintaxe Java que implementa a herança.

```
...
public class Corrente extends Conta {
        int digitoVerificador;

        public void obterSaldo( ); // método polimórfico
}
...
```

Figura 57 - Fragmento de código para especialização de conta

```
...
public class Poupanca extends Conta {
        date dataVencimento;

        public void obterSaldo( ); // método polimórfico
}
...
```

Figura 58 - Fragmento de código para especialização de Conta

Para os relacionamentos por associação entre as classes Agência – Conta e Histórico – Conta a codificação para Java é semelhante ao processo descrito para associação entre Conta e Cliente. Diferenciando apenas em relação a classes envolvidas, a navegabilidade estabelecida e a cardinalidade inserida.

Passo 4) Não há elementos de anotação;

Passo 5) A execução deste passo deverá ser realiza no momento em que o processo de geração de código esteja automatizado;

Passo 6) Arquivo **Conta.class**

```
// Arquivo fonte Conta.java
public class Conta {
    String numero;
    int codigoCliente;
    double saldo;

    Conta( ) { numero = 000; saldo = 20; }

    public void depositar( ); {
            // implementação do método depositar
    }
    public void debitar( ); {
            // implementação do método debitar
    }
    public void obterSaldo( ); {
            // implementação do método obterSaldo
    }
    public void criar( ); {
            // implementação do método criar
    }
    public void fechar( ); {
            // implementação do método fechar
    }
    public void obterExtrato( ); {
            // implementação do método obterExtrato
    }
}
// Arquivo fonte Corrente.java
public class Corrente extends Conta {
        Corrente( ) { super( ); }
        public void obterSaldo( ) { // código específico }
}
// Arquivo fonte Poupanca.java
public class Poupanca extends Conta {
        Poupanca( ) { super( ); }
        public void obterSaldo( ) { // código específico}
}
```

Figura 59 - Código para as classes conta, corrente e poupança da Aplicação Contas

Como fechamento do estudo de caso cabe o comentário a respeito dos relacionamentos por associação e generalização/especialização que são mais freqüentes em aplicações que envolvem a movimentação de dados. Neste sentido, observa-se que a inserção de atributos em classes associadas aponta para algumas questões(centralizado-

distribuído, chaves-índices, modelo de banco de dados, entre outros) de projeto de banco de dados que devem ser consideradas em aplicações desta natureza.

4.3.4 Estudo de Caso 4 – "Sistema de Informações Escolares Hipotético"

Nesta aplicação pretende-se enfatizar a codificação de relacionamentos por agregação e composição. Por isso escolheu-se um sistema de informações escolares hipotético baseado em **[BOO00]** que, de acordo com as especificações abaixo, implementa os relacionamentos citados acima.

A seguir, é apresentado o conjunto de requisitos do sistema de informações relativo a uma escola.

Como requisitos, para o sistema de informações escolares, têm-se:

a) Alunos freqüentam cursos, onde cada aluno pode freqüentar vários cursos e cada curso poderá ter vários alunos;
b) Os professores ministram cursos. Para cada curso, há pelo menos um professor e todo professor poderá ministrar zero ou mais cursos;
c) Uma escola tem zero ou mais alunos, cada aluno poderá estar registrado em uma ou mais escolas, uma escola tem um ou mais departamentos, cada departamento pertence exatamente a uma única escola;
d) A escola possui alunos e departamentos, ou seja, eles são as partes que compõem a escola e não existem fora dela;
e) Todo professor está alocado em um ou mais departamentos, significando que a estrutura de departamento é composta pelo conjunto de professores;
f) Para cada departamento existe somente professor que é responsável (chefe), sendo possível que um professor possa ser responsável por apenas um departamento e alguns professores não são responsáveis por nenhum departamento.

Abaixo, apresenta-se o diagrama de classes que modela os requisitos estruturais dos sistema de informações escolares.

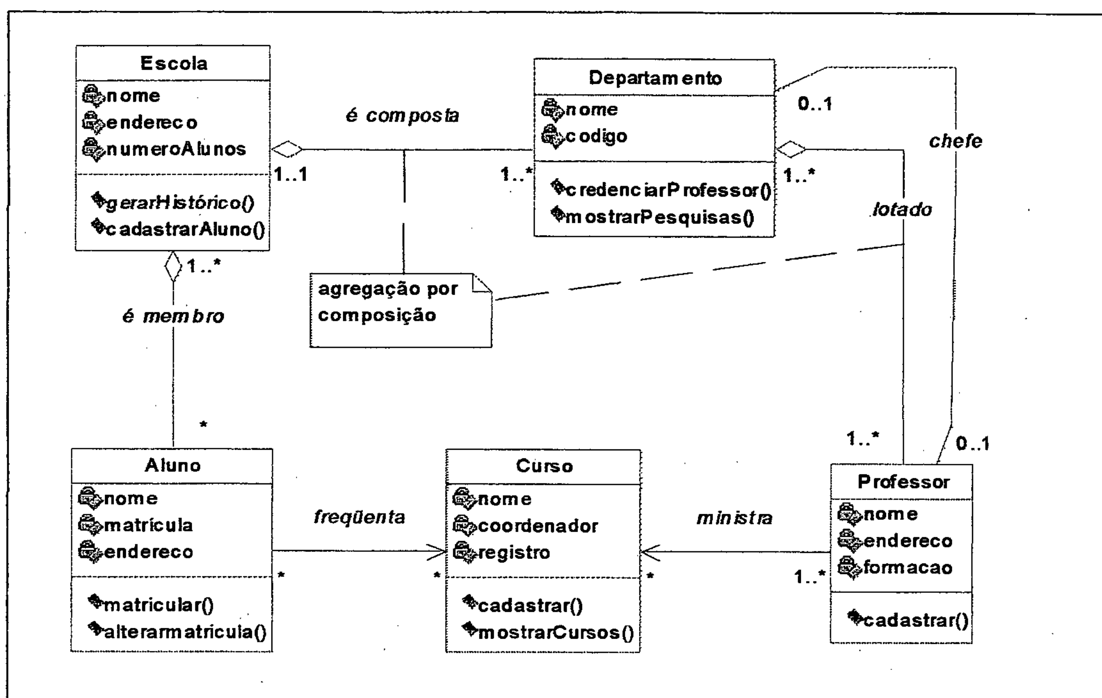


Figura 60 - Diagrama de classes para Sistema de Escola hipotético

Como funções, que o sistema deverá prover, têm-se:

a) A escola deverá gerar histórico conforme solicitação de alunos;
b) A escola deverá prover um mecanismo de cadastro (inclusão, alteração, exclusão e consulta) de dados de alunos;
c) O departamento deverá credenciar professores para ministrar aulas nos cursos;
d) O departamento deverá apresentar um relatório de pesquisas realizadas por professor lotado;
e) O professor terá a possibilidade de cadastro de seus dados;
f) O aluno terá condições de matricular-se e alterar a sua matrícula através do sistema;
g) O sistema fornecerá condições para o cadastramento de cursos bem como relatório de cursos cadastrados.

A seguir, através do diagrama de use-case, pode-se perceber a representação das funcionalidades do sistema.

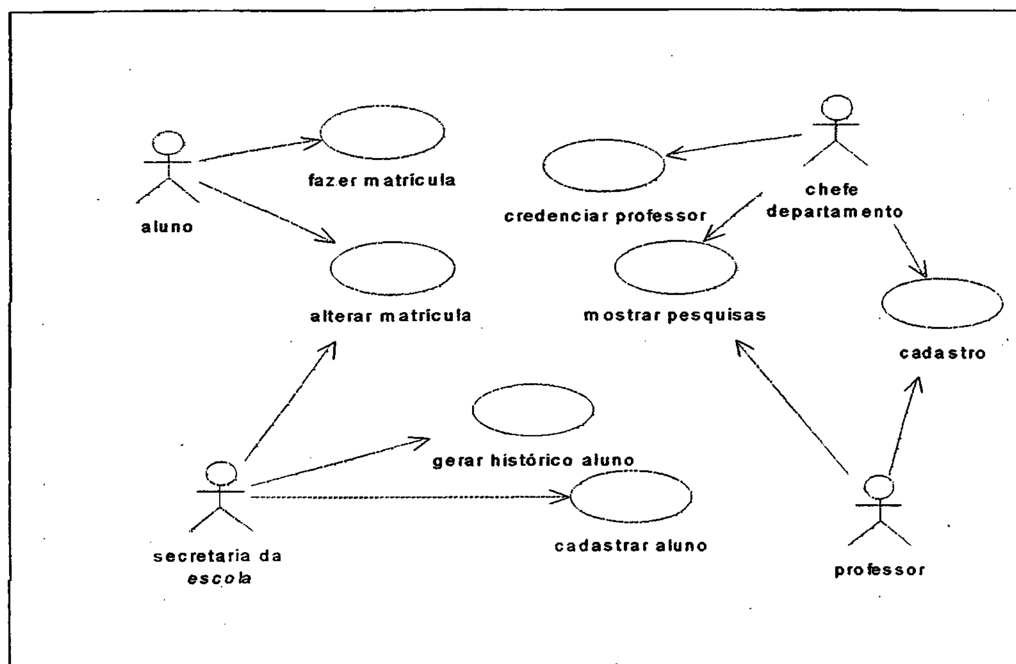


Figura 61 - Diagrama Use-case para o Sistema de Informações Escolares Hipotético

A seguir tem-se a aplicação do processo de mapeamento para Sistema de Informações Escolares.

Passo 1) Construção do modelo

Passo 2) A execução deste passo deverá ser realiza no momento em que o processo de geração de código esteja automatizado;

Passo 3) Tradução

Para os relacionamentos por composição, quando um elemento não depende da criação de outro elemento, o processo será de mapear os objetos de acordo com a Tab. 1 de correspondências UML – Java. Por outro lado, quando ocorrer relacionamento por agregação deve-se ligar o objeto dependente, através de um atributo de ligação, ao

objeto detentor da agregação. Esta ligação deve ter o sentido parte->todo. O fragmento, a seguir representa como seria a codificação Java para relacionamentos por agregação.

```
...
public class Aluno {
        String nome;
        String endereco;
        int matricula;
        Escola ligacao; // atributo responsável pelo relacionamento do tipo agregação

        public void matricular( ) { // implementação do método
        }
        public void alterarMatricula( ) { // implementação do método
        }
}
...
```

Figura 62 - Fragmento de código Java para relacionamento do tipo agregação

Passo 4) Os elementos de anotação são convertido em comentários em Java;

Passo 5) A execução deste passo deverá ser realiza no momento em que o processo de geração de código esteja automatizado;

Passo 6) Arquivo **.class** para cada uma das classes envolvidas na modelagem. Ver Fig. 62 - Fragmento de código Java para relacionamento do tipo agregação.

4.3.5 Estudo de Caso 5 – "Jogo de Dados - Craps"

Este estudo de caso, baseado em **[DEI98]**, pretende ilustrar a tradução de classes abstratas, atributos final e interfaces para Java.

O jogo de dados funciona da seguinte forma: um jogador joga 2 dados, cada dado possuindo 6 lados. Onde as faces dos dados contém números consecutivos de 1 a 6. Após os dados serem jogados pelo jogador, a soma dos lados de cima são calculadas. Se a soma for 7 ou 11 na primeira jogada, o jogador é vencedor. Se a soma for 2,3, ou 12 na primeira jogada ocorreu "craps", ou seja, a "casa" ganha. Se a soma for 4,5,6,8,9, ou 10 na primeira jogada, então ela torna-se pontos para o jogador. Para vencer, é necessário que os pontos do jogador sejam iguais à soma dos dados. Caso a soma dos dados for igual a 7 o jogador perde após primeira jogada.

A Fig. 63 apresenta a tela de execução do aplicativo dados.

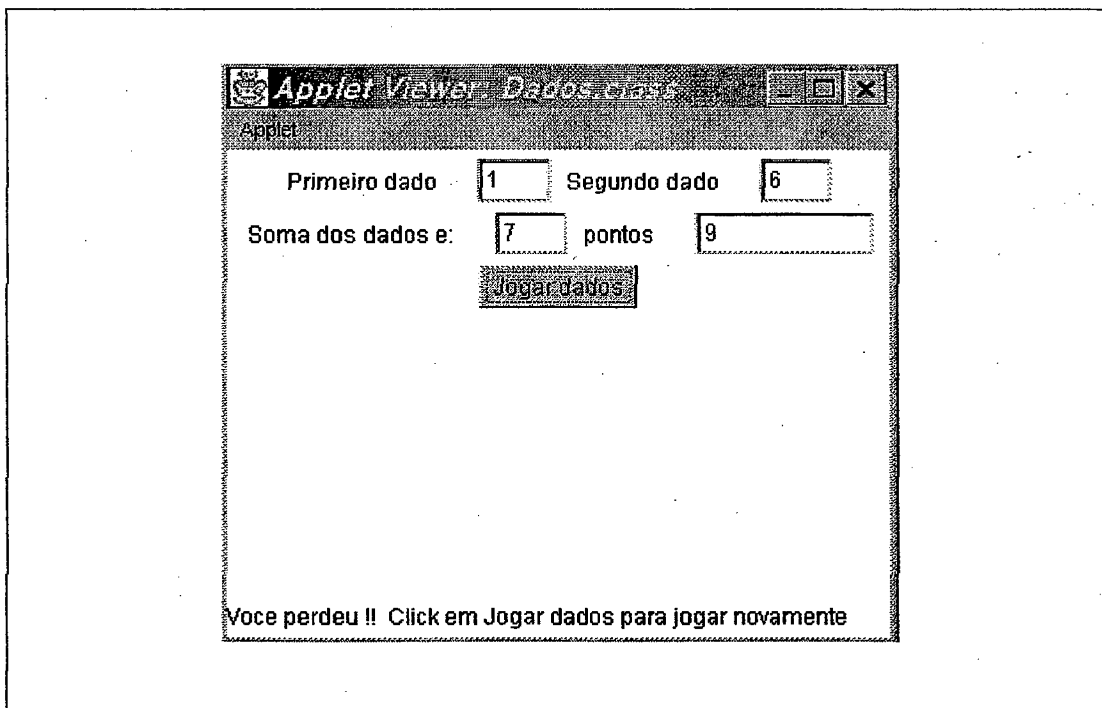


Figura 63 - Tela de execução do aplicativo dados

A seguir tem-se a aplicação do processo de mapeamento para o jogo de dados – "craps".

Passo 1) Construção do modelo

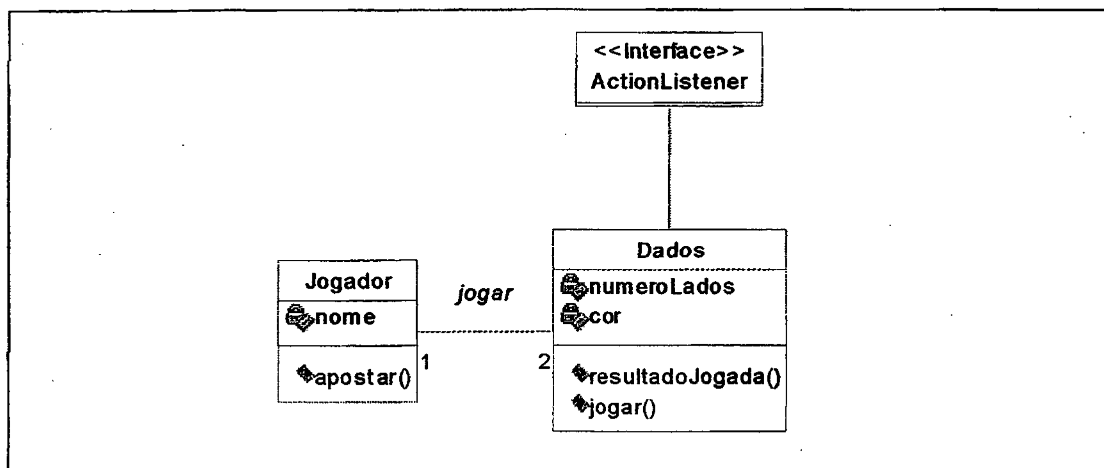


Figura 64 - Diagrama de classes para o exemplo do jogo de dados

Passo 2) A execução deste passo deverá ser realiza no momento em que o processo de geração de código esteja automatizado;

Passo 3) Tradução

Num primeiro momento deve-se inserir os pacotes integrantes das implementações das classe da aplicação jogos de dados. A inserção da palavra reservada import seguida da hierarquia de pacotes é suficiente para o estabelecimento de vínculos entre as classes do jogo de dados e as classes pré-defindas de Java. Pode-se perceber a inclusão dos pacotes da linguagem a partir da Fig. 65.

```
import java.awt.*;
import java.applet.Applet;
import java.awt.event.*;

. . .
```

Figura 65 - Fragmento de código Java com inserção de pacotes e classes para o Jogo de dados

Num segundo momento far-se-á a definição da classe que será responsável pelo gerenciamento da aplicação do tipo applet. Esta classe deverá ser implementada a partir de uma interface que comanda as ações de eventos ocorridos. Para tanto, deve-se inserir, junto a definição de classe, a palavra reservada implements seguido do nome da interface.

Num terceiro momento utiliza-se a referência atributo constante conforme Tab. 1 de correspondência UML – Java para mapeamento dos atributos final WON, LOST e CONTINUE. A Fig. 66 representa o código Java para atributos final.

```
. . .
final int WON = 0, LOST = 1, CONTINUE = 2; // constantes
. . .
```

Figura 66 - Fragmento de código para atributos final

Num quarto momento deve-se criar um método chamado de init( ) que será responsável pela inicialização e criação dos objetos que farão parte da applet. O fragmento de código a seguir ilustra a implementação do método init( ).

```
. . .   public void init () {

            die1Label = new Label(" Primeiro dado ");
            add(die1Label);
            firstDie = new TextField(2);
            firstDie.setEditable(false);
            add(firstDie);

            die2Label = new Label(" Segundo dado ");
            add(die2Label);
            secondDie = new TextField(2);
            secondDie.setEditable(false);
            add(secondDie);

            sumLabel = new Label( " Soma dos dados e: ");
            add(sumLabel);
            sum = new TextField(2);
            sum.setEditable(false);
            add(sum);

            pointLabel = new Label(" pontos ");
            add(pointLabel);
            point = new TextField(10);
            point.setEditable(false);
            add(point);

            roll = new Button("Jogar dados");
            add(roll);

            roll.addActionListener(this);

    } . . .
```

Figura 67 - Fragmento de código Java para método init( )

Num quinto momento implementa-se o método play( ) que será responsável pela definição da soma dos lados dos dados e pela geração aleatória dos números dos lados dos dados. A Fig. 68 mostra a implementação do método play( ).

```
. . . public void play() {
        if (firstRoll) {
            sumOfDice = rollDice();
            switch (sumOfDice) {
               case 7: case 11:
                  gameStatus = WON;
                  point.setText( "" );
                  break;
               case 2: case 3: case 12:
                  gameStatus = LOST;
                  point.setText( "" );
                  break;
               default:
                  gameStatus = CONTINUE;
                  myPoint = sumOfDice;
                  point.setText(Integer.toString(myPoint));
                  firstRoll = false;
                  break;
            }
         } else {
            sumOfDice = rollDice();

            if (sumOfDice == myPoint ) {
               gameStatus = WON;
               showStatus("Voce venceu numero de pontos igual a soma
dos dados");
            } else
               if (sumOfDice == 7)
                  gameStatus = LOST;
         }
         if (gameStatus == CONTINUE)
            showStatus (" Jogue novamente ... ");
         else {
            if (gameStatus == WON)
               showStatus("Voce venceu !! " +
                          "  Click em Jogar dados para jogar
novamente");
            else
               showStatus("Voce perdeu !! "+" Click em Jogar dados
para jogar novamente");
            firstRoll = true;
         }
      } . . .
```

Figura 68 - Fragmento de código para método play( )

Num sexto momento implementa-se o método actionPerformed( ) responsável pela chamada do método play( ) após a ocorrência do evento *roll.addActionListener(this)*, ou seja, depois que o usuário clicar no botão jogar.

```
. . .       public void actionPerformed (ActionEvent e) {
          play();
      } . . .
```

Figura 69 - Fragmento de código de implementação método actionPerformed( )

Passo 4) Os elementos de anotação são convertido em comentários em Java;

Passo 5) A execução deste passo deverá ser realiza no momento em que o processo de geração de código esteja automatizado;

Passo 6) Arquivo **.class** para cada uma das classes envolvidas na modelagem. Ver Anexo 3 - Código Java – exemplo: Interface e final.

4.4 ENGENHARIA REVERSA JAVA - UML

A engenharia reversa (a criação de modelos a partir de código) do mundo real de volta para os diagramas da UML é de grande valia, interessante em sistemas distribuídos que sofrem alterações ou mudanças constantes.

Desenvolver um mecanismo que implemente engenharia reversa é tão ou mais complicado do que a engenharia de sistemas computacionais, pois teríamos numa primeira análise que criar a seguinte metodologia para efetivação do processo:

- Escolha da linguagem alvo para a engenharia reversa;
- Domínio da gramática da linguagem alvo;
- Armazenamento de dados em estruturas de dados adequadas; e
- Desenvolvimento de um ambiente para geração dos modelos.

Este trabalho não trata de mecanismos ou faz análise para que haja engenharia reversa de Java para a UML, mas pode ajudar em trabalhos futuros que tenham ênfase neste processo.

CAPÍTULO 5. CONCLUSÃO

Para garantir que aspectos de complexidade no desenvolvimentos de sistemas computacionais sejam minimizados é necessário que o desenvolvimento de software seja abordado sob outro prisma. Esta mudança de enfoque, implica necessariamente na utilização de novos conceitos e ferramentas que garantam:

- reutilização de componentes;
- facilidade de percepção do sistema sob aspectos dinâmicos, funcionais e estruturais;
- rapidez e a eficiência de processos que atinjam padrões de garantia da qualidade de software; e
- criação de modelos de representação de sistemas, pois devido a diversidade da criação de modelos e metodologias que vêm prejudicando e acabando causando problemas de interpretação, entendimento, documentação e comunicação no desenvolvimento de sistemas.

Esta mudança de enfoque passa necessariamente pela adoção de um padrão de modelagem de sistemas, a Linguagem de Unificada de Modelagem (UML), que possibilita, dentre outros ganhos, a redução do tempo, de custos, de riscos e acaba com a diversidade de modelos para o desenvolvimento de sistemas. Esse padrão, de notação de sistemas, faz com que os modelos desenvolvidos hoje tenham a mesma nomenclatura/notação amanhã, facilitando assim a produção de software com qualidade.

No entanto, não se pode ocultar as dificuldades para a utilização de conceitos, padrões de notações ou ferramentas que implementam essas fundamentações. Tais complexidades iniciam a partir da fundamentação teórica, do entendimento de conceitos fundamentais do paradigma de programação a objetos, passam pelas filosofias de análise de fluxo de dados, análise por modelagem de dados, análise de sistemas em

tempo real, análise orientada a objetos, pelo conhecimento de linguagens de programação, chegando até padrões de notação de sistemas, como a UML. Ainda não se pode deixar de citar que a UML está em desenvolvimento e agrega um conjunto de conceitos amplos que devem ser perfeitamente entendidos para a utilização de todo o seu potencial.

Para tanto, torna-se imperioso que se desenvolva modelos ou até ferramentas CASE cuja a preocupação esteja em incorporar os conceitos fundamentais de desenvolvimento de sistemas. Desta forma, estar-se-á beneficiando a fase de codificação, que será, cada vez mais, substituída pela geração automática de código.

Assim, os avanços recentes na área de engenharia de software estimulam e permitem muitas expectativas em relação à utilização do enfoque de modelagem por objetos aliado à tecnologia de linguagens de programação no desenvolvimento de sistemas que suportem melhor o tempo agregado a um aperfeiçoamento da sua funcionalidade, permitindo um menor custo de manutenção e uma melhoria da qualidade de software produzido.

Neste enfoque não podemos deixar de referenciar as ferramentas de modelagem de sistemas orientadas a objetos que, além da quantidade reduzida desenvolvida, o alto preço de aquisição e a pouca disponibilidade de suas versões educacionais tornam o trabalho de análise e estudo demorado e complicado.

O objetivo deste trabalho foi de realizar um estudo e análise da possibilidade da criação de um mapeamento de sistemas representados em UML em Java. Ele pretende ter a sua terminalidade numa proposta de mapeamento de modelos UML em Java, que futuramente poderá ser utilizado para implementação de uma ferramenta CASE para estudo e aplicação educacional ou comercial.

Como resultados deste trabalho pode-se citar:

- o desenvolvimento de templates UML-Java;

- a elaboração de estudos de casos para aplicação do processo de tradução;
- as limitações UML-Java e Java-UML; e
- a dificuldade no estabelecimento do mapeamento UML-Java.

Sugere-se como trabalhos futuros:

- a criação editor de modelos;
- melhoria e adição de novos templates para suporte a um conjunto maior de aplicações;
- implementação em alguma linguagem de programação do mecanismo de tradução UML – Java;
- Retomada do estudo e proposta de um trabalho aprofundado realçando a superação da limitações deste trabalho; e
- A efetiva construção de uma ferramentas de geração de código automática em várias linguagens de programação.

Percebe-se então, que este trabalho pode ser continuado no sentido de se obter a construção de uma ferramenta CASE robusta/completa que possa ser utilizada nas mais diversas aplicações. No entanto, é importante destacar que a complexidade, o tempo e o número de pessoas a serem envolvidas são variáveis vitais para análise e projeto de trabalhos futuros que venham a surgir.

Pode-se ainda, observar que, trabalhos tanto em nível de graduação quanto de pós-graduação (especialização ou mestrado) podem ser pensados a partir deste. Pode-se citar, como possíveis trabalhos de graduação, seja de pesquisa científica ou conclusão de curso, os seguintes:

- Projeto e Análise de um Editor de Modelos baseado em UML, contemplando as visões referente a análise do sistema. Neste projeto, o importante é que o editor de modelos gere uma saída que possa ser "entendida" formalmente pelo "motor" ou mecanismo de mapeamento, para que então seja possível a

geração de código. Este editor de sistemas seria um sub-sistema de uma ferramenta CASE futura;

- Um estudo em relação a extensão (ampliação e robustez) dos templates pode ser realizado com alunos de graduação e também de pós-graduação. Este estudo teria como resultados a criação de um tradutor que, a partir das entradas (saídas do editor de sistemas) fosse capaz de mapear numa linguagem destino pré-estabelecida pelo projetista do sistema;
- Outro ponto bastante interessante é a preocupação com a representação total de sistemas, através de modelos, numa linguagem de programação. Um estudo aprofundado poderia responder questões do tipo: qual a percepção do quanto e do que seria "perdido" numa transformação de um modelo para uma linguagem de programação? que pontos ou partes do sistema modelado não teriam a sua representação explícita numa linguagem de programação ? ;
- Um estudo sobre testes de software, poderiam responder questões como: que(ais) teste(s) software seria(m) necessário(s) para o atendimento completo da ferramenta CASE ? que metodologia de testes seria adequada, já que se tem vários sub-sistemas em desenvolvimento ? (editor de modelos ? tradutor(es) ? ferramenta CASE completa ?); e
- Seria importante também que, num futuro trabalho fosse pensado na questão do desenvolvimento de software utilizando a internet como meio para a troca e estabelecimento de pontos de controle e verificação de requisitos. Seria interessante que o tradutor pudesse também gerar código para internet. Isto daria condições para que o projetista do sistema pudesse perceber, visualizar e acompanhar o desenvolvimento do mesmo não tendo que se deslocar até o local do desenvolvimento, através de um controle de acesso em nível de usuário, o projetista poderia alterar o modelo-sistema através da Web. A possibilidade de alteração via

Web dá outra dinâmica e pode economizar tempo de desenvolvimento do sistema, pois os esforços de deslocamento e contato podem ser minimizados.

Enfim, é importante ressaltar que o perfil da equipe seja multidisciplinar, que tenha interesse em trabalhar nesta área do conhecimento, que possua afetividade e interesse pelo estudo de linguagens e seus mecanismos de tradução (compiladores e linguagens formais) e também tenha a percepção dos conceitos que norteiam a engenharia de software.

BIBLIOGRAFIA

[FUR98] FURLAN J. D.
Modelagem de Objetos através da UML.
Ed Makron Books, 1998.

[MAR98] Mark Collins-Cope of Ratio Group.
Object Oriented Analsys and Design Using UML.
http://www.ratio.co.uk/white.html. 1998.

[CON98] CONALLEN J.
Modeling Web Application Design with UML.
http://www.rational.com/uml/resources/whitepapers. 6-Jun-98.

[SUR99] SURVEYER J.
Java and UML.
http://www.devx.com/upload.free/features/javapro.

[UDE99] UDELL J.
Uses of Unified Modeling Language (UML). July 13, 1999.
http://www.webbuildermag.com/upload/free/features/webbuilder.

[ALH98a] ALHIR S. S.
What is the Unified Modeling Language. July 1, 1998.
http://home.earthlink.net/~salhir/whatistheuml.html.

[ALH98b] ALHIR S. S.
Applying the Unified Modeling Language (UML).August 1, 1998.
http://home.earthlink.net/~salhir/applyingtheuml.html.

[NEW97] NEWMAN A. et al.
Usando Java.
Editora Campos. Rio de Janeiro 1997.

[CES99] CESTA A. A.
Tutorial "A Linguagem Java".
Unicamp - Instituto de Computação. 1999.

[SUN95] Sun Microsystems.
The Java Language: An Overview. Sun Microsystems, 1994, 1995.
http://java.sun.com/docs/books/tutoril/index.html.

[HOP97] HOPSON K.C. e INGRAM E. S.
Desenvolvendo Applets com Java.
Rio de Janeiro. Editora Campus, 1997.

[PRA96] PRATT T. W.
Programming Languages: Design and Implementation.
Ed Pretince Hall, 1996.

[GOS96] GOSLING J. e McGILTON H.
The Java Language Environment A White Paper. Maio 1996.
http://java.sun.com/docs/books.

[YOU99] YOURDON, Eduard.
Análise e Projeto Orientado a Objetos.
São Paulo, Makron Books, 1999.

[DEM91] DE MARCO, Tom.
Análise estruturada e especificação de sistemas.
Rio de Janeiro: Campus, 1991. (Série Yourdon Press)

[SHL90] SHLAER, Sally; MELLOR, Stephen J.
Análise de sistemas orientada para objetos.
São Paulo: Mc Graw-Hill, 1990.

[YOU94] YOURDON, Eduard.
Análise estruturada moderna.
Rio de Janeiro : Campus, 1994.

[AMB97] AMBLER, SCOTT W.
Análise e Projeto Orientados a Objeto.
Rio de Janeiro : Infobook, 1997.

[DEI98] DEITEL, Harvey M.; P. J. Deitel.
Java: How to Program.
Pretince-Hall, 1998.

[REZ99] REZENDE, D. A.
Engenharia de software e sistemas de informações.
Rio de Janeiro: Brasport, 1999.

[PRE95] PRESSMAN, Roger S.
Engenharia de Software.
Makron Books, 1995.

[ERI98] ERIKSSON, Hans-Erik & PENKER, Magnus.
UML Toolkit.
Editora Wiley, 1998.

[RAT99] **Rational Corporation.**
Documentação Oficial da UML.
http://www.rational.com/uml.

[SUR99] **Surveyer, Jacques.**
Java and UML
http://www.devx.com/upload/free/features/javapro/1999/10mid99/js1399/js1399.asp

[WAR99] **WARMER, Jos.; KLEPPE, Anneke.**
OCL: The Constraint Language of the UML
http://www.joopmag.com

[FOW00] **FOWLER, Martin & SCOTT Kendal.**
UML Essencial:um breve guia para a linguagem padrão de modelagem de objetos.
Editora Bookman, 2000.

[DOU] **DOUGLAS, POWEL Bruce.**
The UML for Systems Engineering
http://www.ilogix.com

[DOU98] **DOUGLAS, POWEL Bruce.**
UML for Systems Engineering
Computer Design. November 1998.

[BOO00] **BOOCH, Grady; RUMBAUGH James; JACOBSON Ivar.**
Guia do Usuário – UML
Editora Campus, 2000.

[COA91] **COAD, Peter; YOURDON Edward.**
Análise Baseada em Objetos – Série Yourdon Press
Editora Campus, 1991.

[GCA00] **GCA Organization.**
UML to Java as Example
http://www.gca.org/papers/xmleurope2000/papers/s36-02.html

ANEXO 1 – COMPARAÇÃO ENTRE FERRAMENTAS CASE

re	Aonix Software-thru-Pictures 7.1	Excel Software WinA&D 3.0	Oracle Designer 2.1	Popkin Software System Architect 2001	Select Software Select Enterprise	Visio Corp. Visio Enterprise 5.0
diagramming	**Nine UML model types—all products do forward/reverse engineering from class diagrams**					
se, class, bject diagrams	yes/yes/yes	yes/yes/yes	yes/yes/no	yes/yes/yes	yes/yes/no	yes/yes/yes
nce/ oration	yes/yes	yes/yes	yes/yes	yes/yes	yes/yes	yes/yes
hart/ y	yes/yes	yes/yes	no/no	yes/yes	yes/yes	yes/yes
onent/ yment	yes/yes	no/yes	no/no	yes/yes	no/no	yes/yes
diagrams	OMT, IM, Structured	Booch, CRC, Coad/ Yourdon, Fusion, IM, OMT, Shlaer/Mellor	ER	Booch, Coad/ Yourdon, OMT, ER, Shlaer/Mellor	BPM, Process hierarchy and thread, ER, OMT	ORM, ER, ORG Chart, hundreds of others
rd/reverse ages ward only, rse only	C++, Forte-f, IDL, Java, ObjectAda-f	C++-r, Delphi-r, C, Basic, Fortran, Java-f/r, Pascal into Structure	C++-f, PL/SQL*, VB*, JavaScript*, (* forms only)	C++, Delphi-f, IDL-f, Java, PowerBuilder-f, VB-f	C++, Forte, Java, VB-f	third-party
ases f/r ward only, rse only	StP/IM	Oracle-f, Sybase-f	DB/2, Oracle, SQL Server, Sybase,ODBC	AS400, DB/2, Informix, Oracle, Sybase	DB/2, Informix, SQL Server, Oracle, Sybase-fr	Informix, SQLServer, Oracle, Sybase
rated code -customizable)	no	yes	yes	yes, VBA	yes	yes, VBA
x methods/ ort	yes	yes	yes	yes	no	no
ss odology	Objectory, DO178b	ER, IM	ER, UML	Catalysis, Gane/ Sarson, SSADM	Perspectives, CBD	ORM, ER
feature	scripting support	robust CRC support	application generation	metamodel customizing	component object animation	smart objects
ase of use	**Visual diagramming with toolbox, drag-and-drop operation**					
objects her model(s)	yes	yes	yes	yes	yes	yes
hronize d models	yes on save	yes	yes	yes	yes	partial
ow panes/ iagrams	yes/yes	yes/yes	yes/yes	yes/yes	yes/yes	yes/yes
ctVue/ erties sheet	yes/yes	yes/no	yes/yes	yes/yes	yes/yes	yes/yes
ed views/ notes	yes/yes	yes/yes	yes/yes	yes/yes	yes/yes	yes/yes
GUI enience	tailor desktop layout	link to source code and requirements	cue cards and wizards	tabbed project views	built-in guides	diagramming tools and templates
sitory	**For all diagrams, objects, and attributes**					
pository engine	Sybase ASA	proprietary	Oracle	proprietary	Softlab Enabler	proprietary
r repository ort	JetDB	no	no	MS Rep 2	MS Rep, UREP	no
/version rol	yes/third-party	yes/third-party	yes/yes	yes/third-party	yes/yes	yes/third-party
butes yclopedia	yes	yes	yes	yes	yes	no
s scripting uage	C-like proprietary	extensive 3GL API	proprietary	VBA	no	VBA
ut and rface options	**Diagram and print review, standard reports**					
r-customized rts	yes	yes	yes	yes	yes	simple
output/ rams in	yes/yes	yes/yes	yes/third-party	yes/yes	yes/yes	yes/no
nes/linked ode	yes/yes	no/yes	yes/third-party	yes/yes	no/no	no/no
er output ons	ATA DocExpress	Word	Word	Word, CSV	Word	several
F/XMI or L	XMI planned	CDIF	CDIF, XMI	no	no	no/no
orts/exports third-party ls	Doors 4.x, ObjectAda	CRC	ODBC links	RequisitePro, Dynasty, Magic	CaliberRM, Lawson's ERP, PeopleSoft's ERP	GDPro, Visual Studio, Visual Cafe
platforms	Win NT, AIX, HP/Ux, IRIX, Solaris	Mac, Win 9x/NT	Win 9x/NT	Win 9x/NT	Win NT	Win 9x/NT
e per eloper-seat	$6,500	$1,995	$5,995	$3,295	$2,990	$995
available at	www.aonix.com	www.excelsoftware.com	www.oracle.com	www.popkin.com	www.princetonsoftech.com	www.visio.com

le 1 Comparison of UML Design Tools These tools fall into three categories: structured CASE tools, pre-UML OOAD tools, and new, Java-oriented tools.

dvanced Software chnologies DPro 3.0	Sterling Software COOL:Jex	Platinum Technology Paradigm Plus	Rational Software Rational Rose	Together Software Together/J 2.1	ObjectDomain 2.5	Riverton Software How 3.0	Softera Soft Modeler/ Business 1.5
s/yes/yes	yes/yes/yes	yes/yes/yes	yes/yes/no	yes/yes/yes	yes/yes/yes	yes/yes/no	yes/yes/yes
s/yes	yes/yes	yes/yes	yes/yes	yes/yes	yes/yes	yes/yes	yes/yes
s/yes	yes/yes	yes/yes	yes/yes	yes/no	yes/yes	yes/yes	no/no
/yes	yes	yes/yes	yes/yes	no/no	yes/yes	no/yes	yes/no
mmary	OMT	Booch, Clipp, OOCL, Coad/Yourdon, ER, OMT, Shlaer/Mellor, TeamFusion	Booch, OMT	Coad/Yourdon	UML	proprietary domain and query	data based simulations as well as RTE-Round Trip Engineering
++, IDL, Java	C++, IDL, Java, NewEra-f, ObjectAda-f, PowerBuilder-f, Smalltalk, VB	C++, Delphi-f, IDL-f, Java, Forte, VB, PowerBuilder, Smalltalk-f	Ada-f, C++, Forte-f, IDL-f, Java, PowerBuilder-f, Smalltalk-f, VB	C++, IDL, Java	C++, Java, Python	C++-f, Java-f, PowerBuilder-f, VB-f	IDL, Java
ird-party	Informix, Oracle, SQLServer, Sybase f/r,	DB/2-r, Informix-r, Oracle, SQLServer-r, Sybase-r	Oracle, SQLServer-f, Sybase-f	Cocobase	JDBC	Oracle, extensive third-party	Cloudscape, Jasmine
s	yes	yes, VB-like	yes, VBA	no	yes	some	no
	yes	yes	no	no	no	no	no
ML, OMT	OMT, ER	proprietary	unified process	UML	UML	proprietary UML	
stomizable	*multi-user* repository	*ER to class* diagram	*model integrator* merges several sub models	*source driven*	*JPython scripts* can call Java classes of OD itself	*identify components* and generate systems from frameworks	*Java component* and validity checks
s	yes	yes	yes	yes	yes	yes	yes
s	yes	batch synchronization	yes	yes	batch	yes	yes
s/yes	yes/yes	yes/yes	yes/yes	yes/yes	yes/yes	yes/yes	no/yes
s/yes	yes/yes	yes/yes	yes/no	yes/yes	yes/yes	yes/yes	yes/no
s/yes	yes/yes	yes/yes	no/yes	yes/yes	yes/yes	yes/yes	no/yes
sual Studio k	easy configuration management	source/diagram tabs	context menus	thumbnail window	multi-tab diagram pages with themes	navigation dialogs	drag-and-drop objects
et	dBase	ObjectStore	proprietary	source files	source files	proprietary	proprietary
	MS Rep	MS Rep 2, Platinum Rep	no	source in version control	no	no	no
yes/third-party	yes/yes	yes/third-party	yes/third-party	third-party/ third-party	yes/proprietary	yes/third-party	yes/third-party
partial	yes	yes	yes	source in version control	yes	yes	no
VBA, Perl	Tcl/Tk	proprietary	Rosescript	JPython	JPython	no	no
yes	yes	simple	yes	simple	no	simple	no
yes/yes	yes/yes	yes/yes	yes/yes	yes/yes	yes/yes	yes/yes	yes/yes
no/yes	no/no	yes/yes	yes/yes	yes/yes	no/no	no/no	no/yes
RTF	Frame, Interleaf, Word	Doc Express	Word	CSV	JavaDoc	Word	no
no	XMI planned	XMI planned	CDIF	no	no	no	XML, XMI
Rose, Doors 4.x, EasyER/Object, Visual CASE, RequisitePro, Visio Pro	Jaguar CTS, MTS, Doors 4.x, Cool:Biz	Doors 4.x, ERwin, Solstice GDMO Builder	CASEwise, COOL:Jex, StP, Visual Modeler, ClearCase	Rose 98	JDBC links	ERwin, PowerDesigner	JBuilder, VisualAge, Visual Cafe
Win NT, IRIX, HPUX, Solaris	AIX, HP/UX, Solaris, Win NT	Win 9x/NT	Win 9x/NT, Unix	Wins, Linux, any OS with JVM	Wins, Solaris, any OS with JVM	Win 9x/NT	Win, Linux, any OS with JVM
$2,495	$5,000	$4,000	$3,400	$2,400	$995	$2,995	$1,990
www.advancedsw.com	www.sterling.com	www.platinum.com.	www.rational.com	www.togetherj.com	www.objectdomain.com	www.riverton.com	www.softera.com

ANEXO 2 – EXEMPLO DA IDE DO EDITOR DE MODELOS

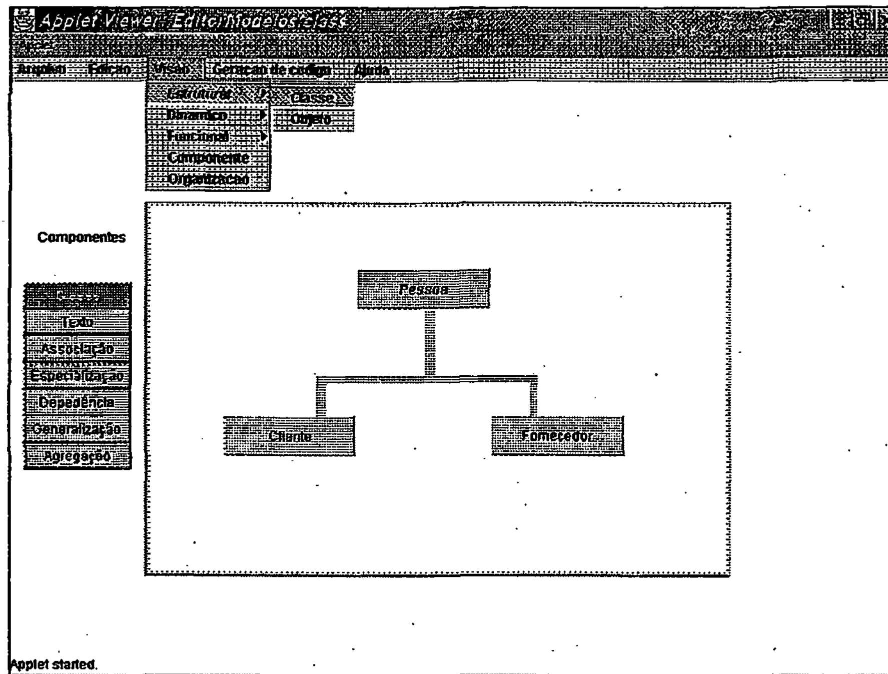


Figura 70 - Exemplo da IDE do Editor de Modelos – Diagrama de classe

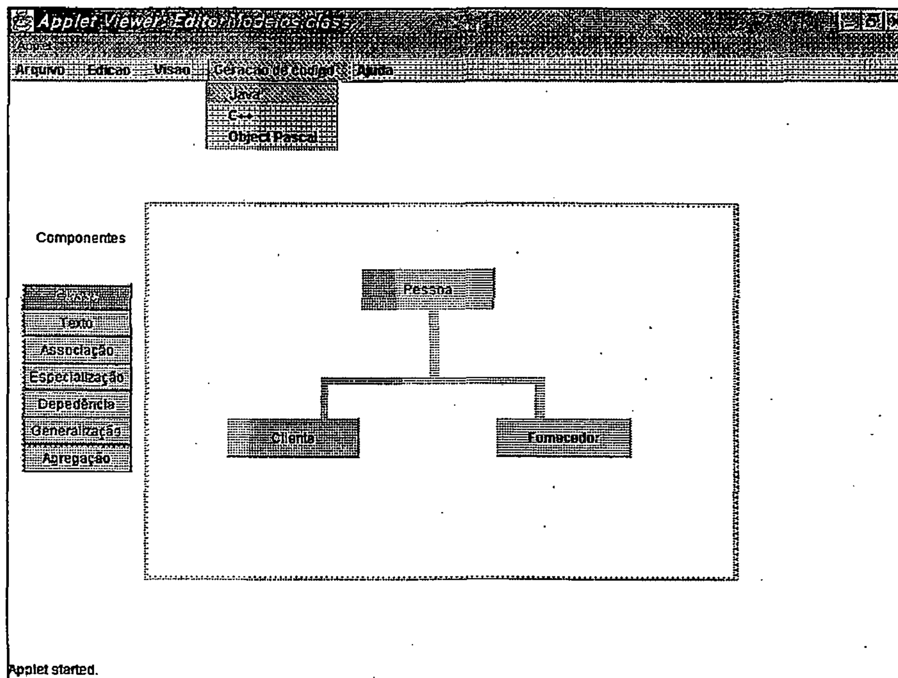


Figura 71 - Exemplo da IDE do Editor de Modelos - Geração de código

ANEXO 3 – CÓDIGO JAVA EXEMPLO: INTERFACE E FINAL

```
import java.awt.*;
import java.applet.Applet;
import java.awt.event.*;

public class Dados extends Applet implements ActionListener  {

       final int WON = 0, LOST = 1, CONTINUE = 2; // constantes
       boolean firstRoll = true;
       int sumOfDice = 0;
       int myPoint = 0;
       int gameStatus = CONTINUE;

       Label die1Label, die2Label, sumLabel, pointLabel;
       TextField firstDie, secondDie, sum, point;
       Button roll;

       public void init () {
                die1Label = new Label(" Primeiro dado ");
                add(die1Label);
                firstDie = new TextField(2);
                firstDie.setEditable(false);
                add(firstDie);

                die2Label = new Label(" Segundo dado ");
                add(die2Label);
                secondDie = new TextField(2);
                secondDie.setEditable(false);
                add(secondDie);

                sumLabel = new Label( " Soma dos dados e: ");
                add(sumLabel);
                sum = new TextField(2);
                sum.setEditable(false);
                add(sum);

                pointLabel = new Label(" pontos ");
                add(pointLabel);
                point = new TextField(10);
                point.setEditable(false);
                add(point);

                roll = new Button("Jogar dados");
                add(roll);

                roll.addActionListener(this);
       }

       public void play() {

          if (firstRoll) {
             sumOfDice = rollDice();
             switch (sumOfDice) {
                case 7: case 11:
```

```
                    gameStatus = WON;
                    point.setText( "" );
                    break;

                 case 2: case 3: case 12:
                    gameStatus = LOST;
                    point.setText( "" );
                    break;

                 default:
                    gameStatus = CONTINUE;
                    myPoint = sumOfDice;
                    point.setText(Integer.toString(myPoint));
                    firstRoll = false;
                    break;
              }
           }
           else {
              sumOfDice = rollDice();
              if (sumOfDice == myPoint ) {
                 gameStatus = WON;
                 showStatus("Voce venceu numero de pontos igual a soma
dos dados");
              }
              else
                 if (sumOfDice == 7)
                    gameStatus = LOST;
           }
           if (gameStatus == CONTINUE)
              showStatus (" Jogue novamente ... ");
           else {
              if (gameStatus == WON)
                 showStatus("Voce venceu !! " +
                            " Click em Jogar dados para jogar
novamente");
              else
                 showStatus("Voce perdeu !! "+" Click em Jogar dados
para jogar novamente");
              firstRoll = true;
           }
        }

        public void actionPerformed (ActionEvent e) {
            play();
        }

        int rollDice() {
            int die1, die2, workSum;

            die1 = 1 + (int) (Math.random() * 6);
            die2 = 1 + (int) (Math.random() * 6);
            workSum = die1 + die2;

            firstDie.setText(Integer.toString(die1));
            secondDie.setText(Integer.toString(die2));
            sum.setText(Integer.toString(workSum));
            return workSum;
        }
}
```