FUNDAÇÃO DE ENSINO “EURÍPIDES SOARES DA ROCHA”
CENTRO UNIVERSITÁRIO “EURÍPIDES DE MARÍLIA” – UNIVEM
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

VINICIUS HUMBERTO SERAPILHA DURELLI
FERNANDO GUTIERRES DAMACENO

ILD – INTERPRETADOR LÉXICO PARA DATILOLOGIA

MARÍLIA
2005

VINICIUS HUMBERTO SERAPILHA DURELLI
FERNANDO GUTIERRES DAMACENO

ILD – INTERPRETADOR LÉXICO PARA DATILOLOGIA

Monografia apresentada à "Fundação de Ensino Eurípides Soares da Rocha", mantenedora do Centro Universitário "Eurípides de Marília", para a obtenção do Título de Bacharel em Ciência da Computação.

Orientador:
Prof. Dr. José Remo Ferreira Brega

MARÍLIA
2005

VINICIUS HUMBERTO SERAPILHA DURELLI
FERNANDO GUTIERRES DAMACENO

ILD – INTERPRETADOR LÉXICO PARA DATILOLOGIA

Banca examinadora da Monografia do Trabalho de Conclusão de Curso do Centro Universitário "Eurípides de Marília" – UNIVEM, para obtenção do Título de Bacharel em Ciência da Computação.

Resultado: _____

ORIENTADOR: Prof. Dr. _____

1º EXAMINADOR:_____

2º EXAMINADOR:_____

Marília, ____ de _____ de 2005.

A minha mãe e minha vó, sem elas
nada teria sido possível.
Vinicius Humberto Serapilha Durelli

A minha família por me guiar durante
todo esse percurso.
Fernando Gutierres Damaceno

AGRADECIMENTOS

À minha mãe por me proporcionar conforto e afeto e também ao professor Remo pelas orientações sempre seguidas de bom humor e confiança, também pelos conselhos e apoio oferecidos.

Vinicius Humberto Serapilha Durelli

À minha família por acreditar e me incentivar sempre principalmente nas horas difíceis e ao professor Remo pela paciência e confiança em nosso trabalho.

Fernando Gutierres Damaceno

É impossivel para aqueles que não conhecem a língua de sinais perceberem sua importância para os surdos, sua enorme influência sobre a felicidade moral e social dos que são privados da audição e sua maravilhosa capacidade de levar o pensamento a intelectos que de outra forma ficariam em perpétua escuridão. Enquanto houver dois surdos no mundo e eles se encontrarem, haverá o uso de sinais.

J. Schuylerhong

A maneira mais provável de o mundo ser destruído, segundo a maioria dos especialistas, é por acidente. É aí que nós entramos: nós somos profissionais de computação. Nós causamos acidentes.

Nathaniel Borenstein

DURELLI, Vinicius; DAMACENO, Fernando. **ILD – INTERPRETADOR LÉXICO PARA DATILOLOGIA.** 2005. Monografia do Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) – Centro Universitário "Eurípides de Marília", Fundação de Ensino "Eurípides Soares da Rocha", Marília, 2005.

RESUMO

O trabalho tem o objetivo de criar um aplicativo que traduz sentenças do português para datilologia, mobilizando a comunidade para criação de novos mecanismos que ajudem a inserir pessoas com deficiência auditiva no contexto social. O aplicativo em questão nomeado como ILD (Interpretador Léxico para Datilologia) foi desenvolvido seguindo algumas das diretrizes do Processo Unificado e utilizando a tecnologia Java. Com O ILD se pretende alcançar a solução para todo o tipo de problema que os deficientes auditivos podem se deparar, dado que não se fornece a tradução das sentenças em LIBRAS somente em datilologia (alfabeto manual).

Palavras-chaves: aplicativo; ILD; Processo Unificado; Java.

DURELLI, Vinicius; DAMACENO, Fernando. **ILD – INTERPRETADOR LÉXICO PARA DATILOLOGIA**. 2005. Monograph of the Work of Conclusion of Course (Bachelor in Computer Science) – Centro Universitário "Eurípides de Marília", Fundação de Ensino "Eurípides Soares da Rocha", Marília, 2005.

ABSTRACT

The work has the objective to create applicative that it translates sentences of the Portuguese for datilologia, mobilizing a community for creation of new mechanisms that help to insert people with auditive deficiency in the social context. The applicative nominated as ILD (Interpreter Lexicon for Datilologia) was developed following some of the guidelines of the Unified Process and using the Java technology. With the ILD we do not intend to reach the solution for all types of problems that auditive deficient can be finding, given that we do not supply the translation of the sentences in LIBRAS, only in datilologia (manual alphabet).

Word-keys: applicative; ILD; Unified Process; Java.

LISTA DE FIGURAS

SUMÁRIO

INTRODUÇÃO

Grande parte da comunidade deficiente auditiva é não alfabetizada em português e se comunica usando a linguagem de sinais (Libras). Tal problema afeta a comunidade dado que pessoas portadoras de deficiência auditiva não alfabetizadas em português não conseguem se comunicar de maneira eficiente com pessoas que não conhecem a Libras, incluindo amigos e familiares. Muitas das pessoas que não possuem deficiência auditiva consideram a linguagem de sinais uma linguagem frívola e também consideram as pessoas portadoras de deficiência auditiva intelectualmente inferiores ou menos interessantes, assim não despertam interesse em aprender a linguagem de sinais e superar essa barreira na comunicação.

O presente trabalho possui o objetivo de oferecer uma solução para a tradução de sentenças do português para datilologia na forma de um aplicativo consideravelmente rápido e fácil de manipular (utilizar) difundindo assim uma forma de se comunicar com os deficientes auditivos.

O setor público pode empregar a solução para atender portadores de deficiência auditiva, levando em consideração que para obter um atendimento mais eficaz deve empregar-se Libras e não datilologia.

O aplicativo foi desenvolvido seguindo algumas diretrizes do processo unificado e utilizando a tecnologia Java. O processo unificado é um processo de desenvolvimento de software *iterativo,* ou seja, subdivide o projeto em subconjuntos de funcionalidades (o estilo *em cascata* subdivide um projeto de acordo com as atividades).

O Capítulo 1 apresenta de maneira sucinta informações sobre metodologias de ensino utilizadas na educação de surdos e mudos, informações sobre Libras,

informações sobre datilologia (alfabeto manual) e sua história, educação dos surdos no Brasil e define a deficiência auditiva.

O Capítulo 2 apresenta o processo unificado, processo que forneceu as diretrizes para o desenvolvimento do aplicativo. Mostra o que é um processo de desenvolvimento, porque as pessoas envolvidas com o desenvolvimento de software precisam de um processo, as fases do processo unificado e resumidamente expõe as características de um processo em cascata e de um processo iterativo. São apresentados também os casos de uso, uma técnica amplamente usada para descobrir requisitos funcionais e como empregá-los de maneira correta. Apresenta também resumidamente o diagrama de casos de uso.

Durante o Capítulo 3 apresenta a Realidade Virtual e seu histórico.

O Capítulo 4 oferece uma abrangente visão sobre a tecnologia Java, técnicas como manipulação de cadeias de caracteres e linhas de execução são tratadas com detalhes.

No Capítulo 5 é fornecida uma descrição do sistema e como ele foi desenvolvido. Este capítulo apresenta as ferramentas empregadas, as dificuldades encontradas, trechos de código do aplicativo e imagens do aplicativo em questão.

CAPÍTULO 1 – METODOLOGIAS DE ENSINO, LIBRAS E DATILOLOGIA.

A linguagem permite ao homem estruturar seu pensamento, exprimir seus sentimentos, transferir e adquirir conhecimentos e comunicar-se de maneira geral. A linguagem marca o ingresso do homem na cultura, tornando-o um ser capaz de realizar coisas nunca antes imaginadas. A linguagem é a prova da inteligência do homem.

Apesar de todos os benefícios provenientes da linguagem é importante ressaltar a ambigüidade trazida pela mesma na maioria dos casos, em um mundo perfeito não haveria ambigüidades na interpretação.

Uma linguagem pode ser um conjunto de palavras ou até mesmo de sinais criados com o intuito de transmitirem algum significado. O homem utiliza instrumentos lingüísticos para efetuar a comunicação escrita, falada ou sinalizada e transmitir sua cultura, assim a língua é um fenômeno sócio cultural à disposição de todos.

Linguagem “natural” é aquela que expressa da forma mais inerente possível a comunicação entre determinada comunidade ou grupo de pessoas.

1.1 Língua e signos lingüísticos

Língua é um sistema abstrato de sinais ou de símbolos de uma comunidade, portanto um instrumento lingüístico particular de determinado grupo.

Para expressar idéias o ser humano utiliza-se de sinais ou palavras que são conhecidos como signos lingüísticos. Signo é a combinação dos complexos sonoros e visuais (por exemplo, “cachorro”) e do significado que este complexo comunica (a idéia do animal doméstico considerado melhor amigo do homem). As partes que constituem o

signo são o significante (palavra ou sinal) e o significado (conceito). Língua então seria um conjunto de signos usados para passar a idéia dos objetos e representá-los, por outro lado, linguagem é a utilização oral, sinalizada (por sinais) ou escrita de uma língua. Alguns tipos de linguagens são: a) afetiva; b) cognitiva; c) conotativa; d) denotativa; e) lúdica; f) simbólica; g) erudita; h) jurídica i) obscena; j) pobre; l) sórdida; m) coloquial; n) cientifica; etc (http://www.ines.org.br/ines_livros/SUMARIO.HTM).

1.2 Educação dos deficientes auditivos no Brasil

Os surdos começaram a serem educados em torno de 400 anos atrás. No primórdio o problema era pouco compreendido e uma pessoa com surdez e conseqüentemente mudez era considerada uma pessoa intelectualmente inferior e na grande maioria dos casos era mantida em asilos.

Os primeiros educadores de surdos surgiram na Europa, no século XVI, criando diferentes metodologias de ensino. Estas metodologias utilizavam o idioma auditivo-oral nativo, língua de sinais, datilologia e códigos visuais e podiam ou não associar diversos meios de comunicação. O grande êxito chegou no século XVIII quando a linguagem de sinais começou a ser bastante difundida.

Devido a avanços tecnológicos no meio do século XIX a oralidade começou a ganhar força e os sinais eram reprimidos.

No Brasil a educação dos surdos teve inicio no período do segundo império, com a chegada do educador francês Hernest Huet (Huet era surdo). Em 26 de setembro de 1857, foi fundado o Imperial Instituto de Surdos Mudos, nesta época o Instituto era um asilo somente para surdos do sexo masculino que vinham de todos os cantos do país,

na maioria das vezes abandonados pelas suas famílias, atualmente Instituto Nacional de Educação dos Surdos (INES).

No século XX os surdos assumiram a direção da primeira universidade para surdos do mundo *Gallaudet University Library* localizada em Washington nos Estados Unidos. Na década de 60 com a visita de Ivete Vasconcelos, educadora de surdos da universidade de Gallaudet, chegou ao Brasil a filosofia da Comunicação Total, na década seguinte, a partir das pesquisas da professora lingüística Lucinda Ferreira Brito sobre a Língua Brasileira de Sinais (LIBRAS) e da professora Eulália Fernandes sobre a educação dos surdos, o Bilingüismo passou a ser difundido. Atualmente as três filosofias educacionais persistem paralelamente no Brasil (Oralismo, Comunicação Total e o Bilingüismo).

1.3 Análise das metodologias de ensino

As três grandes metodologias difundidas para educação dos surdos apresentam vantagens e desvantagens considerando as particularidades inerentes à surdez.

Muitos profissionais se dedicam a estudar os procedimentos que privilegiam ou não a linguagem gestual. Este esforço é justificado por dois motivos: os professores de surdos em sua grande maioria são ouvintes e o meio social e cultural dos surdos é, também, de ouvintes.

Baseado em considerações dessa natureza que o Oralismo se faz presente nos cursos de formação de grande parte dos professores que ensinam em instituições especializadas para surdos. A oralidade baseia-se na crença de que a modalidade oral da língua é a única forma desejável de comunicação para o surdo, e que qualquer forma de

gesticulação deve ser evitada, ou seja, nesta metodologia de ensino a aprendizagem da fala é o foco central.

Para o desenvolvimento e aprendizagem da fala algumas técnicas orais devem ser utilizadas, as técnicas são as seguintes:

- Treinamento auditivo: estimulação da audição para distinção e reconhecimento de ruídos, sons ambientais e sons da fala.
- Desenvolvimento da fala: exercícios objetivando a mobilidade e tonicidade dos órgãos envolvidos na fonação, exercícios de relaxamento e de respiração (chamado também de mecânica de fala).
- Leitura labial: treinamento para identificação da palavra através dos gestos orais do emissor.

Uma característica interessante do Oralismo é a indicação de próteses individuais, que ampliam os sons, admitindo assim a existência de um "resíduo" auditivo em qualquer tipo de surdez (até mesmo na profunda). Então este método busca assim reeducar o indivíduo surdo através da ampliação de sons e técnicas de oralidade.

Este método é defendido por muitos que acreditam que a fala exerça um papel importante no desenvolvimento cognitivo do indivíduo, e ao mesmo tempo criticado pelos que afirmam que a maneira de se expressar não precisa necessariamente do som, ou seja, afirmam que a linguagem não depende da natureza do meio material que utiliza.

Depois de décadas sem muitos progressos, juntamente com a divulgação da linguagem de estudos sobre a língua de sinais, houve uma mudança no enfoque educacional.

O enfoque agora era no método conhecido como Comunicação Total. A Comunicação Total consiste em uma proposta de uso tanto da linguagem gestual quanto

da linguagem oral. A proposta inicial foi expandida e transformada deixando de ser um método para se tornar uma filosofia educacional. A Comunicação Total não impõe claramente procedimentos de ensino sendo então incorporada em vários lugares e em versões muito diferentes. Uma das características da Comunicação Total é a aceitação de vários recursos comunicativos com a finalidade de ensinar a língua majoritária (língua portuguesa no caso do surdo brasileiro) e promover a comunicação.

Comunicação Total não se trata de uma oposição à linguagem oral, mas apresenta-se como um sistema de comunicação complementar. Adeptos da Comunicação Total admitem que a comunicação oral é essencial para que os surdos possam obter uma vida social e cultural de melhor qualidade, mas ressaltam as dificuldades de aquisição da mesma. As crianças surdas demoram muito para apreender a linguagem oral e esta lacuna de comunicação neste período de aprendizado deve ser suprida por algo (http://www.ines.org.br/ines_livros/SUMARIO.HTM).

Nessa perspectiva esta filosofia educacional resulta em diferentes métodos e sistemas de comunicação distintos, entre os quais é possível citar: língua falada de sinais (codificada em sinais), língua falada sinalizada exata (variante do sistema anterior, caracteriza-se por buscar a reprodução precisa da estrutura da língua), implantação de códigos manuais para ajudar na discriminação e articulação dos sons e combinação diversa de sinais, fala, datilologia, gesto, pantomima etc.

A abordagem educacional chamada Bilingüismo se encaixa no contexto citado acima. Nesta abordagem o surdo primeiramente aprende a língua de sinais e posteriormente aprende a língua majoritária do seu país, normalmente em sua modalidade escrita. Uma questão polêmica que divide os educadores dos surdos envolvendo o Bilingüismo é se o surdo deve aprender a linguagem na modalidade escrita, oral ou ainda em ambas, mas há um consenso que o desenvolvimento cognitivo

não depende da audição e sim do desenvolvimento espontâneo da linguagem que deverá auxiliar no convívio social. Algumas pesquisas revelam que “[...] a ausência de convívio social causa severos efeitos negativos na capacidade cognitiva geral” (Katz, Lawrence; Rubin, Manning. Mantenha o Seu Cérebro Vivo: exercícios neuróbicos para ajudar a prevenir a falta de memória e aumentar a capacidade mental. Rio de Janeiro: Editora Sextante, 2000. 160 p.).

1.4 LIBRAS OU LSB: Linguagem de Sinais Brasileira

LIBRAS ou LSB são os acrônimos usados para se referir à Linguagem Brasileira de Sinais. A LIBRAS é a língua materna (natural) usada pelos deficientes auditivos. As línguas de sinais não são universais, cada país tem sua própria linguagem de sinais que sofre influências culturais e como qualquer outra língua apresenta regionalismo (expressões que diferem de região para região dentro de um mesmo país).

Ao contrário do que muitos acreditam as linguagens de sinais não são simplesmente mímicas e pantomimas usadas para facilitar a comunicação. São línguas com suas próprias estruturas gramáticas.

As linguagens de sinais receberam o status de língua porque são compostas pelos seguintes níveis lingüísticos: o fonológico, o morfológico, o sintático e o semântico. Nas línguas de sinais certos parâmetros são usados para configurar os sinais, a saber: a) configuração das mãos: são as formas que as mãos podem assumir (representar), estas formas podem ser da datilologia (alfabeto manual) ou outras formas feitas pela mão predominante (direita para destros e esquerda para canhotos) ou ainda pelas duas mãos; b) Ponto de articulação: local onde o sinal é feito (pode-se tocar alguma parte do corpo ou não); c) movimento: os sinais podem apresentar movimento

ou não; d) expressão facial e/ou corporal: as expressões tanto faciais como corporais são de grande valor para o entendimento real do sinal; e) direção (orientação): os sinais possuem uma direção com relação aos parâmetros citados anteriormente.

1.5 Datilologia e sua história

A datilologia atual também conhecida como alfabeto manual, é utilizada para informar (representar) coisas que ainda não possuem um sinal na LIBRAS, para expressar nomes e palavras de línguas estrangeiras.

Segundo um levantamento histórico feito por Woll (1977) tratando de material impresso sobre a língua de sinais na Inglaterra, mostrou que os primeiros panfletos são datados de 1880 e provavelmente destinados à venda para arrecadação de fundos.

No entanto a datilologia é mais antiga e no século XVI atribui-se a um monge, Pedro Ponce de Léon (1520-1584) a invenção do primeiro alfabeto manual conhecido. Este trabalho está registrado nos livros da instituição religiosa que relata sucesso de uma metodologia que incluía datilologia, escrita e fala.

Em meados do século XVIII a datilologia foi levada a França por Jacob Rodriguez Pereira e depois para os Estados Unidos em 1816 por Gallaudet. A datilologia apresentava algumas variações dependendo da localidade, variando de “alfabeto de uma mão” para “alfabeto de duas mãos”.

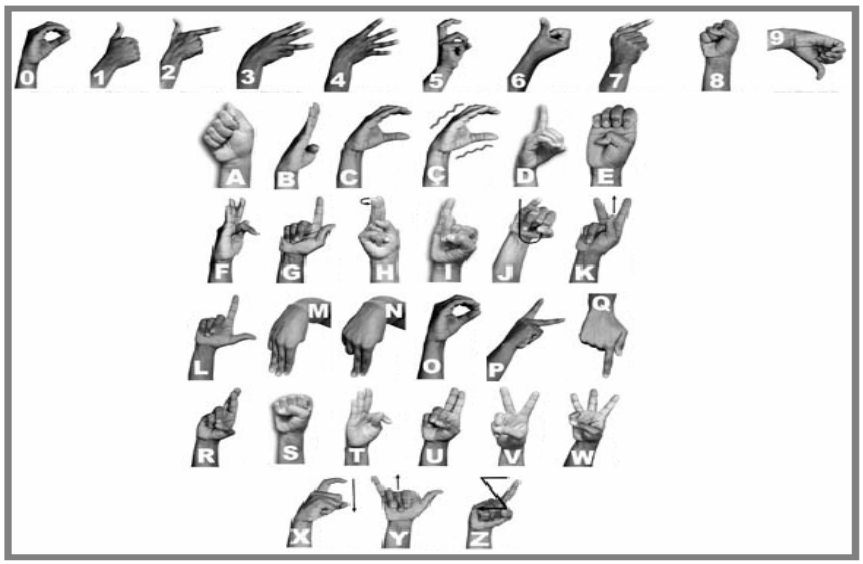

Figura 1: Alfabeto Manual e Números (AJA, 2004)

1.6 A Deficiência Auditiva

Deficiência Auditiva é a diminuição da capacidade auditiva, ou seja, diminuição da percepção dos sons por parte do indivíduo portador da deficiência. A DA (Deficiência Auditiva) pode ser parcial (caso em que a audição é funcional com ou sem aparelhos que ampliem a audição) e surdo cuja audição não é considerada funcional para a vida cotidiana.

CAPÍTULO 2 – O PROCESSO UNIFICADO DE DESENVOLVIMENTO SOFTWARE

De acordo com Pressman (1995, p. 24) “a natureza lógica do software constitui um desafio para as pessoas que o desenvolvem”.

Atualmente pessoas ainda desenvolvem software como em torno de 25 anos atrás, ou seja, de maneira casual.

Esse tipo de abordagem caótica não é apropriado para resolução de problemas dessa natureza.

O desafio de desenvolver software deve ser apoiado por um processo de desenvolvimento. Este capítulo apresenta alguns aspectos do processo de desenvolvimento que “dirigiu” a criação do MA e do MG.

2.1 Processo de desenvolvimento

Um processo define quem faz o que, quando e como, para atingir determinada meta (tal meta é construir um produto de software ou melhorar um existente). Um processo oferece linhas-guia (guidelines) para desenvolvimento de software de qualidade. Um processo de desenvolvimento de software deve capturar e apresentar as melhores práticas do atual estado da arte (JACOBSON; BOOCH; RUMBAUGH; 2001).

Existem quatro coisas que um processo deve levar em consideração tecnologia, ferramentas (tools), pessoas e padrões organizacionais (JACOBSON; BOOCH; RUMBAUGH; 2001).

- Processos são construídos sobre as tecnologias disponíveis (como tecnologia entenda: linguagens, sistemas operacionais etc.);
- Ferramentas e processos podem ser desenvolvidos em paralelos, ferramentas são partes do processo;
- A pessoa (ou grupo de pessoas) responsável pela criação de um processo não deve exagerar no conjunto de habilidades necessárias para que outras pessoas (desenvolvedores, por exemplo) possam “operar” (utilizar, empregar) o processo. Tais habilidades devem ser limitadas para aquilo que os desenvolvedores possuem atualmente ou aquilo que eles possam conseguir sem muito esforço;
- Processos devem se adaptar a realidade das organizações atuais.

2.2 Necessidade de um processo de desenvolvimento

O desenvolvimento de software sério não é somente codificar. Um roteiro que abrange todo o desenvolvimento é necessário, tal roteiro assume a forma de um processo de desenvolvimento. A equipe precisa de um meio organizado para trabalhar e, um processo fornece esse meio, ou seja, colocando de maneira simples (e também informal e sucinta) um processo serve para “liderar” a ordem das atividades da equipe de desenvolvimento.

2.3 O Processo Unificado

Antes de qualquer coisa o processo unificado é um processo de desenvolvimento de software. Um processo de desenvolvimento de software consiste

em um conjunto de atividades necessárias para transformar os requisitos em um sistema de software (JACOBSON; BOOCH; RUMBAUGH; 2001). Isso é ilustrado na figura 2:

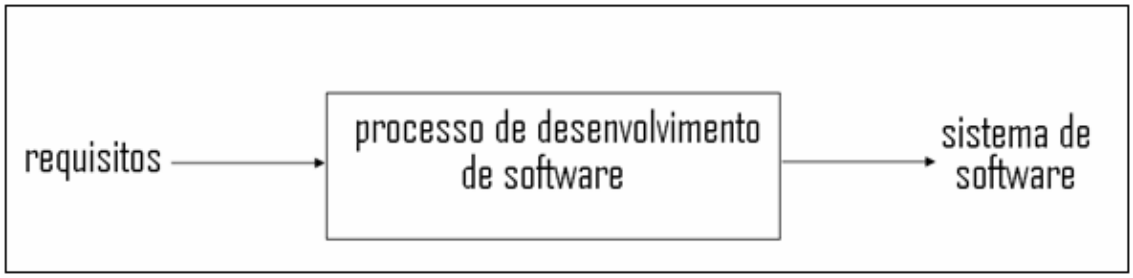

Figura 2: Processo de desenvolvimento de software (JACOBSON; BOOCH; RUMBAUGH; 2001).

Um sistema de software são todos os artefatos que são representados de maneira compreensível pela máquina ou de forma compreensível por humanos a partir da forma compreensível por máquina (JACOBSON; BOOCH; RUMBAUGH; 2001).

Artefato é tanto o recurso produzido como o recurso consumido em uma atividade, um artefato pode estar relacionado a diversas atividades e uma atividade pode estar relacionada com vários artefatos. Exemplos poderiam ser: código de máquina executável (binário), código fonte, manual de padrões plano de trabalho e diagramas de interação.

O processo unificado é mais que um único processo; é uma estrutura genérica de processos que pode ser aplicada em diferentes tipos de organizações, diferentes níveis de competência e projetos de vários tamanhos.

O processo unificado pode ser capturado em: dirigido por casos de uso, centrado em arquitetura, interativo e incremental.

2.4 O papel dos casos de uso

No processo unificado os casos de uso desempenham um papel muito importante. Eles aparecem e atuam como ferramentas centrais para a detecção de funcionalidades do sistema.

Os casos de uso são desenvolvidos junto com a arquitetura do sistema. E os dois seguem amadurecendo conforme o ciclo continua. Mais detalhes sobre este assunto serão expostos adiante.

2.5 Arquitetura de software

De acordo com Jacobson, Booch e Rumbaugh (2001) a regra de arquitetura de software é similar à regra da arquitetura (na construção de edifícios, casas etc.). A arquitetura convencional analisa vários aspectos: estrutura, serviços, condução de calor, eletricidade etc. Isso permite uma visão completa antes da construção iniciar. Similarmente a arquitetura em sistemas de software é descrita na forma de diferentes visões (aspectos) do sistema sendo construído.

O conceito de arquitetura de software é focado nos aspectos estáticos e dinâmicos mais importantes do sistema (JACOBSON; BOOCH; RUMBAUGH; 2001).

Entretanto a arquitetura também é influenciada por outros fatores como: plataforma na qual o software deve executar (arquitetura do computador), Sistema Operacional, blocos reutilizáveis disponíveis entre outros (JACOBSON; BOOCH; RUMBAUGH; 2001).

A arquitetura pode ser vista tanto como substantivo assim como verbo (LARMAN, 2005). Como substantivo inclui: organização e estrutura dos principais

elementos do sistema além do comportamento. Aparecendo como verbo ela seria a investigação (e em parte trabalho do projeto).

2.6 O desenvolvimento iterativo e incremental

O processo unificado propõe muitas práticas, mas talvez a mais interessante delas seja o desenvolvimento iterativo. Nessa abordagem o trabalho é dividido em mini-projetos de duração fixa (iterações). O resultado de cada iteração é um subconjunto do sistema final.

Cada mini-projeto é uma iteração que resulta em um incremento (JACOBSON; BOOCH; RUMBAUGH; 2001). Iterações podem ser consideradas passos nas atividades (mais detalhes sobre o significado de atividades no contexto do processo unificado serão esclarecidos mais tarde nesse trabalho) e incrementos como um aumento do produto.

Larman (2005, p. 38) afirma:

> O ciclo de vida iterativo é baseado em refinamentos e incrementos de um sistema por meio de múltiplas iterações, com realimentação (feedback) e adaptação cíclicas como principais propulsores para um sistema adequado.

É importante ressaltar que os resultados das iterações são subconjuntos do sistema com qualidade final e não protótipos. Segundo Larman (2005, p. 40) “a saída de uma iteração não é um protótipo experimental ou descartável, assim como o desenvolvimento iterativo não é prototipação”.

Uma iteração não precisa necessariamente tratar da criação de um subconjunto do sistema, também são válidas iterações que melhorem um subconjunto do sistema já existente.

2.7 Especificação Suplementar

A Especificação Suplementar serve para documentar e expor atributos de qualidade do sistema e restrições (de implementação, de projeto, etc.).

Larman (2005, p.110) define:

> Especificação Suplementar documenta outros requisitos, informações que não são facilmente documentadas pelo Glossário, incluindo atributos de qualidade ou requisitos FURPS+ [maiores informações consultar o capítulo referente aos casos de uso] aplicáveis a todo o sistema.

A Especificação Suplementar também pode possuir as regras de domínio. É importante ressaltar que as regras de domínio não são requisitos da aplicação.

2.8 Casos de uso

Os casos de uso são amplamente usados para resolução de requisitos (mais especificamente requisitos funcionais), tal idéia surgiu em 1986, introduzida por Ivar Jacobson.

Devido sua utilidade e facilidade os casos de uso podem ser usados por uma ampla variedade de pessoas interessadas em seus benefícios (não necessariamente somente analistas experientes).

Este capítulo trata das características, aplicação, benefícios e apresenta onde os casos de uso atuam no processo unificado. O capítulo é baseado em obras atuais e é consistente com os trabalhos mais recentes de Alistair Cockburn. Cockburn contribui com o passo mais amplo, coerente e notavelmente influente sobre o que são casos de uso (LARMAN, 2005).

Casos de uso de maneira informal e sucinta são narrativas de utilização do sistema, essa narrativa expõe (às vezes não tão claramente) as funcionalidades que o sistema subjacente deve realizar.

Fowler (2005, p.104) escreve:

> [...] casos de uso são uma técnica para captar requisitos funcionais de um sistema [...] servem para descrever as interações entre os usuários de um sistema e o próprio sistema, fornecendo uma narrativa sobre como o sistema é utilizado.

Uma definição mais precisa pode ser fornecida, mas para isso é necessária a apresentação de alguns outros conceitos.

Uma possível definição de *ator* segundo Larman (2005, p.69): “algo com comportamento, tal como uma pessoa (identificada por seu papel), um sistema de computador ou uma organização”. Definir cenário nesse contexto também se faz útil. Cenário é uma seqüência específica de ações e interações envolvendo o sistema em questão, uma história particular de uso do sistema (LARMAN, 2005).

De acordo com Fowler (2005, p.104): “cenário é uma seqüência de passos que descreve uma interação entre o usuário e o sistema”. Usuários têm objetivos, os cenários “ilustram” esses objetivos. Então uma definição ainda informal, porém sutilmente mais completa é escrita por Larman (2005, p.69): “um caso de uso é uma coleção de cenários

relacionados de sucesso e fracasso, que descrevem atores usando um sistema como meio para atingir um objetivo".

A atitude ideal a ser tomada no trabalho com casos de uso é focar em oferecer "um valor observável" para o usuário e não meramente tratar os requisitos como uma lista de compras (LARMAN, 2005).

2.9 Categoria de requisitos

Nem todos os requisitos são obtidos da mesma forma e, nem todos se enquadram em uma só categoria.

Primeiramente uma definição sobre o que é um requisito se faz necessária. Requisitos são necessidades ou características que o software deve atender (realizar) ou se encaixar.

Sommerville (2003, p. 82) escreve: "as descrições das funções e das restrições são os requisitos para o sistema".

O termo requisito não é empregado pela indústria de software de maneira consistente, algumas vezes é visto como uma declaração abstrata (alto nível), e outras vezes uma definição detalhada, matematicamente formal (SOMMERVILLE, 2003).

Fowler (2005, p. 47) acrescenta: "a atividade de análise de requisitos procura descobrir o que os usuários e clientes de um produto de software querem que ele faça".

Existe um modelo com o intuito de categorizar os requisitos, denominado FURPS+, o processo unificado emprega esse modelo, por isso o mesmo foi adotado para categorizar os atributos do MA. Este mnemônico possui o seguinte significado:

- Funcionais: características, capacidades.

- Usabilidade: fatores humanos, recursos de ajuda, documentação.
- Confiabilidade: possível freqüência de falhas e capacidade de recuperação.
- Desempenho: tempo de resposta, uso de recursos.
- Facilidade de Suporte: internacionalização, configurabilidade e manutenção.

O “+” em FURPS+ representa aspectos adicionais como: implementação, interface, operações, empacotamento e questões legais (LARMAN, 2005).

Normalmente os requisitos são chamados de atributos de qualidade e também muitos usualmente classificam os requisitos como “funcionais” ou “não-funcionais” (no caso todos os requisitos que não fossem considerados funcionais seriam “não-funcionais”).

Os casos de uso são usados na maioria dos processos modernos para investigação de requisitos funcionais, mas não somente este tipo de requisito (LARMAN, 2005).

De acordo com Sommerville (2003, p.99): “requisitos funcionais são declarações de funções que o sistema deve fornecer ou são descrições de como alguns cálculos devem ser realizados”.

Requisitos funcionais descobertos em um caso de uso se enquadram na letra “F” do modelo de categorização FURPS+, dado que “F” são requisitos funcionais (comportamentais).

Muitos profissionais consideram os requisitos como “a lista de coisas que o sistema deve fazer”, não é bem assim e, os casos de uso devem amenizar essa

consideração “forçando-os” a escrever pensando nos requisitos funcionais (LARMAN, 2005).

2.10 Tipos de casos de uso

Antes de expor os tipos de casos de uso é interessante desmistificar que os casos de uso são “diagramas bonitinhos e coloridos”. Na verdade casos de uso são documentos de texto e Larman (2005, p.71) acrescenta: “a modelagem de casos de uso é basicamente um ato de redigir textos, não de desenhar”. Entretanto a UML (Unified Modeling Language) oferece diagramas para representação de casos de uso.

Praticamente todo o valor dos casos de uso se encontra na parte textual (conteúdo), o diagrama é de valor bastante limitado (FOWLER, 2005).

Voltando a tipos de casos de uso, não existe uma maneira padronizada de casos de uso, diferentes formatos são empregados. Apesar disso, alguns formatos são mais usados e recomendados, tais formatos serão apresentados nas seções seguintes.

2.11 Caso de uso de caixa-preta

Os casos de uso não devem focar detalhes de implementação. Casos de uso de caixa-preta são recomendados e também mais comuns, tal tipo não descreve o funcionamento interno do sistema (LARMAN, 2005). A Figura 3 ilustra um conceito correto de caixa-preta e outro incorreto (com detalhes relacionados ao funcionamento interno também conhecido como caixa-branca):

Figura 3: conceito de caixa-preta e conceito de caixa-branca

2.12 Formalidades

Os casos de uso podem ser escritos com base em três graus de formalidade: informal, resumido e completo (LARMAN, 2005). No informal são mostrados vários parágrafos que cobrem vários cenários.

Por sua vez o resumido trata-se de um parágrafo e, geralmente do cenário principal de sucesso (LARMAN, 2005).

O formato completo de acordo com Larman (2005, p.71) possui o seguinte perfil: “todos os passos de suporte como pré-condições e garantias de sucesso”.

Cockburn sugere um esquema de níveis diferente, os níveis são: mar, peixe e pássaro (FOWLER, 2005).

2.13 Estilo Essencial

Cockburn (2001) apud Larman (2005): “deixe de fora a interface de usuário, focalize a intenção”. O estilo essencial é o estilo de escrita que evita detalhes de interface gráfica e dá ênfase nas intenções de usuário.

Cada passo em um caso de uso é equivalente a interação entre o ator e o sistema, dito isso cada passo deve ser uma simples declaração da interação em questão e não mecanismos do que o ator faz, assim você não descreve a interface com o usuário no caso de uso (FOWLER, 2005).

Larman (2005, p.89) destaca: “escreva casos de uso utilizando um estilo essencial; deixe de lado a interface com o usuário e enfoque na intenção do ator”.

Um caso de uso essencial permanece independente de tecnologia e detalhes de interface com o usuário.

O estilo em que detalhes de interface com o usuário estão presentes é chamado de caso de uso concreto, este tipo de caso de uso pode ser empregado como auxilio no projeto de interface com o usuário, mas não são adequados para a especificação inicial de requisitos (LARMAN, 2005).

2.14 Utilizando casos de uso

Fowler (2005, p. 108) fornece algumas dicas valiosas:

- Com os casos de uso, você concentra sua energia no texto e não no diagrama de casos de uso;

- Um grande perigo dos casos de uso é que as pessoas os tornam complicados demais e não conseguem prosseguir [mantenha os casos de uso simples];
- Normalmente você terá menos problemas fazendo pouco do que fazendo demais.

CAPÍTULO 3 – REALIDADE VIRTUAL

A Realidade Virtual (RV) tem aplicações na maioria das áreas do conhecimento. Aqui no Capítulo 3 será apresentada uma visão geral sobre tal assunto.

3.1 Realidade Virtual

Apesar de a Realidade Virtual ter sido inventada há mais de 30 anos ela tem evoluído consideravelmente a cada ano.

O seu custo permaneceu alto durante muito tempo por causa do alto custo dos equipamentos utilizados que utilizam alta tecnologia, mas devido à evolução das indústrias de computadores e ao grande avanço tecnológico atualmente é possível utilizá-la a um preço acessível fazendo com que deixasse de ser exclusividade de instituições de pesquisa ou governamentais (ANTONIO VALERIO NETO; LILIANE DOS SANTOS MACHADO; MARIA CRISTINA F. OLIVEIRA; 2002).

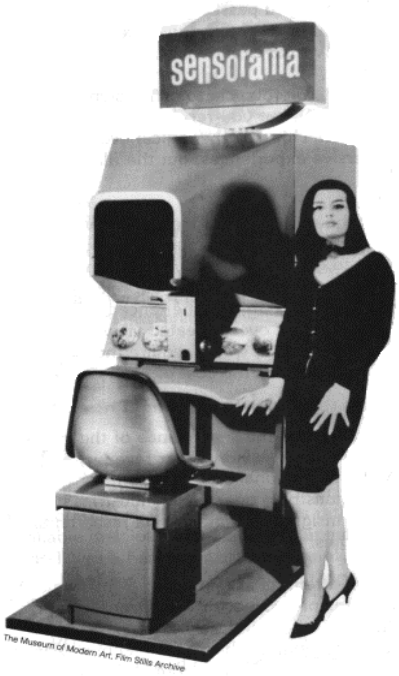

Figura 4 :O "Sensorama" (GRIGORE BURDEA;PHILIPPE COIFFET; 1994)

3.2 Histórico

A realidade virtual teve seus primeiros vestígios em 1962 quando Morton Heiling criou um árcade intitulado “Sensorama Simulator”. Ele imaginou uma maquina que pudesse substituir a experiência cinematográfica da sua época (GRIGORE BURDEA; PHILIPPE COIFFET; 1994).

O Sensorama era uma espécie de cabine, que combinava filmes 3D, som estéreo, vibrações mecânicas, aromas e ar movimentado por ventiladores, isso fazia com que o espectador tivesse uma viagem multisensorial(ANTONIO VALERIO NETO; LILIANE DOS SANTOS MACHADO ; MARIA CRISTINA F. OLIVEIRA ; 2002).

Em 1965 Ivan Sutherland teve a idéia de usar os computadores para desenhar projetos diretamente na tela do computador utilizando uma caneta ótica, foi o inicio da computação gráfica, Sutherland tornou-se o precursor da atual indústria CAD e também inventou o primeiro videocapacete desenvolvido para o projeto “Ultimate Display”.

Com o uso desse capacete o usuário poderia ver de acordo com a movimentação de sua cabeça os diferentes lados de uma estrutura de arame que formavam um cubo flutuando no espaço (PIMENTEL, 1995).

A realidade virtual passou também a ter um papel importante no treinamento de pilotos de avião, depois da segunda guerra a Força Aérea dos Estados Unidos desenvolveu um simulador de vôo onde treinavam seus pilotos (ANTONIO VALERIO NETO; LILIANE DOS SANTOS MACHADO; MARIA CRISTINA F. OLIVEIRA; 2002).

Na mesma época em que surgia o videocapacete, Myron Krueger experimentava combinar computadores com sistemas de vídeo criando a Realidade Artificial. Em 1975 Krueger inventou um capacete com uma câmera de vídeo que capturava os participantes em imagens 2D e jogava-os em uma grande tela, o equipamento foi chamado de

VIDEOPLACE.Os participantes interagiam uns com os outros e com os objetos que eram projetados na tela, seus movimentos eram constantemente processados, essa técnica passou a ser conhecida também por Realidade Virtual de projeção (JACOBSON, 1994).

Thomas Furness, em 1982 demonstrava para a Força Aérea Americana o VCASS (*Visually Coupled Airborne Systems Simulator*), conhecido como "*Super Cockpit*" - um simulador que imitava a cabine de um avião através do uso de computadores e videocapacetes interligados representando um espaço gráfico 3D (Pimentel, 1995). Os videocapacetes integravam a parte de áudio e vídeo. Assim, os pilotos podiam aprender a voar e lutar em trajetórias com 6 graus de liberdade, sem decolar verdadeiramente, ficando praticamente isolados do mundo ao seu redor. O VCASS possuía uma alta qualidade de resolução nas imagens e era bastante rápido no *rendering* de imagens complexas. No entanto apresentava um problema: milhões de dólares eram necessários apenas para o capacete (Pimentel, 1995).

Figura 5:Piloto utilizando o capacete "Super Cockpit"(PIMENTEL,1995)

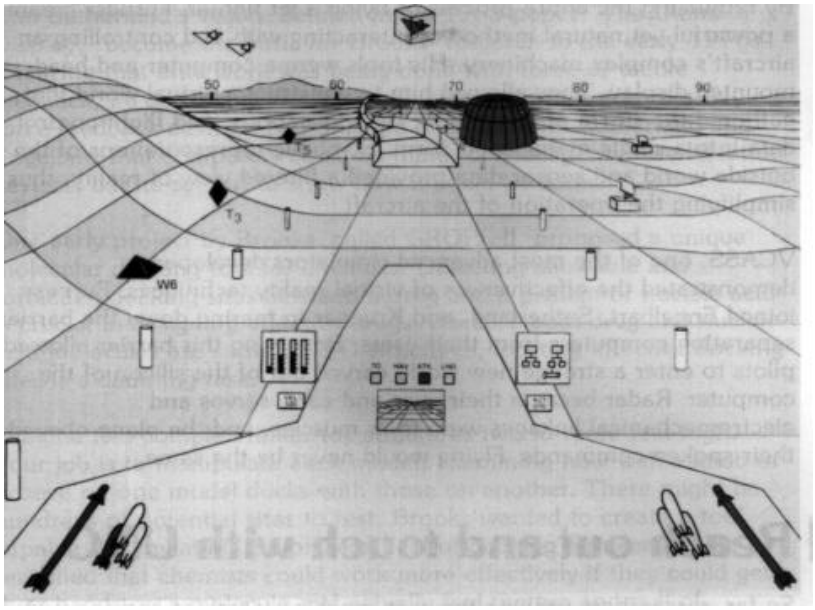

Figura 6:Visão do usuário do "Super Cockpit"(PIMENTEL,1995).

Através do uso de uma nova tecnologia de visores de cristal líquido (LCD) Michael McGreevy começou a trabalhar no projeto VIVED (*Virtual Visual Environment Display*) em 1984 na NASA, no qual as imagens seriam estereoscópicas (Pimentel, 1995). A resolução das imagens era limitada em comparação ao VCASS, mas o custo era bastante atrativo (Rheingold, 1991). A parte de áudio e vídeo foi então montada sobre uma máscara de mergulho utilizando dois visores de cristal líquido com pequenos auto-falantes acoplados. Scott Fisher juntasse a esse projeto no ano de 1985 com o objetivo de incluir nele: luvas de dados, reconhecimento de voz, síntese de som 3D, e dispositivos de *feedback* tátil (Pimentel, 1995).

No final de 1986 a equipe da NASA já possuía um ambiente virtual que permitia aos usuários ordenar comandos pela voz, escutar fala sintetizada e som 3D, e manipular objetos virtuais diretamente através do movimento das mãos. O mais importante é que através deste trabalho foi possível verificar a possibilidade de comercialização de um conjunto de novas tecnologias, sendo que o preço de aquisição e desenvolvimento tornava-se mais acessível (Pimentel, 1995).

A conscientização de que os empreendimentos da NASA baseavam-se em equipamentos comercializáveis deu início a inúmeros programas de pesquisa em Realidade Virtual no mundo inteiro (Pimentel, 1995). Organizações variando de firmas de software até grandes corporações de informática começaram a desenvolver e vender produtos e serviços ligados à Realidade Virtual (Jacobson, 1994).

Em 1989 a AutoDesk apresentava o primeiro sistema de Realidade Virtual baseado num computador pessoal (PC) (Jacobson, 1994).

3.3 O termo Realidade Virtual

O termo Realidade Virtual é creditado a Jaron Lanier, que nos anos 80 sentiu a necessidade de um termo para diferenciar as simulações tradicionais por computação dos mundos digitais que ele tentava criar. O termo é bastante abrangente assim acadêmicos, desenvolvedores de *software* e pesquisadores procuram definir Realidade Virtual baseados em suas próprias experiências. Pimentel (1995, p.15 e 17) define Realidade Virtual como o uso da alta tecnologia para convencer o usuário de que ele está em outra realidade.

O termo Realidade Virtual em geral refere-se a uma experiência interativa e imersiva baseada em imagens gráficas 3D geradas em tempo-real por computador (Pimentel, 1995). Machover (1994, p.15) afirma que a qualidade dessa experiência em RV é crucial, pois deve estimular ao máximo e de forma criativa e produtiva o usuário - a “realidade” precisa reagir de forma coerente aos movimentos do participante, tornando a experiência consistente. O principal objetivo desta nova tecnologia é fazer com que o participante desfrute de uma sensação de presença no mundo virtual (Jacobson, 1994).

Para propiciar esta sensação de presença o sistema de RV integra sofisticados dispositivos. Estes dispositivos podem ser luvas de dados, óculos, capacetes, etc.

Dois fatores bastante importantes em sistemas de RV são imersão e interatividade. A imersão pelo seu poder de prender a atenção do usuário, e a interatividade no que diz respeito à comunicação usuário-sistema (Pimentel, 1995).

CAPÍTULO 4 – A TECNOLOGIA JAVA

Java é uma linguagem orientada a objetos, e possui forte suporte para técnicas adequadas de Engenharia de Software (DEITEL, J. P., 2003; DEITEL, M. H., 2003).

Devido ao fato da linguagem Java ter sido empregada para o desenvolvimento do ILD, serão apresentadas logo em seguida algumas técnicas como o *multithreading* (multiescalonamento), que possibilita que os programadores escrevam programas com atividades paralelas.

Serão apresentados também alguns aspectos chave da API (*Applications Programming Interfaces* – interfaces de programas aplicativos) Java 3D.

Alguns detalhes da classe *String* (pacote *java.lang*). Como o aplicativo precisa fazer manipulação (quase que constante) de entradas baseadas em texto é conveniente um estudo sobre como a linguagem Java ou tecnologia Java -considerando que quando emprega se Java implicitamente refere-se a API Java, a linguagem em questão e a máquina virtual Java (JVM – *Java Virtual Machine*)- manipula *strings*. Durante a obra *string* é uma seqüência de símbolos incluindo caracteres, números e pontuação; por sua vez *String* se refere a classe *String* da linguagem Java.

O usuário necessita que a tradução não atrapalhe e seja realizada com certo paralelismo das outras atividades. É claro que se o usuário possuir uma máquina com somente um processador alcançaremos somente um pseudo-paralelismo. A linguagem Java nos fornece um meio de se criar *threads*, ou seja, linhas de execução que podem ser executadas paralelamente.

O MA deve traduzir a cadeia de caracteres fornecida, mas a interface com o usuário deve continuar atendendo aos eventos gerados pelo usuário. Assim um estudo

sobre *threads* se faz necessário e, este capítulo apresenta uma "visão" abrangente sobre o assunto.

4.1 O tratamento de strings na linguagem Java

Uma definição muito superficial sobre *string* seria: uma seqüência de caracteres interpretada como um todo (DEITEL, J. P., 2003; DEITEL, M. H., 2003).

Tal seqüência de caracteres é criada, tratada e representada de várias maneiras de acordo com a tecnologia (linguagem) utilizada.

Em Java *strings* de caracteres são representados como objetos da classe *java.lang.String*. Exatamente como acontece com outros objetos é possível criar a instância de uma *String* usando a palavra chave *new*. Alguns exemplos de instanciação podem ser vistos na Figura 10 que apresenta um trecho de código:

```
String str1 = new String();
byte[] arrayBytes = { (byte)'H', (byte)'a', (byte)'r', (byte)'r',
                      (byte)'y', (byte)'P', (byte)'o', (byte)'t',
                      (byte)'t', (byte)'e', (byte)'r' };
String str2 = new String( arrayBytes );// HarryPotter
String str3 = new String( arrayBytes, 5, 6 );// Potter
char[]  arrayChar  = { 'H', 'e', 'r', 'm', 'i', 'o', 'n', 'e',
                       'G', 'r', 'a', 'n', 'g', 'e', 'r' };
String str4 = new String( arrayChar ); //HermioneGranger
String str5 = new String( arrayChar, 8, 7 );// Granger
String str6 = new String( str5 ); //Granger
StringBuffer strBuffer = new StringBuffer("Rony Weasley");
String str7 = new String( strBuffer );
StringBuilder strBuilder = new StringBuilder("Severo Snape");
String str8 = new String( strBuilder );
String str9 = "Vinicius Durelli";
```

Figura 7: Algumas maneiras de se "criar" uma String

Um aspecto interessante envolvendo *String´s* em Java é a imutabilidade. Não se pode utilizar uma variável de referência de String para modificar o conteúdo de um objeto *String* (mais detalhes serão vistos posteriormente).

Em Java cada caractere de uma *String* é um caractere *Unicode* de 16 bits.

A classe *String* representa *strings* de caracteres, todas as *strings* de caracteres tal qual "abc" são implementadas como instâncias dessa classe (http://java.sun.com/j2se/1.5.0/docs/api).

4.2 Explicando a imutabilidade

Quando nos referimos à imutabilidade com relação à classe *String*, nos referimos ao fato de que depois de instanciado um objeto *String* o seu conteúdo não pode ser alterado.

A imutabilidade é explicada a seguir. Para criarmos uma nova *String*:

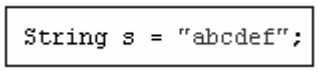

Figura 8:Instanciando um objeto String

Agora queremos uma segunda referência para o objeto *String* criado posteriormente, para isso fazemos:

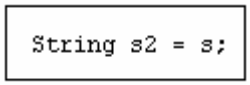

Figura 9:Atribuindo mais uma referência a um objeto instanciado anteriormente

Até aqui os objetos String estão se comportando exatamente como outros objetos. Quando fazemos:

```
s = s.concat(" mais algo"); //anexa " mais algo" no final
```

Figura 10: Alterando o conteúdo de um objeto String

Anexamos *“ mais algo”* no final da string *“abcdef”*, porém os objetos String não são imutáveis?

O que acontece realmente é que a JVM pega o conteúdo do objeto *String* “abcdef” e adiciona “ mais algo” no final, assim obtemos o valor “abcdef mais algo”. Dado que objetos *String* são imutáveis, a JVM não pode inserir essa nova *string* (a saber: “abcdef mais algo”) naquele objeto referenciado por s, portanto, criou um novo objeto *String*, deu a ele o valor “abcdef mais algo” e fez com que s o referenciasse.

Ressaltando que até esse ponto têm-se três objetos String: o primeiro “abcdef”, o segundo “ mais algo” (sim, ele também é um novo objeto *String*) e o terceiro “abcdef mais algo”. Entretanto tem-se referências somente para “abcdef” (referenciado por s2) e “abcdef mais algo” (referenciado por s2). A Figura 11 mostra o que acontecerá na memória quando se reatribuir uma variável de referência. A linha tracejada aponta uma referência excluída (a mercê do coletor de lixo).

Um outro exemplo:

```
String x = "123";
x.concat("456");
```

Figura 11:Tentando alterar o conteúdo de um objeto String

No exemplo posterior criamos um objeto *String* com o valor “123” e referenciaremos com x. Depois a JVM cria uma segundo objeto *String* com o valor “456” mais sem algo que o referencie e logo em seguida outro com o valor “123456”, também sem referência (repare que não há nenhuma atribuição); ninguém poderá alcançá-lo. A variável de referência x continuará referenciando o objeto *String* original (“123”).

4.3 Objetos String e a memória

As linguagens de programação mais sofisticadas procuram fazer um uso eficiente da memória, é comum que *strings* ocupem enormes espaços de memória (geralmente há também muita redundância no universo de *strings* de um programa). Para tornar a linguagem Java mais eficiente no uso de memória, a JVM cria uma área especial conhecida como “pool constante de *Strings*” e assim que o compilador encontra uma string literal, ele analisa o “pool constante de *Strings*” para ver se já existe uma idêntica, caso uma coincidência seja encontrada a referência ao novo valor literal será direcionada para *String* existente e nenhum objeto *String* novo será criado, agora é possível perceber porque tornar os objetos *String* inalteráveis é uma boa idéia (SIERRA; BATES, 2004).

É importante ressaltar que a concatenação de *Strings* em Java apresenta uma desempenho pobre, uma ligeira melhoria no desempenho pode ser obtida empregando a classe *StringBuffer*.

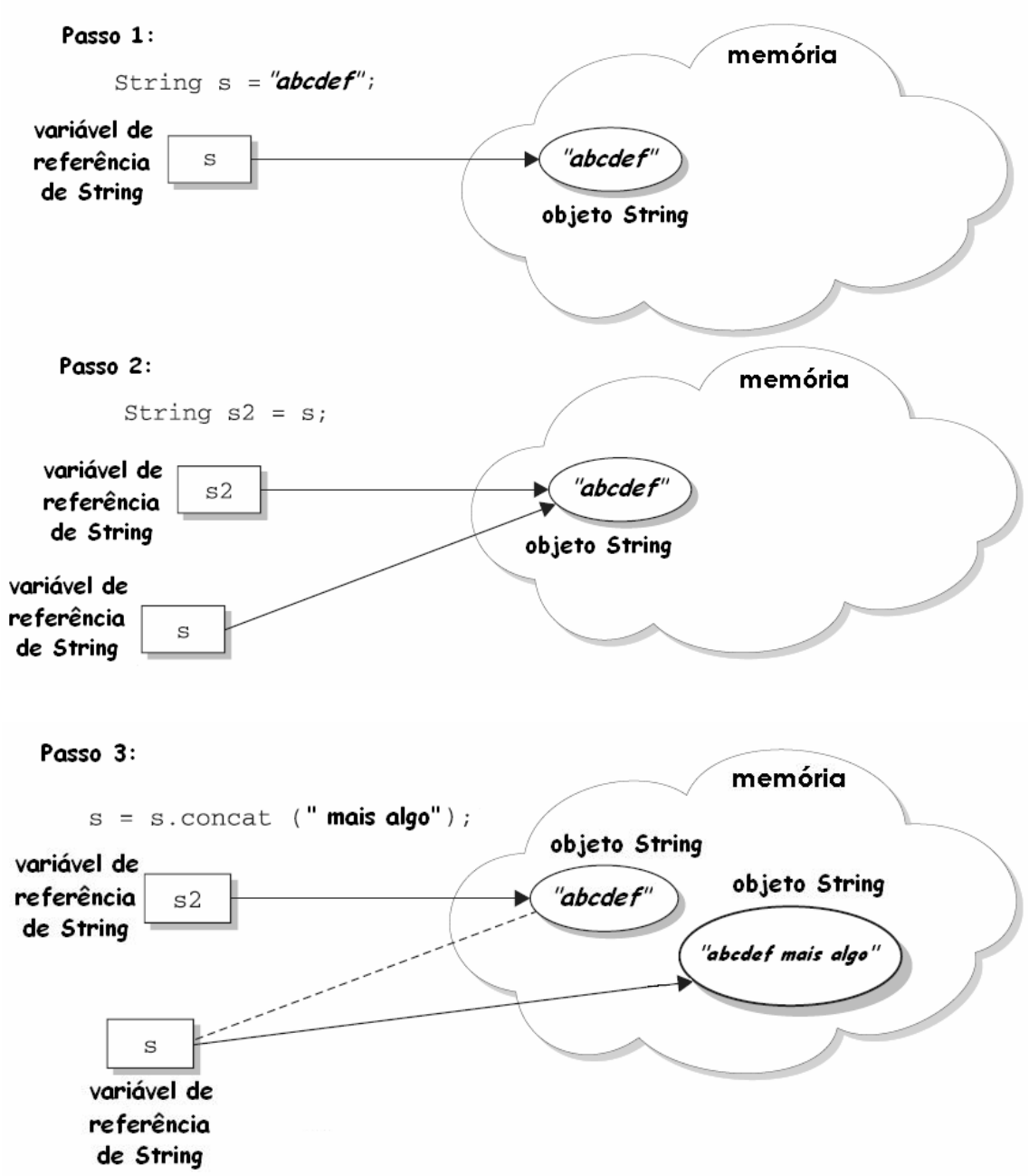

Figura 12:Ilustrando a imutabilidade dos objetos String em Java

A classe *String* é marcada como *final* para que tal funcionalidade seja sempre mantida.

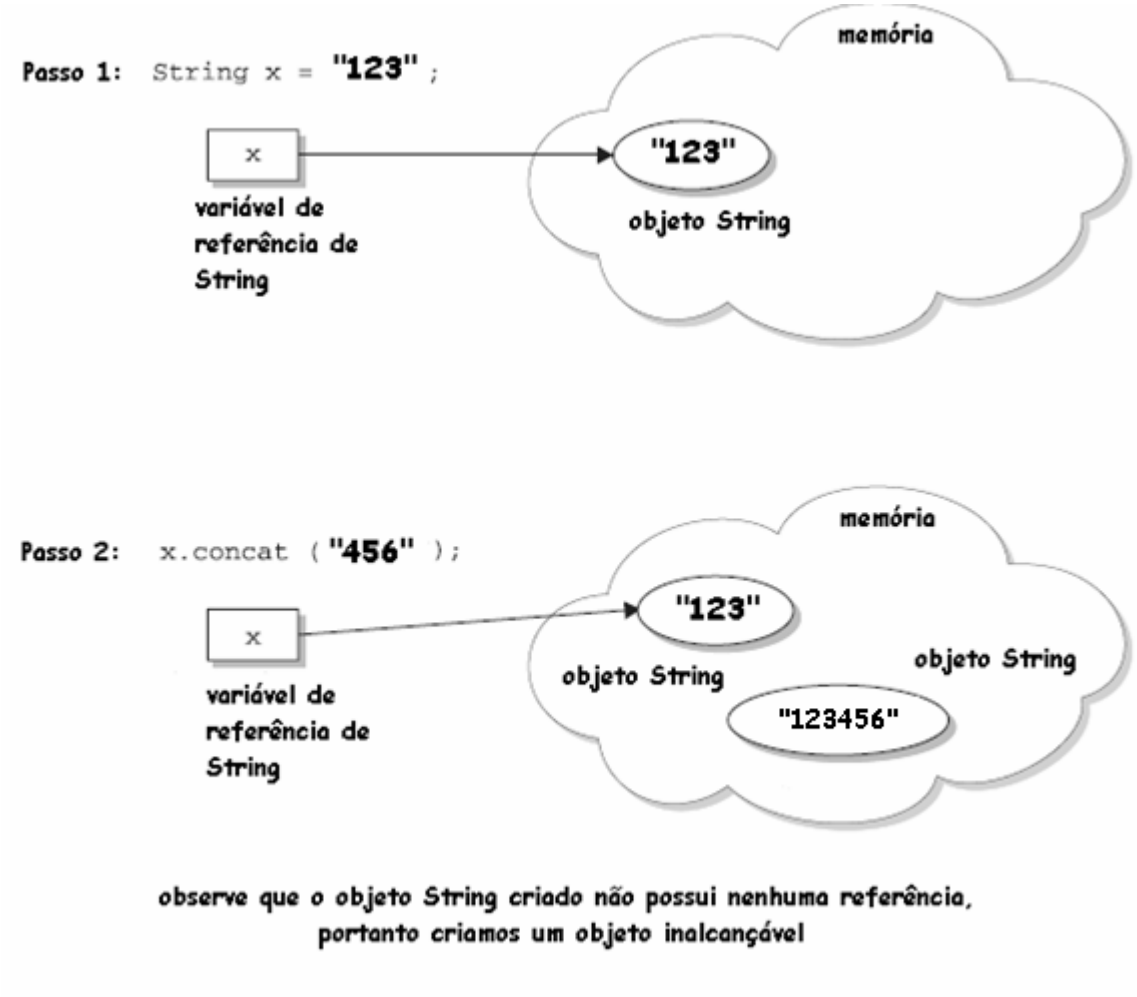

Figura 13: Um objeto String abandonado após a criação

4.4 Alguns métodos da classe String

Um erro freqüentemente cometido por pessoas iniciantes na linguagem Java é usar o operador de igualdade para verificar se dois objetos *String* são iguais. Este é um erro sutil, quando se emprega o operador de igualdade ao invés de compararmos o conteúdo dos objetos, comparamos o conteúdo das referências, ou seja, na verdade esta sendo comparado se as duas referências são para o mesmo objeto, e não o conteúdo dos objetos em questão.

A maneira correta de se realizar tal operação é usar o método *equals*. Este método compara o conteúdo de duas *Strings* (objetos) de maneira lexicográfica – compara os valores numéricos dos caracteres correspondentes nos objetos *String*, ou seja, os valores *Unicode* que representam cada caractere em cada um dos objetos *String* são comparados. O cabeçalho do método é:

```
public boolean equals(Object anObject)
```

Figura 14: Cabeçalho do método equals

Pode se comparar o conteúdo de dois objetos *String* ignorando se as diferenças entre maiúsculas e minúsculas usando *equalsIgnoreCase*, que possui a seguinte assinatura:

```
public boolean equalsIgnoreCase(String anotherString)
```

Figura 15: Cabeçalho do método equalsIgnoreCase

O método *compareTo* também pode ser usado para comparar o conteúdo de dois objetos *String* lexicograficamente. O método *compareTo* devolve zero se o conteúdo dos objetos *String* forem iguais, um número negativo se o objeto *String* que invoca *compareTo* for menor que o objeto *String* passado como argumento e um positivo caso o objeto *String* que invoca o *compareTo* for maior que o objeto *String* passado como argumento. Sua assinatura é:

```
public int compareTo(String anotherString)
```

Figura 16: Cabeçalho do método compareTo

Para comparar-se parte de dois objetos *String* emprega-se o método *regionMatches*. Sua assinatura:

```
public boolean regionMatches(int toffset,
                             String other,
                             int ooffset,
                             int len)
```

Figura 17: Cabeçalho do método regionMatches

4.5 Definição de Threads

Antes de definir o que são *threads* um pouco de contextualização pode ser útil para compreendermos sua importância e sua aplicação.

Seria realmente muito bom se fosse possível fazer uma coisa de cada vez (e fazer o melhor possível), mas as coisas não funcionam assim. Um exemplo interessante é este trabalho que você está lendo, ele foi desenvolvido durante o período de aproximadamente oito meses, e de maneira pseudo-paralela com atividades (coisas) que os autores não poderiam negligenciar, como: as outras disciplinas (sistemas distribuídos, redes de computadores entre outras), famílias e namoradas. É importante deixar claro que mesmo assim esta obra teve uma prioridade bem alta dentre as outras citadas posteriormente.

Quando programas fazem *download* de arquivos grandes como clipes de áudio ou videoclipe da *world wide web*, não se deseja ou às vezes não se pode esperar até que

o *download* termine antes de iniciar a reprodução; neste caso o navegador emprega múltiplas *threads* para trabalhar: uma descarrega o arquivo e a outra reproduz, assim estas tarefas trabalham de maneira concomitante (DEITEL, J. P., 2003; DEITEL, M. H., 2003).

Uma *thread* (algumas vezes chamada de processo leve) é um único fluxo seqüencial de controle (http://java.sun.com/docs/books/tutorial/essential/threads/).

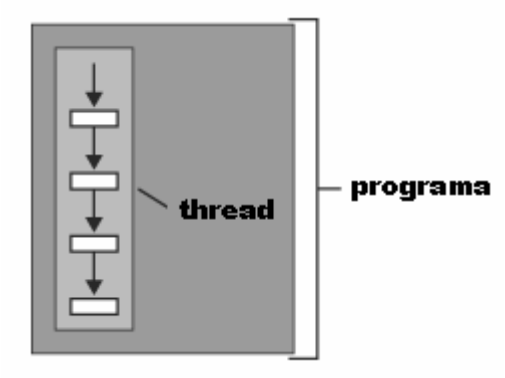

Figura 18:Thread como um fluxo seqüencial de controle

(http://java.sun.com/docs/books/tutorial/essential/threads/definition.html)

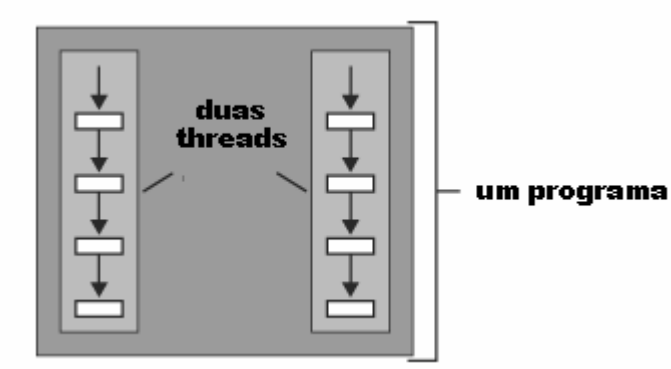

Figura 19: Várias threads em um programa propondo vários fluxos seqüências de controle

(http://java.sun.com/docs/books/tutorial/essential/threads/definition.html)

Usualmente usa-se *threads* quando se pretende "isolar" alguma tarefa e quando tal processamento é de certa forma independente e pode ser realizado em paralelo.

Durante o texto foi empregado *Thread* quando a referência for a classe que implementa o funcionamento de uma linha de execução na linguagem Java e, *thread* quando nos referimos a uma instância da classe *Thread* ou o próprio conceito de linha de execução.

4.6 Ciclo de vida de uma Thread

Threads nem sempre estão "em ação", ou seja, executando. Ocasionalmente elas ficam bloqueadas ou adormecem. A causa dos bloqueios nem sempre são as mesmas, veremos algumas causas serão vistas mais adiante neste capítulo.

As linhas de execução podem estar em um dos quatro estados: nova, passível de execução, bloqueada ou morta (HORSTMANN; CORNELL, 2003).

A Figura 20 ilustra o ciclo que vida ao qual uma *thread* se submete:

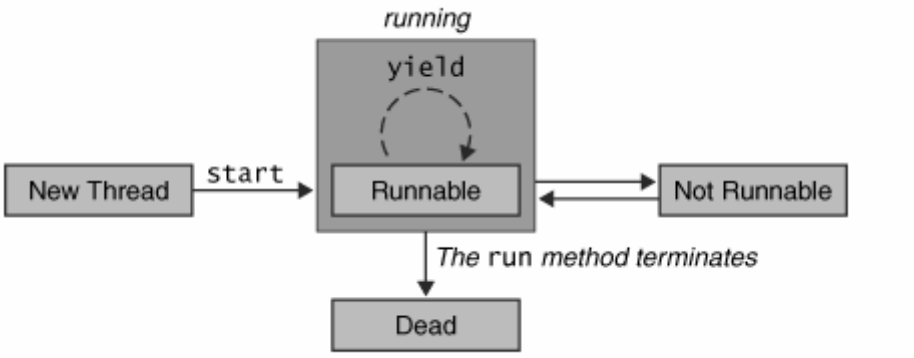

Figura 20: Ciclo de vida de um objeto da classe Thread

(http://java.sun.com/docs/books/tutorial/essential/threads/lifecycle.html)

4.6.1 Linhas de execução novas

Quando uma nova linha de execução é criada utilizando o operador *new* como no exemplo abaixo, a linha de execução criada ainda não está no estado de execução, portanto significa que ela está no estado novo, ou seja, no estado *New Thread* segundo a figura acima (HORSTMANN; CORNELL, 2003).

```
Thread t = new MinhaThread();
```

Figura 21: Instanciando um objeto da classe Thread

Uma preparação é necessária para que a linha de execução possa ser executada, tal preparação e alocação de recursos necessários são tarefas do método *start* (HORSTMANN; CORNELL, 2003).

4.6.2 Linhas de execução passível de execução

Logo depois que o método *start* "inicializa" (faz os ajustes necessários) a linha de execução passa do estado novo para o estado passível de execução (*Runnable*).

Uma linha de execução no estado passível de execução não está necessariamente em execução, que está em execução somente quando o código dentro do seu método *run* começar a ser executado, entretanto a documentação da plataforma Java não considera isso um estado separado (HORSTMANN; CORNELL, 2003).

A decisão de quando uma linha de execução deve entrar em execução fica por conta do Sistema Operacional (SO) em questão, isso porque somente o SO pode fornecer os ciclos da unidade central de processamento (UCP), assim o pacote de linhas de execução de Java interage com o SO (HORSTMANN; CORNELL, 2003).

Deve-se lembrar sempre que uma linha de execução passível de execução pode ou não estar sendo executada em determinado momento (HORSTMANN; CORNELL, 2003).

4.6.3 Linhas de execução bloqueadas

Vários motivos podem causar o bloqueio das linhas de execução. Alguns motivos são: a *thread* chama o método estático *sleep*, a linha de execução se bloqueia esperando a conclusão de uma operação de entrada/saída, a linha de execução chama o método *wait*, a linha de execução tenta obter o bloqueio de um objeto já bloqueado por outra linha de execução (mais detalhes posteriormente), ou alguém chama o método resume que foi desaprovado e não deve mais ser usado (HORSTMANN; CORNELL, 2003).

Quando uma *thread* é bloqueada ou quando ela morre outra *thread* é colocada em execução.

Em uma máquina com vários processadores, cada processador pode executar uma linha de execução, assim teremos várias linhas de execução executando em paralelo, dessa maneira uma linha de execução pode deixar de executar quando outra linha de execução de prioridade mais alta se tornar passível de execução e não houver nenhum processador disponível para executá-la (HORSTMANN; CORNELL, 2003).

4.6.4 Deixando um estado de bloqueio

Quando uma *thread* precisa sair do estado bloqueado ela deve fazer a rota oposta. Se uma linha de execução foi colocada para dormir (chamou *sleep*), ela deve então esperar o número de milisegundos expirar, se ela estiver esperando por entrada ou

saída então a operação deve ser completada, caso a *thread* tenha chamado *wait* qualquer outra *thread* deve chamar *notify* ou *notityAll* e caso uma *thread* esteja esperando pelo bloqueio de um objeto já bloqueado por outra *thread* o bloqueio deve ser liberado primeiro (HORSTMANN; CORNELL, 2003).

Caso um método que não seja compatível com o estado da *thread* seja chamado a JVM lançará uma exceção *IllegalThreadStateException*.

4.6.5 Linhas de execução mortas

Uma linha de execução pode tornar-se morta por duas razões

- "Morte natural": quando o método *run* encerra normalmente (retorna);
- "Morte abrupta": quando uma exceção não capturada ocorre dentro do método *run*, fazendo com que o mesmo expire.

Outra maneira de ser "matar" uma linha de execução é chamando o seu método *stop* que lança o erro *ThreadDeath*, mas este método foi depreciado e não se deve mais empregá-lo.

Depois de considerada morta a *thread* continua sendo um objeto da classe *Thread*, mas não é mais uma linha de execução, caso *start* seja invocado em uma instância de *Thread* morta você receberá a indesejável *IllegalThreadStateException* (SIERRA; BATES, 2004).

4.7 Prioridades

Em Java é possível atribuir prioridades para as linhas de execução. Pode-se aumentar ou diminuir a prioridade das *threads* usando o método *setPriority*, valores de um até dez podem ser usados para esse propósito. *Thread*.MIN_PRIORITY é a prioridade mínima, *Thread*.NORM_PRIORITY é uma prioridade intermediária e *Thread*.MAX_PRIORITY corresponde a maior prioridade (os valores são respectivamente 1, 5, 10).

Quando o agendador de linhas de execução tem a oportunidade de escolher uma linha de execução, usualmente ele escolhe a linha de execução com a maior prioridade e que seja passível de execução (HORSTMANN; CORNELL, 2003).

Não confie nas prioridades para controlar a sua aplicação com múltiplas linhas de execução, o escalonamento das linhas de execução de acordo com as suas prioridades não é garantido (SIERRA; BATES, 2004).

A Figura 22 mostra como ajustar a prioridade de uma linha de execução.

```
Thread t = new MinhaThread();
t.setPriority( Thread.MAX_PRIORITY );
t.start();
```

Figura 22:Ajustando a prioridade de um objeto Thread

Quando uma linha de execução é criada e não tem sua prioridade ajustada ela recebe a prioridade da linha de execução de origem (que instanciou o objeto *Thread*), a prioridade default é *Thread*.NORM_PRIORITY, ou seja, cinco.

4.8 Sincronização

Em muitos aplicativos com múltiplas linhas de execução duas ou mais *threads* precisam usualmente compartilhar o acesso aos mesmos objetos.

Em um cenário como esse o que pode acontecer de duas linhas de execução chamarem um método (que pode alterar o estado do objeto) no mesmo objeto. Dependendo em que os dados forem acessados o resultado pode ser um objeto danificado (inconsistente), tal situação é conhecida como situação de corrida (HORSTMANN; CORNELL, 2003).

Então quando as linhas de execução não são executadas de maneira “isolada“ deve-se analisar com cuidado os trechos que devem ser sincronizados.

O sincronismo funciona usando travas (*locks*); cada objeto possui uma trava embutida e quando uma linha de execução chama um método sincronizado, caso este objeto esteja livre, ou seja, nenhuma linha de execução possui o bloqueio do objeto (nenhuma outra *thread* chamou um método sincronizado do objeto em questão) a *thread* responsável pela chamada obtêm o bloqueio do objeto. Após obter o bloqueio do objeto o método é executado de maneira atômica.

Por “maneira atômica” não queremos dizer que a linha de execução não poderá ser interrompida, mas que enquanto ela não terminar a execução do método sincronizado ou chamar *wait* nenhuma linha de execução consegue chamar métodos sincronizados do objeto em questão.

Alguns pontos chave são: somente métodos podem ser sincronizados, cada objeto possui somente uma trava (*lock*), nem todos os métodos da classe devem ser sincronizados e métodos estáticos também podem ser sincronizados (SIERRA; BATES, 2004).

Caso dois métodos sejam sincronizados em uma classe, somente uma linha de execução pode acessar um deles, isso é somente uma linha de execução obtêm o bloqueio sobre o objeto, assim nenhuma outra *thread* pode entrar em qualquer um dos métodos sincronizados naquele objeto. Quando a classe possui tanto métodos sincronizados como não-sincronizados as linhas de execução podem continuar a acessar os métodos não sincronizados sem se preocupar com restrições (SIERRA; BATES, 2004).

Uma *thread* pode possuir mais de uma trava (*lock*). Por exemplo, a *thread* chama um método sincronizado do objeto A, logo depois ela chama um método sincronizado do objeto B.

Torna-se um método sincronizado usando a palavra chave *synchronized* (Figura 23).

```
synchronized void nomeMetodo() {
    //seu código aqui
}
```

Figura 23: Método sincronizado

Também é possível a criação de trechos sincronizados, é conveniente usar trechos sincronizados para as aplicações que devem ser executadas de maneira atômica em um método muito grande (Figura 24).

```
public void nomeMetodo() {
    //um trecho sincronizado
    synchronized ( this ) {
       //aqui fica o código que deve ser executado de maneira atômica
    }
}
```

Figura 24: Trecho sincronizado dentro de um método

4.9 O métodos wait, notify e notifyAll

Os métodos *wait*, *notify* e *notifyAll* servem para aperfeiçoar o mecanismo de sincronização de Java. Quando *wait* é chamado dentro de um método *synchronized*, a linha de execução corrente é bloqueada e vai para uma fila de espera dentro do objeto subjacente, liberando assim o bloqueio do objeto (HORSTMANN; CORNELL, 2003).

Tanto o método *wait* como *notify* e *notifyAll* são definidos na classe Object (raiz da hierarquia das classes Java) e não na classe Thread.

Diferentemente de uma *thread* tentando invocar um método *synchronized*, uma *thread* que chamou o método *wait* entra na fila de espera do objeto. Enquanto essa *thread* não for removida da lista de espera ela não tem chances de executar, o agendador simplesmente a ignora. A *thread* não pode sair por conta própria, é neste ponto que entram os métodos *notify* e *notifyAll*.

Para se remover uma *thread* da fila de espera de um objeto alguma outra *thread* deve chamar *notify* ou *notifyAll*.

Quando há mais de uma *thread* na lista de espera não se pode determinar qual delas será removida da lista de espera quando chamamos *notify*. Por sua vez *notifyAll* libera todas.

Deve-se chamar *notifyAll* quando o estado de um objeto mudar de um modo que seria vantajoso para as *threads* que estiverem esperando (HORSTMANN; CORNELL, 2003).

CAPÍTULO 5 – DESCRIÇÃO DO SISTEMA

As informações desde capítulo são referentes somente ao Módulo Analisador (MA) e ao Módulo Gerador (MG). Também serão apresentadas informações concernentes ao desenvolvimento de ambos e as ferramentas empregadas.

Este Capítulo tem o intuito de apresentar algumas das técnicas utilizadas durante o desenvolvimento do ILD. Além disso, apresentar as dificuldades encontradas durante as várias fases. Serão mostrados trechos de código do ILD e alguns artefatos criados durante o seu desenvolvimento.

5.1 Especificação

O objetivo do MA é servir como “ponte” entre o usuário do sistema em questão, ou seja, para o usuário do ILD com o Módulo Gerador. Além dessa tarefa o MA também se encarrega de “filtrar” possíveis caracteres inválidos, caracteres de pausa (como espaço em branco e vírgula), exibir mensagens de status entre outras funções que serão esclarecidas posteriormente.

A Figura 25 serve para conceituar a posição do MA e do MG no contexto do sistema em questão.

A seguir serão apresentados alguns esclarecimentos da abordagem utilizada para o desenvolvimento do MA e do MG e também tecnologia e ferramentas.

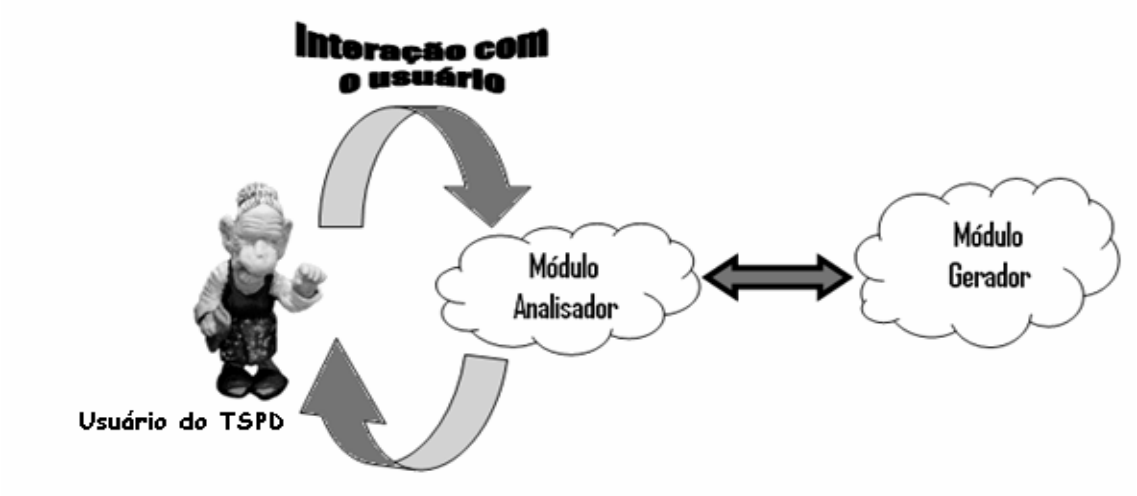

Figura 25: Posição do MA no contexto do sistema

5.2 Módulo Analisador e Módulo Gerador

Inicialmente o ILD poderia ser definido da seguinte maneira: aplicativo com o objetivo de traduzir sentenças (letras, palavras ou mesmo frases) do português para datilologia. O desenvolvimento do MA e do MG foi iterativo e incremental, seguindo algumas diretrizes do Processo Unificado (PU). Para informações adicionais sobre o PU ver o capítulo ulterior sobre este assunto.

5.3 Usabilidade, funcionalidade e desempenho

No desenvolvimento de um software que pode ser utilizado tanto por deficientes auditivos como por pessoas que não possuem uma deficiência auditiva, certos aspectos devem ser levados em consideração. Alguns aspectos são: o software não deve fazer utilização de sons para “avisar” o usuário (emitir alertas sobre o estado do software) e também as mensagens não devem possuir gírias ou onomatopéias (na maioria das vezes

os deficientes auditivos não têm conhecimento dos seus significados). O ILD deve apresentar uma interface simples e intuitiva, dispensando assim treinamento.

Quanto à funcionalidade do ILD a principal delas é fornecer a tradução de sentenças apresentando um desempenho tolerável e resistente à falhas. Resistente a falhas neste caso, seria que a "tradução" deve continuar mesmo que um caractere inválido seja encontrado na sentença, neste caso o usuário deve ser informado da ocorrência do caractere (ou caracteres) inválido e continuar com o processo de tradução.

O desempenho do ILD deve ser razoável, ou seja, não deve irritar o usuário

5.4 ILD e suas possíveis aplicações

O ILD pode ser utilizado por usuários interessados em aprender ou ensinar datilologia. Neste caso o aplicativo em questão deve ser usado como um recurso para ajudar na memorização da datilologia.

Ele (ILD) também pode ser utilizado como ponte para a comunicação com deficientes auditivos (apenas ressaltando que tal comunicação seria mais eficiente se fosse realizada em LIBRAS) e pode ser usado também para auxiliar na alfabetização de deficientes auditivos. Outras aplicações poderiam ser:

1- Atendimento a pessoas portadoras de deficiência auditiva em repartições públicas e concessionárias de serviços públicos (como determina a lei).
2- Atendimento de emergência a pessoas portadoras de deficiência auditiva em hospitais públicos, delegacias e tribunais.

Em todos os casos deve ser considerado que o deficiente auditivo conheça datilologia, além disso, no segundo caso (2) que o mesmo esteja em condições de se comunicar e tal comunicação não acarretará em problemas posteriores.

5.5 Desenvolvimento

Para o desenvolvimento de um software com qualidade considerável devemos utilizar métodos eficazes para todas as fases de desenvolvimento, melhores técnicas para a garantia de qualidade do software e um processo que defina quem faz as coisas quando e como – isso pode ser resumido em uma disciplina, a Engenharia de Software.

Uma definição para tal disciplina é:

> [...] uma área de conhecimento voltada para especificação, desenvolvimento e manutenção de sistemas de software aplicando tecnologias e práticas de ciência da computação, gerência de projetos e outras disciplinas objetivando a produtividade e a qualidade (http://pt.wikipedia.org/wiki/Engenharia_de_software).

Para Bauer (1969) apud Pressman (1995):

> O Estabelecimento e uso de sólidos princípios de engenharia para que se possa obter economicamente um software que seja confiável e que funcione eficientemente em máquinas reais.

A Engenharia de Software é extremamente necessária para a criação ou/e aperfeiçoamento de produtos de software.

Larman (2005) afirma: “conhecer uma linguagem orientada a objetos (como Java) é um primeiro passo necessário, mais insuficiente, para criar sistemas orientados a objeto”.

Assim o ILD foi desenvolvido seguindo princípios da análise e projeto orientados a objeto com o intuito de se obter um software de qualidade (entenda qualidade como: bem-projetado, manutenível e robusto).

5.6 Capturando Requisitos

O desenvolvimento de software tendo como processo de desenvolvimento o processo unificado usualmente começa com um estudo de viabilidade; a concepção. Esta análise inicial foi “ignorada”, então o primeiro passo rumo à construção do aplicativo foi a descoberta dos requisitos. Inicialmente um esforço maior foi empregado para a resolução dos requisitos funcionais.

Esse tipo de requisito pode ser considerado como declarações de funções, ou seja, funções que o sistema em questão deve fornecer (SOMMERVILLE, 2003).

Para descobrir esses requisitos sem interagir com nenhum *stakeholder* emprega se os casos de uso.

Stakeholder são pessoas que exercem influência direta ou indireta sobre os requisitos (SOMMERVILLE, 2003).

Em seguida (Figura 26) um caso de uso criado com a intenção de descobrir requisitos funcionais do ILD no formato informal.

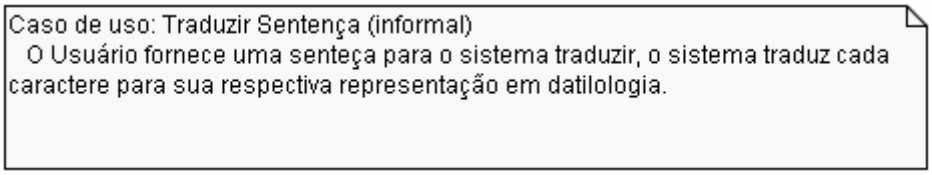

Figura 26:Caso de uso no formato informal

Um caso de uso no formato informal é útil para se obter uma visão daquilo que o sistema deve realizar, apresentando tal informação de maneira bem resumida em uma única sentença. Para obtermos mais detalhes devemos expandir o caso de uso para o formato completo. O formato completo apresenta muito mais detalhes como: pré-condições, extensões, freqüência de ocorrência, ator principal, interessado e interesses, lista de variações tecnológicas e até mesmo requisitos especiais.

O caso de uso completo para a funcionalidade Traduzir Sentença do ILD é apresentado em seguida.

Apesar disso, somente escrever casos de uso ainda não é suficiente, existem outros tipos de requisitos para serem identificados (LARMAN, 2005).

Estes outros requisitos e restrições são registrados na Especificação Suplementar. A Especificação Suplementar é um artefato do processo unificado.

Restrições não são comportamentos, são restrições de opções de projeto; recebem este nome para enfatizar a sua influência restritiva (LARMAN, 2005).

Uma Especificação Suplementar descreve atributos de qualidade FURPS+ aplicáveis ao sistema (LARMAN, 2005).

Caso de uso: Traduzir Sentença (completo)
Ator Principal: Usuário
Interessados e interesses:
- Usuário deseja uma tradução correta, com velocidade ajustável as suas necessidades e não quer complicações.

Pré-condições: #
Garantia de sucesso (pós-condições): #

Cenário de sucesso principal:
1. O Usuário digita* uma conjunto de caracteres (sentença, frase)
2. O Usuário solicita a tradução
3. O sistema exibi a representação do caracter corrente
4. O sistema avança na sentença (muda para o próximo caracter)
Repete o passo 3 e 4 até o final da sentença

Extensões (fluxos alternativos):

a* nenhuma sentença é fornecida e o usuário solicita a tradução
1. O Usuário deve ser informado que uma sentença deve ser fornecida

2a. O caracter não possui uma representação em datilologia
1. O Usuário é informado que o caracter não possui uma representação em datilologia
2. O sistema continua a partir do passo 4

2b. Já existe uma tradução em progresso
1. O Usuário é informado que deve aguarda o final da tradução atual ou interrompe-la.

Requisitos Especiais:
- Representação das mãos simbolizando as letras em datilologia em 3D
- Vários ângulos das mãos

Lista de variações tecnológicas e de dados
1a* Os caracteres que devem ser traduzidos podem ser obtidos a partir de uma arquivo .txt

Frequência de ocorrência: poderia ser quase continua

Figura 27:Caso de uso no formato completo

5.7 Mitigando os riscos

O processo unificado se baseia em quatro fases: concepção, elaboração, construção e transição.

A concepção oferece uma “visão inicial” e um entendimento um pouco mais que superficial sobre o problema, baseado neste estudo inicial (não confiável) a decisão de prosseguir ou não deve ser efetuada.

Na elaboração os requisitos encontrados na concepção são refinados e, novos requisitos devem ser descobertos e refinados. Nessa fase também ocorre a implementação da arquitetura.

A construção cuida da implementação dos elementos menos arriscados e a preparação para implantação, por sua vez a fase de transição cuida dos testes beta e da implantação.

Segundo Larman (2005, p. 41) o desenvolvimento iterativo possui os seguintes benefícios:

- Mitigação precoce, em vez de tardia, de altos riscos (técnicos, requisitos, objetivos, usabilidade, etc.);
- Progresso visível desde o início;
- Realimentação, envolvimento do usuário e adaptação imediatos, levando a um sistema refinado que atenda, de forma mais adequada, às reais necessidades dos interessados no projeto;
- A complexidade é administrada, a equipe não é sobrecarregada pela “paralisia da análise” ou por passos muito longos e complexos;
- O aprendizado obtido em uma iteração pode ser metodicamente usado para melhorar a próprio processo de desenvolvimento, iteração por iteração.

Para o desenvolvimento do aplicativo a concepção não foi realizada. Na primeira iteração da elaboração o foco foi o MA. A primeira versão do ILD foi desenvolvida com o intuito de observar os detalhes necessários para se fornecer uma tradução adequada, ou seja, quais funcionalidades referentes a tradução das cadeias de

caracteres o aplicativo deveria fornecer. A Figura 28 mostra o aplicativo desenvolvido na primeira iteração.

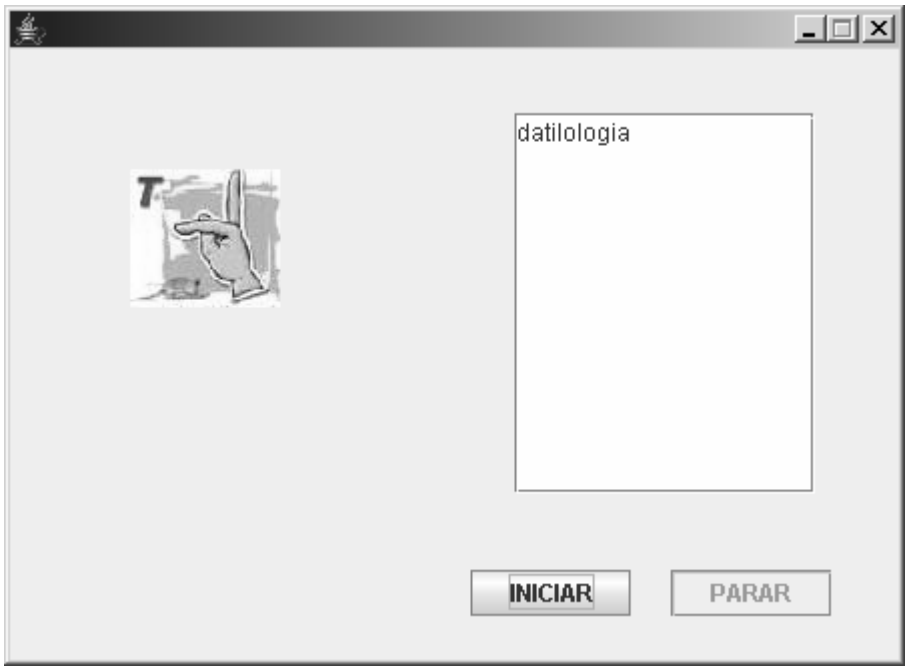

Figura 28: aplicativo desenvolvido na primeira iteração da elaboração

Durante a utilização do aplicativo criado na primeira iteração da elaboração algumas funcionalidades necessárias foram descobertas, também foi descoberto que tornar o campo onde o usuário digita as sentenças menor e permitir que arquivos de texto (extensão *.txt*) sejam carregados e traduzidos tornaria a utilização do aplicativo mais fácil. Então novas interfaces foram projetadas e testadas. A Figura 29 mostra um dos modelos de interface testados.

Figura 29: um dos "modelos" de interface testados

Como foi visto anteriormente as primeiras iterações foram focadas no desenvolvimento de uma "tradução eficiente" (funcionalidades que melhorariam a tradução) e de uma interface amigável, ou seja, de fácil entendimento e utilização.

5.8 Criando a Classe Tradutora

Durante a tradução das sentenças para datilologia é desejável que o usuário consiga interromper o progresso da tradução para iniciar uma nova tradução e consiga acessar (utilizar) as outras opções da interface. Dessa maneira a tradução deve ser realizada como uma tarefa pseudo-paralela, para conseguirmos tal coisa em Java devemos criar uma subclasse da classe *Thread* ou implementar a inteface *Runnable* e passar uma referência da classe que implementou *Runnable* para uma instância de *Thread*. Claro que uma Tradutora não é uma linha de execução (*Thread*), mas usamos o relacionamento de herança por questões técnicas e de simplicidade ao invés de criamos uma classe que implementa *Runnable*.

O código a seguir ilustra o relacionamento de herança entre a classe Tradutora e a classe *Thread*, tal relacionamento é obtido com o uso da palavra chave *extends*. O trecho de código a seguir também mostra o construtor da classe Tradutora, o construtor inicializa as variáveis de instância (Figura 30).

```
public class Tradutora extends Thread {
    private final Universo universo;
    private final String statement;
    private final JTextArea infoField;
    private final int velocidadeTraducao;

    public Tradutora( String sentenca, Universo universo, JTextArea infoField, int velTrad
    {
        super();// invoca o construtor da superclasse
        this.universo = universo;
        //passa todos os caracteres para maiusculo e elimina os espaços
        //em branco no começo e no final da sentença
        statement = sentenca.toUpperCase().trim();
        this.infoField = infoField;
        //ajusta a velocidade da tradução
        velocidadeTraducao = velTrad;
    }
```

Figura 30: O construtor da classe Tradutora

O código crucial da classe Tradutora está no método *run*, inicialmente o método *run* exibe uma mensagem informando o usuário que a tradução já foi iniciada e está em progresso. Depois dentro de um laço *for* cada caractere é passado para um método (*selecionaCodLetra*), logo após isso a linha de execução corrente é colocada para dormir. O tempo que a linha de execução corrente deve dormir é baseado na variável de referência *velocidadeTraducao*. O método *run* da classe Tradutora é mostrado em seguida na Figura 31.

```
public void run()
{
    try {
        infoField.setText("status: realizando \ntradução...");

        for ( int i = 0; i < statement.length(); i++ ) {
            selecionaCodLetra( statement.charAt( i ));
            sleep( velocidadeTraducao );
        }
        infoField.setText("status: tradução \nfinalizada");
    }
    catch(InterruptedException exception)
    {
        JOptionPane.showMessageDialog( universo, "A tradução não pode ser" +
                " concluída", "AVISO", JOptionPane.WARNING_MESSAGE );
    }
}
```

Figura 31: Método run da classe Tradutora

O método que envia mensagens para a classe *Universo* (que será abordada mais adiante neste capítulo) é o método *selecionaCodLetra*. Este método recebe como parâmetro um caractere (*char*) que é comparado no comando *switch*. Quando o nó *switch* encontra o valor correspondente de letra ele usa a referência ao objeto Universo que ele possui e chama o método *mudaLetraDatilologia*, este método será explicado posteriormente.

Um trecho do método *selecionaCodLetra* é mostrado na Figura 32.

Quando um espaço em branco é encontrado na sentença o ILD “retarda” a tradução por aproximadamente 20 milisegundos. E caso o caractere não possua um representação em datilologia, ou seja, o caractere não é uma letra válida então o usuário é informado que não há nenhuma representação em datilologia para a letra em questão.

```
private void selecionaCodLetra( final char letra )
{
    switch ( letra )
    {
        case 'B':
            universo.mudaLetraDatilologia( 1 ); //muda para 'B'
        break;
        case 'C':
            universo.mudaLetraDatilologia( 2 ); //muda pra 'C'
        break;
        case 'D':
            universo.mudaLetraDatilologia( 3 ); //muda para 'D'
        break;
        case 'E':
            universo.mudaLetraDatilologia( 4 ); //muda para 'E'
        break;
        case 'F':
            universo.mudaLetraDatilologia( 5 ); //muda para 'F'
        break;
        case 'G':
            universo.mudaLetraDatilologia( 6 ); //muda para 'G'
        break;
        case 'H':
            universo.mudaLetraDatilologia( 7 ); //muda para 'H'
        break;
        case 'I':
            universo.mudaLetraDatilologia( 8 ); //muda para 'I'
        break;
```

Figura 32: Trecho do método selecionaCodLetra

Em seguida apresentamos o código em questão pertencente a classe *Tradutora*.

```
case ' ':

    try {
        Thread.sleep( 20 ); //pausa quando encontra espaços em branco
    }
    catch( InterruptedException exception ) { /*não trata*/ }

break;
//informa que não há nenhuma representação em datilologia para o
//caractere atual
default:infoField.setText("Nenhuma\nrepresentação\n\npara: " +
        letra + "");
        universo.mudaLetraDatilologia( -1 );//passa valor inválido
```

Figura 33:Trecho que interrompe a tradução

5.9 A Classe Universo

A classe *Universo* é uma subclasse de *JInternalFrame* (pacote *javax.swing.JInternalFrame*). A classe foi nomeada assim porque é nela que o *SimpleUniverse* é instanciado, colocando de maneira mais simples, é nela que as mãos em três dimensões são mostradas.

Esta classe é a mais complexa do ILD porque apresenta várias técnicas não-triviais. Neste capítulo apenas uma visão parcial das técnicas empregadas na classe *Universo* será fornecida, mais detalhes sobre Java3D serão apresentados posteriormente neste capítulo.

A classe Universo troca mensagens com a classe *Tradutora*, mais especificamente um objeto da classe *Tradutora* envia mensagens para um objeto da classe *Universo* "coordenando" quais mãos (letras em datilologia) devem ser exibidas e quando.

Um ponto importante no "chaveamento" (troca) das mãos é o nó *Switch* (*javax.media.j3d.Switch*). Este nó controla o que deve ser renderizado em determinado momento. Para adicionarmos um novo filho a este nó usamos o método *addChild*. Por sua vez para ajustarmos o filho que deve ser exibido em determinado momento utilizamos o método *setWhichChild*.

Quando se tem como objetivo carregar um arquivo com a extensão *.obj* deve se empregar um *ObjectFile*. A Figura 34 mostra a criação de um objeto *ObjectFile*.

```
ObjectFile obj =
      new ObjectFile(ObjectFile.RESIZE,(float)(10.0 * Math.PI / 180.0));
```

Figura 34:Instanciando um objeto ObjectFile

A Figura 35 mostra como carregar uma arquivo de extensão *.obj;* o arquivo em questão é o *V.obj* e se localiza na pasta hands.

```
v = obj.load("hands\\V.obj");
```

Figura 35: Carregando um arquivo .obj

5.10 A Classe TSPDMain

A classe principal do aplicativo, ou seja, aquela que possui o método estático e público *main* é a classe *TSPDMain*. Esta classe cria a interface gráfica com o usuário e define algumas classes internas usadas no tratamento de eventos gerados pelo usuário.

A classe interna *TraduzHandler* responde a eventos gerados pelo botão *'Traduz'* da interface com o usuário. Ela testa se a sentença fornecida não é vazia e se já existe uma tradução sendo realizada, caso ambos não sejam verdadeiros ela instância um novo objeto da classe *Tradutora* passando os parâmetros necessários para o construtor. A classe *TraduzHandler* é definida dentro do corpo da classe *TSPDMain* (Figura 36).

Outra classe interna definida no corpo da classe *TSPDMain* é a classe *PararHandler*. Este classe responde a eventos gerados pelo botão *'Parar'* da interface com o usuário; tecnicamente ela interrompe a *thread* encarregada da tradução e muda para uma mão que não representa nenhum símbolo em datilologia. Nessa classe também foram colocados alguns comandos para otimizar o uso da memória (Figura 37).

```
//classe interna responsável pelo tratamento dos eventos
//de tradução (acionada pelo botão TRADUZ)
private class TraduzHandler implements java.awt.event.ActionListener {

    public void actionPerformed(java.awt.event.ActionEvent evento) {
        String conteudo = jTextField1.getText();

        //não há nada para ser traduzido
        if (conteudo.equals("")) {
            javax.swing.JOptionPane.showMessageDialog(TSPDMain.this,
                    "Insira o texto", "AVISO",
                    javax.swing.JOptionPane.WARNING_MESSAGE);
            jTextField1.setText("Seu texto aqui!!!");
        } else {
            jInternalFrameUniverso.setVisible(true); //mostra form
            //testa se não existe uma tradução em progresso
            if (tradutora == null || !tradutora.isAlive()) {
                jTextArea1.setText("Traduzindo...");
                tradutora = new Tradutora(jTextField1.getText(),
                        jInternalFrameUniverso, jTextArea1, velocidade);
                tradutora.start();
                //instanciar a classe responsável pela tradução
            } else {
                javax.swing.JOptionPane
                        .showMessageDialog(
                                TSPDMain.this,
                                "Já existe uma tradução em progresso, "
                                        + "clique em PARAR e logo após em TRADUZ para "
                                        + "traduzir a nova sentença.",
                                "AVISO",
                                javax.swing.JOptionPane.WARNING_MESSAGE);
            }
        }
    }
}
```

Figura 36:A classe interna TraduzHandler

```
private class ParарHandler implements java.awt.event.ActionListener {

    public void actionPerformed(ActionEvent evento) {
        if (tradutora == null) {
            jInternalFrameUniverso.mudaLetraDatilologia(0);
            System.gc();//chama o coletor de lixo
            return;
        } else {
            tradutora.interrupt();
            jInternalFrameUniverso.mudaLetraDatilologia(0);
            System.runFinalization();//roda o finalize dos objetos
            //sem referência que serão
            //desalocados
            tradutora = null;
        }
    }
}
```

Figura 36: A classe interna ParарHandler

5.11 Uma visão geral

Para uma compreensão melhor do aplicativo a Figura 38 mostra um diagrama resumido. Resumido porque não apresenta nas classes o compartimento onde ficam as operações e nem o compartimento dos atributos; apenas um esboço das classes principais.

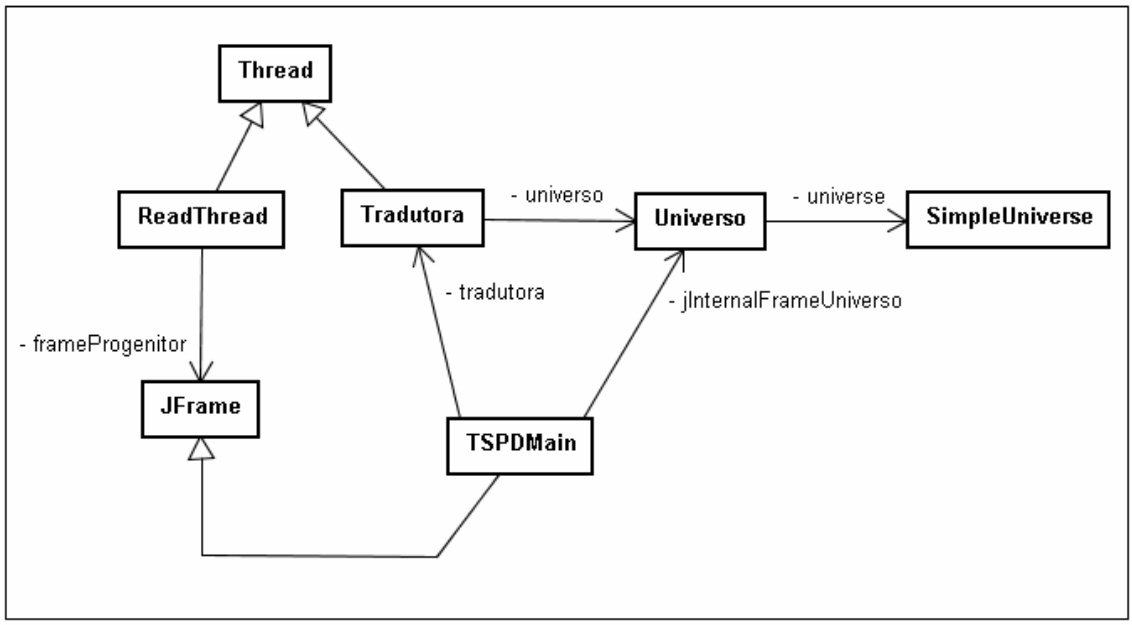

Figura 37*:* Esboço das classes principais

5.12 Modelagem das mãos

A primeira mão virtual foi desenvolvida no formato Wavefront (OBJ) devido à dificuldade de modelagem em Java3D. Com o uso de ferramentas de modelagem 3D a modelagem fica mais prática, permitindo que a mão fique mais perfeita o que poderia levar muito tempo no desenvolvimento direto, ou seja, utilizando a API Java 3D.

O primeiro protótipo da mão foi feito com a ajuda da ferramenta *Wings 3D* (Figura 38), uma ferramenta de modelagem poligonal livre e de fácil utilização (http://www.wings3d.com/).

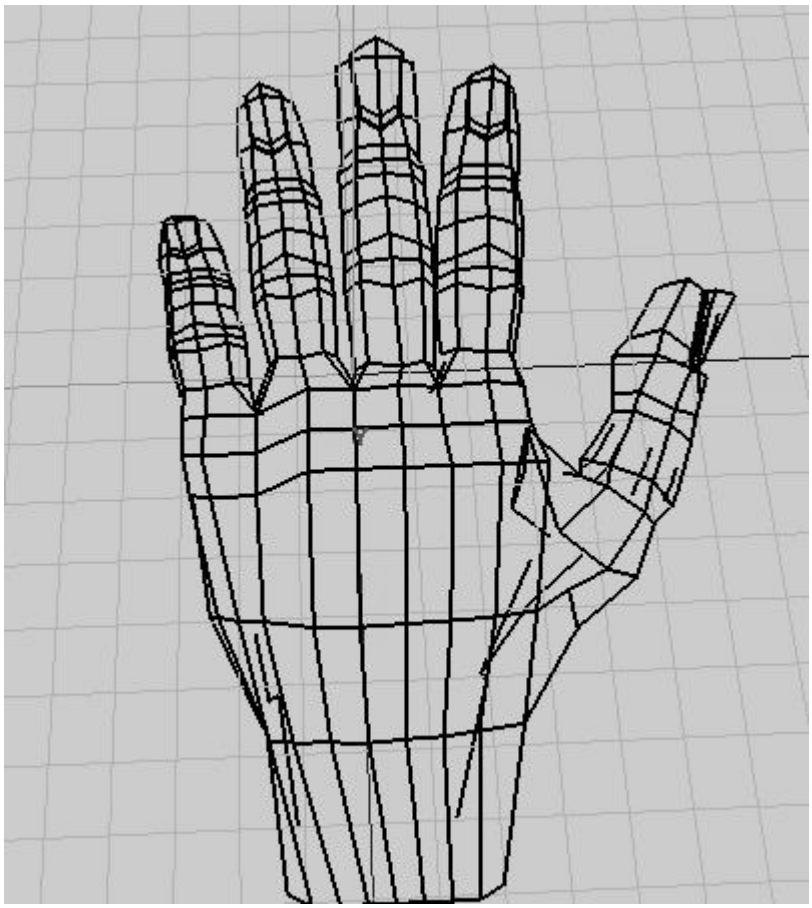

Figura 38: Protótipo da mão feita na ferramenta Wings 3D

Através de um cubo (Figura 40), foram feitos os dedos e palmas e muito depois foram feitos os detalhes das unhas. Após a modelagem básica da estrutura da mão foi feito uma renderização para suavizar, melhorando consideravelmente o visual da mão conforme pode ser visto na Figura 41.

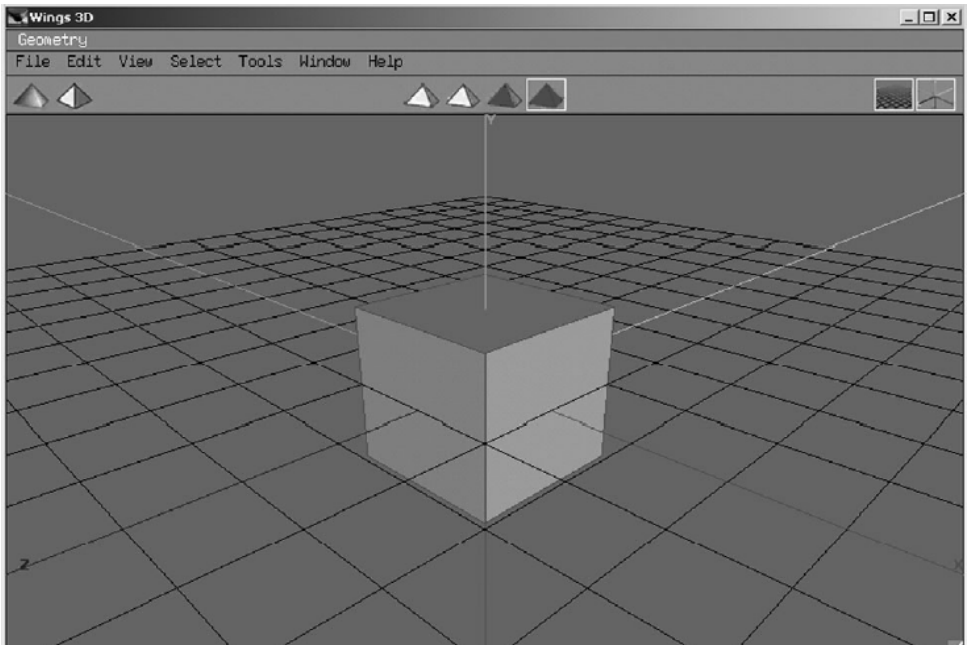

Figura 39: Cubo, o primeiro passo para iniciação de criação da mão

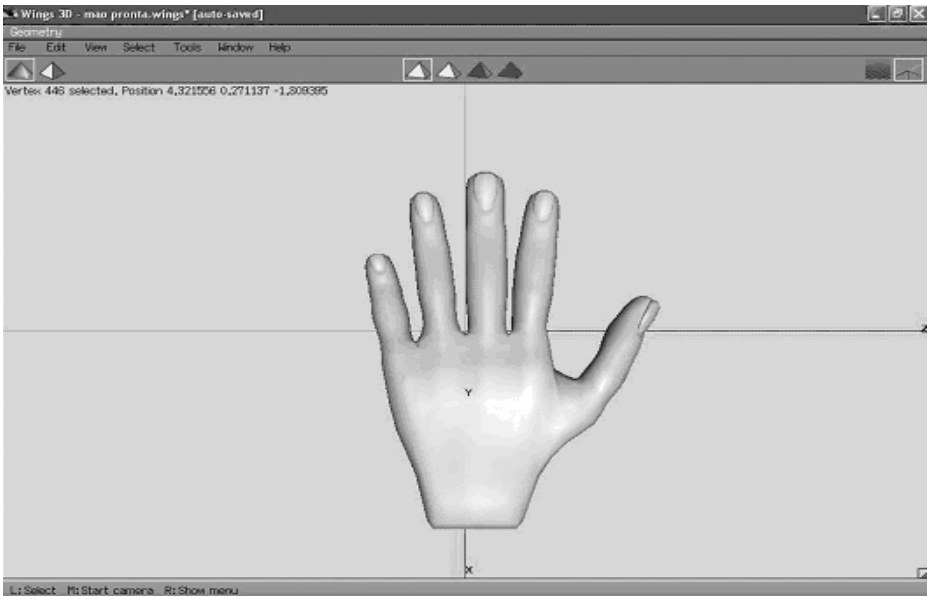

Figura 40: Mão renderizada

Depois de desenvolvida a mão, foi necessário fazer as transformações para cada uma das letras em datilologia, para isso foi utilizado outro software chamado *Blender*, a mudança de software foi necessária devido a maior facilidade para transformações geométricas.

Na Figura 42 é mostrado a mão transformada na letra Z, para isso foram feitas várias rotações e translações nos dedos.

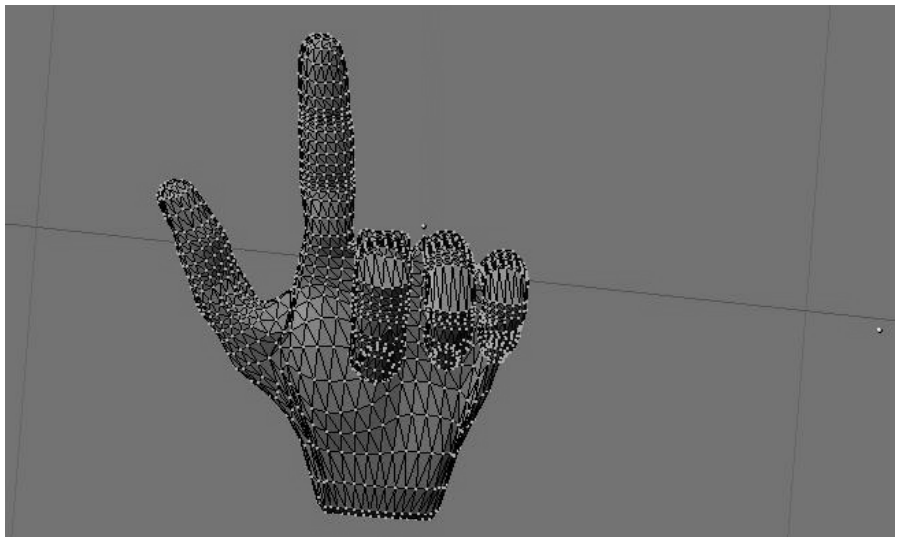

Figura 41: Letra Z modelada com a ferramenta Blender 3D

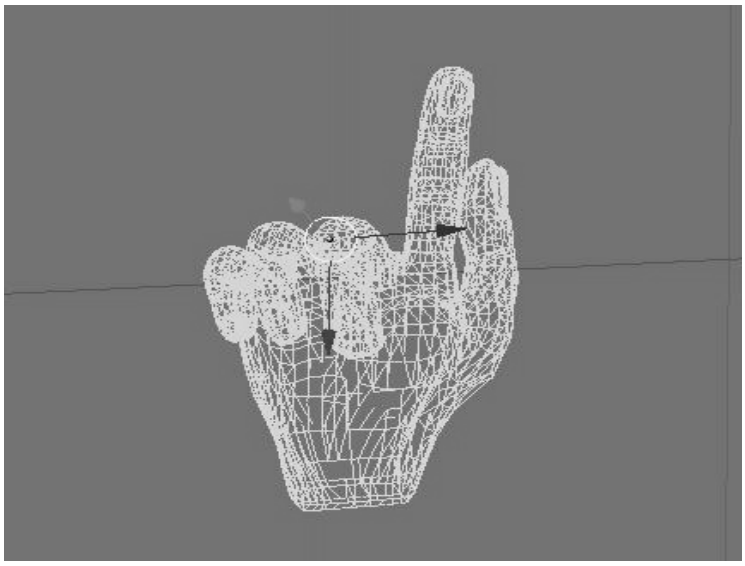

Figura 42:Mão em wireframe

Os recursos de wareframe, seleção e criação de *Edges Loop* facilitam a modelagem das letras.

5.13 O aplicativo ILD

Um aplicativo que pode ser utilizado por pessoas interessadas em aprender ou se aperfeiçoar em datilologia deve apresentar uma interface simples de se interagir, dado isso, durante o desenvolvimento do ILD existiu grande preocupação em manter a interface simples (de fácil utilização).

Um aplicativo voltado para deficientes auditivos não deve apresentar onomatopéias e palavras em um idioma diferente (inglês, por exemplo); isso dificulta a compreensão.

Figura 43: ILD realizando tradução de uma cadeia de caracteres

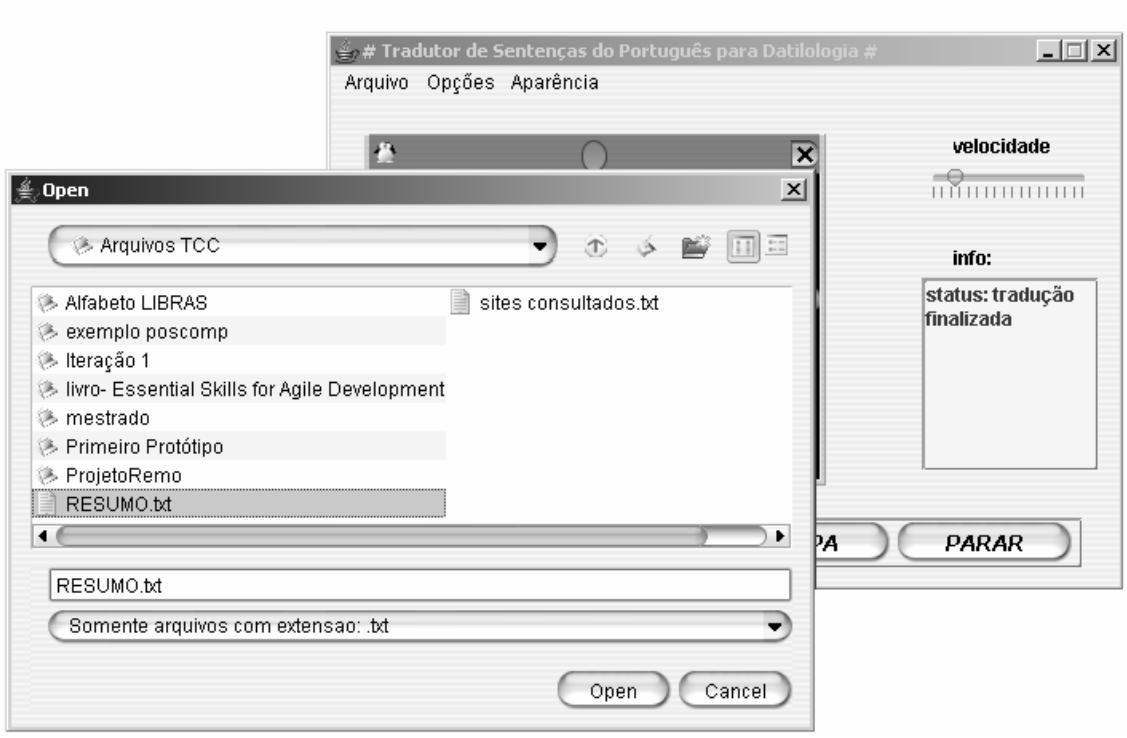

Figura 44: Selecionando um arquivo de texto para tradução

Figura 45: Ajustando o tamanho de exibição

O aplicativo fornece um meio para se ajustar a velocidade da “tradução” (Figura 46).

Figura 46: Barra de velocidade

CONCLUSÃO

Para a realização do trabalho em questão foi necessário inicialmente um estudo sobre as metodologias de ensino difundidas para educação dos surdos. Com isso, compreender em quais metodologias o ILD poderia ser utilizado como uma ferramenta auxiliar na aprendizagem.

A linguagem Java é uma linguagem complexa, e para o desenvolvimento do aplicativo algumas técnicas como a manipulação de *Strings* e uma técnica uma pouco mais "avançada" como *threads* (linhas de execução/processos peso-leve) foram estudadas; aspectos para a criação da interface gráfica não apresentaram nenhuma complexidade.

A API Java 3D foi estudada para realizar o "carregamento" das mãos em .obj, a criação do universo onde as mãos são apresentadas (*SimpleUniverse*) e principalmente para realizar o chaveamento entre as mãos. Para a modelagem das mãos em questão foram estudadas parcialmente duas ferramentas, uma delas para se realizar a modelagem e a outra para criar o arquivo *.obj*. Respectivamente as ferramentas são *Wings 3D* e *Blender*.

O ILD captura uma cadeia de caracteres e traduz para datilologia, ou seja, exibe a sua representação em datilologia. Caso um símbolo que não possua um correspondente em datilologia seja encontrado na cadeia de caracteres o usuário é informado do fato (caracteres de pontuação, por exemplo, não possuem representação em datilologia).

Um possível aperfeiçoamento do trabalho poderia ser a inclusão de LIBRAS, quando o programa reconhecesse uma palavra que possui representação em LIBRAS a mesma é traduzida e, caso não possua correspondente em LIBRAS a datilologia é

empregada na tradução. Isso possibilitaria uma tradução mais eficiente dado que em LIBRAS uma palavra é usualmente representada como um único gesto, as palavras em outra língua, inglês, por exemplo, seriam traduzidas usando datilologia. Ao invés de mostrar somente uma mão neste caso seria necessária uma representação do corpo humano completo ou pelo menos da cintura para cima.

Espera-se que o presente trabalho contribua para difundir a Libras, a datilologia, as metodologias de ensino mais empregadas e atue como inspiração para que novos aplicativos dessa categoria sejam criados, ajudando assim os deficientes auditivos a se expressarem e serem entendidos.

Este trabalho foi de grande valia no aprendizado dos autores. Oferecendo um entendimento mais aprofundando da linguagem Java, uma introdução na API Java 3D, um estudo sobre métodos de análise (casos de uso) e desenvolvimento utilizando um processo iterativo.

REFERÊNCIAS

DEITEL, H. M.; DEITEL, P. J. **Java: Como programar.** Porto Alegre: Bookman, 2003. 1386 p.

LAWRENCE, K.; MANNING, R. **Mantenha o Seu Cérebro Vivo: exercícios neuróbicos para ajudar a prevenir a falta de memória e aumentar a capacidade mental**. Rio de Janeiro: Editora Sextante, 2000. 142 p.

LARMAN, C. **Utilizando UML e Padrões**. Porto Alegre: Bookman, 2005.

FOWLER, M. **UML Essencial**. Porto Alegre: Bookman, 2005. 608 p.

SIERRA, K; BATES, B. **Java 2: Certificação SUN para Programador e Desenvolvedor Java 2**. Rio de Janeiro: Alta Books, 2004. 442 p.

PRESSMAN, R. **Engenharia de Software**. São Paulo: Makron Books, 1995. 1056 p.

HORSTMANN, C.; CORNELL, G. **Core Java 2: Recursos Avançados**. São Paulo: Makron Books, 2000. 850 p.

PIMENTEL, K.; TEIXEIRA, K. **Virtual reality - through the new looking glass**. 2.ed. New York, McGraw-Hill, 1995.

Jacobson, L. **Realidade virtual em casa**. Rio de Janeiro, Berkeley, 1994

MACHOVER, C.; TICE, S.E. VIRTUAL REALITY. **IEEE Computer Graphics & Applications,** Jan. 1994.

RHEINGOLD, H. **Virtual reality**. New York, Touchstone, 1991.

SUN MICROSYSTEM. **Overview (Java 2 Plataform SE 5.0)**. Disponível em: <http://java.sun.com/j2se/1.5.0/docs/api/>. Acessado em 3 de Novembro de 2005.

SUN MICROSYSTEM. **Lesson: Threads: Doing Two or More Tasks At Once**. Disponível em: <http://java.sun.com/docs/books/tutorial/essential/threads/ >. Acesso em 13.jul.2005.

RYBENÁ. Disponível em: < http://www.rybena.org.br >. Acesso em: 15.jun.2005.

SUN MICROSYSTEM. **Lesson: I/O: Reading and Writing (but no 'rithmetic)**. Disponível em: < http://java.sun.com/docs/books/tutorial/essential/io/index.html >. Acesso em 25.jul.2005.

ALFABETO MANUAL. Disponível em: <http://site.surdosol.com.br/index.php?comunidade=alfabetos >. Acesso em 25.mai.2005

WINGS 3D **Software livre de modelagem 3D tutoriais** Disponível em: http://www.wings3d.com.br/tutoindex.htm Acesso em 05.mar.2005

BLENDER 3D **Using Blender** Disponivel em: http://www.blender3d.com/cms/Using_Blender.80.0.html Acesso em 18.jun.2005

SUN'S JAVA 3D ENGINEERING TEAM ; 2000. Disponível em : <http://java.sun.com/developer/onlineTraining/java3d/> Acesso em 10.ago.2005

ISABEL HARB MANSSOUR **Introdução à Java 3D**. Disponível em: < http://www.inf.pucrs.br/~manssour/Java3D/ Acesso em 10.Jul.2005

A.L. BICHO, L.G. DA SILVEIRA JR, A.J.A. DA CRUZ E A.B. RAPOSO. **Programação Gráfica 3D com OpenGL, Open Inventor e Java 3D**. REIC - Revista Eletrônica de Iniciação Científica. v. II, n. I, março, 2002. Disponível em: <http://www.sbc.org.br/reic/edicoes/2002e1/tutoriais/ProgramacaoGrafica3DcomOpenGLOpenInventoreJava3D.pdf>. Acesso em 20.Jul.2005

Libras – Alfabeto Manual e Números. Disponível em: < http://www.libras.org.br/Thumbnails/FrameSet.htm >. Acesso em 12.jun.2005