INSTITUTO DE PESQUISAS TECNOLÓGICAS DO ESTADO DE SÃO PAULO - IPT

CRISTIANO GARCIA COSTA

FATORLEX - Compilador para Linguagem FatoRH/W

Dissertação apresentada ao Instituto de Pesquisas Tecnológicas do Estado de São Paulo - IPT, para obtenção do título de Mestre em Engenharia de Computação
Área de Concentração: Engenharia de Software

Prof. Dr. Marco Dimas Gubitoso
Orientador

São Paulo

2004

Ficha Catalográfica
Elaborada pelo Centro de Informação Tecnológica do IPT

C837f Costa, Cristiano Garcia
FATORLEX - Compilador para linguagem FatoRH/W. / Cristiano Garcia Costa. São Paulo, 2004.
90p.

Dissertação (Mestrado em Engenharia de Computação) - Instituto de Pesquisas Tecnológicas do Estado de São Paulo. Área de concentração: Engenharia de Software.

Orientador: Prof. Dr. Marco Dimas Gubitoso

1. Compilador FATORLEX 2. FatoRH/W (Linguagem de programação) 3. JAVA (Linguagem de programação) 4. Linguagem de Modelagem Unificada - UML 5. Engenharia de software 6. Tese I. Instituto de Pesquisas Tecnológicas do Estado de São Paulo. Centro de Aperfeiçoamento Tecnológico II. Título

05-19 CDU 004.4'422(043)

“The best way to predict the future is to invent it.”
— SIR ALAN KAY

AGRADECIMENTOS

Agradeço à minha esposa, Luciana e ao meu filho, Gabriel, que me deram apoio durante o período em que me dediquei exclusivamente ao trabalho. Agradeço também ao meu ilustre orientador Dr. Gubitoso que me apresentou o maravilhoso mundo do LaTeX.

RESUMO

Este trabalho foi realizado com o intuito de solucionar um problema real de uma empresa de desenvolvimento de software[1] que possuía um sistema legado cuja função era interpretar as fórmulas de cálculo das verbas do *Sistema de Folha de Pagamento*[2].

O trabalho em questão trata de um compilador escrito em Java - daqui em diante chamado de Fatorlex - para linguagem FatoRH/W, linguagem esta desenvolvida pela Soft Trade para definição de fórmulas pelo usuário final de seus sistemas. Tanto a linguagem quanto os componentes da solução serão apresentados no decorrer deste trabalho.

A solução encontrada foi desenvolvida dentro dos padrões da Engenharia de Software e além de ser baseada na tecnologia Java, utiliza ferramentas[3] *Open Source* na composição da sua arquitetura.

Palavras-chave: Engenharia de Software, Compilador, Java, UML.

[1] A empresa em questão trata-se da Soft Trade Engenharia de Sistemas Ltda

[2] FatoRH/W - Sistema de Administração de Pessoal

[3] JLex - Analisador Léxico e Java CUP - Analisador Sintático

FATORLEX - Compiler to FatoRH/W language

ABSTRACT

The purpose of this paper was to solve a real problem facing a software development company [4]. This company had inherited a system whose function was to interpret the formulas of calculation for the *Payroll System* [5].

This work deals with a compiler written in Java - from now on denominated Fatorlex - for language FatoRH/W. This language was developed by Soft Trade to define formulas for the end user of its systems. The language and the components of the solution will be presented in this work.

The solution that was found was developed according to the recognized standards of Software Engineering and besides being based on Java technology, it uses *Open Source* tools [6] in the composition of its architecture.

Keywords: Software Engineering, Compiler, Java, UML.

[4] The company is Soft Trade Engenharia de Sistemas Ltda

[5] FatoRH/W - Payroll Administration System

[6] JLex - A lexical analyzer generator for Java and Java CUP - a parser generator for Java

LISTA DE ILUSTRAÇÕES

SUMÁRIO

1 INTRODUÇÃO

> A Engenharia de Software é uma disciplina que integra *métodos, ferramentas e procedimentos* para o desenvolvimento de software de computadores. Uma série de diferentes paradigmas da engenharia de software foram propostos, cada um exibindo potencialidades e fragilidades, mas todos tendo uma série de fases genéricas em comum. As três fases, *definição, desenvolvimento e manutenção*, são encontradas em todo desenvolvimento de software, independentemente da área de aplicação, tamanho do projeto ou complexidade (PRESSMAN, 1995).

A flexibilidade na configuração dos sistemas tornou-se um diferencial competitivo na acirrada disputa por clientes no mercado brasileiro de software. Quanto mais as empresas disponham recursos em seus aplicativos que permitam definir as regras do seu negócio, menos dispendiosos ficam a manutenção e a evolução dos sistemas adquiridos ou dos que por ventura venham a ser adotados.

Neste cenário nasceu a linguagem FatoRH/W. Da necessidade de configurar as fórmulas de um Sistema de Folha de Pagamento, foi criada uma linguagem de definição de regras, totalmente em português, para poder ser operada pelo próprio usuário da aplicação, com total flexibilidade na maneira de escrever as regras. Foram introduzidas estruturas disponíveis em linguagens de programação [1], além de tipos de dados [2] e a possibilidade de extensão da linguagem com o conceito de funções do usuário e do sistema.

Inicialmente foi criado um compilador para o linguagem FatoRH/W utilizando a linguagem de programação Visual Basic da Microsoft. O desenvolvimento deste software demandou cerca de 6 meses de trabalho integral de uma equipe formada por 3 recursos. Atualmente este software encontra-se em produção em uma base de aproximadamente 100 clientes.

Como a demanda do mercado guia a estratégia das empresas e o mercado clama por soluções multiplataformas, principalmente com a popularização de sistemas de código aberto, como o Linux, a Soft Trade adotou o Java como nova plataforma de desenvolvimento para suas aplicações. O objetivo deste trabalho é o desenvolvimento de uma versão Java do compilador da linguagem FatoRH/W. O projeto foi batizado de **Fatorlex** e está sendo desenvolvido conforme padrões estabelecidos pela Engenharia de Software em seu estado de arte.

O trabalho encontra-se dividido em 3 partes que correspondem às 3 fases genéricas encontradas em todos paradigmas da Engenharia de Software: Definição, Desenvolvimento e Conclusão.

---

[1] Exemplo: Desvio Condicional:SE e Loop Condicional:FAÇA ENQUANTO

[2] Texto, Número, Data e Lógico.

A fase de Definição inicia com a análise do sistema, onde é estabelecido o que será produzido como resultado deste trabalho, no caso o Fatorlex, além dos recursos usados para a montagem da solução.

Seguindo na fase de Definição temos o planejamento do projeto, onde é definido o plano do mesmo com a especificação de itens relacionados à viabilidade de desenvolvimento do software, tais como: estimativas de risco, alocação de recursos, prazo de entrega, programação de tarefas, entre outros.

A análise de requisitos do software finaliza a parte do trabalho dedicada a Definição do sistema. Esta fase é concluída com a especificação dos requisitos da linguagem FatoRH/W.

A fase de Desenvolvimento é voltada ao projeto, implementação, teste e manutenção da solução especificada na fase de Definição. O Desenvolvimento inicia com o projeto do Fatorlex, onde os requisitos são traduzidos numa representação do software usando como linguagem de modelagem a UML [3].

Por se tratar do desenvolvimento de um compilador, o processo de compilação é detalhado nessa fase, bem como os componentes usados na solução: Java, JLex e Java CUP.

Seguindo no Desenvolvimento temos a especificação da linguagem FatoRH/W através da definição dos símbolos no arquivo JLex e da gramática da linguagem no arquivo Java CUP.

O Desenvolvimento é concluído com a abordagem dos pontos referentes a verificação, liberação e manutenção do software produzido neste trabalho.

A última fase do trabalho é formada pela Conclusão. Nela são relatadas as conclusões de fato tiradas da experiência adquirida ao longo de todas as fases do projeto.

Durante a Conclusão também são discutidas questões relacionadas a limitações e possiveis extensões do trabalho.

[3] Linguagem de Modelagem Unificada

Parte I

Definição

2 ANÁLISE DO SISTEMA

A análise do sistema é o primeiro passo na fase de definição do software. Essa etapa dará as bases do que se pretende desenvolver e como isso será realizado. Em relação ao que se pretende fazer, vale a pena ressaltar que o objetivo do trabalho não é criar uma linguagem e sim um compilador, pois a linguagem FatoRH/W já existe e está sendo usada por cerca de 100 clientes da Soft Trade.

2.1 Especificação do Sistema

2.1.1 Introdução

O desenvolvimento de compiladores é uma disciplina que vem evoluindo ao longo do século e a criação das linguagens de programação juntamente com seus compiladores que reconhecem a sintaxe da linguagem e produzem código de mais baixo nível, permitiram que o desenvolvimento de software evoluisse para diversas áreas de aplicações (científica, comercial, industrial, entre outras) e que várias linguagens fossem criadas aumentando a produtividade do desenvolvimento de software e facilitando a construção de programas.

O trabalho da construção de compiladores foi explorado de tal maneira que hoje possuímos ferramentas confiáveis de geração de código de compiladores, apesar de uma implementação definitiva de um compilador normalmente necessitar de otimizações manuais do código gerado pela ferramenta.

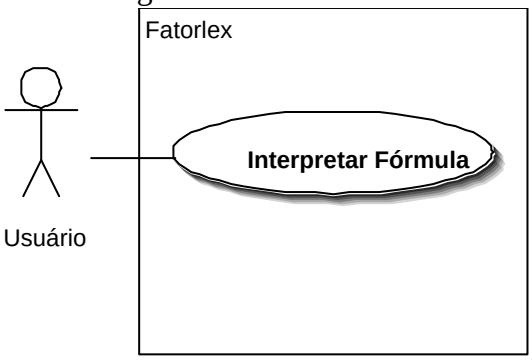
Dois tipos de ferramentas são atualmente difundidos na criação de compiladores: Geradores Léxicos e Geradores Sintáticos. Uma vez especificada a linguagem, no nosso caso a linguagem FatoRH/W - com seus tipos, operadores, comandos e outros tipos de declaradores - temos condições de utilizar as ferramentas acima na criação do compilador Fatorlex para linguagem FatoRH/W.

2.1.2 Escopo e propósito do documento

O objetivo deste documento é definir o que se espera entregar no final do trabalho de mestrado e como será composta a solução proposta, com suas funcionalidades e arquitetura. O diagrama UML[1](BOOCH; RUMBAUGH; JACOBSON, 1999) de Caso de Uso do Fatorlex expõe as funcionalidades que o software irá prover. Na segunda parte do trabalho, que descreve a fase de Desenvolvimento, veremos o projeto detalhado em diagramas UML bem como sua implementação usando os componentes da solução Fatorlex.

[1]Linguagem de Modelagem Unificada

Figura 2.1: Diagrama de Caso de Uso do Fatorlex



2.1.3 Visão Geral

2.1.3.1 Objetivos

Deverá ser criado um programa de computador em linguagem Java (Fatorlex), que reconheça/traduza ou aponte erros em um texto de entrada escrito na linguagem FatoRH/W. A linguagem FatoRH/W será detalhada em capítulo próprio. A tradução do texto de entrada será feita da linguagem FatoRH/W para linguagem Java.

Pela natureza da linguagem de desenvolvimento, no caso Java, o Fatorlex será multiplaforma, ou seja, estará disponível para todas plataformas onde o Java seja portável. Pelo menos duas plataformas serão utilizadas no desenvolvimento e validação do compilador: Windows e Linux.

2.1.3.2 Restrições

O Fatorlex deverá manter compatibilidade com a versão atual do compilador desenvolvida em Visual Basic e disponibilizada para os clientes.

2.1.4 Arquitetura do Sistema

Para aumentar a produtividade e ganhar flexibilidade no desenvolvimento serão utilizadas ferramentas de criação de compiladores. Duas ferramentas foram selecionadas para compor a solução: JLex e Java CUP.

O JLex fará a análise léxica do texto de entrada. Nesta fase todas as palavras do texto serão separadas em tokens e identificadas como um dos símbolos da linguagem FatoRH/W.

A fase seguinte será a análise sintática do texto de entrada. Nela, o analisador sintático Java CUP receberá os tokens do JLex e validará a estrutura do texto verificando se o texto é um programa válido da linguagem FatoRH/W.

3 PLANEJAMENTO DO PROJETO DE SOFTWARE

Passada a fase inicial da análise do sistema, onde foi definido o motivo pelo qual o software deverá ser criado, o papel que ele desempenhará, bem como seu escopo, o engenheiro de software deverá, então, elaborar um plano para o Desenvolvimento do mesmo. Este plano deverá levar em conta varios itens ligados a viabilidade de desenvolvimento do software, tais como: estimativas de risco, alocação de recursos, prazo de entrega, programação de tarefas, entre outros.

3.1 Plano do Projeto de Software

3.1.1 Introdução

Cada fase do desenvolvimento do software deverá ser acompanhada por um documento que reproduza os aspectos relevantes da fase em questão. Nesta fase do projeto abordaremos os aspectos relativos a esforço, recursos, custos e tempo.

3.1.2 Escopo e propósito do documento

O Plano do Projeto de Software é o documento elaborado pelo Engenheiro de Software contendo as diretrizes que conduzirão o desenvolvimento do software. O mesmo contém estimativas referentes prazos e riscos, bem como a alocação dos recursos do projeto. Este documento poderá servir de apoio a tomada de decisão relacionada a continuidade do desenvolvimento projeto e projeção dos custos de desenvolvimento.

3.1.3 Objetivos do projeto

- **Objetivos**

  Deverá ser criado um programa de computador em linguagem Java (Fatorlex), que reconheça/traduza ou aponte erros em um texto de entrada escrito na linguagem FatoRH/W.

- **Funções principais**

  Análise de um texto contendo instruções em linguagem FatoRH/W. O texto deverá ser interpretado e transformado em um programa Java, caso o mesmo, seja um programa válido na sintaxe da linguagem FatoRH/W. Caso o texto não seja um programa FatoRH/W, o sistema deverá informar o motivo que levou o mesmo a concluir que o programa era inválido, ou seja, a mensagem de erro relativa ao programa que esta sendo interpretado.

- **Questões de desempenho**

O programa antigo usado pela Soft Trade e compilado na linguagem Visual Basic, leva no máximo 1 segundo para interpretar e gerar o código de um programa de 50 linhas. Espera-se que o novo software obtenha desempenho equivalente ao programa atual.

- **Restrições técnicas e administrativas**

  Uma diferença significativa em relação ao software antigo versus o novo software é o fato do sistema anterior gerar uma procedure na linguagem do banco de dados usado pelo cliente [1], enquanto o software novo deverá gerar um programa Java.

3.1.4 Estimativas de projeto

A previsibilidade do projeto com base em estimativas confiáveis é fundamental para diversos itens do gerenciamento do projeto, tais como prazos de entrega, orçamentos de custos, entre outros. A estimativa deve preferencialmente utilizar dados históricos de projetos anteriores e alguma técnica estatisticamente aceitável para cálculos dos valores estimados.

- **Dados históricos usados nas estimativas**

  O Fatorlex foi criado para substituir um software legado desenvolvido pela própria Soft Trade [2]. Logo, a base de dados históricos natural para o cálculo das estimativas é o projeto FatoRH/W.

  O projeto da linguagem FatoRH/W consumiu 3000 horas, alocadas em 3 recursos dedicados 8 horas por dia cada um; o projeto passou por todas as fases (Análise, Projeto, Implementação e Testes) da engenharia de software em 6 meses. Foram desenvolvidas cerca de 10000 linhas de código fonte em Visual Basic.

- **Técnicas de estimativa**

  A técnica usada para o projeto Fatorlex foi a relação linha de código versus esforço homem-meses [3](MCCONNELL, 1996). O esforço homem-meses é o número de meses que somente um desenvolvedor levaria para completar o sistema do início ao fim, dedicando-se integralmente ao projeto. Este valor é estimado com base em dados coletados de projetos[4] registrados no Instituto de Engenharia de Software (SEI) de 3 tipos de produtos: Sistemas, Negócios e Pacotes. Ou seja, tendo-se uma idéia da quantidade de linhas do projeto e o tipo de produto a ser desenvolvidos, pode-se estimar o esforço homem-meses e o menor número de meses em que o produto pode ser desenvolvido. O tamanho da equipe pode ser calculado pela divisão do esforço homem-meses pelo número de meses estimados para o desenvolvimento do projeto.

- **Estimativas**

  Pela tabela "Efficient Schedules"(MCCONNELL, 1996), temos o valor 24 para o esforço homem-meses de um produto do tipo Sistema de 10 mil linhas e o valor 5 para um produto do tipo Negócio. O Fatorlex tem em média 80% do tipo Sistema[5]

[1] PL/SQL do Oracle ou T-SQL do Microsoft SQL Server
[2] O software a ser substituído é o FatoRH/W
[3] Tabela: Efficient Schedules (Lines of code x Effort man-months)
[4] Software Engineering Economics (Boehm 1981)
[5] Relativo a parte da linguagem e do compilador

e 20% de tipo Negócio[6]; isso corresponde a um valor aproximado de 20 homem-meses (24 * 80% + 5 * 20%), ou seja, 1 ano e 8 meses de trabalho de um único desenvolvedor em tempo integral e pela mesma tabela um prazo mínimo de 7 meses.

A tabela de valores homem-meses é limitada ao mínimo de 10 mil linhas, pois projetos inferiores a essa quantidade de linhas normalmente são desenvolvidos por um único desenvolvedor e a faixa de valores é muito ampla, dependendo muito mais da produtividade do desenvolvedor do que algum padrão de processo.

Porém, em função da adoção de geradores léxicos e sintáticos e da utilização de uma linguagem orientada o objetos, teremos uma redução de 90% do número de linhas de código da aplicação. Este ganho em linhas de código viabilizou a adoção deste projeto como trabalho de tese de mestrado de um único desenvolvedor, com prazo estimado de 1 ano de projeto em dedicação parcial(menos de 4 horas por dia).

3.1.5 Riscos do projeto

> Embora seja fútil tentar eliminar o risco e questionável tentar minimizá-lo, é fundamental que os riscos assumidos sejam os riscos certos (DRUCKER, 1974).

O risco está presente no desenvolvimento de qualquer projeto e por isso faz-se necessária a sua identificação, avaliação e administração. Qualquer indivíduo que já tenha desenvolvido algum software que entrou em produção, sabe o quanto é improvável que se consiga, na fase de planejamento, predizer todos os riscos envolvidos no projeto. Por isso o Engenheiro de Software deverá tentar identificar os riscos com maior probabilidade de ocorrência e os que poderão causar os maiores impactos no projeto, monitorando-os, e tomando ações pró-ativas, para que os mesmos interfiram o mínimo no desenvolvimento do software.

3.1.5.1 Análise dos riscos

- **Identificação**

  Entre os principais riscos do projeto Fatorlex, temos a utilização do analisador léxico JLex e do analisador sintático Java CUP. O sucesso do projeto depende da viabilidade de uso dos analisadores, pois caso contrário o ganho estimado pelo uso de geradores de código não ocorrerá, aumentando o número de linhas de código a ser desenvolvido e por conseqüência o prazo de desenvolvimento do projeto, inviabilizando a entrega do projeto dentro do prazo da tese de mestrado.

  Outro risco que tem maior impacto na utilização final do software, é a adoção da linguagem Java como resultado da compilação dos programas escritos na linguagem FatoRH/W, pois no software legado, as linguagens adotadas são nativas do banco de dados, como é o caso do Transact-SQL do Microsoft SQL Server e o PL/SQL da Oracle. O fato de usar as linguagens nativas aumenta o desempenho dos programas gerados, pois os mesmos ficam compilados dentro do banco de dados e não precisam gerenciar conexões todas as vezes que são executados. Porém ao usar o Java como linguagem intermediária entre o código FatoRH/W e o banco de dados, apesar do ganho de portabilidade e independência das linguagens proprietárias dos

[6] Possui regras exclusivas da área de Recursos Humanos

fabricantes de banco de dados, corremos o risco do software não obter um desempenho dentro dos limites aceitos pelos usuário do software legado, e este risco poderia inviabilizar a implantação do Fatorlex.

Um risco que sempre está presente em qualquer projeto é o cumprimento do prazo do mesmo. No nosso caso não é diferente, temos a data limite da entrega da tese que deve ser respeitada como data limite do projeto Fatorlex.

- **Estimativa dos riscos**

  Estima-se que o risco da utilização dos analisadores léxicos e sintáticos seja algo em torno de 10%, pois já se tem um protótipo funcional com 90% dos comandos, operadores e funções das linguagens FatoRH/W, o que aumenta a segurança do uso de analisadores/geradores no software Fatorlex.

  Porém, com relação ao risco de desempenho, temos um quadro mais preocupante - por volta de 20% de risco - o que equivale a média de tempo de 0.01 segundos do software legado, multiplicado pelo fator 8 vezes 10, menos 1 segundo. O fator 8 foi encontrado em testes de desempenho em relação a média de tempo de execução de programas em linguagem nativa do banco de dados versus aplicações escritas em Java com acesso ao banco de dados. Como o tempo de espera tolerável pelo usuário para o cálculo da folha de pagamento de um funcionário é de 1 segundo e cada funcionário tem em média 10 verbas [7], temos uma margem de 0.20 segundos para os programas gerados em Java pelo Fatorlex.

  Em relação à entrega do projeto dentro do prazo da tese, temos um risco médio, devido à alta complexidade do projeto e ao baixo número de recursos envolvidos: apenas um desenvolvedor.

- **Avaliação**

  Uma vez identificados e estimados os riscos, uma avaliação deverá ser realizada na tentativa da definição de níveis de riscos do projeto. Os níveis de risco do projeto são definidos em função dos riscos identificados, da probabilidade de ocorrência de cada risco e do impacto que cada um poderá causar no projeto, de acordo com a matriz Impacto/Probabilidade:

Impacto/Prob	M.Alto	Alto	Médio	Baixo	M.Baixo
Catastrófico	Alto	Alto	Moderado	Moderado	Baixo
Crítico	Alto	Alto	Moderado	Baixo	—
Aceitável	Moderado	Moderado	Baixo	—	—
Insignificante	Moderado	Baixo	Baixo	—	—

  Construímos nossa tabela de risco baseada na matriz Impacto/Probabilidade do risco:

Impacto/Prob	M.Alto	Alto	Médio	Baixo	Muito Baixo
Catastrófico	-	-	prazo	-	uso de analisadores
Crítico	-	-	-	desempenho	-
Aceitável	-	-	-	-	-
Insignificante	-	-	-	-	-

  No projeto Fatorlex foram definidos 3 níveis de risco, conforme tabela abaixo:

[7] Cada verba (Provento/Desconto) da folha de pagamento equivale a um programa Fatorlex

Nível	Descrição do Nível	Riscos envolvidos
1	Alto	-
2	Moderado	Prazo
3	Baixo	Analisadores e Desempenho

3.1.6 Administração dos riscos

- **Monitoração dos riscos**

  Temos riscos situados em dois níveis: Moderado para o prazo de entrega e Baixo para o uso de analisadores léxicos e sintáticos e baixo também para as questões relativas ao desempenho. Um plano detalhado de como o risco será prevenido e/ou amenizado deve ser criado. Um planejamento de ação pode ser usado para amenizar o risco via uma resposta imediata. A probabilidade de um risco ocorrer ou o impacto potencial de um risco pode ser minimizado ao se lidar com o problema o mais cedo possível no projeto. O planejamento de contingência pode ser usado para controlar o risco e invoca uma resposta determinada antecipadamente. Um gatilho deve ser armado, e caso seja disparado, o plano de contingência deve ser posto em ação.

  O processo de identificação, análise, planejamento e monitoração dos riscos deve estar definido antes do projeto ter início, pois se os riscos ficarem fora de controle todo o projeto estará sujeito a não cumprir a sua finalidade.

4 ANÁLISE DE REQUISITOS

4.1 Especificação de Requisitos de Software

4.1.1 Introdução

Um problema comum no desenvolvimento de software é a instabilidade dos requisitos, que ocorre quando o cliente, os usuários ou a equipe do projeto trazem novos requisitos ou alterações de requisitos já levantados, ou mesmo solicitam a exclusão de algum requisito. Com o objetivo de recomendar um processo formal de desenvolvimento de software para minimizar o problema descrito acima, descreveremos algumas diretrizes da Gestão de Requisitos do nível 2 do CMM (AHERN; CLOUSE; TURNER, 2003), que permitem a definição e o controle dos requisitos nos quais se baseiam os compromissos assumidos no projeto. São elas:

- Identificação de requisitos
- Cadastramento dos requisitos
- Controle de alterações dos requisitos
- Monitoração dos requisitos

4.1.2 Objetivos

O objeto da análise de requisitos é estabelecer e manter a concordância do cliente em relação ao que o software deve fazer, fornecendo aos desenvolvedores um melhor entendimento dos requisitos. A definição dos limites de atuação do software, juntamente com uma base para planejamento do projeto deve fornecer uma referência para as estimativas de custo e tempo para o desenvolvimento. Nessa fase também deverão ser estabelecidas as interfaces de usuário do software.

4.1.3 Identificação de requisitos

Nesta etapa são identificadas as necessidades dos interessados em relação ao software a ser desenvolvido. As informações coletadas junto aos interessados servem para gerar uma lista de requisitos do software. Esta etapa é realizada em conjunto com representantes do cliente, usuários chave e outros especialistas da área de aplicação quando necessário. Para identificar os requisitos do software são consideradas as informações obtidas durante as entrevistas com os usuários / representantes / especialistas. Os requisitos estatutários e regulamentares aplicáveis ao software e os itens identificados e estabelecidos em acordo com o cliente (produtos a serem entregues, tecnologia utilizada, prazos de entrega, forma

de entrega, etc.), devem ser repassados pelo responsável à equipe do projeto. Também são levadas em consideração as informações obtidas a partir de trabalhos desenvolvidos no cliente, bem como versões já existentes do software. Ainda nesta etapa é elaborado o Dicionário de Dados, descrevendo os dados levantados, sem se preocupar com a especificação física dos atributos. Pode-se utilizar, neste caso, a própria ferramenta de modelagem de banco de dados para a geração do Dicionário de Dados. Caso o cliente forneça algum produto para ser incorporado ao software ou para ser consultado com a finalidade de extrair do mesmo informações relevantes ao desenvolvimento do software em questão, devem ser seguidas as diretrizes estabelecidas para controlar os produtos fornecidos pelo cliente descritas na Gestão de Projeto.

4.1.4 Cadastramento dos requisitos

Os requisitos podem ser classificados em:

- Requisitos funcionais - São identificados a partir das necessidades dos interessados e descrevem as funcionalidades que o software deve disponibilizar para gerar algum benefício mensurável para os usuários.

- Requisitos não funcionais - Incluem os requisitos de desempenho, requisitos lógicos de dados, requisitos de qualidade do software, requisitos estatutários e regulamentares, e requisitos de documentação.

  - Requisitos de desempenho - São requisitos tais como: número de terminais suportados, número de usuários simultâneos, volume de informação que deve ser tratado, número de transações por unidade de tempo, condições de normalidade e de pico. Os requisitos de desempenho são especificados de forma quantitativa e mensurável e são medidos nos testes de aceitação.
  - Requisitos lógicos de dados - São caracterizados pelas estruturas de dados a serem usadas pelo software, tais como: tabelas em bancos de dados, arquivos convencionais, arquivos partilhados entre o software e outros sistemas, as fontes e destinos dos dados assim como os formatos destes, os relatórios gerados, etc.
  - Requisitos de qualidade - São características não funcionais requeridas pelo cliente, baseadas nas características e sub-características definidas pela norma ISO-9126, tais como: funcionalidade, confiabilidade, usabilidade, eficiência, manutenibilidade e portabilidade, além dos requisitos de desempenho. Os requisitos estatutários e regulamentares dizem respeito às leis e normas aplicáveis ao negócio e/ou ao software a ser desenvolvido.

  Uma definição completa de requisitos não funcionais pode incluir outros aspectos como:

  - Restrições de memória principal ou secundária
  - Modos de operação, por exemplo: processamento em lote ou manutenção
  - Requisitos de adaptação a ambientes específicos (por exemplo, que aspectos devem ser configuráveis durante a instalação em determinado local).

Uma vez levantados, os requisitos devem ser controlados e acompanhados quanto às alterações que se fizerem necessárias. Para isso eles são cadastrados com as seguintes informações:

- Importância essencial, desejável ou opcional:
  - Essencial : requisito cujo não atendimento torna o software inaceitável.
  - Desejável : requisito cujo atendimento aumenta o valor do software, mas cuja ausência pode ser relevada em caso de necessidade (por exemplo de prazo).
  - Opcional : requisito a ser cumprido se houver disponibilidade de prazo e orçamento, depois de atendido os demais requisitos.
- Complexidade - É a complexidade estimada de se implementar o requisito, considerando o esforço e os riscos de implementação, em comparação com outros requisitos do projeto; pode ser muito alta, alta, média, baixa ou muito baixa.
- Instabilidade - É a estimativa da probabilidade de que o requisito venha a ser alterado no decorrer do projeto, com base na experiência em projetos correlatos; pode ser muito alta, alta, média, baixa ou muito baixa.
- Status - estado do requisito desde a sua identificação:
  - Original : quando o requisito permanece inalterado desde sua identificação.
  - Adicionado : quando o requisito é incluído após a aprovação do Documento de Visão.
  - Alterado : quando o requisito sofre alguma alteração.
  - Excluído : quando um requisito deixa de ser necessário.

Uma vez identificados e cadastrados, os requisitos funcionais e não funcionais devem ser especificados até um nível de detalhe suficiente para as atividades de análise e projeto do software. As especificações dos requisitos devem ser validadas com os envolvidos e uma vez validadas devem ser inseridas no controle de versão. Para as especificações validadas é coletado um Aceite de Produtos de Projeto. Os requisitos do software podem sofrer alterações no decorrer do projeto por diversos motivos:

- Descoberta de defeitos e inadequações nos requisitos originais
- Falta de detalhamento suficiente nos requisitos originais
- Melhor entendimento do problema por parte dos usuários ou dos desenvolvedores
- Alterações no contexto do projeto
- Fatores externos (por exemplo: mudança de legislação, alterações tecnológicas, gerenciais, legais e políticas, etc.).

4.1.5 Controle de alterações dos requisitos

Considerando-se o impacto que as alterações de requisitos geralmente têm nos prazos e custos dos projetos, as mesmas devem acontecer de forma controlada. As alterações nos requisitos podem ser solicitadas a qualquer momento durante o projeto, por qualquer membro da equipe, e devem ser feitas através dos documentos de Requisição de Alteração, que são encaminhados ao Gerente do Projeto para análise e aprovação/reprovação. Caso as alterações solicitadas sejam aprovadas, o documento de Controle de Requisitos deve ser revisado para atualizar o status dos requisitos. O status reflete as modificações solicitadas e aprovadas: um requisito pode ser excluído, adicionado ou alterado. A alteração de um requisito pode se referir ao seu conteúdo (por exemplo, um detalhe de um caso de uso ou uma característica de um requisito não funcional), ou pode se referir a um de seus atributos (por exemplo, um requisito pode ser rebaixado de essencial para desejável). Algumas alterações têm impacto no escopo do projeto e, consequentemente no prazo e custo, portanto o Plano do Projeto e o Cronograma do Projeto também devem ser revisados e atualizados para absorverem este impacto.

4.1.6 Monitoração dos requisitos

As requisições de alteração possuem campos que contêm, para cada requisito alterado, a data da alteração, o tipo da alteração e o motivo da alteração, o que permite localizar as causas mais comuns de mudanças de requisitos. Essas informações devem ser usadas em conjunto para analisar futuras solicitações de alterações em requisitos, nesse projeto ou em projetos similares.

4.2 Linguagem FatoRH/W

O principal requisito do Fatorlex é a linguagem que será implementada pelo compilador. Sem uma especificação clara do que deverá ser disponibilizado na linguagem, seria impossivel começar o projeto do compilador. Seguimos, então, com a especificação da linguagem FatoRH/W.

4.2.1 Introdução

Sendo voltada para o usuário final, a estrutura da linguagem foi concebida com o objetivo de facilitar o aprendizado. Para tanto, os comandos e funções foram definidas em português e os tipos de dados limitam-se a apenas quatro; os operadores também são bastante simplificados.

4.2.2 Tipos de Dados

O definição de tipos de dados necessita de um contexto para ser mais facilmente compreendida e este contexto se refere ao programa ou função que está sendo descrita.

Normalmente para desenvolver um programa precisamos criar variáveis que recebem valores que serão usados no decorrer no programa em comparações, atribuições, argumentos de função e numa infinidade de combinações que por vezes só fazem sentido se forem relacionadas com valores do mesmo tipo, ou seja, normalmente não fará sentido o usuário somar um texto a uma data, isto produziria um resultado indefinido, difícil de ser separado posteriormente.

Este e outros motivos vindos da história da computação nos levaram a implementar somente os quatro tipos básicos na linguagem:

- **Texto**

  Este tipo de dados contém cadeia de caracteres delimitados por aspas. (“) O texto que está entre aspas não possui restrições em relação ao conteúdo do texto, podendo conter números, letras e símbolos.

  ```
  “Este é um texto livre com 40 caracteres ”
  “”
  “O texto acima é uma string vazia”
  ```

- **Número**

  Tipo de dado usado para armazenar valores numéricos. Pode receber valores inteiros ou decimais.

  ```
  3492
  -52
  83,30
  ```

- **Data**

  As datas serão armazenadas em um tipo de dado específico que usa os delimitadores {} na definição do conteúdo. Os formatados disponíveis são Dia/Mês/Ano ou Mês/Ano. O ano pode ser expresso com 2 ou 4 dígitos.

```
{09/03/2000} Nove de março de 2000.
{12/2003} Primeiro de dezembro de 2003.
{25/12/04} Vinte e cinco de dezembro de 2004.
```

- **Lógico**

  Este tipo de dado guarda valores booleanos, pode ser definido pelas palavras reservadas **SIM NÃO** ou pelo número zero **0** para valores falsos e qualquer número diferente de zero para valores verdadeiros.

```
NÃO
0
SIM
1
```

4.2.3 Variáveis

Dois pontos importantes a ressaltar quanto a utilização das variáveis. Primeiro, não haverá limite para a criação de variáveis. Segundo, deve ser respeitado o tipo do dado inicialmente armazenado na variável, ou seja, se a variável foi iniciada com um dado do tipo numérico, posteriormente ela não poderá armazenar um texto.

4.2.4 Operadores / Expressões

- **Operadores Lógicos**
  - Operador **E**

    Usado para verificar se duas ou mais condições são verdadeiras, retornando um verdadeiro caso isso ocorra.
  - Operador **Ou**

    Usado para verificar se pelo menos uma entre duas condições é verdadeira, retornando um verdadeiro nesta ocorrência. Este operador trabalha de forma não exclusiva.

- **Operadores Relacionais**

  Devolve um verdadeiro se a comparação entre os valores à sua esquerda e à sua direita satisfizerem o operador. Os operadores são:

  - Operador $=$ Igual
  - Operador $<$ Menor que
  - Operador $>$ Maior que
  - Operador $<>$ Diferente
  - Operador $<=$ Menor igual
  - Operador $>=$ Maior igual

- **Operadores Aritméticos**

  Usados para cálculos matemáticos.

  - Operador $+$ Adição

  - Operador $-$ Subtração
  - Operador $*$ Multiplicação
  - Operador / Divisão

- **Operadores Especiais**

  Operadores que representam funções diferenciadas.

  - Operador $:=$ Atribuição
  - Operador $/*$ Início de comentário
  - Operador $*/$ Final de comentário
  - Operador () Alterador de prioridade de execução

4.2.5 Comandos / Estruturas

A linguagem possui comandos de controle de fluxo, como o SE e ENQUANTO.
Estrutura Básica
Comando SE:

```
Se <Condição> então
   Comandos
Fim
```

Se a <Condição> for verdadeira, serão executados os comando contidos entre o SE e o FIM.

O comando SE possui algumas variantes na sua utilização, como demonstrado a baixo:

```
Se <condição> então
   Comandos1
Senão
   Comandos2
Fim
Se <expressão1> então
   Comandos1
Ou se <expressão2>
   Comandos2
Ou se <expressão3>
   Comandos3
Fim
```

A utilização do SENÃO faz com que os comandos que estão entre ele o Fim sejam executados, caso a <Condição> seja falsa.

Já o sub-comando OU SE fará um novo teste com outra condição, caso o questionamento anterior tenha sido falso.

Também temos a estrutura do “looping”, com o comando ENQUANTO. Vejamos como:

```
Enquanto <Condição>  Faça
   Comandos
Fim
```

Os comandos contidos entre o ENQUANTO e o FIM, serão executados enquanto a <Condição> for verdadeira. O comando SAIA força a saída do laço independente da resposta da <Condição>. Já o comando REAVALIE faz com que o controle de execução retorne ao ENQUANTO antes do comando FIM, reavaliando a condição.

Parte II

Desenvolvimento

5 PROJETO DE SOFTWARE

> O projeto de software é o processo pelo qual os requisitos são traduzidos numa representação do software. Inicialmente, a representação descreve uma visão holística do software. Subsequentes refinamentos levam a uma representação que está muito próxima ao código-fonte. (PRESSMAN, 1995).

Se na fase de definição foi determinado o que se deve fazer, na fase de desenvolvimento descreveremos como será feito e implementaremos a solução. O primeiro passo na fase de desenvolvimento será a elaboração de um projeto consistente com a descrição do problema elaborado na fase de análise. Abordaremos a fase de projeto com uma especificação sugerida pelo SWEBOK (IEEE, 2001) e a modelagem de sistemas usando UML, recomendada pelo processo unificado da Rational (KRUCHTEN, 2003).

5.1 Especificação do Projeto

5.1.1 Introdução

Na especificação do projeto, os requisitos do software darão origem a documentos que serão o esqueleto do sistema e os seguintes elementos deverão ser projetados:

- Estruturas de banco de dados.
- Interfaces de usuário.
- Elementos arquiteturais que compõem o software e suas interações.
- Especificações dos componentes até o nível de suas interfaces.

5.1.2 Objetivos

Esta etapa envolve o projeto das especificações dos requisitos (funcionais e não funcionais) e são produzidos documentos que guiarão o restante do processo de desenvolvimento, tais como:

- Diagramas de fluxos de dados ;
- Diagramas de entidades e relacionamentos ;
- Diagrama hierárquico funcional ;
- Dicionário de dados ;

- Especificações e lay out dos relatórios e telas ;

Nesta etapa o analista de sistemas elabora os Diagramas de Fluxo de Dados (DFD), representando os processos, depósitos de dados e agentes externos encontrados no levantamento do sistema. O Dicionário de Dados é refinado especificando a estrutura física dos atributos. Pode-se também alimentar o Dicionário de Dados nesta etapa. Para projetos que envolvem processamento batch, nesta etapa devem ser especificados e documentados os fluxos das rotinas batch. Nesta etapa o projetista de banco de dados especifica a forma de armazenamento das entidades persistentes, quando existirem, e define o comportamento que deve ser implementado no banco de dados, elaborando o Diagrama de Entidades e Relacionamentos (DER) que descreve detalhadamente toda a estrutura do banco de dados. Também é especificado qualquer comportamento definido na base de dados, tais como: stored procedures, triggers, constraints, etc. Para o projeto do banco de dados podem ser realizadas as seguintes tarefas:

- Definir as entidades persistentes;
- Definir as tabelas, views, triggers, stored procedures;
- Especificar as restrições de integridade (chaves-primárias, chaves-estrangeiras, relacionamentos);
- Especificar a cardinalidade e totalidade dos relacionamentos;
- Otimizar o modelo dos dados (definir índices);
- Especificar o acesso aos dados;
- Especificar as características de armazenamento;
- Especificar as regras de negócio sobre os dados.

Também são especificados os elementos arquiteturais que compõem o software e suas interações. A arquitetura do software apoia questões importantes do projeto, tais como: organização do software em componentes e módulos, estruturas de controle globais, protocolos de comunicação, composição dos elementos do projeto, e a designação da funcionalidade dos componentes do projeto. A arquitetura do software é definida de maneira a atender à especificação dos requisitos do software previamente estabelecidos. Nesta etapa os componentes individuais do software também são refinados até o nível de suas interfaces. Detalhes são adicionados em termos de estruturas de dados e algoritmos, de modo a permitir que os componentes individuais do software sejam futuramente implementados. Para tal, são tomadas decisões de projeto como definição das bases de dados, das linguagens de programação, dos padrões de integração e dos estilos de interfaces. Nesta etapa o projetista de interface de usuário especifica o desenho externo (gráfico e funcional) das interfaces e das classes correspondentes (classes de fronteira). Em geral o projeto das interfaces de usuário acontece de forma interativa, submetendo as soluções encontradas a vários ciclos de avaliação por parte dos usuários. Os requisitos de qualidade de produtos de software, tais como funcionalidade, confiabilidade, usabilidade e eficiência também são considerados nesta etapa.

5.2 Softwares Orientado a Objetos

5.2.1 Introdução

O conceito da orientação a objetos já vem sido discutido há muito tempo, desde o lançamento da 1ª linguagem orientada a objetos, a SIMULA. Vários "papas"da engenharia de software mundial como Peter Coad, Edward Yourdon e Roger Pressman (PRESSMAN, 1995) abordaram extensamente a análise e projeto orientado a objetos como realmente um grande avanço no desenvolvimento de sistemas. Mas, mesmo assim, eles citam que não existe (ou que não existia no momento de suas publicações) uma linguagem que possibilitasse o desenvolvimento de qualquer software utilizando a análise orientada a objetos.

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

- A orientação a objetos é uma tecnologia para a produção de modelos que especifiquem o domínio do problema de um sistema.
- Quando construídos corretamente, sistemas orientados a objetos são flexíveis a mudanças, possuem estruturas bem conhecidas e provêm a oportunidade de criar e implementar componentes totalmente reutilizáveis.
- Modelos orientados a objetos são implementados convenientemente utilizando uma linguagem de programação orientada a objetos. A engenharia de software orientada a objetos é muito mais que somente utilizar mecanismos de sua linguagem de programação, é saber utilizar da melhor forma possível todas as técnicas da modelagem orientada a objetos.
- A orientação a objetos não é só teoria, mas uma tecnologia de eficiência e qualidade comprovadas usada em inúmeros projetos e para construção de diferentes tipo de sistemas.

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

5.2.2 Orientação a objetos

O paradigma da orientação a objetos está fortemente ligado a forma atual de desenvolvimento de software. Esta seção descreve o que esse termo significa e justifica por que a orientação a objetos é importante para a modelagem de sistemas (BEZERRA, 1999).

Segundo o Aurélio[1], um paradigma pode ser entendido como: uma forma de abordar um problema. Pode-se concluir, então, que o termo "paradigma da orientação a objetos"é uma nova abordagem no desenvolvimento de sistema, comparado ao paradigma anterior, o estruturado.

No final dos anos 60, Alan Kay, um dos pais do paradigma da orientação a objetos, formulou a chamada "analogia biológica". Nessa analogia, ele imaginou como seria um sistema de software que funcionasse como um ser vivo. Nesse sistema, cada "célula"interagiria com outras células através do envio de mensagens para realizar um objetivo comum. Adicionalmente, cada célula se comportaria como uma unidade autônoma.

De uma forma mais geral, Kay pensou em como construir um sistema de software a partir de agentes autônomos que interagem entre si. Ele, então, estabeleceu os seguintes princípios da orientação a objetos:

[1]Dicionário Aurélio Século XXI

1. Qualquer coisa é um objeto.

2. Objetos realizam tarefas através da requisição de serviços a outros objetos.

3. Cada objeto pertence a uma determinada classe. Uma classe agrupa objetos similares.

4. A classe é um repositório para comportamento associado ao objeto.

5. Classes são organizadas em hierarquias.

Mas o que o paradigma da orientação a objetos tem a ver com a modelagem de sistemas? Antes da orientação a objetos, um outro paradigma era utilizado na modelagem de sistemas: o paradigma estruturado. Nesse paradigma, os elementos são dados e processos. Processos agem sobre dados para que um objetivo seja alcançado. Por outro lado, no paradigma da orientação a objetos, há um elemento, o objeto, uma unidade autônoma que contém seus próprios dados que são manipulados pelos processos definidos para o objeto e que interage com outros objetos para alcançar um objetivo. É o paradigma da orientação a objetos que os seres humanos utilizam no cotidiano para a resolução de problemas. Uma pessoa atende a mensagens (requisições) para realizar um serviço; essa mesma pessoa envia mensagens a outras para que estas realizem serviços. Por que não aplicar essa mesma forma de pensar a modelagem de sistemas?

O paradigma da orientação a objetos visualiza um sistema de software como uma coleção de agentes interconectados chamados objetos. Cada objeto é responsável por realizar tarefas específicas. É através da interação entre objetos que uma tarefa computacional é realizada.

Pode-se concluir que a orientação a objetos, como técnica para modelagem de sistemas, diminui a diferença semântica entre a realidade sendo modelada e os modelos construídos. Aqui descrevemos o importante papel da orientação a objetos na modelagem de sistemas de software atualmente. Explícita ou implicitamente, as técnicas de modelagem de sistemas aqui descritas utilizam os princípios que Alan Kay estabeleceu há mais de 30 anos.

Um sistema de software orientado a objetos consiste de objetos em colaboração com o objetivo de realizar as funcionalidades desse sistema. Cada objeto é responsável por tarefas específicas. E através da cooperação entre objetos que a computação do sistema se desenvolve.

O construção de um sistema orientado a objetos envolve varios conceitos (MEYER, 1997), veremos detalhamente alguns deles.

5.2.3 Classes e Objetos

O mundo real é formado de coisas. Como exemplos de coisas pode-se citar um cliente, uma loja, uma venda, um pedido de compra, um fornecedor, este documento etc. Na terminologia de orientação a objetos, essas coisas do mundo real são denominadas objetos.

Seres humanos costumam agrupar os objetos. Provavelmente, os seres humanos realizam esse processo mental de agrupamento para tentar gerenciar a complexidade de entender as coisas do mundo real. Realmente, é bem mais fácil entender a idéia cavalo do que entender todos os cavalos que existem. Na terminologia da orientação a objetos, cada idéia é denominada classe de objetos, ou simplesmente classe. Uma classe é uma

descrição dos atributos e serviços comuns a um grupo de objetos. Sendo assim, pode-se entender uma classe como sendo um molde a partir do qual objetos são construídos. Ainda sobre terminologia, diz-se que um objeto é uma instância de uma classe.

Por exemplo, quando se pensa em um cavalo, logo vem a mente um animal de quatro patas, cauda, crina etc. Pode ser que algum dia você veja dois cavalos, um mais baixo que o outro, um com cauda maior que o outro, ou mesmo, por um infeliz acaso, um cavalo com menos patas que o outro. No entanto, você ainda terá certeza de estar diante de dois cavalos. Isso porque a idéia (classe) cavalo está formada na mente dos seres humanos, independentemente das pequenas diferenças que possa haver entre os exemplares (objetos) da idéia cavalo.

É importante notar que uma classe é uma abstração das características de um grupo de coisas do mundo real. Na maioria das vezes, as coisas do mundo real são muito complexas para que todas as suas características sejam representadas em uma classe. Além disso, para fins de modelagem de um sistema, somente um subconjunto de características pode ser relevante. Portanto, uma classe representa uma abstração das características relevantes do mundo real.

5.2.4 Mensagens

Objetos não executam suas operações aleatoriamente. Para que uma operação em um objeto seja executada, deve haver um estímulo enviado a esse objeto. Se um objeto for visto como uma entidade ativa que representa uma abstração de algo do mundo real, então faz sentido dizer que tal objeto pode responder a estímulos a ele enviados (assim como faz sentido dizer que seres vivos reagem a estímulos que eles recebem). Independentemente da origem do estímulo, quando ele ocorre, diz-se que o objeto em questão está recebendo uma mensagem requisitando que ele realize alguma operação.

Quando se diz na terminologia de orientação a objetos que objetos de um sistema estão trocando mensagens significa que esses objetos estão enviando mensagens uns aos outros com o objetivo de realizar alguma tarefa dentro do sistema no qual eles estão inseridos.

5.2.5 Abstração

Uma abstração é qualquer modelo que inclui os aspectos mais importantes, essenciais de alguma coisa, ao mesmo tempo em que ignora os detalhes menos importantes. Abstrações permitem gerenciar a complexidade e concentrar a atenção nas características essenciais de um objeto. Note que uma abstração é dependente da perspectiva: o que é importante em um contexto pode não ser importante em outro.

5.2.6 Encapsulamento

Objetos possuem comportamento. O termo comportamento diz respeito a operações realizadas por um objeto e também ao modo pelo qual essas operações são executadas. O mecanismo de encapsulamento é uma forma de restringir o acesso ao comportamento interno de um objeto. Um objeto que precise da colaboração de outro objeto para realizar alguma tarefa simplesmente envia uma mensagem a este último. O método que o objeto requisitado usa para realizar a tarefa não é conhecido dos objetos requisitantes.

Certamente, o objeto requisitante precisa conhecer quais as tarefas que um outro objeto sabe fazer ou que informação ele conhece. Para tanto, a classe de um objeto descreve o seu comportamento. Na terminologia da orientação a objetos, diz-se que um objeto possui uma interface. Em termos bastante simples, a interface de um objeto é o que ele

conhece e o que ele sabe fazer, sem descrever como o objeto conhece o faz. A interface de um objeto define os serviços que ele pode realizar e conseqüentemente as mensagens que ele recebe. Uma interface pode ter várias formas de implementação. Mas, pelo Princípio do Encapsulamento, a implementação de um objeto requisitado não importa para um objeto requisitante.

Através do encapsulamento, a única coisa que um objeto precisa saber para pedir a colaboração de outro objeto é conhecer a sua interface. Nada mais. Isso contribui para a autonomia dos objetos. Cada objeto envia mensagens a outros objetos para realizar certas tarefas, sem se preocupar em como as tarefas são realizadas.A aplicação da abstração, neste caso, está em esconder os detalhes de funcionamento interno de um objeto.

5.2.7 Polimorfismo

O polimorfismo indica a capacidade de abstrair várias implementações diferentes em uma única interface. No polimorfismo um objeto pode enviar a mesma mensagem para objetos semelhantes, mas que implementam a sua interface de formas diferentes. O que importa é a interface do objeto e não sua implementação.

5.2.8 Herança

A herança é outra forma de abstração utilizada na orientação a objetos. Já vimos que as características e o comportamento comuns a um conjunto de objetos podem ser abstraídos em uma classe. A herança pode ser vista como um nível de abstração acima da encontrada entre classes e objetos.

Na herança, classes semelhantes são agrupadas em hierarquias. Cada nível de uma hierarquia pode ser visto como um nível de abstração. Cada classe em um nível da hierarquia herda as características das classes nos níveis acima. Esse mecanismo facilita o compartilhamento de comportamento comum entre um conjunto de classes semelhantes. Além disso, as diferenças ou variações de uma classe em particular podem ser organizadas de forma mais clara.

5.2.9 Vantagens da orientação a objetos

Ao escolher desenvolver um software pelo paradigma da orientação a objetos, o desenvolvedor procura obter uma série de vantagens decorrentes das características dessa abordagem:

- O uso de objetos na modelagem torna mais fácil descrever as estruturas e o comportamento existente no mundo real. Essa proximidade faz com que os clientes possam ser identificados mais diretamente com os problemas nos modelos.

- O encapsulamento do conhecimento em componentes isola o comportamento, o que permite que as mudanças nos requisitos possam também ser isoladas em cada componente, sem afetar o sistema como um todo.

- O uso de classes e objetos facilita a integração das fases do processo de desenvolvimento, porque ao contrário de outros modelos, onde cada fase possui técnicas e paradigmas próprios, na orientação a objetos o mesmo paradigma é conduzido da análise à construção.

- O encapsulamento favorece o teste dos sistemas de software, que pode ser isolado para cada componente. O resultado é um aumento na qualidade do sistema.

- O encapsulamento permite ainda que os componentes possam ser desenvolvidos por fornecedores diferentes.
- A reutilização, decorrente do encapsulamento, reduz custos e prazos no desenvolvimento de software, porque possibilita que o mesmo comportamento seja usado em vários projetos.
- A herança associada ao encapsulamento permite abordar problemas mais complexos do que com outras abordagens de desenvolvimento. A herança cria uma família de objetos com uma complexidade crescente, os quais podem ser aplicados em vários problemas diferentes.

5.3 Linguagem de Modelagem Unificada (UML)

A Unified Modeling Language (UML) é uma linguagem gráfica para visualizar, especificar, construir e documentar os artefatos de um sistema. A UML lhe dá meios de descrever os planos do sistema, cobrindo itens conceituais como processos de negócios e funcionalidades do sistema, bem como itens concretos como classes e componentes.

5.3.1 Introdução

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

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

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

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

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

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

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

Diante desta diversidade de conceitos, "os três amigos"(BOOCH; RUMBAUGH; JACOBSON, 1999) decidiram criar uma Linguagem de Modelagem Unificada. Eles disponibilizaram inúmeras versões preliminares da UML para a comunidade de desenvolvedores e a resposta incrementou muitas novas idéias que melhoraram ainda mais a linguagem. Os objetivos da UML são:

- A modelagem de sistemas (não apenas de software) usando os conceitos da orientação a objetos;
- Estabelecer uma união fazendo com que métodos conceituais sejam também executáveis;
- Criar uma linguagem de modelagem usável tanto pelo homem quanto pela máquina.

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

5.3.2 Uso da UML

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

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

- Sistemas de Informação: Armazenar, pesquisar, editar e mostrar informações para os usuários. Manter grandes quantidades de dados com relacionamentos complexos, que são guardados em bancos de dados relacionais ou orientados a objetos.
- Sistemas Técnicos: Manter e controlar equipamentos técnicos como de telecomunicações, equipamentos militares ou processos industriais. Eles devem possuir interfaces especiais do equipamento e menos programação de software de que os sistemas de informação. Sistemas Técnicos são geralmente sistemas real-time.
- Sistemas Real-time Integrados: Executados em simples peças de hardware integrados a telefones celulares, carros, alarmes etc. Estes sistemas implementam programação de baixo nível e requerem suporte real-time.
- Sistemas Distribuídos: Distribuídos em máquinas onde os dados são transferidos facilmente de uma máquina para outra. Eles requerem mecanismos de comunicação sincronizados para garantir a integridade dos dados e geralmente são construídos em mecanismos de objetos como CORBA, COM/DCOM ou Java Beans/RMI.
- Sistemas de Software: Definem uma infra-estrutura técnica que outros softwares utilizam. Sistemas Operacionais, bancos de dados, e ações de usuários que executam ações de baixo nível no hardware, ao mesmo tempo que disponibilizam interfaces genéricas de uso de outros softwares.
- Sistemas de Negócios: descreve os objetivos, especificações (pessoas, computadores etc.), as regras (leis, estratégias de negócios etc.), e o atual trabalho desempenhado nos processos do negócio.

É importante perceber que a maioria dos sistemas não possui apenas uma destas características acima relacionadas, mas várias delas ao mesmo tempo. O Fatorlex possui características de sistemas de software com regras de negócio de recursos humanos.

5.3.3 A Notação UML

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

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

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

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

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

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

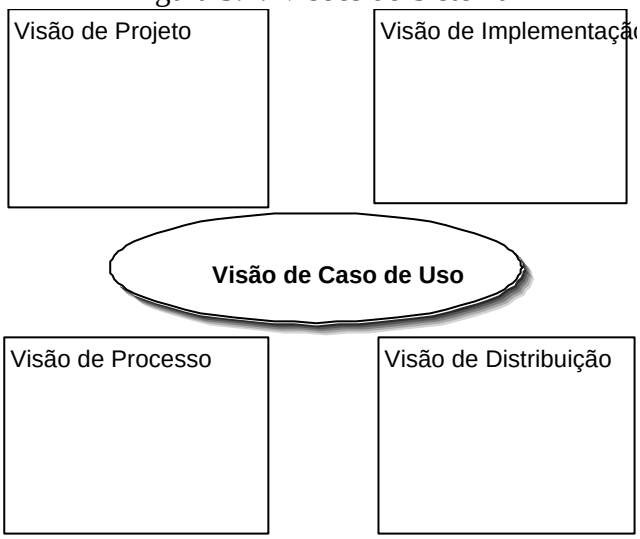
5.3.4 Visões do Sistema

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

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

!htp]

Figura 5.1: Visões do Sistema



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

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

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

- Visão de Implementação: É uma descrição da implementação dos módulos e suas dependências. É executada principalmente por desenvolvedores e consiste nos diagramas de pacotes e componentes.

- Visão de Processos: Trata a divisão do sistema em processos e processadores. Este aspecto, que é uma propriedade não funcional do sistema, permite uma melhor utilização do ambiente onde o sistema se encontrará, se o mesmo possui execuções paralelas, e se existe dentro do sistema um gerenciamento de eventos assíncronos.

Uma vez dividido o sistema em linhas de execução de processos concorrentes (threads), esta visão de concorrência deverá mostrar como se dá a comunicação e a concorrência destas threads. A visão de concorrência é suportada pelos diagramas dinâmicos, que são os diagramas de estado, seqüência, colaboração e atividade, e pelos diagramas de implementação, que são os diagramas de componente e distribuição.

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

5.3.5 Modelos de Elementos

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

Apesar de alguns conceitos já terem sido definidos quando falamos de orientação a objetos, iremos agora exemplificar sua representação.

5.3.5.1 Classes

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

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

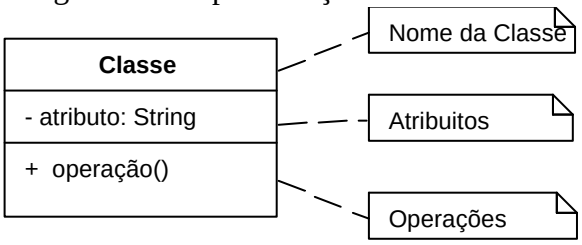
Identificar as classes de um sistema pode ser complicado e deve ser feito por experts no domínio do problema a que o software modelado se baseia. As classes devem ser retiradas do domínio do problema e serem nomeadas pelo que elas representam no sistema. Quando procuramos definir as classes de um sistema,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

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

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

- Existem classes de bibliotecas, componentes ou modelos externos a serem utilizados pelo sistema modelado? Se sim, normalmente essas classes, componentes e modelos conterão classes candidatas ao nosso sistema.
- Qual o papel dos atores dentro do sistema? Talvez o papel deles possa ser visto como classes, por exemplo, usuário, operador, cliente e daí por diante.

Figura 5.2: Representação de uma Classe

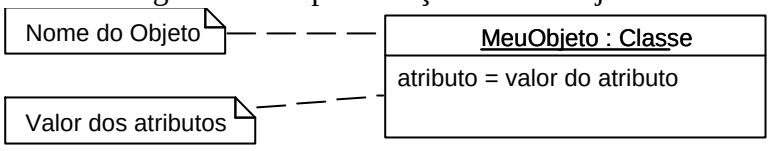


Em UML as classes são representadas por um retângulo dividido em três compartimentos: o compartimento de nome, que conterá apenas o nome da classe modelada, o de atributos, que possuirá a relação de atributos que a classe possui em sua estrutura interna, e o compartimento de operações, que serão os métodos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

5.3.5.2 Objetos

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

Figura 5.3: Representação de um Objeto

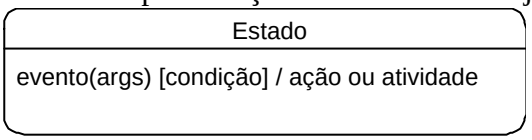


Em UML um objeto é mostrado como uma classe só que seu nome (do objeto) é sublinhado e o nome do objeto pode ser mostrado opcionalmente precedido do nome da classe.

5.3.5.3 Estados

Todos os objetos possuem um estado que significa o resultado de atividades executadas pelo objeto e é normalmente determinada pelos valores de seus atributos e ligações com outros objetos. Um objeto muda de estado quando acontece algo. O fato de acontecer alguma coisa com o objeto é chamado de evento. Através da análise da mudança de estados dos tipos de objetos de um sistema, podemos prever todos os possíveis comportamentos de um objeto de acordo com os eventos que o mesmo possa sofrer.

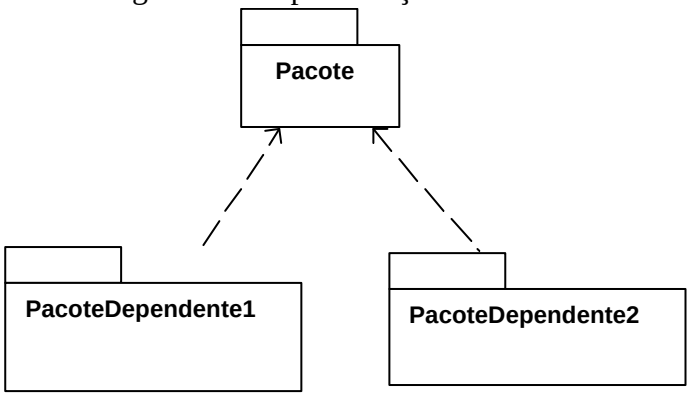
Figura 5.4: Representação de Estados do Objeto



Um estado, em sua notação, pode conter dois compartimentos. O primeiro mostra o nome do estado. O segundo compartimento é opcional e é chamado de compartimento de atividade, onde eventos e ações podem ser listados. Três eventos padrões podem ser mostrados no compartimento de atividades de um estado: entrar, sair e fazer. O evento entrar pode ser usado para definir atividades no momento em que o objeto entra naquele estado. O evento sair, define atividades que o objeto executa antes de passar para o próximo estado e o evento fazer, define as atividades do objeto enquanto se encontra naquele estado.

5.3.5.4 Pacotes

Figura 5.5: Representação de Pacotes



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

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

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

5.3.5.5 Componentes

Um componente pode ser tanto um código em linguagem de programação como um código executável já compilado. Por exemplo, em um sistema desenvolvido em Java,

Figura 5.6: Representação de Componentes

cada arquivo .java ou .class é um componente do sistema e será mostrado no diagrama de componentes que os utiliza.

5.3.5.6 Relacionamentos

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

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

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

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

5.3.6 Diagramas

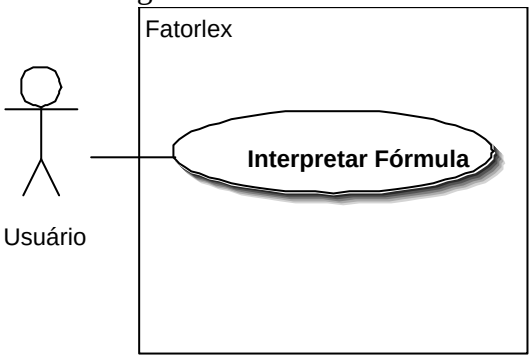
Os diagramas utilizados pela UML são compostos de nove tipos: diagrama de casos de uso, de classes, de objeto, de estado, de seqüência, de colaboração, de atividade, de componente e o de distribuição.

Todos os sistemas possuem uma estrutura estática e um comportamento dinâmico. A UML suporta modelos estáticos (estrutura estática), dinâmicos (comportamento dinâmico) e funcionais. A Modelagem estática é suportada pelo diagrama de classes e de objetos, que consiste nas classes e seus relacionamentos. Os relacionamentos podem ser de associações, herança (generalização), dependência ou refinamentos. A modelagem dinâmica é suportada pelos diagramas de estado, seqüência, colaboração e atividade. E a modelagem funcional é suportada pelos diagramas de componente e distribuição. Abordaremos agora cada um destes tipos de diagrama.

5.3.6.1 Diagrama de Casos de Uso

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

Figura 5.7: Diagrama de Caso de Uso do Fatorlex



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

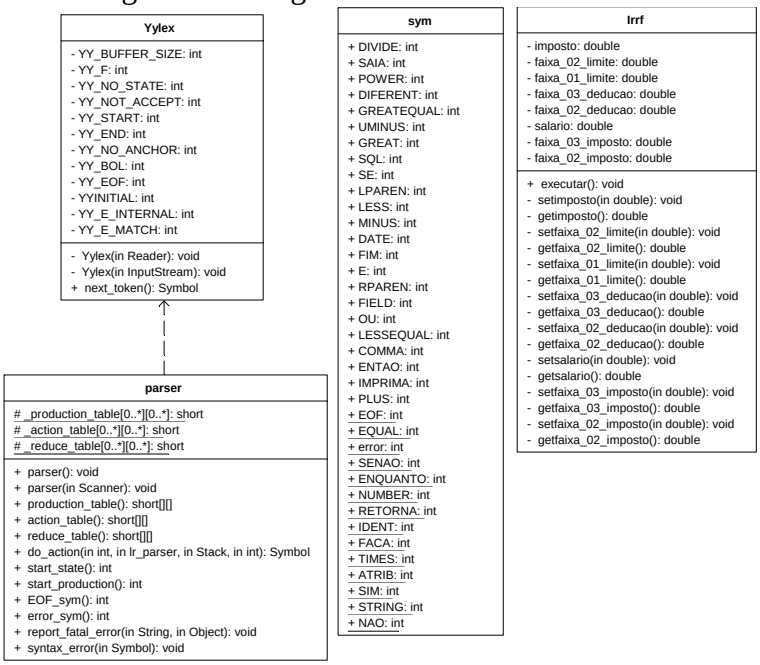
O diagrama de Casos de Uso acima demonstra as funções de um ator externo do Fatorlex que foi modelado na fase de análise deste trabalho. O diagrama especifica que funções

o usuário poderá desempenhar. Pode-se perceber que não existe nenhuma preocupação com a implementação de cada uma destas funções, já que este diagrama apenas se resume a determinar que funções deverão ser suportadas pelo sistema modelado.

5.3.6.2 Diagrama de Classes

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

Figura 5.8: Diagrama de Classes do Fatorlex



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

O diagrama de classes exibido no exemplo é o resultado produzido pelos analisadores léxicos e sintáticos [2] usados no Fatorlex. Também foi incluida uma classe gerada diretamente pelo Fatorlex. A classe Irrf é o resultado do programa escrito na linguagem Fatorhw que será comentado mais adiante.

[2] JLex e Java CUP

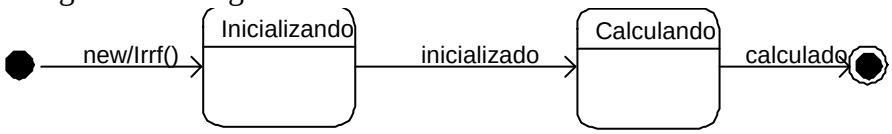
5.3.6.3 Diagrama de Objetos

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

5.3.6.4 Diagrama de Estado

O diagrama de estado é tipicamente um complemento para a descrição das classes. Este diagrama mostra todos os estados possíveis em que objetos de uma certa classe podem se encontrar e mostra também quais são os eventos do sistema que provocam tais mudanças. Os diagramas de estado não são escritos para todas as classes de um sistema, mas apenas para aquelas que possuem um número definido de estados conhecidos e onde o comportamento das classes é afetado e modificado pelos diferentes estados.

Figura 5.9: Diagrama de Estado do Cálculo de IRRF do Fatorlex



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

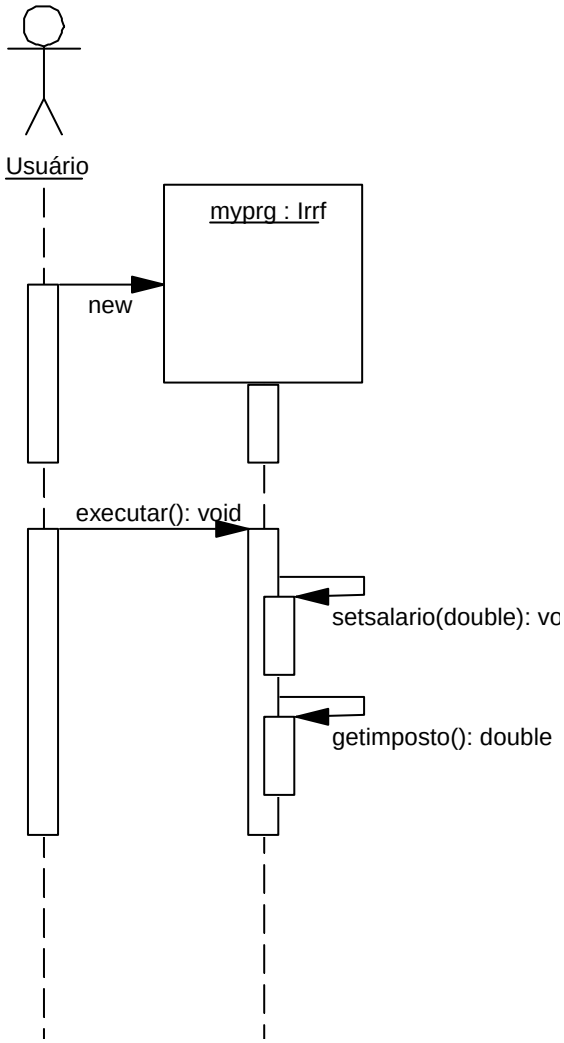
Diagramas de estado possuem um ponto de início e vários pontos de finalização. Um ponto de início (estado inicial) é mostrado como um círculo todo preenchido e um ponto de finalização (estado final) é mostrado como um círculo em volta de um outro círculo menor preenchido. Um estado é mostrado como um retângulo com cantos arredondados. Entre os estados estão as transições, mostrados como uma linha com uma seta no final de um dos estados. A transição pode ser nomeada com o seu evento causador. Quando o evento acontece, a transição de um estado para outro é executada ou disparada. Uma transição de estado normalmente possui um evento ligado a ela. Se um evento é anexado a uma transição, esta será executada quando o evento ocorrer. Se uma transição não possuir um evento ligado a ela, a mesma ocorrerá quando a ação interna do código do estado for executada (se existir ações internas como entrar, sair, fazer ou outras ações definidas pelo desenvolvedor). Então, quando todas as ações forem executadas pelo estado, a transição será disparada e serão iniciadas as atividades do próximo estado no diagrama de estados.

5.3.6.5 Diagrama de Seqüência

Um diagrama de seqüência mostra a colaboração dinâmica entre os vários objetos de um sistema. O mais importante aspecto deste diagrama é que a partir dele percebe-se a seqüência de mensagens enviadas entre os objetos. Ele mostra a interação entre os

objetos, alguma coisa que acontecerá em um ponto específico da execução do sistema. O diagrama de sequência consiste em um número de objetos mostrado em linhas verticais. O decorrer do tempo é visualizado observando-se o diagrama no sentido vertical de cima para baixo. As mensagens enviadas por cada objeto são simbolizadas por setas entre os objetos que se relacionam.

Figura 5.10: Diagrama de Seqüência do Cálculo de IRRF no Fatorlex



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

No eixo horizontal estão os objetos envolvidos na seqüência. Cada um é representado por um retângulo de objeto (similar ao diagrama de objetos) e uma linha vertical pontilhada chamada de linha de vida do objeto, indicando a execução do objeto durante a seqüência. Ccomo exemplo citamos: mensagens recebidas ou enviadas e ativação de objetos. A comunicação entre os objetos é representada como linha com setas horizontais simbolizando as mensagens entre as linhas de vida dos objetos. A seta especifica se a mensagem é síncrona, assíncrona ou simples. As mensagens podem possuir também números seqüenciais; eles são utilizados para tornar mais explícitas as seqüências no

diagrama.

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

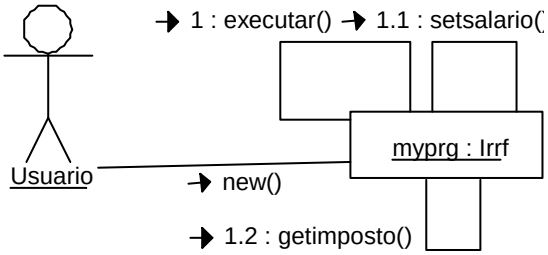
Os diagramas de seqüência podem mostrar objetos que são criados ou destruídos como parte do cenário documentado pelo diagrama. Um objeto pode criar outros objetos através de mensagens. A mensagem que cria ou destrói um objeto é geralmente síncrona, representada por uma seta sólida.

5.3.6.6 Diagrama de Colaboração

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

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

Figura 5.11: Diagrama de Colaboração do Cálculo de IRRF no Fatorlex



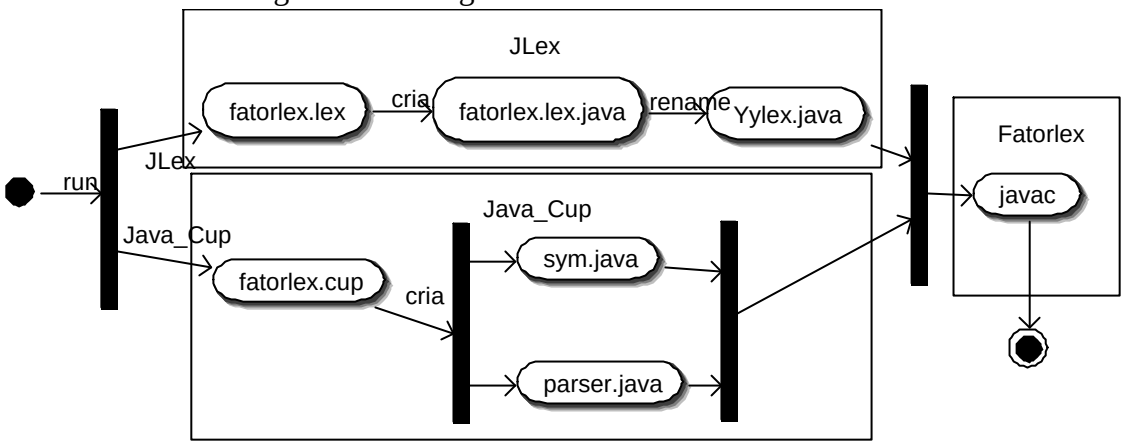
O diagrama de colaboração é desenhado como um diagrama de objeto, onde os diversos objetos são mostrados juntamente com seus relacionamentos. As setas de mensagens são desenhadas entre os objetos para mostrar o fluxo de mensagens entre eles. As mensagens são nomeadas e numeradas para que entre outras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

5.3.6.7 Diagrama de Atividade

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

Os estados no diagrama de atividade mudam para um próximo estágio quando uma ação é executada (sem ser necessário especificar nenhum evento como no diagrama de estado). Outra diferença entre o diagrama de atividade e o de estado é que podem ser colocadas como raias de natação (swim lanes). Uma raia de natação agrupa atividades,

Figura 5.12: Diagrama de Atividades do Fatorlex



com respeito a quem é responsável e onde estas atividades residem na organização e é representada por retângulos que englobam todos os objetos que estão ligados a ela (raia).

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

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

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

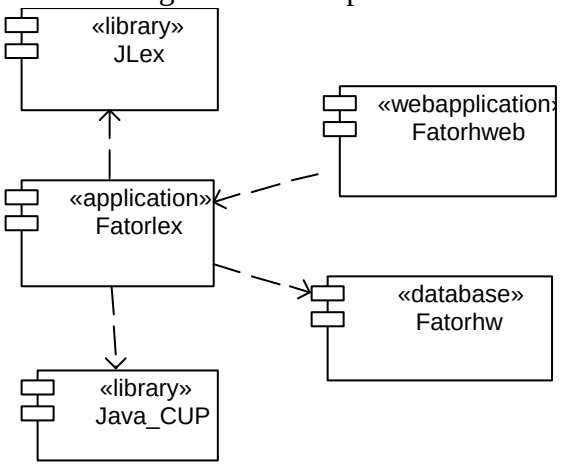
O diagrama de atividade mostra o fluxo seqüencial das atividades. É normalmente utilizado para demonstrar as atividades executadas por uma operação específica do sistema. Consistem em estados de ação, que contém a especificação de uma atividade a ser desempenhada por uma operação do sistema. Decisões e condições, como execução paralela, também podem ser mostradas no diagrama de atividade. O diagrama também pode conter especificações de mensagens enviadas e recebidas como partes de ações executadas.

5.3.6.8 Diagrama de Componente

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

O diagrama de componente descreve os componentes de software e suas dependências entre si, representando a estrutura do código gerado. Os componentes são a implementação na arquitetura física dos conceitos e da funcionalidade definidos na arquitetura lógica

Figura 5.13: Diagrama de Componentes do Fatorlex



(classes, objetos e seus relacionamentos). Eles são tipicamente os arquivos implementados no ambiente de desenvolvimento.

Um componente é mostrado em UML como um retângulo com uma elipse e dois retângulos menores do seu lado esquerdo. O nome do componente é escrito abaixo ou dentro de seu símbolo. Componentes são tipos, mas apenas componentes executáveis podem ter instâncias. Um diagrama de componente mostra apenas componentes como tipos. Para mostrar instâncias de componentes, deve ser usado um diagrama de distribuição, onde as instâncias executáveis são alocadas em nodes.

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

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

5.3.6.9 Diagrama de Distribuição

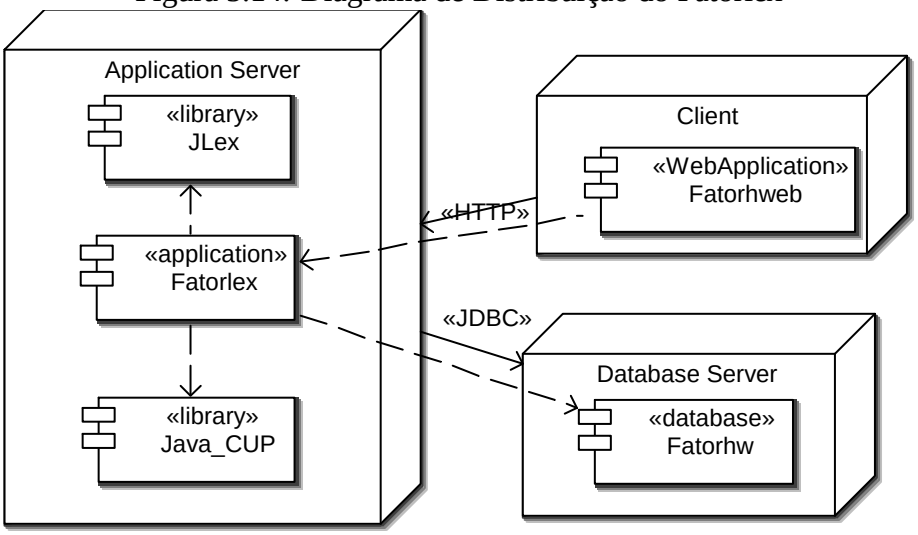
O diagrama de distribuição mostra a arquitetura física do hardware e do software no sistema. Pode mostrar os atuais computadores e periféricos, juntamente com as conexões que eles estabelecem entre si e pode mostrar também os tipos de conexões entre esses computadores e periféricos. Especificam-se também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

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

O diagrama de distribuição é composto por componentes, que possuem a mesma simbologia dos componentes do diagrama de componentes, nodes, que significam objetos físicos que fazem parte do sistema, podendo ser uma máquina cliente numa LAN, uma

máquina servidora, uma impressora, um roteador, etc., e conexões entre estes nodes e componentes que juntos compõem toda a arquitetura física do sistema.

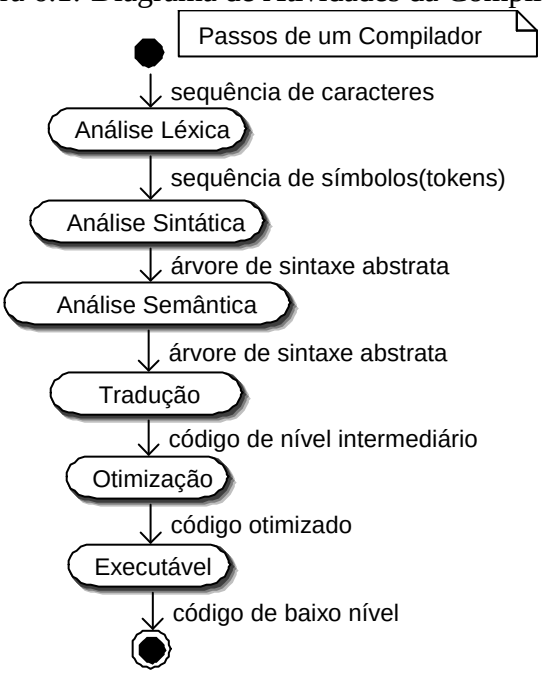
Figura 5.14: Diagrama de Distribuição do Fatorlex



6 COMPILADOR FATORLEX

6.1 Compiladores

Figura 6.1: Diagrama de Atividades da Compilação



O trabalho de um compilador (VERMEIR, 2001) é traduzir uma linguagem de alto nível em código de baixo nível e esta tradução é organizada em várias fases ou passagens. Cada uma das fases será apresentada a seguir.

6.1.1 Análise Léxica

Um analisador léxico (scanner ou tokenizer) converte um fluxo de caracteres em um fluxo de símbolos.

Símbolos são elementos como:

- Palavras chaves tais como: SE, ENTÃO, ENQUANTO, FAÇA.
- Caracteres especiais tais como: +, -, (, e <.
- Nome de variáveis.
- Constantes como FALSO, 10, SIM.

O analisador léxico normalmente é uma sub-rotina do analisador sintático. Cada símbolo é uma única entidade. Um código numérico normalmente é designado a cada tipo de símbolo.

Analisadores léxicos executam:

- Reconstrução de linha
- Exclusão de comentários
- Exclusão de brancos em excesso
- Anulam retrocesso de caráter
- Executam substituição de texto

Tradução Léxica / identificação de símbolos:

- Símbolo + -> SOMA
- Símbolo ; -> EOF
- Símbolo MinhaVar -> IDENTIFICADOR
- Símbolo "Texto-> STRING

Palavras chaves assim como operadores são fáceis de traduzir, pois envolvem um conjunto finito de elementos. Já os identificadores, números e seqüências de caracteres são bem mais difíceis de identificar, por haver uma quantidade infinita deles, porém os mesmos podem ser expressos através do uso de expressões regulares.

6.1.2 Análise Sintática

O analisador sintático, também conhecido como Parser é responsável pela validação da estrutura sintática das sentenças que compõem o programa. Árvores de sintaxe são usadas para expor a estrutura do programa. O parser recebe uma seqüência de símbolos e constrói a arvore sintática abstrata representando o programa. O conjunto de todos os programas de uma dada linguagem é normalmente especificado usando uma lista de regras conhecida como gramática livre de contexto.

Uma gramática livre de contexto como a do Fatorlex possui quatro componentes:

- Um conjunto de símbolos conhecido como símbolos terminais
- Um conjunto de símbolos não terminais
- Um conjunto de expressões, separadas em dois lados, onde o lado esquerdo da expressão consiste num símbolo não terminal e o lado direito consiste de símbolos terminais e/ou não terminais.
- A designação de um símbolo não terminal como símbolo inicial da estrutura da linguagem.

6.1.3 Tratamento de erros

Os erros podem ocorrer em qualquer fase da compilação, por exemplo: caracteres de entrada de inválidos, erros de sintaxe, erros semânticos, etc. Bons compiladores tentarão identificar erros e passar informações apropriadas relativas aos erros.

6.1.4 Tabela de Símbolos

A tabela de símbolo é uma estrutura de dados usada em todas as fases do compilador para acompanhar os símbolos definidos pelo usuário (e às vezes as palavras chaves também). Durante as fases prévias (análise léxica e sintática) os símbolos são identificados e incluídos na tabela de símbolo. Nas fases seguintes os símbolos são checados para validar o seu uso.

Entre as atividades típicas da tabela de símbolos temos:

- Adicionar um novo símbolo a tabela
- Adicionar informação para um símbolo
- Informar acessos a um símbolo
- Determinar se um símbolo esta presente na tabela
- Retirar um símbolo
- Definir escopo do símbolo

Possíveis implementações de uma tabela de símbolos:

- Lista linear
- Lista classificada
- Hashtable
- Estrutura de árvore

Informações tipicamente armazenadas em uma tabela de símbolos:

- Valor armazenado
- Tipo
- Nível do escopo
- Valor inicial

6.1.5 Análise Semântica

O analisador semântico completa a tabela de símbolos com informação nas características de cada símbolo. A tabela de símbolos normalmente é iniciada durante a analise sintática. Uma entrada é criada para cada identificador e constante. O escopo do símbolo é levado em conta. Duas variáveis diferentes com o mesmo nome terão entradas diferentes na tabela de símbolos.

Cada nó da árvore sintática que contém um identificador ou uma constante deve possuir uma indicação na tabela de símbolos. O analisador semântico completa a tabela usando informações das declarações e o contexto onde os identificadores ocorreram. O analisador semântico faz a checagem de tipos, de fluxo de controle e a checagem de ambigüidades na declaração de identificadores.

O objetivo é identificar erros semânticos estaticamente. Por exemplo:

- Identificadores não declarados
- Identificadores fora do escopo
- Código nunca alcançado
- Métodos chamados com o número errado de parâmetros ou com parâmetros do tipo errado.

6.1.6 Geração e otimização de código

Antes de gerar o código para uma arquitetura específica, a maioria dos compiladores gera o código em uma linguagem intermediária. Este código normalmente pode ser manipulado e otimizado.

A otimização pode ser simples:

- Converter a:=a+1 para a++
- Converter 3*5 para 15

Ou mais complexa:

- Reorganizar dados e acessos a dados para eficiência de cache.

A otimização pode melhorar o desempenho do programa e freqüentemente também diminui o tamanho do mesmo.

6.2 Java

Analisando a evolução das linguagens de programação, pode-se estabelecer a seguinte seqüência: de B surgiu a linguagem C, de C desenvolveu-se o C++ e esta preparou a caminho para a linguagem Java.

Entender Java é entender as razões que guiaram a sua criação, as forças que a formaram e o legado por ela herdado. Assim como suas antecessoras, Java é uma mistura dos melhores elementos de sua rica herança, combinado com conceitos inovadores aplicados em um ambiente único.

Java está relacionado a C++, que é descendente direto da linguagem C. Muito do caráter de Java é herdado dessas duas linguagens. Da linguagem C, Java deriva sua sintaxe. Já boa parte das características de orientação a objetos em Java foi influência de C++. Na verdade a linguagem Java é o resultado de três décadas de evolução da orientação a objetos bem como do amadurecimento das linguagens de programação.

A linguagem Java foi concebida por James Gosling, Patrick Naughton, Chris Warth, Ed Frank e Mike Sheridan nos laboratórios da Sun Microsystems Inc em 1991. Foram 18 meses de desenvolvimento até o lançamento da sua primeira versão. A linguagem foi inicialmente batizada de Oak, mas posteriormente renomeada para Java em 1995.

Entre a implementação inicial de Oak em meados de 1992 e o anúncio público de Java no início de 1995, muitos outros desenvolvedores também contribuíram para o projeto e evolução da linguagem. O advento da internet colaborou para a divulgação e o uso do Java como linguagem de programação para Web, porém uma observação importante é o fato de Java não ter sido criado inicialmente para ser uma linguagem voltada para internet. Sua principal motivação foi a criação de uma linguagem independente da plataforma, para ser usada em equipamentos eletrônicos, como microondas e controle remoto, equipamentos esses que normalmente usam diversos tipos de CPUs, o que inviabiliza o uso de linguagem C ou C++, pois estas são projetadas para plataformas especificas.

A solução para os problemas de portabilidade inicialmente voltada para dispositivos eletrônicos ganhou escala quando foi voltada como solução para internet e a semelhança com as linguagens C/C++ facilitou a adoção pela comunidade de desenvolvedores.

6.3 JLex

Um gerador de analisador léxico para Java (BERK, 1997)

6.3.1 Introdução

O trabalho de um analisador léxico é separar um fluxo de entrada de caracteres em símbolos (tokens). Escrever um analisador léxico do zero é um trabalho complexo e dispendioso. O JLex assume a tarefa da produção de um analisador léxico em Java a partir de um dado arquivo de especificação.

6.3.2 Especificação do JLex

Um arquivo de especificação JLex é organizado em três seções, separados por %%. Uma especificação adequada de JLex tem o seguinte formato:

```
< Código Java >
%%
< Diretrizes JLex >
%%
< Expressão Regular>
```

A diretiva %% distingüe seções do arquivo de entrada e deve ser colocada no começo de sua linha. A continuação da linha contendo o %% pode ser descartada e não deve ser usada para abrigar declarações adicionais nem código.

A seção de código Java — a primeira seção do arquivo de especificação — é copiada diretamente no arquivo resultante de produção. Esta área da especificação fornece espaço para a implementação de classes uteis ou tipos de retorno.

As diretrizes JLex compõem a segunda parte do arquivo de entrada. Aqui, definições de macros são dadas e os estados são declarados.

A terceira seção contém as regras de análise léxica, cada uma consiste em três partes: uma lista opcional de estado, uma expressão regular e uma ação.

6.3.2.1 Código Java

O código Java precede o primeiro %%. Este código é copiado literalmente no início do arquivo fonte do analisador léxico produzido pelo JLex. Portanto, se o analisador léxico necessitar de algum pacote ou da importação de uma classe externa, a seção de código Java deve começar com a declaração correspondente. Esta declaração então será copiada no início do arquivo fonte gerado.

6.3.2.2 Diretrizes JLex

A seção diretrizes JLex começa depois do primeiro %% e continua até o segundo %%. Cada diretriz JLex deve estar contida em uma única linha e deve iniciar a mesma.

- **Código Interno**

  A diretriz `%{...%}` permite escrever código de Java que será copiado diretamente na classe do analisador léxico. Esta diretriz é usada como segue:

  ```
  %{
  < Código Interno >
  %}
  ```

  Para ser reconhecido, o `%{` e `%}` devem estar situados no começo de cada linha. O código Java especificado em `< Código Interno >` será então copiado na classe do analisador léxico criado pelo JLex.

  ```
  class Yylex {
  ... < Código Interno > ...
  }
  ```

  Isto permite a declaração de variáveis e funções internas para a classe gerada do analisador léxico. Variáveis iniciadas com yy devem ser evitadas, pois elas são reservadas para uso do analisador léxica gerado.

- **Código para Inicialização**

  A diretriz `%init{... %init}` permite escrever código Java para o construtor da classe do analisador léxico.

```
%init{
< Código >
%init}
```

  As diretrizes `%init{` e `%init}` devem estar no começo de cada linha. O código especificado de Java em `< Código >` será então copiado no construtor da classe do analisador léxico.

```
class Yylex {
  Yylex() {
  ... < Código > ...
  }
}
```

  Esta diretriz permite inicializar os membros da classe do analisador léxico de dentro de seu construtor. Variáveis iniciadas com yy devem ser evitadas, pois são reservadas para o analisador léxico.

  O código definido na diretriz `%init{... %init}` potencialmente pode gerar uma exceção. Para declarar esta exceção, use a diretriz `%initthrow{...%initthrow}`.

```
%initthrow{
< exceção [1]>[, < exceção [2]>, ...]
%initthrow}
```

  O código Java especificado aqui será copiado na declaração do construtor do analisador léxico.

```
Yylex()
throws < exceção [1]>[, < exceção [2]>, ...]
{... < Código > ... }
```

  Se o código Java dado na diretriz `%init{... %init}` gerar uma exceção que não é declarada, o arquivo fonte do analisador léxico poderá não compilar com êxito.

- **Código de Fim de Arquivo**

  A diretriz `%eof{... %eof}` permite escrever código Java a ser copiado na classe do de analisador léxico para execução depois que o fim de arquivo é alcançado.

```
%eof{
< Código >
%eof}
```

As diretrizes `%eof{` e `%eof}` devem estar situadas no começo das linhas. O código Java especificado em `< Código >` será executado imediatamente depois que o fim de arquivo é alcançado para a entrada de caracteres processada pela classe do analisador léxico.

O código dado na diretriz `%eof{... %eof}` potencialmente pode gerar uma exceção. Para declarar esta exceção, use a diretriz `%eofthrow{...%eofthrow}`.

```
%eofthrow{
<exception[1]>[, <exception[2]>, ... ]
%eofthrow}
```

O código Java especificado aqui será copiado na declaração do analisador léxico especificado na diretriz `%eof`.

- **Definição de Macros**

  A definição de macros segue a seguinte sintaxe:

  ```
  <macro> = <definição>
  ```

  O nome da macro deverá ser um identificador válido e a definição deverá ser uma expressão regular. A definição da macro poderá conter outra macro previamente definida, e a mesma será referenciada pela sintaxe `{<macro>}`.

- **Contagem de Linhas**

  A contagem de linhas é desativada por default, caso você queria ativá-la, poderá usar a diretiva line:

  ```
  %line
  ```

  O número da linha poderá ser acessado então pela variavel inteira yyline.

- **Compatibilidade com o parser Java CUP**

  A compatibilidade com o Java CUP se dá pela diretiva:

  ```
  %cup
  ```

  Mais adiante detalharemos o uso da parser Java CUP.

- **Character Sets**

  A configuração default do JLex é a alfabética, caracteres entre 0 e 127 inclusive. Pode-se usar o diretiva `%full` para aceitar os caracteres da tabela ASCII.

  ```
  %full
  ```

  Se a diretiva `%full` é usada, o JLex irá gerar um analizador léxico que aceitará caracteres entre 0 e 255 inclusive.

  O Java fornece uma classe String que suporta caracteres de 16-bits representados pelo Unicode. A diretiva `%unicode` pode ser usada para indicar o suporte a unicode.

```
%unicode
```

- **Especificando fim de arquivo**

  A diretiva `%eofval{ ... %eofval}` especifica o valor do fim de arquivo. O código java especificado será executado quando o fim do arquivo de entrada for alcançado. Esta diretiva normalmente é usada para retornar o simbolo de fim de arquivo, conforme exemplo abaixo:

```
%eofval{
 return (new Symbol(sym.EOF));
%eofval}
```

6.3.2.3 Regras JLex

A terceira parte do arquivo de especificação JLex consiste nas regras descritas através de expressões regulares para separações dos tokens identificados no arquivo de entrada. A definição das expressões regulares segue a seguinte sintaxe:

```
<Expressão Regular> { <Ação> }
```

Nesta parte do arquivo iremos identificar os símbolos terminais da linguagem FatoRH/W. A expressão regular identifica o token e a ação gera uma instância do símbolo terminal.

```
{ENQUANTO} { return new Symbol(sym.ENQUANTO); }
{DIGIT}+ { return new Symbol(sym.NUMBER,
  new String(yytext())); }
```

6.4 Java CUP

O Java CUP [1] (HUDSON, 1999) é o gerador de analisadores sintáticos usado no Projeto do Fatorlex. A gramática da linguagem FatoRh/W dará origem a uma arquivo de especificação que será processado pelo CUP e este irá gerar as classes Java que farão parte da solução Fatorlex.

O CUP faz uso da integração com o JLex para identificar os tokens, ou símbolos terminais. Um dos primeiros passos da criação da gramática (METSKER, 2001) da linguagem é a definição dos símbolos terminais ,identificados pelo analisador léxico, e dos não terminais, resolvidos pela regra da gramática da linguagem.

Na linguagem Fatorlex temos os seguintes símbolos terminais:

```
terminal PLUS, MINUS, TIMES, POWER, LPAREN,
    RPAREN, UMINUS, STRING, COMMA, SE, SENAO,
    ATRIB, NUMBER, DIVIDE, FIELD, IMPRIMA,
    DATE, ENTAO, FIM, SQL, FACA, ENQUANTO,
    E, OU, IDENT, DIFERENT, GREATEQUAL,
    LESSEQUAL, EQUAL, LESS, GREAT, SAIA,
    RETORNA, SIM, NAO;
```

E os seguintes símbolos não terminais:

[1] Constructor of Useful Parsers

```
non terminal expr_list, expr, statement_list,
    statement, funcao, variavel, programa,
    sairloop, relat, condicao , atribuicao,
    seentao, opt_senao, facaenquanto,
    imprimir, retornar;
```

Uma regra normalmente seguida é o uso de caracteres maiúsculos para símbolos terminais e minúsculos para símbolos não terminais.

Uma vez definido os símbolos, o próximo passo será a criação das regras da gramática. Um programa Fatorlex é identificado pelo símbolo não terminal `program` e as regras começarão por este símbolo.

```
start with programa;
```

O programa possui uma lista de procedimentos que será representado pelo símbolo `statement_list`

```
programa ::= statement_list:sl
```

A especificação de uma regra possui duas partes: do lado esquerdo está o símbolo não terminal a ser identificado e do lado direito os símbolos terminais ou não terminais que formarão o símbolo da esquerda.

Seguido ao nome do símbolo podemos adicionar uma variável que será usada para referenciar o conteúdo associado ao símbolo. No exemplo acima, o nome da variável é `sl` e ela deve ser precedida por dois pontos :.

A lista de procedimentos/declarações pode ser composta por um simples procedimento ou um encadeamento de procedimentos. Isto será representado pela regra abaixo:

```
statement_list ::=
                    statement:sp
                  | statement_list:sl statement:sp
```

Neste ponto temos um elemento chave na definição de gramáticas - um simbólo não terminal pode ser usado de forma recursiva na sua própria definição. Traduzindo o exemplo acima temos: um procedimento simples é uma lista de procedimentos e uma lista de procedimentos também é formada por uma lista seguida de um procedimento simples.

Este recurso é usado para empilhar elementos e tratá-los como uma unidade, ou seja, a lista de procedimentos será tratada como um único programa.

Seguindo na gramática do Fatorlex, um procedimento poderá ser identificado por uma das seguintes situações:

```
statement ::= atribuicao:at
                  | seentao:se
                  | facaenquanto:fe
                  | imprimir:imp
                  | retornar:ret
                  | sairloop:sai
                  | SQL:s
```

Dentre as opções temos os comandos da linguagem FatoRH/W [2] e o símbolo terminal SQL que será simplesmente reproduzido dentro do código final.

Para a atribuição, temos:

```
atribuicao     ::= IDENT:i ATRIB expr_list:exl
```

A atribuição é identificada pelo símbolo terminal IDENT, que foi definido no JLex como sendo uma string iniciada com caracter alfanumérico [3], seguido do símbolo terminal ATRIB (:=) e de uma lista de expressões (`expr_list`).

Os seguintes exemplos poderiam se encaixar dentro da regra de atribuição:

```
variavel := 34
valor := 34 * 2 - 2
total := valor + variavel
```

A gramática completa do Fatorlex poderá ser consultada no apendice deste trabalho, juntamente com exemplos de códigos na linguagem FatoRH/W bem como a conversão desses programas em código Java.

6.5 IDE Fatorlex

Como produto do processamento dos arquivos de especificação do JLex e Java CUP temos as classes [4] que formam o Fatorlex. A aplicação dessas classes será ilustrada com a ajuda de uma interface visual onde no lado esquerdo serão inseridos os comandos na linguagem FatoRH/W e no lado direito será exibido o código produzido em Java.

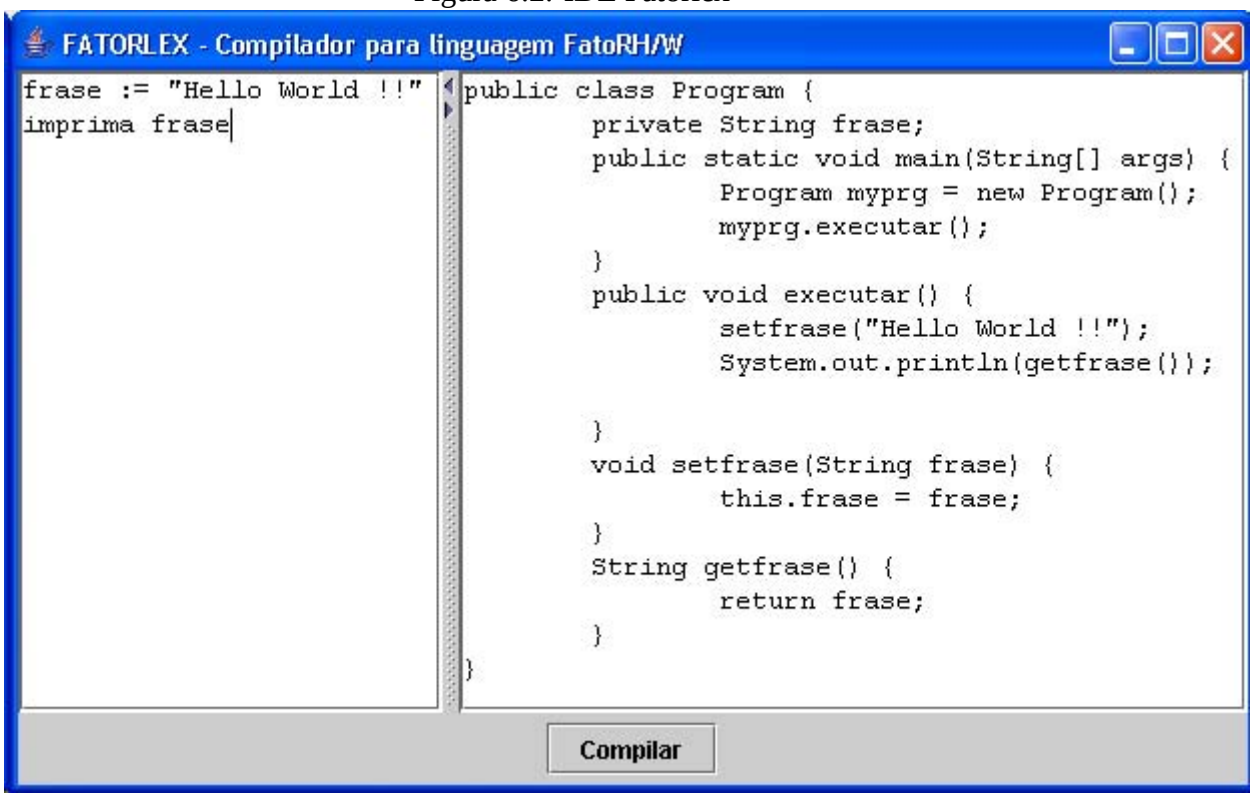
A figura IDE Fatorlex ilustra essa interface. O código fonte deste aplicativo poderá ser encontrado no apendice deste trabalho.

[2] Atribuição, SE ENTÃO, ENQUANTO, IMPRIMIR, RETORNAR, SAIR

[3] Veja Especificação do Fatolex.lex no apendice

[4] parser, Yylex e sym

Figura 6.2: IDE Fatorlex



7 VERIFICAÇÃO, LIBERAÇÃO E MANUTENÇÃO

7.1 Especificação de Teste

Para realização de testes do Fatorlex usaremos como base as fórmulas de eventos de Folha de Pagamento criadas pelos clientes do FatoRH/W. Existe cerca de 1000 fórmulas que já estão em uso pelo sistema legado.

Esses eventos deverão passar pela regra da gramática definida para o Fatorlex, pois um dos requisitos é a compatibilidade com o sistema legado. Para garantir esta etapa, um backup da base de dados de produção de um cliente da Soft Trade deverá ser usado nos testes.

Numa primeira fase, será criada uma interface Java cujos métodos são os essenciais para o cálculo da folha de pagamento. Essa interface será implementada por uma classe Java que irá disparar os procedimentos [1] criados no banco de dados pelo software legado. O cálculo da folha de pagamento do cliente deverá então ser executado e o tempo de execução de cada procedimento será registrado, bem como o resultado de seu cálculo.

A segunda fase dos testes inicia com a criação das classes Java pelo Fatorlex a partir de todas as fórmulas da base de cálculo do cliente. Essas classes deverão implementar a interface especificada na primeira fase. Caso alguma fórmula da base de dados do cliente não seja validada pelo Fatorlex, dependendo da natureza do erro encontrado, uma alteração deverá ocorrer no arquivo de especificação léxica ou então na especificação sintática.

Um benefício do uso do Fatorlex é a manutenção localizada nos arquivos de especificações, extremamente pequenos se comparado a grande quantidade de linhas de código do seu antecessor.

Como etapa final dos testes, a folha de pagamento do cliente deverá ser mais uma vez executada e o tempo de execução de cada classe Java deverá ser registrado, bem como o resultado de seu cálculo. Os resultados obtidos pela execução das classes Java serão comparados com os resultados apurados pelo disparo das procedures, permitindo assim a validação da performance no uso das classes Java e também a verificação da precisão dos novos valores cálculados.

O teste final será um paralelo da folha de pagamento rodando as verbas criadas pelo Fatorlex versus as verbas criadas pelo software legado.

[1] Procedures Oracle PL/SQL ou Microsoft Transact-SQL.

7.2 Liberação do Produto

Por tratar da rotina de cálculo da folha de pagamento dos clientes da Soft Trade, o produto Fatorlex faz parte do núcleo da solução da mesma. Ele desempenha um papel crítico no sistema devido a manipulação de valores financeiros elevados e envolve alto risco toda vez que sofre uma alteração.

Pelos motivos acima o Fatorlex somente entrará em produção após estar totalmente compatível com o sistema anterior e possuir resultados satisfatórios no quesito de performance e confiabilidade dos cálculos, além de um processo de implementação nos clientes bem definido.

Porém, para efeito da avaliação deste trabalho de mestrado, a versão disponível no anexo está operacional e pronta para ser usada.

7.3 Solicitação de Manutenção

A Manutenção do Fatorlex será assumida pela Soft Trade após o produto entrar em produção nos clientes e as solicitações de manutenção entrarão no processo adotado pela empresa. No modelo atual o cliente abre um chamado no Help Desk da empresa solicitando a manutenção do produto e a Soft Trade dá um parecer sobre o ocorrência do chamado de manutenção.

Parte III

Conclusão

8 CONCLUSÃO

A motivação do tema Fatorlex para a dissertação de mestrado veio da necessidade de criar uma nova versão para o compilador e linguagem criados e adotados pela Soft Trade.

Essa necessidade decorreu de algumas limitações do software legado usado até então pela Soft Trade, como por exemplo:

- Versão do software somente para sistema operacional Windows;
- Dependência de software proprietário no núcleo do software e no código gerado;
- Uma versão do software para cada banco de dados utilizado: Oracle, MSSQL, Sybase, etc;
- Alta complexidade na manutenção do software, dificultando a implementação de novas funcionalidades.

Para superar essas limitações foi proposta a criação de uma nova versão do compilador da linguagem FatoRH/W. Nesta nova versão o principal requisito é a linguagem Java como plataforma de desenvolvimento, pois a adoção da linguagem Java permite o ganho direto da portabilidade entre diversos sistemas operacionais, quebrando assim a principal limitação do software legado que é o uso restrito ao sistema operacional Windows.

Porém, com a criação do Fatorlex espera-se também obter outros ganhos, como por exemplo:

- A unificação do compilador da linguagem FatoRH/W em um componente de software, acabando com a necessidade da criação de uma versão separada para cada banco de dados conforme ocorre no software legado;
- A facilidade na manutenção e a flexibilidade na criação de novas funcionalidades, uma vez que o núcleo do software estará concentrado nos arquivos de especificação e não mais em um emaranhado com mais de mil linhas de código;
- O uso de software livre na composição da solução em oposição ao software proprietário do legado.

A criação de um novo compilador para linguagem FatoRH/W a partir do uso de geradores de analisadores léxicos e sintáticos mostrou-se uma opção viável para o uso em aplicações comerciais através da combinação das ferramentas JLex e Java CUP. As ferramentas usadas nasceram no mundo acadêmico para dar suporte ao problema de compiladores, facilitando a criação de novas linguagens a partir de arquivos de especificação onde

são definidos os símbolos e a gramática da linguagem. Essa abordagem resolve de forma elegante e simples um problema complexo para ser tratado sem o uso dos geradores.

O desenvolvimento do compilador em termos de arquivos de especificação facilita a manutenção pois as alterações na estrutura de linguagem ficam concentradas em poucos arquivos. Já a integração com a linguagem Java aumenta a flexibilidade na criação de novas funcionalidades através do uso das bibliotecas disponíveis em Java.

O resultado deste trabalho trouxe como benefício para Soft Trade a visualização do Fatorlex como um novo argumento de vendas, pois no software legado, todo projeto envolvendo alterações na linguagem FatoRH/W era visto como um complicador que encarecia o orçamento enviado aos clientes, devido a complexidade do código legado.

No novo modelo de componentes criado com o uso do Fatorlex, o analista de negócio sabe que a inclusão de novas funcionalidades na linguagem FatoRH/W está concentrada em um único componente bem definido, parametrizável e facilmente configurável ao gosto do cliente. Estas características liberam a criatividade dos analistas motivando a criação de novas funcionalidades relacionadas a linguagem FatoRH/W.

Como exemplificação do benefício citado acima, temos a inclusão de um novo comando na linguagem FatoRH/W, neste caso o comando PARACADA.

O comando PARACADA é uma nova estrutura na linguagem FatoRH/W e tem funcionalidade semelhante ao comando ENQUANTO[1], porém a estrutura de repetição é baseada em uma entidade do sistema - por exemplo o Funcionário - e não em uma condição como é o caso do comando ENQUANTO.

Para implementação deste novo comando na software legado foi necessario duas semanas de trabalho integral de um desenvolvedor para o entendimento do código e localização das partes afetadas pela inclusão. Com o Fatorlex esta alteração poderia ser afetuada em algumas horas pela alteração do arquivo de especificação da gramática da linguagem FatoRH/W, incluindo o novo comando PARACADA.

```
paracada  ::=  PARACADA entidade:e
                 statement_list:s  FIM
   {:  Tratamento do comando PARACADA  :};
```

8.1 Limitações e Avaliações da Implementação

A versão do Fatorlex entregue neste trabalho de mestrado foi apresentada aos analistas da Soft Trade. A seguir serão comentadas as limitações apontadas bem como os pontos avaliados nesta apresentação.

Um dos pontos mais discutidos foi a geração automática de código a partir das ferramentas JLex e Java CUP. Isso num primeiro momento gerou insegurança nos analistas, essas foram minimizadas pelo fato de ambas as ferramentas possuírem código aberto, e possíveis limitações encontradas no uso profissional poderem ser incluídas no código fonte da ferramenta.

Ao identificar os arquivos de especificação usados pelas ferramentas como os arquivos fontes do Fatorlex, surgiu a limitação da impossibilidade do uso de um ambiente integrado para o desenvolvimento do código Java embutido no arquivo de especificação do Java CUP. No caso os profissionais indicaram que seria inviável usar o Eclipse - o software usado pelos desenvolvedores da Soft Trade - para edição do ‘Fatorlex.cup’, isso pode prejudicar a manutenção no código Java do Fatorlex.

[1] ver Linguagem FatoRH/W - Comandos / Estruturas

Outra limitação apontada foi a ausência de uma camada de acesso ao banco de dados, isto dificulta a implantação do Fatorlex em ambiente de produção, porém o tratamento desta camada merece um trabalho separado pois uma boa implementação envolveria o mapeamento objeto-relacional das entidades do sistema de folha de pagamento da Soft Trade.

A preocupação com a performance e o gerenciamento da memória usada pelo conjunto de classes gerado pelo Fatorlex é evidente. A implementação completa do Fatorlex como um produto em ambiente de produção nos clientes necessita de mecanismos para o acompanhamento da performance de execução das classes e também de uma ferramenta para o gerenciamento da memória consumida pelas classes.

O Fatorlex foi criado dentro de uma especificação fechada, pois deveria manter compatibilidade com o sistema legado. Após a entrada em produção o mesmo poderia incluir novos recursos na linguagem, como a criação de novos tipos ou comandos, pois a linguagem FatoRH/W possui poucos recursos, fato este decorrente da complexidade do código legado.

8.2 Sugestão de Trabalhos Futuros

Atualmente, o sistema de folha de pagamento da Soft Trade possui uma versão disponível para acesso via internet. Essa funcionalidade foi decorrente da concentração de esforços visando à migração do sistema legado para um novo sistema cujo acesso ocorre através do navegador de paginas HTML.

O uso do compilador da linguagem FatoRH/W se dá pelas paginas HTML que enviam solicitações para as servlets executadas no servidor Web da aplicação. A criação de serviços disponibilizados em forma Web Services poderia fazer a ponte entre as chamadas das servlet e as páginas HTML. O uso de Web Services poderia fornecer serviços de interpretação da linguagem FatoRH/W gerando classes Java, esses serviços poderiam ser usados não somente pelos usuários que acessam as páginas HTML, mas também por outros aplicativos que possuem o interpretador da linguagem FatoRH/W em seu núcleo.

A seqüência deste trabalho poderia envolver o mapeamento das entidades do sistema e a criação de interfaces que disponibilizarão os serviços a serem implementados pelo Web Service.

Outra linha de extensão do trabalho poderia envolver implementação dos conceitos da orientação a objetos na linguagem FatoRH/W, como por exemplo: a criação de tipos abstratos de dados, o conceito de herança e polimorfismo, entre outros. Seguindo esta linha, este trabalho poderia dar origem a uma nova linguagem orientada a objetos e voltada para o usuário leigo.

REFERÊNCIAS

AHERN, D. M.; CLOUSE, A.; TURNER, R. ***CMMI distilled***: a practical introduction to integrated process improvement, second edition. Boston, MA: Addison Wesley, 2003.

BERK, E. ***JLex***: a lexical analyzer generator for java. http://www.cs.princeton.edu/ appel/modern/java/JLex/.

BEZERRA, E. ***Princípios de Análise e Projeto de Sistemas com UML***. Rio de Janeiro, RJ: Campus, 1999.

BOOCH, G.; RUMBAUGH, J.; JACOBSON, I. ***The Unified Modeling Language User guide***. Boston, MA: Addison Wesley, 1999.

DRUCKER, P. F. ***Management***: tasks, responsibilities, practices. New York, NY: Harper & Row, 1974.

HUDSON, S. E. ***CUP User's Manual***. http://www.cs.princeton.edu/ appel/modern/java/CUP/.

IEEE. ***SWEBOK***: the guide to the software engineering body of knowledge. New York, NY: IEEE, 2001.

KRUCHTEN, P. ***Introdução ao RUP***: rational unified process. Rio de Janeiro: Ciência Moderna, 2003.

MCCONNELL, S. ***Rapid Development***: taming wild software schedules. Redmond: Microsoft Press, 1996.

METSKER, S. J. ***Building Parsers with Java***. Boston, MA: Addison Wesley, 2001.

MEYER, B. ***Object-Oriented Software Construction***. 2nd.ed. Santa Barbara, CA: Interactive Software Engineering Inc., 1997.

PRESSMAN, R. S. ***Engenharia de Software***. São Paulo: Makron Books do Brasil, 1995.

VERMEIR, D. ***An Introduction to Compilers***. 2001. Tese (Doutorado em Ciência da Computação) — Universidade de Bruxelas, Bruxelas.

APÊNDICE A ARQUIVOS DE ESPECIFICAÇÕES

A.1 Fatorlex.lex

```
package fatorlex;
import java_cup.runtime.Symbol;
%%
%full
%public
%cup
%line
%eofval{
  return (new Symbol(sym.EOF));
%eofval}
DIGIT=[0-9]
ALPHA=[a-zA-Z_]
IDENT={ALPHA}({ALPHA}|{DIGIT})*
SPACE=[ \t\r\n\f\b]
SENAO=[sS][eE][nN][ãÃ][oO]
IMPRIMA=[iI][mM][pP][rR][iI][mM][aA]
ENTAO=[eE][nN][tT][ãÃ][oO]
FACA=[fF][aA][çÇ][aA]
ENQUANTO=[eE][nN][qQ][uU][aA][nN][tT][oO]
SE=[sS][eE]
FIM=[fF][iI][mM]
SAIA=[sS][aA][iI][aA]
RETORNA=[rR][eE][tT][oO][rR][nN][aA]
SIM=[sS][iI][mM]
NAO=[nN][ãÃ][oO]
OU=[oO][uU]
E=[eE]
STRING=\"[^\"]*\"
SSTRING=\’[^\’]*\’
DATE=\{[^\}]*\}
FIELD=\[[^\]]*\]
SQL=[sS][qQ][lL].*[fF][iI][mM]{SPACE}[sS][qQ][lL]
COMMENT=\/\*([^*]|[*][^/])*\*\/

%%
```

```
";" { return new Symbol(sym.EOF); }
"," { return new Symbol(sym.COMMA); }
"+" { return new Symbol(sym.PLUS); }
"-" { return new Symbol(sym.MINUS); }
"*" { return new Symbol(sym.TIMES); }
"^" { return new Symbol(sym.POWER); }
"(" { return new Symbol(sym.LPAREN); }
")" { return new Symbol(sym.RPAREN); }
":=" { return new Symbol(sym.ATRIB); }
"<>" { return new Symbol(sym.DIFERENT); }
">=" { return new Symbol(sym.GREATEQUAL); }
"<=" { return new Symbol(sym.LESSEQUAL); }
"=" { return new Symbol(sym.EQUAL); }
"<" { return new Symbol(sym.LESS); }
">" { return new Symbol(sym.GREAT); }
"/" { return new Symbol(sym.DIVIDE); }
{ENTAO} { return new Symbol(sym.ENTAO); }
{SENAO} { return new Symbol(sym.SENAO); }
{SE} { return new Symbol(sym.SE); }
{DATE} { return new Symbol(sym.DATE); }
{FIELD} { return new Symbol(sym.FIELD); }
{SQL} { return
    new Symbol(sym.SQL, new String(yytext())); }
{SSTRING} { return
    new Symbol(sym.STRING, new String(yytext())); }
{STRING} { return
    new Symbol(sym.STRING, new String(yytext())); }
{COMMENT} { System.out.println(yytext()); }
{FIM} { return new Symbol(sym.FIM); }
{FACA} { return new Symbol(sym.FACA); }
{ENQUANTO} { return new Symbol(sym.ENQUANTO); }
{SAIA}  { return new Symbol(sym.SAIA); }
{RETORNA}  { return new Symbol(sym.RETORNA); }
{IMPRIMA}  { return new Symbol(sym.IMPRIMA); }
{SIM}  { return new Symbol(sym.SIM); }
{NAO}  { return new Symbol(sym.NAO); }
{E} { return new Symbol(sym.E); }
{OU} { return new Symbol(sym.OU); }
{DIGIT}*"."{DIGIT}* { return
   new Symbol(sym.NUMBER, new String(yytext())); }
{DIGIT}+ { return
   new Symbol(sym.NUMBER, new String(yytext())); }
{IDENT} { return
   new Symbol(sym.IDENT, new String(yytext())); }
{SPACE} {;}
. { System.err.println("Caracter Invalido: "+yytext()+
    " na linha " + (yyline + 1)); }
```

A.2 Fatorlex.cup

```
package fatorlex;
import java_cup.runtime.*;
import java.util.*;

action code {:
public Hashtable TabSym = new Hashtable();
public Hashtable TabSymVar = new Hashtable();
:}

parser code {:
    public String meuPrograma = "";

    public String toString() {
          return meuPrograma;
    }

public static void main(String args[])
throws Exception {
           new parser(new Yylex(System.in)).parse();
}

public void report_fatal_error(String message, Object info)
    { done_parsing();
      System.err.println(message);
      System.exit(1);
     }

public void syntax_error(Symbol cur_token)  {
   try{
   report_error("Erro de Sintaxe: "+cur_token.value.toString(),null);
   }
   catch (Exception e)
   {
   report_error("Erro de Sintaxe: " + cur_token.toString() , null);
   }
}
            :}

terminal PLUS, MINUS, TIMES, POWER, LPAREN, RPAREN, UMINUS, STRING,
    COMMA, SE, SENAO, ATRIB, NUMBER, DIVIDE, FIELD, IMPRIMA,
    DATE, ENTAO, FIM, SQL, FACA, ENQUANTO, E, OU, IDENT,
    DIFERENT, GREATEQUAL, LESSEQUAL, EQUAL, LESS, GREAT, SAIA,
    RETORNA, SIM, NAO;

non terminal expr_list, expr, statement_list, statement, funcao,
    variavel, programa, sairloop;
non terminal relat, condicao , atribuicao, seentao, opt_senao,
```

```
    facaenquanto, imprimir, retornar;

precedence left E, OU;
precedence left MINUS, PLUS;
precedence left TIMES, DIVIDE;
precedence left UMINUS;
precedence left POWER;

start with programa;

programa ::= statement_list:sl
             {: RESULT = sl;

   String myprg = "public class Program{\n";
   String getset = "";
   Enumeration IdentTabSym = TabSymVar.keys();
   while (IdentTabSym.hasMoreElements())
     {
     String Ident = IdentTabSym.nextElement().toString() ;
     myprg += "private " + TabSymVar.get(Ident)
           + " " + Ident + ";\n";
     getset += "void set" + Ident + "(" + TabSymVar.get(Ident)
            + " " + Ident + ") {\n" ;
     getset += "this." + Ident + "=" + Ident +";}\n" ;
     getset += TabSymVar.get(Ident) + " get" + Ident + "() {\n" ;
     getset += "return " + Ident +";}\n" ;
     }
    myprg += "public static void main(String[] args){\n";
    myprg += "Program myprg = new Program();\n";
    myprg += "myprg.executar();\n";
   IdentTabSym = TabSymVar.keys();
   while (IdentTabSym.hasMoreElements())
     {
     String Ident = IdentTabSym.nextElement().toString() ;
     }
   myprg += "}\n";
   myprg += "public void executar(){\n";

   myprg += RESULT + "}\n" + getset + "}\n";
   System.out.println( myprg );
   parser.meuPrograma = myprg;
      :}
             |
error:er
         {: parser.report_fatal_error(er.toString(), null); :};

statement_list ::= statement_list:sl statement:sp
                   {: RESULT = sl.toString() + sp.toString()+"\n";
```

```
      :}
                  | statement:sp
                    {: RESULT = sp.toString() + "\n";
    :};

statement ::= atribuicao:at
                    {: RESULT = at;
   :}
                  | seentao:se
                    {: RESULT = se;
   :}
                  | facaenquanto:fe
                    {: RESULT = fe;
   :}
                  | imprimir:imp
       {: RESULT = imp;
   :}
                  | retornar:ret
                    {: RESULT = ret; :}
                  | sairloop:sai
                    {: RESULT = sai; :}
                  | SQL:s {: RESULT = s.toString();
     :};

seentao        ::= SE condicao:c ENTAO statement_list:s opt_senao:o FIM
                    {: RESULT =
                       new String("if(" + c + "){\n"+ s + "\n}\n" + o );
     :};

opt_senao      ::= SENAO statement_list:s
                    {: RESULT = new String("else {\n" + s + "\n}\n");
     :}
                  | {: RESULT = new String("");
                    :}
     ;

facaenquanto   ::= ENQUANTO condicao:c FACA statement_list:s FIM
                    {: RESULT =
                       new String("while (" + c + "){\n" + s + "\n}\n" )
                     :};

sairloop       ::= SAIA {: RESULT = new String("break;"); :};

retornar       ::= RETORNA {: RESULT = new String("return;"); :};

imprimir       ::= IMPRIMA expr_list:expimp
                    {: RESULT =
                       new String("System.out.println(" + expimp + ");\n
```

```
                    :};

atribuicao     ::= IDENT:i ATRIB expr_list:exl
                   {: RESULT = new String("set" + i + "(" + exl + ");")
                     if ( ! TabSym.containsKey(i.toString()) )
                      {
                       TabSym.put(i.toString(),
                                  TabSym.get(exl.toString()).toString()
                       TabSym.put("get" + i.toString() + "()",
                                  TabSym.get(exl.toString()).toString()
                       }
                     else
                        if ( !
TabSym.get(i.toString()).toString().equals(
                        TabSym.get(exl.toString()).toString()))
                   parser.report_fatal_error("Tipos Incompativeis " + i
                        + "("+ TabSym.get(i.toString()) + ") " + i+ "("
                        + TabSym.get(exl.toString()) + ")" , null);

                     if ( ! TabSymVar.containsKey(i.toString()) )
                       TabSymVar.put(i.toString(), TabSym.get(
                       exl.toString()).toString());
    :};

expr_list ::= expr:e
      {: RESULT= e.toString();
               if ( ! TabSym.containsKey(e.toString()) )
                  TabSym.put(e.toString(), "Nao Definido");
       :};

expr      ::= expr:l PLUS:p expr:r
      {: RESULT=new String(l + "+" + r );
               if ( ! TabSym.containsKey(l.toString()) )
                   parser.report_fatal_error("Expressão Indefinida "
                   + l, null);
               if ( ! TabSym.containsKey(r.toString()) )
                   parser.report_fatal_error("Expressão Indefinida "
                   + r, null);
               if ( ! TabSym.get(l.toString()).toString().equals(
               TabSym.get(r.toString()).toString()))
                   parser.report_fatal_error("Tipos Incompativeis "
                   + l + "("+ TabSym.get(l.toString()) + ") " + r+
                   "("+ TabSym.get(r.toString()) + ")", null);
              TabSym.put(RESULT, TabSym.get(l.toString()).toString());
       :}
            | expr:l MINUS expr:r
      {: RESULT=new String( l.toString() + "-" + r.toString() );
               if ( ! TabSym.containsKey(l.toString()) )
```

```
                parser.report_fatal_error("Expressão Indefinida "
                + l, null);
            if ( ! TabSym.containsKey(r.toString()) )
                parser.report_fatal_error("Expressão Indefinida "
                + r, null);
            if ( ! TabSym.get(l.toString()).toString().equals(
            TabSym.get(r.toString()).toString()))
                parser.report_fatal_error("Tipos Incompativeis " +
                l + "("+ TabSym.get(l.toString()) + ") " + r+ "("+
                TabSym.get(r.toString()) + ")", null);
           TabSym.put(RESULT, TabSym.get(l.toString()).toString());
:}
  | expr:l TIMES expr:r
    {: RESULT=new String(l.toString() + "*" + r.toString() );
            if ( ! TabSym.containsKey(l.toString()) )
                parser.report_fatal_error("Expressão Indefinida "
                + l, null);
            if ( ! TabSym.containsKey(r.toString()) )
                parser.report_fatal_error("Expressão Indefinida "
                + r, null);
            if ( ! TabSym.get(l.toString()).toString().equals(
             TabSym.get(r.toString()).toString()))
                parser.report_fatal_error("Tipos Incompativeis "
                + l + "("+ TabSym.get(l.toString()) + ") "
                + r+ "("+ TabSym.get(r.toString()) + ")", null);
           TabSym.put(RESULT, TabSym.get(l.toString()).toString());
:}
  | expr:l DIVIDE expr:r
    {: RESULT=new String( l.toString() + "/" + r.toString() );
            if ( ! TabSym.containsKey(l.toString()) )
                parser.report_fatal_error("Expressão Indefinida "
                + l, null);
            if ( ! TabSym.containsKey(r.toString()) )
                parser.report_fatal_error("Expressão Indefinida "
                + r, null);
            if ( ! TabSym.get(l.toString()).toString().equals(
            TabSym.get(r.toString()).toString()))
                parser.report_fatal_error("Tipos Incompativeis "
                + l + "("+ TabSym.get(l.toString()) + ") "
                + r+ "("+ TabSym.get(r.toString()) + ")", null);
           TabSym.put(RESULT, TabSym.get(l.toString()).toString());
:}
          | MINUS expr:r
    {: RESULT=new String("-" + r.toString());
            if ( TabSym.get(r.toString()).toString().equals("double"
            {
               if ( ! TabSym.containsKey(RESULT) )
                  TabSym.put(RESULT, "double");                 }
```

```
                else
                   parser.report_fatal_error("Inválido " + RESULT, null);
  :}
      %prec UMINUS
    | expr:l POWER expr:r
      {: RESULT=new String("power(" + l.toString() + ","
       + r.toString() +")" );
  :}
    | LPAREN expr:e RPAREN
      {: RESULT=e;
  :}
    | NUMBER:n
      {: RESULT=n.toString();
              TabSym.put(n.toString(), "double");
  :}
    | STRING:n
      {: RESULT=n.toString();
              TabSym.put(n.toString(), "String");
  :}
    | variavel:n
      {: RESULT= n.toString();
  :}
    | funcao:n
      {: RESULT=n.toString();
  :}
    ;

variavel ::=  IDENT:n
      {: RESULT= "get" + n.toString() + "()";
       :}
          ;

funcao   ::=  IDENT:f LPAREN expr_list:l COMMA expr_list:r RPAREN
              {: RESULT=new String("Funcao:"  +f + "(" + l.toString()
               + "," + r.toString()+")" );
       :}
           ;


condicao ::= condicao:c1 E condicao:c2
      {: RESULT= c1.toString() + " && " + c2.toString();
       :}

          | condicao:c1 OU condicao:c2
      {: RESULT= c1.toString() + " || " + c2.toString();
       :}
          | LPAREN condicao:c RPAREN
              {: RESULT= "( " + c.toString() + " )";
```

```
        :}
           | expr:l relat:re expr:r
        {: RESULT=new String(l.toString() + re.toString()
         + r.toString() );
                if ( ! TabSym.containsKey(l.toString()) )
                    parser.report_fatal_error("Expressão Indefinida "
                    + l, null);
                if ( ! TabSym.containsKey(r.toString()) )
                    parser.report_fatal_error("Expressão Indefinida "
                    + r, null);
                if ( ! TabSym.get(l.toString()).toString().equals(
                TabSym.get(r.toString()).toString()))
                    parser.report_fatal_error("Tipos Incompativeis "
                    + l + "("+ TabSym.get(l.toString()) + ") "
                    + r+ "("+ TabSym.get(r.toString()) + ")", null);
               TabSym.put(RESULT, TabSym.get(l.toString()).toString());

        :}            ;

relat::=   LESS           {: RESULT=new String("<");
:}
    |  EQUAL          {: RESULT=new String("==");
:}
    |  LESSEQUAL      {: RESULT=new String("<=");
:}
    |  GREAT       {: RESULT=new String(">");
:}
    |  DIFERENT       {: RESULT=new String("!>");
:}
    |  GREATEQUAL     {: RESULT=new String(">=");
:}
    ;
```

APÊNDICE B EXEMPLO DE PROGRAMAS EM FATORH/W

B.1 IRRF

B.1.1 Programa Fonte

```
salario:=7000
faixa_01_limite := 1058
faixa_02_limite := 2215
faixa_02_imposto := 15
faixa_02_deducao := 158.7
faixa_03_imposto := 27.5
faixa_03_deducao := 423.08
se salario <= faixa_01_limite então
   imposto := 0
senão
   se salario  <= faixa_02_limite então
      imposto := salario * faixa_02_imposto / 100
                 - faixa_02_deducao
   senão
      imposto := salario * faixa_03_imposto / 100
                 - faixa_03_deducao
   fim
fim
imprima "Para o salário: "
imprima salario
imprima "O imposto devido é: "
imprima imposto
```

B.1.2 Código Produzido em Java

```
public class Irrf {
private double imposto;
private double faixa_02_limite;
private double faixa_01_limite;
private double faixa_03_deducao;
private double faixa_02_deducao;
private double salario;
private double faixa_03_imposto;
private double faixa_02_imposto;
```

```
public static void main(String[] args) {
Irrf myprg = new Irrf();
myprg.executar();
}
public void executar() {
setsalario(7000);
setfaixa_01_limite(1058);
setfaixa_02_limite(2215);
setfaixa_02_imposto(15);
setfaixa_02_deducao(158.7);
setfaixa_03_imposto(27.5);
setfaixa_03_deducao(423.08);
if (getsalario() <= getfaixa_01_limite()) {
setimposto(0);
} else {
if (getsalario() <= getfaixa_02_limite()) {
setimposto(
getsalario() * getfaixa_02_imposto() / 100
- getfaixa_02_deducao());
} else {
setimposto(
getsalario() * getfaixa_03_imposto() / 100
- getfaixa_03_deducao());
}
}
System.out.println("Para o salário: ");
System.out.println(getsalario());
System.out.println("O imposto devido é: ");
System.out.println(getimposto());
}

void setimposto(double imposto) {
this.imposto = imposto;
}
double getimposto() {
return imposto;
}
void setfaixa_02_limite(double faixa_02_limite) {
this.faixa_02_limite = faixa_02_limite;
}
double getfaixa_02_limite() {
return faixa_02_limite;
}
void setfaixa_01_limite(double faixa_01_limite) {
this.faixa_01_limite = faixa_01_limite;
}
double getfaixa_01_limite() {
return faixa_01_limite;
```

```
}
void setfaixa_03_deducao(double faixa_03_deducao) {
this.faixa_03_deducao = faixa_03_deducao;
}
double getfaixa_03_deducao() {
return faixa_03_deducao;
}
void setfaixa_02_deducao(double faixa_02_deducao) {
this.faixa_02_deducao = faixa_02_deducao;
}
double getfaixa_02_deducao() {
return faixa_02_deducao;
}
void setsalario(double salario) {
this.salario = salario;
}
double getsalario() {
return salario;
}
void setfaixa_03_imposto(double faixa_03_imposto) {
this.faixa_03_imposto = faixa_03_imposto;
}
double getfaixa_03_imposto() {
return faixa_03_imposto;
}
void setfaixa_02_imposto(double faixa_02_imposto) {
this.faixa_02_imposto = faixa_02_imposto;
}
double getfaixa_02_imposto() {
return faixa_02_imposto;
}
}
```

B.2 Fatorial

B.2.1 Programa Fonte

```
fatorial_de := 12
se fatorial_de < 0 então
   imprima
   "O cálculo do fatorial aplica-se somente a inteiros positivos"
   retorna
fim
textoderetorno := "Resultado do cálculo do fatorial"
imprima "Calculando Fatorial de: "
imprima fatorial_de
resultado := 1
enquanto fatorial_de > 1 faça
   resultado := resultado * fatorial_de
```

```
   fatorial_de := fatorial_de -1
   se fatorial_de > 100 ou resultado > 100000*100000 então
      imprima textoderetorno +
      " acima da capacidade do processador"
      saia
   fim
fim
imprima textoderetorno
imprima resultado
```

B.2.2 Código Produzido em Java

```
public class Fatorial {
private String textoderetorno;
private double resultado;
private double fatorial_de;
public static void main(String[] args) {
Fatorial myprg = new Fatorial();
myprg.executar();
}
public void executar() {
setfatorial_de(12);
if (getfatorial_de() < 0) {
System.out.println(
"O cálculo do fatorial aplica-se somente a inteiros positivos");
return;
}
settextoderetorno("Resultado do cálculo do fatorial");
System.out.println("Calculando Fatorial de: ");
System.out.println(getfatorial_de());
setresultado(1);
while (getfatorial_de() > 1) {
setresultado(getresultado() * getfatorial_de());
setfatorial_de(getfatorial_de() - 1);
if (getfatorial_de() > 100 ||
   getresultado() > 100000 * 100000) {
System.out.println(
gettextoderetorno()
+ " acima da capacidade do processador");
break;
}
}
System.out.println(gettextoderetorno());
System.out.println(getresultado());
}
void settextoderetorno(String textoderetorno) {
this.textoderetorno = textoderetorno;
}
String gettextoderetorno() {
```

```
return textoderetorno;
}
void setresultado(double resultado) {
this.resultado = resultado;
}
double getresultado() {
return resultado;
}
void setfatorial_de(double fatorial_de) {
this.fatorial_de = fatorial_de;
}
double getfatorial_de() {
return fatorial_de;
}
}
```

B.3 Teste

B.3.1 Programa Fonte

```
x:="teste"
a := 2 b := 9*(3+3) se  3+2>=3 e (b<12+3-1 ou b<2)
 e a<2 ou a>0 então y:= 454 y:=y+45

x:= "concatena"
conta := 1
 senão x:="acabopu" fim
/*sql select * from funcionario fim sql*/
z := 3*y + (1-y/2)
t :=  2-4*5
enquAntO  conta<5 faça
/* um comentario fora de lugar */
y:= 454*y
y:=y+4
conta:=conta+1
imprima x  + " um -- "
fim
imprima "passou no teste"
se a > 9 então
y := y + 3 *1+t

fim
```

B.3.2 Código Produzido em Java

```
/*sql select * from funcionario fim sql*/
/* um comentario fora de lugar */
public class Program {
private double b;
private String x;
```

```
private double a;
private double t;
private double z;
private double conta;
private double y;
public static void main(String[] args) {
Program myprg = new Program();
myprg.executar();
}
public void executar() {

setx("teste");
seta(2);
setb(9 * 3 + 3);
if (3 + 2 >= 3
&& (getb() < 12 + 3 - 1 || getb() < 2)
&& geta() < 2
|| geta() > 0) {
sety(454);
sety(gety() + 45);
setx("concatena");
setconta(1);

} else {
setx("acabopu");

}

setz(3 * gety() + 1 - gety() / 2);
sett(2 - 4 * 5);
while (getconta() < 5) {
sety(454 * gety());
sety(gety() + 4);
setconta(getconta() + 1);
System.out.println(getx() + " um -- ");

}

System.out.println("passou no teste");

if (geta() > 9) {
sety(gety() + 3 * 1 + gett());

}

}

void setb(double b) {
```

```
this.b = b;
}
double getb() {
return b;
}
void setx(String x) {
this.x = x;
}
String getx() {
return x;
}
void seta(double a) {
this.a = a;
}
double geta() {
return a;
}
void sett(double t) {
this.t = t;
}
double gett() {
return t;
}
void setz(double z) {
this.z = z;
}
double getz() {
return z;
}
void setconta(double conta) {
this.conta = conta;
}
double getconta() {
return conta;
}
void sety(double y) {
this.y = y;
}
double gety() {
return y;
}
}
```

APÊNDICE C IDE FATORLEX

C.1 Aplicativo.java

```
import java.awt.BorderLayout;
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.WindowEvent;
import java.awt.event.WindowListener;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.WindowConstants;

public class Aplicativo extends JFrame
       implements WindowListener,ActionListener
{
private JSplitFatorlex areaPanel;
private JPanel geral = new JPanel(new BorderLayout());
public Aplicativo()
{
super();
this.setSize(800,570);
        this.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
this.setTitle("FATORLEX - Compilador para linguagem FatoRH/W");
this.setContentPane(geral);
this.addWindowListener(this);
montaTela();
this.show();

}
public static void main(String[] args)
{
new Aplicativo();
}
public void windowOpened(WindowEvent e) {}
public void windowClosing(WindowEvent e) {}
public void windowClosed(WindowEvent e) {}
```

```
public void windowIconified(WindowEvent e) {}
public void windowDeiconified(WindowEvent e) {}
public void windowActivated(WindowEvent e) {}
public void windowDeactivated(WindowEvent e) {}
public void actionPerformed(ActionEvent arg0)
{
if (arg0.getActionCommand().equals("Compilar"))
{
areaPanel.compila();
}
}
public void montaTela()
{
try
{
areaPanel = new JSplitFatorlex();
geral.add(areaPanel,BorderLayout.CENTER);
        JPanel botoes = new JPanel(new BorderLayout());
        JPanel Termina = new JPanel(new FlowLayout());
        JPanel Proximo = new JPanel(new FlowLayout());
JButton proximo = new JButton("Compilar");
JButton terminar = new JButton("Terminar");
proximo.setVisible(true);
terminar.setVisible(true);
proximo.addActionListener(this);
terminar.addActionListener(this);
Proximo.add(proximo);
        Termina.add(terminar);
        botoes.add(Proximo,BorderLayout.CENTER);
geral.add(botoes,BorderLayout.SOUTH);
this.setContentPane(geral);
}
catch (Exception e)
{
e.printStackTrace();
}
}
}
```

C.2 JSplitFatorlex.java

```
import java.awt.Color;
import java.awt.BorderLayout;
import java.awt.Font;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.StringReader;
import javax.swing.JScrollPane;
```

```
import javax.swing.JSplitPane;
import java.util.ArrayList;
import javax.swing.JPanel;
import javax.swing.JTextArea;
import javax.swing.event.ChangeEvent;
import javax.swing.event.ChangeListener;
import fatorlex.Yylex;
import fatorlex.parser;

public class JSplitFatorlex extends JPanel
       implements ChangeListener, ActionListener
{
private ArrayList botoesDeRadio;
private JTextArea panelFatorlex;
private JTextArea panelJava;

public JSplitFatorlex() throws Exception
{
super();
botoesDeRadio = new ArrayList();
this.setLayout(new BorderLayout());
this.setBackground(Color.WHITE);
JPanel corpo = new JPanel(new BorderLayout());
corpo.setBackground(Color.WHITE);
JSplitPane cima = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT);
JPanel esq = new JPanel(new BorderLayout());
JPanel dir = new JPanel(new BorderLayout());
panelFatorlex = new JTextArea();
Font courier = new Font("Courier New",Font.PLAIN,14);
panelFatorlex.setFont(courier);
panelJava = new JTextArea();
panelJava.setFont(courier);
esq.add(panelFatorlex);
dir.add(panelJava);
JScrollPane scrollEsq = new JScrollPane(esq);
scrollEsq.setBackground(Color.WHITE);
JScrollPane scrollDir = new JScrollPane(dir);
scrollDir.setBackground(Color.WHITE);
cima.add(scrollEsq);
cima.add(scrollDir);
cima.setOneTouchExpandable(true);
cima.setDividerLocation(300);
corpo.add(cima);
this.add(corpo,BorderLayout.CENTER);

}
public void stateChanged(ChangeEvent arg0) {}
```

```
public void compila()
{
parser prs = new parser(new Yylex(
  new StringReader(panelFatorlex.getText())));
try {
prs.parse();
} catch (Exception e) {
e.printStackTrace();
}
panelJava.setText(prs.toString());
}
public void actionPerformed(ActionEvent arg0) {}

}
```