UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO – BACHARELADO

COMPILADOR JAVA 5.0 PARA GERAR CÓDIGO C++ PARA PLATAFORMA PALM OS

JÚLIO VILMAR GESSER

BLUMENAU
2007

2007/1-22

JÚLIO VILMAR GESSER

COMPILADOR JAVA 5.0 PARA GERAR CÓDIGO C++ PARA PLATAFORMA PALM OS

Trabalho de Conclusão de Curso submetido à Universidade Regional de Blumenau para a obtenção dos créditos na disciplina Trabalho de Conclusão de Curso II do curso de Ciências da Computação — Bacharelado.

Profa. Joyce Martins, Mestre - Orientadora

BLUMENAU
2007

2007/1-22

COMPILADOR JAVA 5.0 PARA GERAR CÓDIGO C++ PARA PLATAFORMA PALM OS

Por

JÚLIO VILMAR GESSER

Trabalho aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II, pela banca examinadora formada por:

Presidente:	Profa. Joyce Martins, Mestre – Orientadora, FURB
Membro:	Prof. Jomi Fred Hübner, Doudor – FURB
Membro:	Prof. Adilson Vahldick, Especialista – FURB

Blumenau, 11 de julho de 2007

Dedico este trabalho a todas as pessoas que me apoiaram durante a realização do mesmo, especialmente minha noiva Sílvia.

AGRADECIMENTOS

À minha família, que sempre esteve ao meu lado, apoiando e incentivando.

À minha noiva, Sílvia Hêdla Correia de Sales, por ter compreendido minha ausência durante o desenvolvimento do trabalho, pelo companheirismo, e pelo apoio e incentivo.

Aos amigos, que sempre se fizeram presentes.

À minha orientadora, Joyce Martins, pela excelente professora que sempre foi, e por ter acreditado na conclusão deste trabalho.

Aos desenvolvedores que contribuíram encontrando erros na gramática implementada.

A todos os professores que contribuíram para minha formação no decorrer do curso.

O verdadeiro sinal de inteligência não é o conhecimento, mas a imaginação.

Albert Einstein

RESUMO

Este trabalho apresenta o desenvolvimento de um compilador para linguagem de programação Java 5.0, cujo código objeto deve ser executável nativo para plataforma Palm OS. Para tanto, o compilador gera código C++ como código intermediário, sendo necessário gerar código para simular o coletor de lixo do Java. Para auxiliar o desenvolvimento do compilador, foi utilizado o JavaCC para gerar os analisadores léxico e sintático, bem como para construir a *Abstract Syntax Tree* (AST). A análise semântica e a geração de código foram também desenvolvidas em Java e ambas fazem varreduras na AST obtida na análise sintática. Foram ainda implementadas em C++ algumas bibliotecas da *Application Programming Interface* (API) do Java. Por fim, é mostrado um estudo de caso implementado em Java para testar o compilador.

Palavras-chave: Palm OS. Java. C++. Compilador.

ABSTRACT

This work presents the development of a compiler for the Java 5.0 programming language, whose object code must be native executable for the Palm OS platform. For this, the compiler generates C++ code as intermediate code, being necessary to generate code to simulate the Java garbage collector. To assist the development of the compiler, the JavaCC was used to generate the lexical and syntactic analyzers, as well to construct the AST. The semantic analysis and the code generation were developed in Java and both go through the AST obtained in the syntactic analysis. Still some libraries of the Java API had been implemented in C++. Finally a case study implemented in Java to test the compiler is shown.

Key-words: Palm OS. Java. C++. Compiler.

LISTA DE ILUSTRAÇÕES

LISTA DE SIGLAS

API – *Application Programming Interface*

ASCII – *American Standard Code for Information Interchange*

AST – *Abstract Syntax Tree*

AWT – *Abstract Window Toolkit*

BNF – *Backus-Naur Form*

BSD – *Berkeley Software Distribution*

CLR - *Common Language Runtime*

DOL – *DObject Library*

EA – Enterprise Architect

EBCDIC – *Extended Binary Coded Decimal Interchange Code*

EBNF – *Extended Backus-Naur Form*

GC – *Garbage Colletion*

GLC – Gramática Livre de Contexto

J2SE – Java 2 Standard Edition

Java EE – Java Platform Enterprise Edition

Java ME – Java Platform Micro Edition

Java SE – Java Platform Standard Edition

JavaCC – Java Compiler Compiler

JDBC – *Java Database Connectivity*

JDK – Java Development Kit

JRE – Java SE Runtime Environment

JVM – Java Virtual Machine

LALR – *Look Ahead Left to right Rightmost derivation*

LL – *Left to right Leftmost*

LP – Linguagem de Programação

LR – *Left to right Rightmost*

MSIL - *Microsoft Intermediate Language*

PDA – *Personal Digital Assistant*

PODS - Palm OS Developer Suite

RF – Requisito Funcional

RMI – Remote Method Invocation

RNF – Requisito Não Funcional

SO – Sistema Operacional

STL – *Standard Type Library*

UIAS – *User Interface Application Shell*

UML – *Unified Modeling Language*

LISTA DE SÍMBOLOS

K - kilobyte

SUMÁRIO

1 INTRODUÇÃO

Bachmann e Foster (2005, p. XXIII) afirmam que a conveniência, o poder e a facilidade de utilização de *handhelds* com Sistema Operacional (SO) Palm OS tornam os mesmos atrativos para uma grande variedade de usuários. Mais de 35 milhões de *handhelds* com Palm OS encontraram seu lugar nos bolsos da camisa dos médicos, dos advogados, do pessoal de vendas, dos profissionais de negócio e dos outros segmentos da sociedade não acostumados a utilizar este tipo de equipamento eletrônico. Com mais de 320.000 desenvolvedores registrados e 20.000 aplicativos de terceiros, o Palm OS também provou ser popular entre os desenvolvedores de software.

As Linguagens de Programação (LPs) nativas do Palm OS são o C e C++. A Access, empresa que mantém o Palm OS, disponibiliza um ambiente de programação compatível com estas linguagens. No entanto, apesar de C e C++ serem LPs de alto nível e C++ trabalhar com o paradigma de orientação a objetos, desenvolver aplicativos para Palm OS utilizando estas linguagens pode ser uma tarefa complicada. Segundo Sebesta (2000, p. 89), a linguagem C não possui verificação de tipos, tornando-a flexível e ao mesmo tempo insegura. Além disso, não possui o conceito de orientação a objetos, conceito este que segundo Varejão (2004, p. 18), torna a linguagem mais rápida e confiável para o desenvolvimento de sistemas. Varejão (2004, p. 22) afirma ainda que C++ foi projetada para ser uma extensão de C com orientação a objetos, mas tornou-se uma LP muito complexa.

Como alternativa, pode-se desenvolver aplicativos para Palm OS utilizando o Java Micro Edition (Java ME) que é uma versão reduzida do Java Standard Edition (Java SE), voltada para dispositivos móveis. Conforme Wilding-McBride (2003, p. 1), o Java ME é dividido em configurações, perfis e pacotes opcionais. Esta divisão permite que o Java ME possa ser suportado por diferentes tipos de dispositivos móveis, com diferentes características e limitações. Desta forma, a máquina virtual de cada aparelho pode implementar o que for possível para sua capacidade. Porém, a máquina virtual disponível para o Palm OS não implementa todas as configurações possíveis do Java ME, apesar de existirem poderosos *palmtops* com o SO, alguns possuindo inclusive capacidade de processamento superior aos primeiros computadores que executaram Java SE.

Neste sentido, é proposto o desenvolvimento de um compilador para Palm OS para uma linguagem robusta e ao mesmo tempo simples, com orientação a objetos, que aproveite os recursos nativos do SO, tenha boa *performance* e seja produtiva. Varejão (2004, p. 22)

afirma que Java é uma LP fortemente baseada em C++, mas é bem mais simples. É orientada a objetos, não utiliza explicitamente o conceito de ponteiros e tem se tornado amplamente utilizada por causa da sua confiabilidade e portabilidade. Desta maneira, utilizar a LP Java 5.0 para desenvolver programas nativos para Palm OS demonstra ser uma solução viável e adequada, podendo tornar prático e produtivo o desenvolvimento de aplicativos para Palm OS, tirando do desenvolvedor uma série de formalismos necessários para desenvolver um programa nativo em C ou C++ e oferecendo a segurança e a confiabilidade do Java.

Para tornar mais simples a geração de código executável nativo, o compilador deve gerar código intermediário C++. Este código intermediário poderá ser facilmente portado para outras arquiteturas. Considerando que a maioria dos computadores possui compilador C++ para sua arquitetura, será possível, com pequenas implementações, a compilação de programas Java para execução nativa em outros computadores.

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é desenvolver um tradutor para linguagem Java 5.0 que gere código C++ para Palm OS.

Os objetivos específicos são:

a) fazer as análises léxica, sintática e semântica dos programas Java, mostrando erros de compilação, inclusive referentes à utilização de recursos e bibliotecas não suportadas;
b) traduzir os programas Java para código intermediário em C++ e, a partir dele, gerar código executável para Palm OS;
c) implementar em C++ algumas bibliotecas que fazem parte da API do Java, incluindo bibliotecas de componentes visuais (`java.util, java.awt`) para serem utilizadas nos programas Java.

1.2 ESTRUTURA DO TRABALHO

Este trabalho está dividido em quatro capítulos. O segundo capítulo apresenta os

assuntos estudados durante o desenvolvimento do trabalho. Nele, de forma geral, são discutidos paradigmas e critérios de avaliação de linguagens de programação e, em específico, sobre Java e C++, sobre compiladores e ainda sobre a plataforma Palm OS.

No terceiro capítulo é apresentado o desenvolvimento do trabalho, iniciando pelos requisitos que o compilador deve atender e seguido pela especificação do mesmo. Então é feita uma apresentação da implementação, incluindo um estudo de caso, e logo em seguida são discutidos os resultados do desenvolvimento.

Por fim, o quarto capítulo apresenta as conclusões e sugestões para extensões em trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

A implementação de um compilador requer o estudo de vários conceitos e técnicas acerca deste assunto, sendo necessário conhecer a linguagem que se pretende compilar e o código alvo que se pretende gerar. Assim, neste capítulo são apresentados conceitos, técnicas e ferramentas utilizados no desenvolvimento deste trabalho, dentre eles: linguagens de programação, plataforma Java, linguagem de programação C++, compiladores e a plataforma Palm OS. Na última seção são descritos alguns trabalhos correlatos.

2.1 LINGUAGENS DE PROGRAMAÇÃO

As LPs foram criadas para tornar mais produtivo o trabalho dos programadores. Logo, em última instância, o objetivo das LPs é tornar mais efetivo o processo de desenvolvimento de software (VAREJÃO, 2004, p. 3). Segundo Varejão (2004, p. 17-19), as linguagens podem ser classificadas em diferentes grupos de características, denominados paradigmas. Os paradigmas dividem-se em duas categorias principais:

a) imperativo: engloba as linguagens fundamentadas na idéia de computação como um processo que realiza mudanças de estados. Nesse sentido, um estado representa uma configuração qualquer da memória do computador. Assim, os programas especificam como o processamento deve ser feito no computador. Os conceitos fundamentais são de variável, valor e atribuição. É subdividido em três outros paradigmas:
   - estruturado: baseia-se na idéia de desenvolvimento de programas por refinamentos sucessivos. Organiza o fluxo de controle de execução dos programas desestimulando o uso de comandos de desvio incondicional e incentivando a divisão dos programas em subprogramas e em blocos aninhados de comandos. Pascal e C são exemplos de linguagens que adotam esse paradigma,
   - orientado a objetos: oferece conceitos com o objetivo de tornar mais rápido e confiável o desenvolvimento de aplicações. Baseia-se em classes, que são abstrações que definem uma estrutura de dados e um conjunto de operações

que podem ser realizadas sobre elas, permitindo a implementação natural de modularização em unidades que são fáceis de integrar e estender. Outros conceitos importantes são herança e polimorfismo. Smalltalk, C++ e Java são linguagens que suportam esse paradigma,

- concorrente: fundamenta-se na execução de vários processos simultaneamente que concorrem por recursos. Os processos podem estar em um único computador ou distribuídos em vários, compartilhando dados ou dispositivos. Ada e Java são as linguagens mais conhecidas que suportam esse paradigma;

b) declarativo: engloba as linguagens que descrevem de forma abstrata a tarefa a ser resolvida, sem a existência de atribuições de valores a variáveis, uma vez que as variáveis são incógnitas e não unidades de memória. É dividido em dois outros paradigmas:

- funcional: tem o objetivo de definir uma função que retorne um valor como resposta do problema. Um programa funcional é uma função que chama outras funções, podendo fazer chamadas recursivas e passar funções como parâmetro. Lisp e Haskell são exemplos de linguagens funcionais,
- lógico: baseia-se em cálculo de predicados, definido por uma relação entre constantes ou variáveis. Um programa lógico é composto por cláusulas que definem predicados e relações factuais. Utiliza um mecanismo de inferência para deduzir novos fatos para verificar a veracidade das questões. Prolog é o exemplo mais conhecido de linguagem lógica.

Sebesta (2000, p. 24-35) propõe um conjunto de critérios para se avaliar uma LP, dentre os quais:

a) legibilidade: diz respeito a facilidade com que os programas podem ser lidos ou entendidos. Ela determina grande parte da facilidade de manutenção em programas, por isso tornou-se uma importante medida de qualidade dos programas e linguagens. As seguintes características contribuem para legibilidade das linguagens:

- simplicidade global: é importante que a linguagem seja simples para ser entendida mais facilmente. A possibilidade de realizar a mesma operação de mais de uma maneira e a sobrecarga de operadores são fatores que tornam a linguagem mais complexa,
- ortogonalidade: está ligado à validade das regras da linguagem independentemente do contexto. Quando uma operação ou instrução da

linguagem se comporta sempre da mesma maneira independente do contexto, fica mais fácil ler e entender programas escritos nessa linguagem;

b) redigibilidade: é a medida da facilidade de escrever um programa em uma linguagem. A maioria das características que afeta a legibilidade também afeta esta característica, isto porque é comum o programador precisar reler o que já foi escrito durante o desenvolvimento de um programa. Os seguintes fatores possuem grande influência na redigibilidade:
   - simplicidade e ortogonalidade: o grande número de diferentes construções de uma linguagem pode levar ao uso inadequado de algumas delas ou o desuso por falta de conhecimento destas construções. Acidentalmente, pode-se obter resultados inesperados pela combinação de recursos que possuam seu comportamento alterado pela forma ou seqüência com que foram utilizados,
   - suporte a abstração: abstração significa a capacidade de definir e usar estruturas ou operações complicadas de maneira que se possa ignorar muitos dos detalhes. A abstração pode ser de processo, por exemplo, um subprograma que ordena uma lista; ou de dados, por exemplo, uma estrutura de árvore binária;

c) confiabilidade: um programa é considerado confiável quando ele se comporta de acordo com suas especificações sob todas as condições. Os seguintes recursos de linguagem contribuem para a confiabilidade de programas:
   - verificação de tipos: verificar a compatibilidade dos tipos consiste em garantir que as variáveis recebam apenas valores compatíveis com seus tipos. É desejável que esta verificação ocorra em tempo de compilação,
   - manipulação de exceções: consiste em interceptar erros de um programa em tempo de execução, podendo fazer correções e prosseguir com a execução,
   - legibilidade e redigibilidade: quanto mais fácil for escrever um programa, menor será a probabilidade dele possuir erros. E quanto mais fácil for de entender o programa, mais fácil será dar manutenção nele sem criar erros;

d) custo: o custo final de uma LP é influenciado por muitas de suas características, que podem ser divididas nos seguintes custos:
   - custo do treinamento dos programadores para usar a linguagem: depende da simplicidade e ortogonalidade da linguagem e da experiência dos programadores,
   - custo para escrever programas na linguagem: depende da redigibilidade,

- custo para compilar programas na linguagem: alguns compiladores mais antigos exigiam muitos recursos de hardware para compilar,
- custo para executar programas escritos na linguagem: está ligado ao desempenho dos programas,
- custo das ferramentas necessárias para se escrever programas na linguagem: se as ferramentas necessárias para utilizar a linguagem forem de custo elevado, certamente a linguagem será pouco aceita no mercado. Isto explica a rápida aceitação do Java, que desde o início teve seus compiladores distribuídos sem custo algum,
- custo da má confiabilidade: falhas em sistemas críticos, como em uma usina nuclear, podem elevar muito o custo, ou mesmo em sistemas não-críticos, que podem receber ações judiciais em função do software defeituoso.

2.2 PLATAFORMA JAVA

Java é o nome dado ao conjunto de tecnologias e ferramentas desenvolvidas pela Sun Microsystems para o desenvolvimento de aplicativos para as mais diversas áreas de aplicação. Este conjunto de tecnologias que formam o Java é dividido em três plataformas (SUN MICROSYSTEMS, 2007a):

a) Java Platform Micro Edition (Java ME): para o desenvolvimento aplicativos para dispositivos móveis e sistemas embarcados;
b) Java Platform Standard Edition (Java SE): para o desenvolvimento de aplicativos para computadores *desktop*;
c) Java Platform Enterprise Edition (Java EE): para o desenvolvimento de aplicativos empresariais de grande porte.

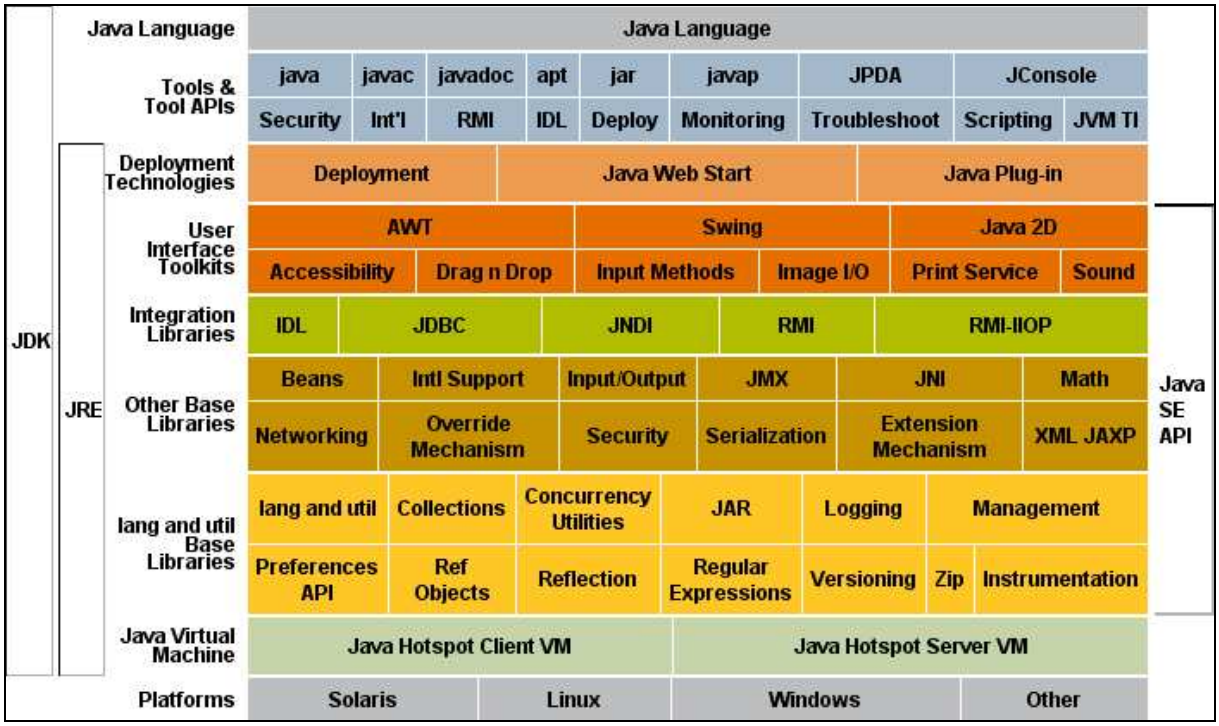
A plataforma utilizada no desenvolvimento do presente trabalho é o Java SE, por isto apenas ela é detalhada neste capítulo. O Java SE é composto por vários componentes que, segundo Sun Microsystems (2007b), podem ser divididos em dois produtos principais:

a) Java SE *Runtime Environment* (JRE): composto pelas bibliotecas, máquina virtual Java e outros componentes para executar *applets* e aplicativos escritos na linguagem de programação Java. Adicionalmente, duas tecnologias chaves para distribuição fazem parte do JRE: Java *Plug-in*, que permite executar *applets* nos

navegadores de internet; e Java *Web Start*, que permite a distribuição de aplicativos pela rede;

b) Java *Development Kit* (JDK): contém, além dos componentes do JRE, ferramentas como compiladores e depuradores, necessárias para o desenvolvimento de *applets* e aplicativos.

A Figura 1 mostra uma visão geral dos componentes que formam a plataforma.



Fonte: Sun Microsystems (2007b).

Figura 1 – Tecnologias da plataforma Java

2.2.1 Breve história

Java teve seu início em 1991, segundo Deitel e Deitel (2003, p. 59), num projeto interno de pesquisa corporativa financiado pela Sun Microsystems, que apostava no futuro do mercado de dispositivos eletrônicos inteligentes. O projeto tinha o codinome *Green* e resultou no desenvolvimento de uma linguagem baseada em C e C++, chamada pelo seu criador, James Gosling, de *Oak* em homenagem a uma árvore de carvalho vista da sua janela na Sun. Posteriormente, descobriu-se que já existia uma linguagem de programação com este nome. Então, quando a equipe da Sun visitou uma cafeteria local, o nome Java (cidade de origem de um café importado) foi sugerido.

Mas o mercado de dispositivos eletrônicos inteligentes não se desenvolveu como o

esperado pela Sun. E o projeto *Green* corria risco de ser cancelado. Por sorte, em 1993 a *World Wide Web* explodiu em popularidade e a equipe da Sun logo percebeu o potencial de utilizar o Java para adicionar conteúdo dinâmico às páginas da web. Isso revigorou o projeto.

A linguagem Java foi anunciada oficialmente pela Sun apenas em maio 1995, em uma importante conferência. Java chamou atenção da comunidade de negócios com interesse na *World Wide Web*. Atualmente, é amplamente utilizada para desenvolver tanto aplicativos corporativos de grande porte como aplicativos para dispositivos móveis.

2.2.2 Linguagem de programação Java

A linguagem de programação Java é uma linguagem de uso geral, concorrente, orientada a objetos. Ela foi desenvolvida para ser simples o bastante para que muitos programadores possam alcançar fluência na linguagem. Java se relaciona com C e C++, mas é organizada diferentemente, com alguns aspectos de C e de C++ omitidos e algumas características de outras linguagens incluídas, explica Gosling et al (2005, p. 1).

Gosling et al (2005, p. 1) afirma que Java é uma linguagem relativamente de alto nível, onde detalhes da representação da máquina não são disponíveis através da linguagem. Ela inclui gerenciamento automático de memória, tipicamente usando um coletor de lixo, para evitar os problemas de segurança de desalocação explícita (como o `free` em C ou o `delete` em C++), e não inclui qualquer construção insegura, como acesso a `array` sem checagem de índice.

Os programas escritos em Java normalmente são compilados para um conjunto de instruções binárias denominadas *bytecode*. A Figura 2 mostra a tela de um programa `Hello World` em Java, cujo código fonte é mostrado no Quadro 1.

Figura 2 – Tela de um programa `Hello World` em Java

```
import java.awt.Button;
import java.awt.Frame;

public class HelloWorld extends Frame {

    public HelloWorld() {
        super("Hello World");
        setSize(320, 320);
        setLayout(null);
        Button button = new Button("OK");
        button.setBounds(122, 147, 76, 26);
        add(button);
    }

    public static void main(String[] args) {
        new HelloWorld().setVisible(true);
    }
}
```

Quadro 1 – Código fonte de um programa `Hello World` em Java

2.2.3 Máquina Virtual Java

A Máquina Virtual Java (Java Virtual Machine - JVM) é a pedra fundamental da plataforma Java. Ela é o componente da tecnologia responsável pela independência de hardware e sistema operacional, pelo pequeno tamanho do código compilado e pela habilidade de proteger os usuários de programas provenientes de fontes não confiáveis ou mal-intencionados (SUN MICROSYSTEMS, 1999). A JVM é uma máquina computacional abstrata. Como uma máquina computacional real, ela possui um conjunto de instruções que manipulam várias áreas de memória em tempo de execução.

O primeiro protótipo de implementação da JVM, feito na Sun Microsystems, emulou o conjunto de instruções da JVM em um software hospedado em um dispositivo de mão que lembra um *Personal Digital Assistant* (PDA) de hoje em dia. A implementação atual da JVM emula a JVM em vários sistemas operacionais e arquiteturas de maneira muito mais sofisticada.

A JVM não conhece a linguagem de programação Java, apenas o formato binário do arquivo `class`, produto da compilação de um arquivo fonte Java. Um arquivo `class` contém instruções da JVM (ou *bytecodes*) e uma tabela de símbolos, assim como outras informações auxiliares. Conforme descrito pela Sun Microsystems (1999) no documento de especificação da JVM, para garantir a segurança, a JVM impõe um formato forte e consistências estruturais no código do arquivo `class`. Porém, qualquer linguagem com funcionalidades que possam ser expressas em termos de um arquivo `class` válido, pode ser instalada na máquina virtual.

2.2.4 Coletor de lixo

Segundo Gosling e McGilton (1996), gerenciamento explícito de memória na implementação de aplicativos tem provado ser uma das maiores fontes de erros, vazamentos de memória[1] e baixa *performance*. A tecnologia Java remove completamente dos programadores a responsabilidade de gerenciar a memória, ou seja, não há comando para liberar memória explicitamente. Para tanto, a JVM disponibiliza um componente muito importante presente em tempo de execução, denominado coletor de lixo (*Garbage Colletion* - GC). O GC monitora os objetos instanciados durante a execução dos programas e caso um objeto não seja mais utilizado, libera automaticamente a memória reservada para ele, permitindo que seja utilizada futuramente.

O modelo de gerenciamento de memória da tecnologia Java é baseado em objetos e referências para objetos, não havendo ponteiros diretos para endereços de memória. O gerenciador de memória monitora todas as referências para objetos. Quando um objeto não possuir mais referências ou for referenciado apenas por objetos candidatos a coleta, este objeto passa a ser candidato para coleta de lixo. Isto porque a coleta efetiva do lixo ocorre apenas quando há ociosidade na execução ou quando não houver mais memória disponível para novos objetos. Isto traz benefícios ao desempenho, pois o tempo gasto com a liberação de memória ocorre apenas se houver ociosidade ou for extremamente necessário.

2.2.5 API

Como pôde ser visto na Figura 1, a plataforma Java possui uma API muito rica em utilitários e estruturas de dados para as mais diversas aplicações. Alguns exemplos de componentes da API:

a) *Abstract Window Toolkit* (AWT): composta por componentes para criação de interface com o usuário, possui vários componentes nativos, robusto modelo de tratamento de eventos, gerenciadores de *layout* e acesso à área de transferência da plataforma (SUN MICROSYSTEMS, 2005a);
b) *Java Database Connectivity* (JDBC): provê acesso universal a dados. É possível

[1] "[...] fenômeno que ocorre em sistemas computacionais quando uma porção de memória precisa ser alocada para uma determinada operação, mas não é liberada depois" (MEMORY..., 2007).

acessar qualquer fonte de dados, de banco de dados relacionais a planilhas de cálculos. Para acessar uma fonte de dados é necessário um JDBC-*driver*, que geralmente é distribuído gratuitamente pelo fornecedor da fonte de dados (SUN MICROSYSTEMS, 2002);

c) *Collections Framework*: estrutura unificada para representação e manipulação de coleções, permitindo que sejam manipuladas independentemente dos detalhes de sua representação. Sua utilização reduz o esforço de programação e promove a reusabilidade. É baseada em quatorze interfaces de coleções, incluindo implementações e algoritmos para manipulá-las (SUN MICROSYSTEMS, 2006).

2.2.6 Novidades na versão 5.0

A versão 5.0 do Java, que possui este nome em homenagem aos cinco anos da segunda geração da plataforma, trouxe alguns recursos interessantes à plataforma e à linguagem (AUSTIN, 2004). Conforme Sun Microsystems (2005b), os principais recursos são:

a) tipos genéricos: permite que um tipo ou um método opere sobre objetos de vários tipos, fornecendo checagem de tipo em tempo de compilação. A estrutura de coleções foi atualizada para utilizar este recurso, o que elimina a utilização excessiva de *casts*;
b) *loop* `for` estendido: simplifica as iterações sobre coleções e `arrays`, eliminando a necessidade de declarar iteradores ou acessar o índice;
c) *autoboxing/unboxing*: elimina a conversão manual entre tipos primitivos (como o `int`) e tipos encapsuladores de primitivos (como o `Integer`);
d) tipos enumerados: permite a criação de tipos enumerados seguros orientados a objeto com métodos e atributos;
e) lista de parâmetros variáveis: permite que um método tenha uma quantidade dinâmica de parâmetros de um determinado tipo;
f) *import* estático: facilita o acesso a membros estáticos de classes, eliminando a necessidade de repetir o nome da classe sempre que um membro for acessado;
g) anotações (*metadata*): permite que informações sejam adicionadas ao programa compilado para consulta em tempo de execução. Também permite a adição de marcações ao código fonte que podem ser interpretadas pelo compilador ou por outros programas, como por exemplo, um aviso sobre a utilização de uma

biblioteca descontinuada.

2.3 LINGUAGEM DE PROGRAMAÇÃO C++

C++ é uma linguagem de programação que evoluiu do C, que por sua vez, evoluiu de duas linguagens de programação anteriores, BCPL e B. BCPL foi desenvolvido em 1967 por Martin Richards como uma linguagem para escrever sistemas operacionais e compiladores. Ken Thompson desenvolveu muitas características em sua linguagem B com base em conceitos contrários aos presentes no BCPL, e usou B para criar versões anteriores do sistema operacional UNIX em 1970 na Bell Laboratories. Tanto B quanto BCPL, eram linguagens sem “tipo”, onde cada item de dado ocupava uma “palavra” na memória.

Segundo Deitel e Deitel (1997, p. 9-10), C foi desenvolvido por Dennis Ritchie na Bell Laboratories em 1972 como uma evolução do B e usando muitos conceitos do BCPL. Adiciona suporte a tipos de dados e outras características. Inicialmente C se tornou muito conhecido como a linguagem de desenvolvimento do sistema operacional UNIX. Hoje a maior parte dos sistemas operacionais são escritos em C ou C++. Com o passar do tempo, C se tornou disponível para a maioria dos computadores, por ser independente de hardware, e até portável, com certos cuidados durante o desenvolvimento.

A variedade de hardwares suportada pela linguagem C acabou gerando certas variações, dificultando o desenvolvimento de aplicativos portáveis. Isso levou a sua padronização, que foi aprovada 1989 e publicada em 1990 como ANSI/ISO 9899: 1990.

C++ foi desenvolvido como uma extensão do C em meados de 1980, por Bjarne Stroustrup, na Bell Laboratories. C++ trouxe várias melhorias, mas a mais significante delas foi o suporte a programação orientada a objetos. Outra grande vantagem foi a *Standard Type Library* (STL), uma biblioteca padrão com diversas classes e funções que podem ser utilizadas nos programas.

A linguagem C++ não é totalmente orientada a objetos, ela é considerada uma linguagem híbrida, pois permite programação estruturada no estilo C, diferente de Smalltalk, que é totalmente orientado a objetos.

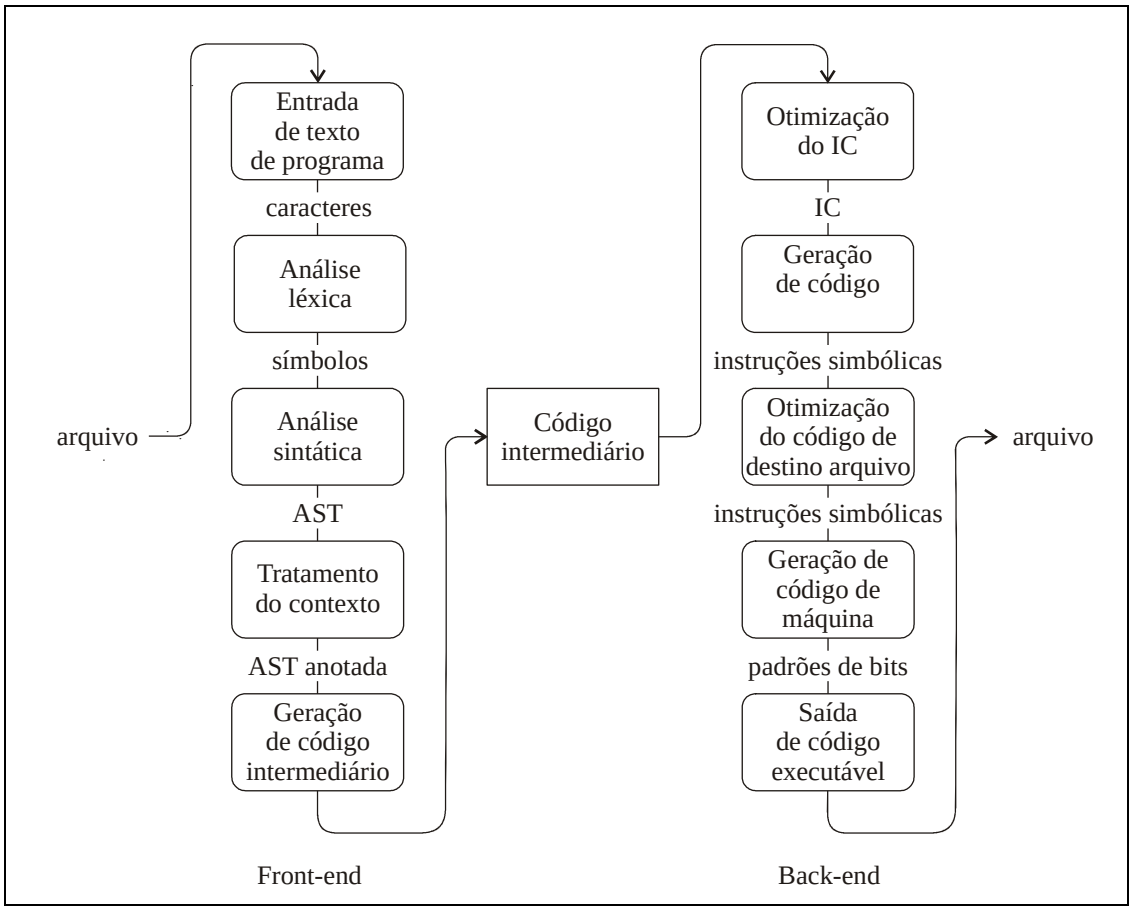
2.4 COMPILADORES

> Um compilador é um programa que lê um programa escrito numa linguagem – a linguagem *fonte* – e o traduz num programa equivalente numa outra linguagem – a linguagem *alvo*. Como importante parte desse processo de tradução, o compilador relata a seu usuário a presença de erros no programa fonte. (AHO; SETHI; ULMAN, 1995, p. 1, grifos dos autores).

Segundo Grune et al (2001, p. 2), a compilação não difere fundamentalmente da conversão de arquivos, embora seja diferente em grau. Um aspecto claro da compilação é que a entrada tem uma propriedade chamada semântica, ou seja, possui um significado. Essa semântica deve ser respeitada pelo processo de conversão e é menos claramente identificável em um programa de conversão de arquivos tradicional, por exemplo um programa que converta *Extended Binary Coded Decimal Interchange Code* (EBCDIC) para *American Standard Code for Information Interchange* (ASCII). Porém, um conversor de GIF para JPEG tem que preservar a impressão visual da figura, o que poderia de algum modo ser chamado de semântica. Então, um compilador é apenas um programa de grande porte para conversão de arquivos.

Um compilador é dividido internamente em fases, com operações lógicas distintas, explica Louden (2004, p. 6-7). Essas fases podem ser consideradas peças separadas, que poderiam inclusive ser escritas independentemente, embora geralmente sejam agrupadas. As fases são: analisador léxico, analisador sintático, analisador semântico, otimizador de código-fonte, gerador de código e otimizador de código-alvo.

Grune et al (2001, p. 2) por sua vez, decompõe o processo de compilação em duas grandes partes: o *front-end* e o *back-end*. O *front-end* é responsável por executar a análise do texto da linguagem fonte, e o *back-end* possui a responsabilidade de fazer a síntese da linguagem alvo. Essas duas partes são subdivididas em vários módulos, apresentados na Figura 3.



Fonte: Grune et al (2001, p. 20).

Figura 3 – Estrutura de um compilador

Como o compilador desenvolvido neste trabalho tem como código alvo C++, apenas os módulos do *front-end* são relevantes, sendo que os módulos do *back-end* são responsabilidade do compilador C++.

2.4.1 Análise léxica

A análise léxica, primeira etapa do processo de compilação, tem como entrada o programa fonte, representado por seqüências de caracteres. Essas seqüências de caracteres são separadas em unidades de informação denominadas *token*. Segundo Louden (2004, p. 32), *tokens* são entidades lógicas geralmente definidas como um tipo enumerado. O Quadro 2 mostra um exemplo de como os tipos de *token* poderiam ser definidos em Java.

```
enum TokenType { IF, THEN, ELSE, PLUS, MINUS, NUM, ID,... }
```

Quadro 2 – Exemplo de tipos de *token* em Java

Existem diversas categorias de *tokens*. Entre elas, as palavras reservadas, como `IF` e

`THEN`, representam as cadeias de caracteres “if” e “then”. Outra categoria é a de símbolos especiais, como os símbolos aritméticos `PLUS` e `MINUS`, que representam os caracteres “+” e “-”. Por fim, existem os *tokens* para representar cadeias múltiplas de caracteres. Alguns exemplos são `NUM` e `ID`, que representam números e identificadores.

Grune et al (2001, p. 54) explica que as formas dos símbolos de uma linguagem podem ser expressas como expressões regulares. Uma expressão regular é uma notação prática para expressar um conjunto de cadeias de caracteres de símbolos terminais (WATT; BROWN, 2000, p. 77). Então, os formatos dos *tokens* de uma linguagem podem ser formalmente descritos como expressões regulares. O Quadro 3 mostra alguns exemplos.

```
letra = [a-zA-Z]
digito = [0-9]
identificador = letra (letra | digito)*
```

Quadro 3 – Exemplos de expressões regulares

2.4.2 Análise sintática

A tarefa da análise sintática é identificar as estruturas sintáticas do programa a partir da seqüência de símbolos gerada pelo analisador léxico (WILHELM; MAURER, 1995, p. 268). A sintaxe de uma LP é normalmente definida pelas regras gramaticais de uma Gramática Livre de Contexto (GLC), semelhante à forma como a estrutura léxica é definida por expressões regulares. Assim como as expressões regulares, as regras gramaticais são definidas sobre um alfabeto, ou conjunto de símbolos. Nas expressões regulares esses símbolos normalmente são caracteres, já no caso das regras gramaticais, o alfabeto é formado pelo conjunto de *tokens* provenientes da definição léxica da linguagem (LOUDEN, 2004, p. 95-98).

As regras gramaticais de uma GLC podem ser descritas pela notação *Backus-Naur Form* (BNF). O Quadro 4 apresenta uma BNF que define um programa composto por uma lista de comandos e menciona o *token* `identificador`, que foi descrito no Quadro 3 por uma expressão regular. Cada linha do quadro representa uma regra gramatical.

```
<programa> ::= { <lista-commando> }
<lista-commando> ::= <comando>
                   | <comando> , <lista-commando>
<comando> ::= ler identificador
            | escrever identificador
```

Quadro 4 – Exemplo de BNF

A notação BNF permite especificar qualquer gramática, porém, não é muito conveniente para expressar repetições e opcionalidades, embora seja possível expressar a repetição através da recursão. Para resolver este problema, foi desenvolvida uma extensão à BNF, chamada de *Extended* BNF (EBNF) (GRUNE et al, 2001, p. 33-34). Segundo Watt e Brown (2000, p. 79), a EBNF pode ser considerada uma combinação da BNF com expressões regulares. O Quadro 5 mostra a BNF do Quadro 4 definida na notação EBNF.

```
<programa> ::= { (<comando>)+ }
<comando> ::= (ler | escrever) identificador
```

Quadro 5 – Exemplo de EBNF

O processo de análise sintática resulta em uma árvore sintática, onde os nós intermediários representam as estruturas sintáticas de um programa refletindo a gramática da linguagem, e os nós folha representam os *tokens* provenientes da análise léxica. Esta árvore é utilizada nos passos seguintes de um compilador. Entretanto, nem todas as informações que ela contém são necessárias. Então a saída real da análise sintática é uma árvore mais eficiente, denominada AST ou árvore sintática abstrata (GRUNE et al, 2001, p. 48-51).

Para realizar a análise sintática existem duas estratégias: a *top-down*, com o método *Left to right Leftmost derivation* (LL); e a *bottom-up*, com os métodos *Left to right Rightmost derivation* (LR) e *Look Ahead Left to right Rightmost derivation (*LALR). Estas estratégias são caracterizadas pela ordem em que os *tokens* são consumidos para construção da árvore sintática (WATT; BROWN, 2000, p. 83).

A estratégia de análise sintática *top-down* constrói a árvore sintática a partir do nó superior, em pré-ordem, ou seja, os nós superiores sempre são criados antes de qualquer um de seus nós inferiores. Para a utilização desta estratégia é necessária uma gramática LL, ou seja, uma gramática fatorada à esquerda e sem recursão à esquerda. Uma gramática que não se enquadre nesses critérios pode ser convertida para tal, porém pode ficar menos legível e deixar de construir a árvore sintática correta. A estratégia *bottom-up* constrói a árvore sintática em *pós-ordem*, ou seja, inicia pelos nós inferiores. Os nós superiores apenas são criados quando todos os seus nós inferiores já tenham sido. Os métodos LR e LALR para esta estratégia possuem a vantagem de suportarem gramáticas com recursão à esquerda, porém são mais complexos e de difícil entendimento (GRUNE et al, 2001, p. 104-160).

2.4.3 Análise semântica

O propósito da análise semântica é verificar se um programa está de acordo com as restrições contextuais da linguagem. Para uma LP típica, as restrições contextuais consistem em: identificação, que aplica as regras de escopo da linguagem para relacionar cada ocorrência de um identificador a sua declaração; e checagem de tipos, que aplica as regras de tipos da linguagem para inferir o tipo de cada expressão e comparar esse tipo com o tipo esperado (WATT; BROWN, 2000, p. 136).

Na análise semântica não existe um método padrão para especificar a semântica, ao contrário da análise sintática, que possui a BNF para especificar a sintaxe. Isto ocorre em parte porque a quantidade e tipos de análise semântica variam muito de uma linguagem para outra. Um método freqüentemente utilizado pelos desenvolvedores de compiladores para descrever a análise semântica é identificar atributos, ou propriedades, de entidades da linguagem e escrever regras semânticas, que expressem como esses atributos se relacionam com as regras gramaticais da linguagem. Este conjunto de atributos é denominado gramática de atributos. São mais utilizados em linguagens que seguem o princípio de semântica dirigida pela sintaxe, onde o conteúdo semântico está fortemente relacionado com a sintaxe (LOUDEN, 2004, p. 260).

Se a análise semântica puder começar depois que toda a análise sintática tiver terminado e a AST estiver construída, a tarefa de implementar a análise semântica se torna consideravelmente mais fácil, resumindo-se a percorrer os nós da AST e aplicar as regras semânticas definidas pela gramática de atributos. Isto faz com que o compilador seja classificado como sendo de múltiplas passadas. Porém, se for necessário efetuar todas as operações (inclusive a geração de código) em uma única passada, a implementação da análise semântica torna-se um processo *ad hoc* de encontrar uma ordem certa e um método adequado para computar as informações semânticas (LOUDEN, 2004, p. 260).

A abordagem utilizada neste trabalho foi a de múltiplas passadas. Então, conforme Watt e Brown (2000, p. 153), a identificação e a checagem de tipos ocorrem com base na AST obtida na análise sintática. Para cada uma destas etapas, é executada uma busca em profundidade na AST, decorando os nós visitados com informações contextuais (WATT; BROWN, 2000, p. 153). Deste modo, o produto da análise semântica é uma AST anotada com informações referentes ao contexto.

2.4.4 Geração de código intermediário

A geração de código se concentra na tradução do programa fonte em código objeto, e depende tanto da linguagem origem quanto da máquina alvo (abstrata ou real) (WATT; BROWN, 2000, p. 250).

A AST proveniente da análise semântica é composta por nós que refletem os conceitos específicos da linguagem fonte. A geração de código intermediário serve para reduzir a quantidade de informações específicas da linguagem a um conjunto reduzido de conceitos gerais que podem ser mais facilmente implementados em máquinas reais ou abstratas. Como a linguagem fonte é de mais alto nível que a linguagem de máquina, a geração de código intermediário normalmente aumenta o tamanho da AST, pois as instruções intermediárias são de mais baixo nível. Por outro lado, a complexidade conceitual é reduzida (GRUNE et al, 2001, p. 254-255).

2.4.5 Geradores automatizados de compiladores

Atualmente existem várias ferramentas que automatizam a construção de um compilador, ou pelo menos parte da construção. Os analisadores léxicos e sintáticos geralmente são gerados com ferramentas chamadas geradores de *parser* ou *compiler-compiler*. Estas ferramentas normalmente têm como entrada as expressões regulares que definem os *tokens* da linguagem para gerar o analisador léxico, e a BNF da linguagem para gerar o analisador sintático. Estas entradas permitem a geração das partes de um compilador de modo muito preciso e eficiente, e possui a vantagem de facilitar a manutenção e o entendimento da sintaxe da linguagem. Existem ainda geradores de analisadores semânticos, porém não são tão eficientes.

Na implementação deste trabalho foi utilizado o gerador de *parser* Java Compiler Compiler (JavaCC). JavaCC é um dos geradores de *parser* mais populares para utilização em aplicações Java. Ele é escrito em Java e gera código puramente Java. Tanto o JavaCC quando os *parsers* por ele gerado podem ser executados em várias plataformas Java.

A analisador sintático gerado pelo JavaCC utiliza o método LL, o que permite a utilização da maioria das gramáticas. As especificações léxicas e sintáticas da LP ficam em um único arquivo. As especificações léxicas são feitas através de expressões regulares e as

definições sintáticas são especificadas utilizando uma mistura de código Java e EBNF. A partir do arquivo de especificação da gramática é possível gerar uma documentação no formato EBNF.

O JavaCC possui muitos recursos que o torna uma ferramenta completa para automatizar uma parte do desenvolvimento de um compilador. Possui ainda uma grande comunidade de usuários, facilitando o encontro de gramáticas, artigos e documentações sobre a ferramenta, que é distribuída gratuitamente sob a licença *Berkeley Software Distribution* (BSD) (JAVACC, 2007).

2.5 PLATAFORMA PALM OS

A plataforma Palm OS foi projetada pela PalmSource, agora conhecida como Access, excepcionalmente para as necessidades do mundo móvel de hoje. Segundo Access (2007), a plataforma teve grande crescimento por causa de seu foco nas necessidades específicas do usuário móvel. Ao invés de tentar colocar todas as características e capacidades de um computador pessoal em um pacote minúsculo, os dispositivos Palm OS são projetados especialmente para gerenciar informação, comunicação e entretenimento móvel. Isto dá ao Palm OS grandes vantagens na flexibilidade, na facilidade de utilização e na compatibilidade.

Bachmann e Foster (2005, p. 2) descrevem as principais razões que tornam o Palm OS uma grande plataforma para muitos desenvolvedores e usuários de *handhelds*, que são:

a) Palm OS é pequeno, rápido, e eficiente: o Palm OS permanece verdadeiro aos seus valores no núcleo e oferece preferivelmente extensibilidade a seus licenciados, permitindo-os construir dispositivos que adicionam características avançadas ao núcleo do sistema operacional;
b) Palm OS é fácil de usar: dispositivos que usam o Palm OS permitem que seus usuários executem tarefas comuns com um mínimo de caixas de diálogos, menus e navegações de telas. Tarefas muito comuns são realizadas com um apertar de botão ou um toque na tela;
c) Palm OS permite sincronização rápida e simples com o *desktop*: o sistema de *HotSync* do Palm OS permite sincronização de dados entre o *desktop* e o *handheld* simplesmente apertando um botão;
d) Palm OS adota a diversidade: o número e a diversidade de licenciados é uma prova

de quão bem os desenvolvedores do Palm OS permitiram aos fabricantes de *handhelds* adaptá-lo a uma grande variedade de tarefas, de multimídia a comunicação sem fio.

2.5.1 Sistema operacional

O Palm OS possui um núcleo multitarefas preemptivo. Porém, o *User Interface Application Shell* (UIAS), parte do SO responsável por gerenciar aplicativos que apresentam interface ao usuário, executa apenas uma aplicação por vez. Geralmente, a única tarefa executada é o UIAS, que chama aplicativos como sub-rotinas. O UIAS não recebe o controle novamente até que a aplicação em execução termine. Neste ponto, o UIAS chama imediatamente o próximo aplicativo como outra sub-rotina. Aplicativos não podem ser *multithread* no Palm OS porque eles devem executar em uma única *thread* do UIAS. Bachmann e Foster (2005, p. 14-15) explicam que existe ainda a possibilidade de um aplicativo sub-executar outro. Uma sub-execução chama outro aplicativo como uma sub-rotina do aplicativo chamador.

Certas chamadas em aplicativos podem fazer com que o SO execute uma nova tarefa. Por exemplo, o aplicativo de *HotSync* inicia uma outra tarefa em adição ao UIAS, que trata a comunicação serial com o computador *desktop*. Porque a tarefa de comunicação serial possui menor prioridade que a tarefa principal de interface ao usuário, o usuário pode interromper a comunicação pressionando um botão na tela ou um botão do hardware. Apenas o SO pode executar uma nova tarefa no Palm OS. Os aplicativos não têm acesso direto às APIs de multitarefas do Palm OS (BACHMANN; FOSTER, 2005, p. 14-15).

A Figura 4 mostra uma tela típica do Palm OS.

Figura 4 – Tela do Palm OS

2.5.2 Programação para Palm OS

Existem muitos ambientes de desenvolvimento para Palm OS. Entre as ferramentas existentes, C e C++ são as LPs mais utilizadas para desenvolver aplicativos para plataforma Palm OS (BACHMANN; FOSTER, 2005, p. 45). Algumas ferramentas trabalham com LPs como Visual Basic, Pascal, Lua e outras.

Os ambientes de programação que utilizam C e C++ mais conhecidos são:

a) Palm OS Developer Suite (PODS): é um pacote de ferramentas integradas baseado no Eclipse 3.0.1 e no *plug-in* C/C++ *Development Toolkit* 2.0.2 (ACCESS, 2005). O PODS é distribuído gratuitamente pela Access;
b) CodeWarrior for Palm OS: é um ambiente de desenvolvimento integrado completo e tem sido por muito tempo o conjunto de desenvolvimento C/C++ comercial mais popular para os desenvolvedores Palm OS. Muitos dos softwares mais bem conhecidos para Palm OS foram escritos usando o CodeWarrior (BACHMANN; FOSTER, 2005, p. 45).

Um aplicativo Palm OS é composto por *resources*, que são blocos que representam dados, código executável, elementos de interface com o usuário e outras peças da aplicação. Em um computador *desktop*, o arquivo que representa um aplicativo possui a extensão `.prc`, e é conhecido como arquivo PRC.

Conforme Bachmann e Foster (2005, p. 16-17), quando o sistema executa um aplicativo, ele chama uma função chamada `PilotMain` (similar à função `main` em um programa C) e passa para ela como parâmetro um código de execução. Um programa pode ser executado com diferentes códigos de execução, sendo os principais:

a) `sysAppLaunchCmdNormalLaunch`: é o código de execução normal, em resposta a ele o aplicativo deve executar normalmente;
b) `sysAppLaunchCmdFind`: pede para o aplicativo buscar um texto em seus dados armazenados;
c) `sysAppLaunchCmdSystemReset`: avisa o aplicativo que o sistema sofrerá um *reset*, e permite que ele execute alguma resposta.

Os aplicativos Palm OS são orientados a eventos. Eles recebem eventos do SO e os tratam ou os devolvem para que sejam tratados pelo próprio SO. Um aplicativo executado com o código de execução normal deve ficar em *loop*, recebendo e tratando eventos de acordo com o tipo, ou ainda despachar o evento para o SO tratar. Um aplicativo padrão permanece

em *loop* até que o sistema envie o evento de fim de execução: `appStopEvent` (BACHMANN; FOSTER, 2005, p. 45).

Os elementos de interface com usuário (formulários, diálogos, botões, listas, etc.) são definidos em um arquivo próprio, e compilados juntamente com os fontes do programa, ficando armazenados em *resource* do arquivo PRC. Existe ainda a possibilidade de criar interfaces dinamicamente através de uma API do SO, porém nem todos os elementos podem ser criados por esta API. O Quadro 6 mostra o código fonte para a definição de uma tela de um programa `Hello World`. A Figura 5 mostra a tela e, logo em seguida, no Quadro 7, o código fonte do programa é apresentado.

```
FORM ID 1000 SAVEBEHIND MODAL
BEGIN
    TITLE "Hello World"
    BUTTON "OK" ID 1001   AT (94 90 56 19)
END
```

Quadro 6 – Código fonte para uma tela de um programa `Hello World` para Palm OS

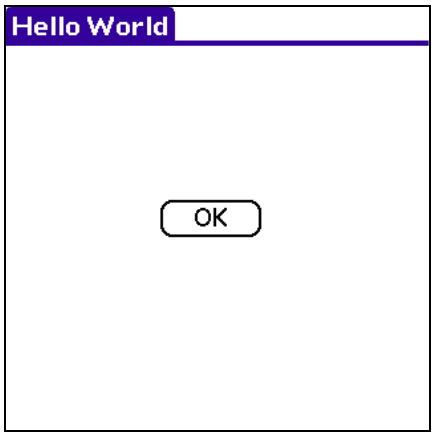


Figura 5 – Tela de um programa `Hello World` para Palm OS

```
#include <PalmOS.h>

UInt32 PilotMain(UInt16 cmd, MemPtr cmdPBP, UInt16 launchFlags) {
	Err error;
	switch (cmd) {
	case sysAppLaunchCmdNormalLaunch:
		error = AppStart();
		if (error) return error;
		FrmGotoForm(MainForm);
		AppEventLoop();

		AppStop();
		break;
	}
	return errNone;
}

static Err AppStart(void) {
	return errNone;
}

static void AppStop(void) {
	FrmCloseAllForms();
}

static void AppEventLoop(void) {
	UInt16 error;
	EventType event;
	do {
		EvtGetEvent(&event, evtWaitForever);
		if (! SysHandleEvent(&event)) {
			if (! MenuHandleEvent(0, &event, &error)) {
				if (! AppHandleEvent(&event)) {
					FrmDispatchEvent(&event);
				}
			}
		}
	} while (event.eType != appStopEvent);
}

static Boolean AppHandleEvent(EventType * eventP) {
	UInt16 formId;
	FormType * frmP;
	if (eventP->eType == frmLoadEvent) {
		formId = eventP->data.frmLoad.formID;
		frmP = FrmInitForm(formId);
		FrmSetActiveForm(frmP);
		switch (formId) {
		case MainForm:
			FrmSetEventHandler(frmP, MainFormHandleEvent);
			break;
		}
		return true;
	}
	return false;
}

static Boolean MainFormHandleEvent(EventType * eventP) {
	Boolean handled = false;
	FormType * frmP;
	switch (eventP->eType) {
	case frmOpenEvent:
		frmP = FrmGetActiveForm();
		FrmDrawForm(frmP);
		MainFormInit(frmP);
		handled = true;
		break;
	}
	return handled;
}
```

Quadro 7 – Código fonte em C de um programa `Hello World` para Palm OS

Para economizar espaço nos programas, o Palm OS utiliza referências de memória de

16 bits, o que limita *jumps* relativos em 32K. Se uma aplicação tentar chamar uma função que está localizada mais de 32K de distância dentro do mesmo *resource* de código, o *jump* vai falhar. Para muitos aplicativos, isto não é um problema porque a maioria consiste em um único *resource* de código com menos de 32K de tamanho. Para aplicações grandes, porém, é necessário estender a distância do processador de *jumps*. Para isto existem algumas técnicas de contorno:

a) alterar a ordem de *link* dos arquivos fonte: isto pode evitar *jumps* maiores que 32K em dois ou mais fontes;
b) rearranjar o código fonte: isto pode evitar *jumps* maiores que 32K em fontes muito grandes;
c) utilizar o modelo de código *smart* no CodeWarrior: isto faz com que o compilador utilize jumps de 32 bits ao invés de jumps de 16 bits para referências fora do alcance. Esta opção pode aumentar consideravelmente o tamanho do executável do programa.

Além desta limitação, a arquitetura de *HotSync* impõe o limite de 64K para qualquer *resource*, incluindo *resources* de código que formam um aplicativo Palm OS. Se for necessário um programa com tamanho maior que 64K, ele deverá ser segmentado em múltiplos *resources* de código (BACHMANN; FOSTER, 2005, p. 827-833).

2.6 TRABALHOS CORRELATOS

Leyendecker (2005) especificou uma linguagem de programação orientada a objetos para a plataforma Microsoft .NET e desenvolveu em C# um compilador para esta linguagem. A linguagem desenvolvida possui sintaxe muito semelhante, embora mais simples, à linguagem C#. Ela tem o objetivo de disponibilizar novas funcionalidades ao C# como expressões relacionais e aritméticas para manipular tipos de data e hora, e intervalo de tempo. O compilador gera código *Microsoft Intermediate Language* (MSIL) que é executado a partir da *Common Language Runtime* (CLR) da plataforma .NET. No desenvolvimento, Leyendecker (2005) utilizou técnicas muito similares às utilizadas no desenvolvimento do presente trabalho, tais como EBNF para especificar a sintaxe da linguagem e AST para análise semântica e geração de código. Do mesmo modo, o código gerado pelo compilador não é executável, precisa ser processado por outra ferramenta, neste caso o montador MSIL

da Microsoft que gera um arquivo executável.

Hildebrandt (2003) apresenta um estudo sobre o processo de desenvolvimento de aplicações para a plataforma Palm OS, utilizando a linguagem C++. Para validar seu estudo, foi implementado um editor gráfico 2D para desenhar e apagar polígonos não-convexos, utilizando a tela do Palm como principal elemento de interface gráfica interativa com o usuário. O estudo também apresenta as principais características da plataforma e as ferramentas de desenvolvimento disponíveis para Linux.

Kleberhoff e Dickerson (2004) deram continuidade ao projeto Jump, originalmente criado por Hewgill (1997). Jump é um programa de código aberto que permite que desenvolvedores programem em Java para Palm OS. Ele funciona lendo arquivos `.class` que contêm *bytecode* proveniente de programas Java, e criando arquivos *assembly* (`.asm`) compatíveis com o processador dos PDAs com Palm OS. Os arquivos `.asm` são montados por outro aplicativo, gerando um executável nativo para Palm OS. O Jump é compatível com o Java 2 Standard Edition (J2SE) 1.4 e os programas desenvolvidos para ele devem ser específicos para Palm OS, ou seja, não é possível implementar um programa que utilize os recursos da API do Java e convertê-lo utilizando o Jump.

Java2cpp (PROGRAMICS.COM, 2003) é um utilitário gratuito que traduz programas Java para C++. Tem como entrada arquivos `.java` e como saída arquivos `.h` e `.cpp` que podem ser compilados para Windows ou para Linux. Compatível com J2SE 1.4, possui uma biblioteca implementada em C++, chamada *DObject Library* (DOL), que transcreve algumas funcionalidades do Java, como *multithreads*, *sockets* e suporte a banco de dados.

3 DESENVOLVIMENTO DO TRABALHO

O desenvolvimento do presente trabalho foi fundamentado no estudo realizado e apresentado no capítulo anterior. Este capítulo apresenta os requisitos que o compilador deve atender, a especificação e o desenvolvimento do mesmo, o estudo de caso utilizado para validar a implementação e, por fim, os resultados obtidos na implementação do compilador.

3.1 REQUISITOS DO COMPILADOR

O compilador desenvolvido deve atender os seguintes requisitos:

a) receber como entrada arquivos fonte Java 5.0 e reportar erros léxicos, sintáticos e semânticos (Requisito Funcional - RF);
b) gerar código C++ como código intermediário do processo de compilação (RF);
c) gerar código para simular o mecanismo de coleta de lixo do Java para liberar memória não utilizada pelo programa (RF);
d) suportar bibliotecas básicas do Java, incluindo bibliotecas para criação de interfaces gráficas (`java.util, java.awt`) (Requisito Não Funcional - RNF);
e) possuir biblioteca Java para utilização de recursos específicos do Palm OS (RNF);
f) ser implementado em Java utilizando o ambiente de programação Eclipse (RNF).

3.2 ESPECIFICAÇÃO DO COMPILADOR

Não faz parte do presente trabalho especificar a linguagem Java 5.0, pois a especificação da mesma já existe e é mantida pela Sun Microsystems (2005c). No documento de especificação da linguagem são detalhadas as definições léxicas, sintáticas e semânticas. Assim sendo, esta seção detalha a especificação do compilador implementado, separando a especificação em três partes principais: analisadores léxico e sintático, analisador semântico e gerador de código C++.

Para facilitar o entendimento da especificação, foram utilizados modelos da *Unified*

Modeling Language (UML) para a especificação dos diagramas de casos de uso, atividades, classes, pacotes e seqüência. A ferramenta utilizada para especificar os diagramas foi o Enterprise Architect (EA). Visando melhorar a leitura dos diagramas de classe, sem comprometer o conteúdo, algumas informações dos diagramas foram suprimidas, tais como métodos e atributos. Ainda, os diagramas foram divididos por atividades ou assunto, de modo a fornecer diagramas menores e mais legíveis.

3.2.1 Diagrama de casos de uso

O compilador é formado por dois casos de uso onde o desenvolvedor Java pode interagir. Os dois casos de uso contemplam todos os requisitos especificados. A Figura 6 apresenta o diagrama de casos de uso, juntamente com os requisitos que cada um contempla.

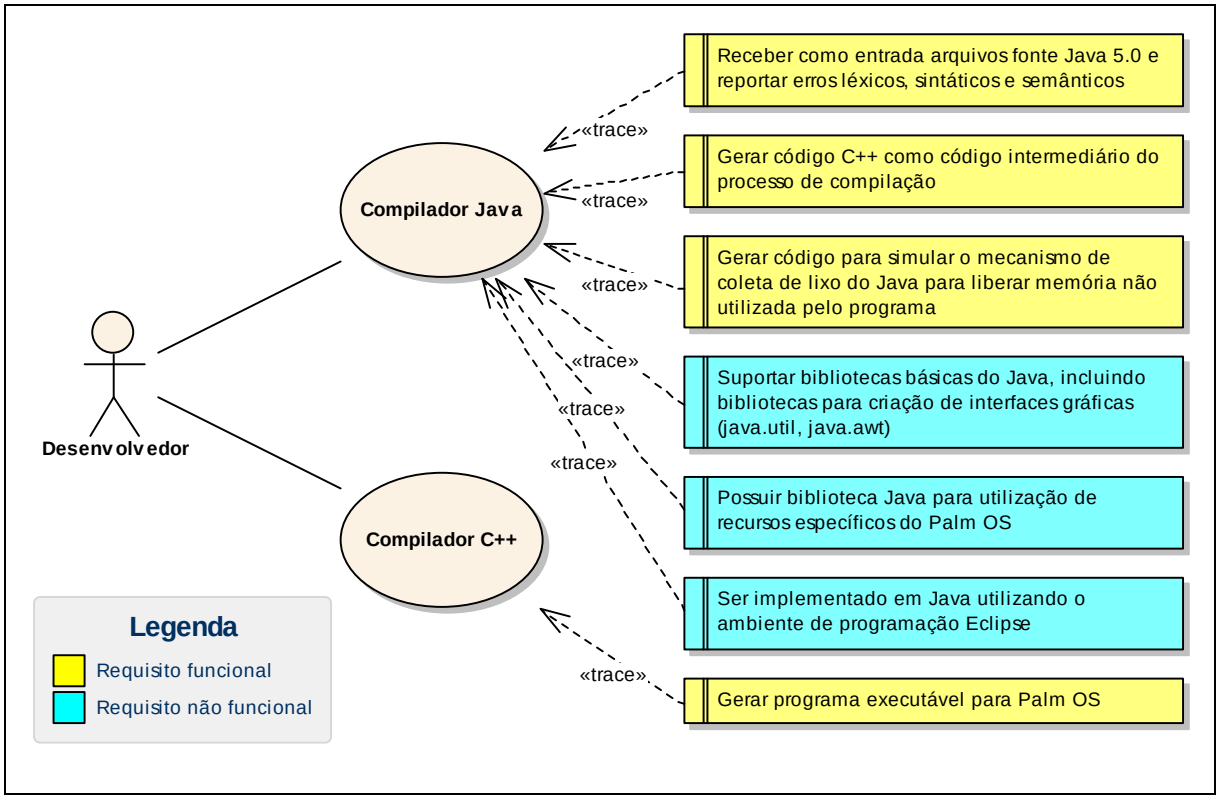


Figura 6 – Diagrama de casos de uso

Apenas o primeiro caso de uso (`Compilador Java`) pertence à especificação deste trabalho, apesar do desenvolvedor ter que interagir com o segundo (`Compilador C++`) para chegar ao produto final esperado: um executável nativo para Palm OS. Deste modo, o segundo caso de uso pode ser realizado por qualquer compilador C++ que atenda o seu único requisito (gerar programa executável para Palm OS). O primeiro caso de uso é detalhado no

Quadro 8 e o segundo no Quadro 9.

Compilador Java	
Descrição	Verifica se os arquivos fonte Java possuem erros de compilação e gera arquivos C++.
Pré-condições	Existir arquivos fonte Java com a extensão `.java`.
Fluxo principal	1. O desenvolvedor executa o compilador, informando como parâmetro os arquivos fonte Java que devem ser compilados. 2. O compilador executa as análises léxica, sintática e semântica dos arquivos de entrada. 3. O compilador gera código C++.
Fluxo alternativo	Não há.
Fluxo de exceção	No passo 2, caso o compilador encontre algum erro nos arquivos fonte: 1. O compilador informa o erro ao desenvolvedor para que o mesmo possa corrigi-lo. 2. O compilador encerra o processo de compilação.
Pós-condições	É gerado um par de arquivos fonte C++ com as extensões `.h` e `.cpp` para cada classe nos arquivos fonte.

Quadro 8 – Detalhamento do caso de uso `Compilador Java`

Compilador C++	
Descrição	Verifica se os arquivos fonte C++ possuem erros de compilação e gera um programa executável nativo para Palm OS.
Pré-condições	Existir arquivos com a extensão `.h` e `.cpp`.
Fluxo principal	1. O desenvolvedor executa o compilador, informando os arquivos fonte C++ conforme a configuração do compilador. 2. O compilador executa as análises léxica, sintática e semântica dos arquivos de entrada. 3. O compilador gera um programa executável nativo para Palm OS.
Fluxo alternativo	Não há.
Fluxo de exceção	No passo 2, caso o compilador encontre erros nos arquivos fonte C++ 1. O compilador informa os erros ao desenvolvedor. Observa-se que neste caso o compilador Java está com erro, pois gerou arquivos C++ com erros de compilação.
Pós-condições	É gerado um programa executável nativo para Palm OS.

Quadro 9 – Detalhamento do caso de uso `Compilador C++`

As atividades executadas pelos casos de uso são detalhadas no diagrama de atividades apresentado na Figura 7. Cada atividade representa uma etapa do processo de compilação.

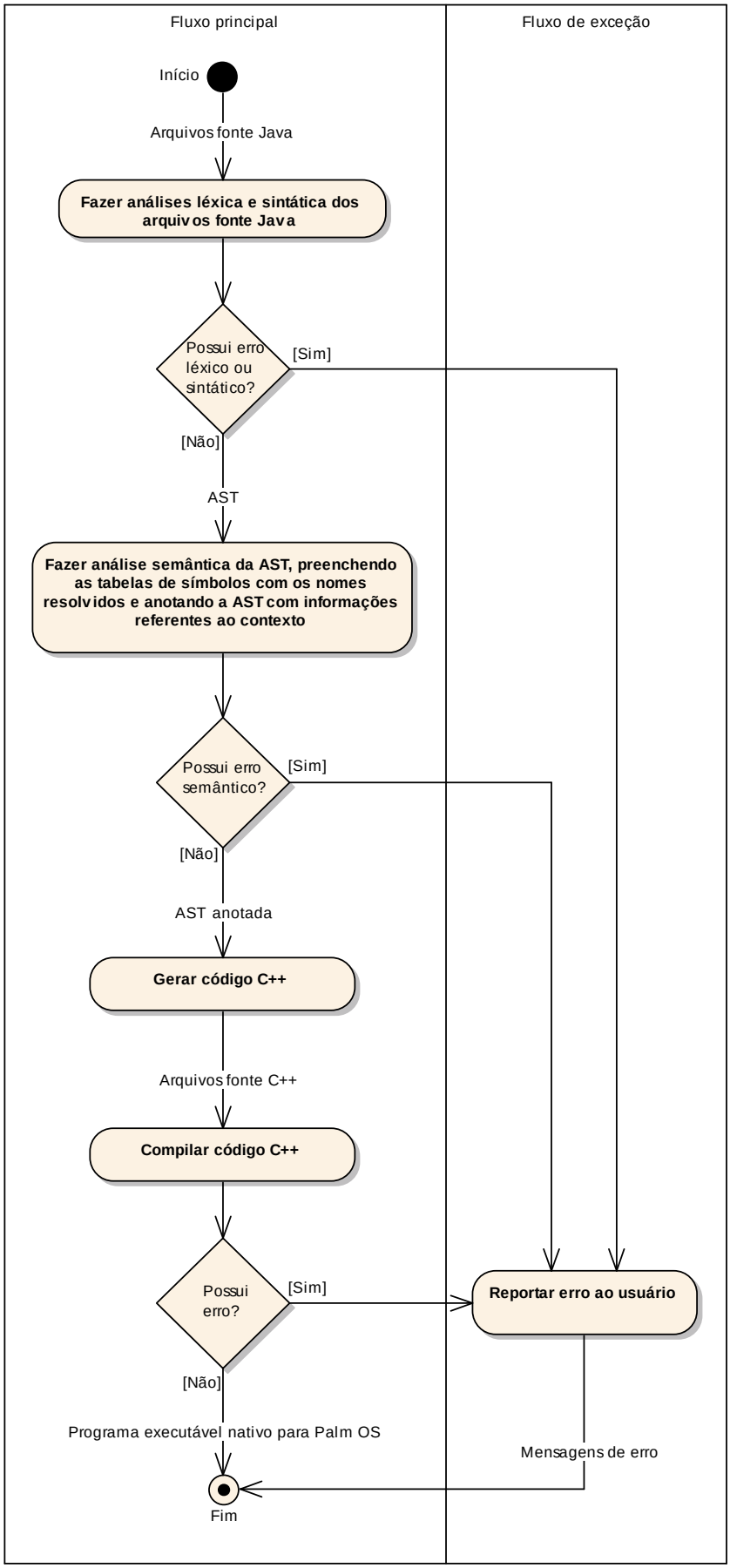


Figura 7 – Diagrama de atividades

3.2.2 Analisadores léxico e sintático

Como a especificação da LP Java 5.0 é pública, existem várias implementações da mesma para vários geradores de *parser*. O presente trabalho baseou-se em uma gramática disponível no repositório de gramáticas do JavaCC desenvolvida por Viswanadha (2004). Essa gramática foi escolhida por ser considerada a mais próxima da especificação sintática do Java e mais simples para ser trabalhada.

A gramática em questão foi alterada para contemplar a geração da AST da LP Java 5.0. Depois de alterada, a mesma foi disponibilizada para utilização pública no repositório de gramáticas do JavaCC. Este fato foi muito importante, pois a comunidade de desenvolvedores informou os erros encontrados, enriquecendo este trabalho com a construção de uma gramática mais coerente com a especificação do Java, uma vez que a mesma foi testada por pelo menos mais oito desenvolvedores que a utilizaram no desenvolvimento de seus próprios projetos. A gramática alterada encontra-se no Apêndice A.

Para construção da AST foram testados três geradores automatizados de árvore sintática: JJTree (JJTREE, 2007), SableCC (SABLECC, 2007) e ANTLR (ANTLR, 2007). Todos eles apresentaram algum tipo de deficiência na qualidade das classes geradas para árvore. Deste modo, os nós da AST foram especificados e implementados manualmente. Com isto, ficaram mais claros e bem definidos em relação aos nós das ASTs geradas. Este fato facilitou a especificação e o desenvolvimento do analisador semântico, que este está fortemente ligado a AST.

O diagrama de classes com as classes responsáveis pelas análises léxica e sintática é apresentado na Figura 8. As classes deste diagrama foram geradas pelo JavaCC a partir da gramática da linguagem Java. Deste modo, não se faz necessário explicar qual a responsabilidade de todas as classes presentes nele. A únicas classes deste diagrama que merecem uma breve explicação são:

a) `JavaParser`: classe responsável por efetuar as análises léxica e sintática de arquivos fonte Java, comandando todo o processo de análise;
b) `ParserException`: classe de exceção que é lançada pelo `JavaParser` ao encontrar um erro léxico ou sintático.

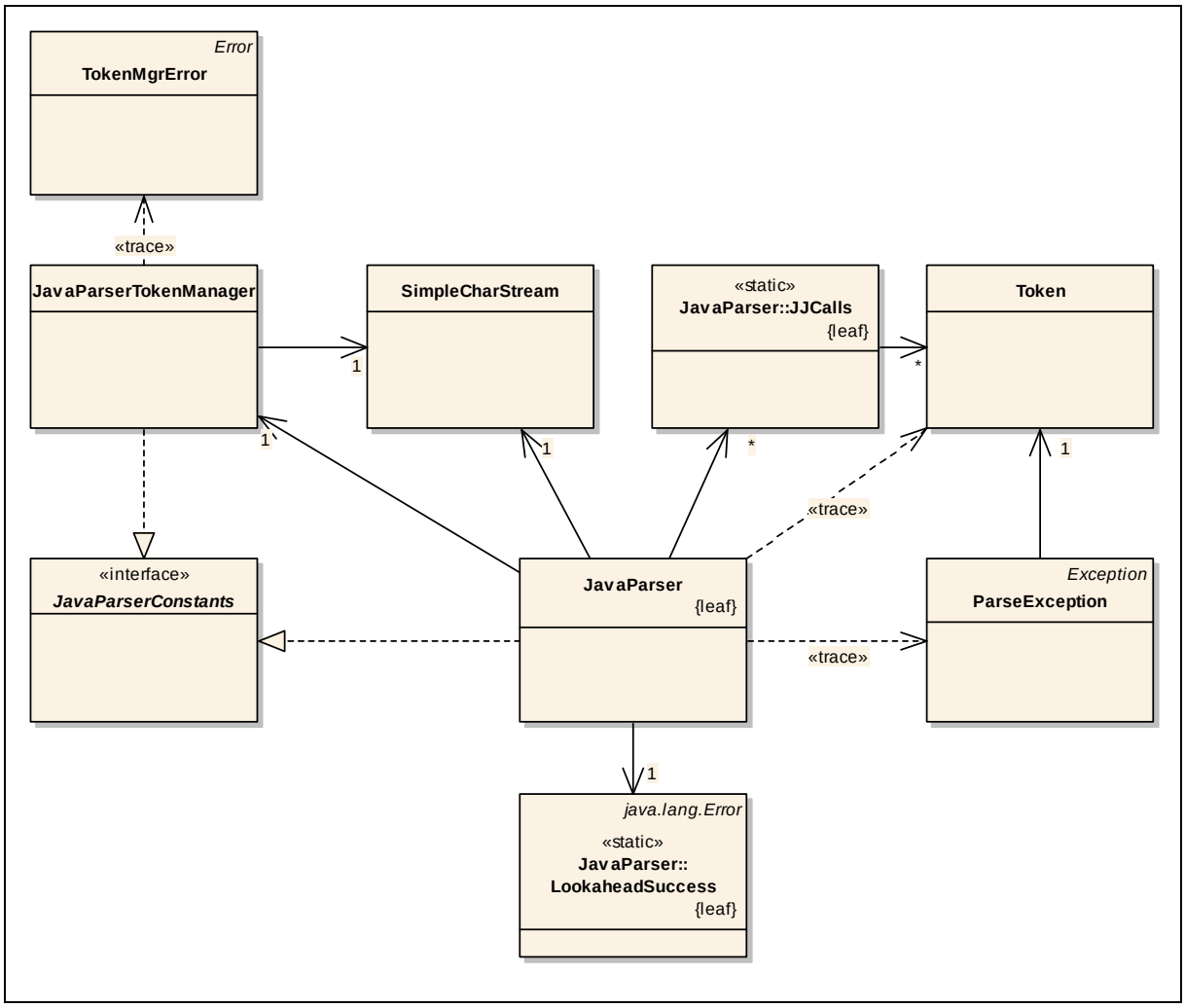


Figura 8 – Diagrama de classes dos analisadores léxico e sintático

O produto da análise sintática é a AST que representa o arquivo fonte compilado. Cada elemento é representado por uma classe diferente, correspondendo a um símbolo terminal ou um símbolo não-terminal da gramática. O diagrama de pacotes apresentado na Figura 9 mostra uma visão geral de todas as classes da AST. As informações guardadas por estas classes são basicamente referências para outros nós, no caso de nós não-terminais, ou valores e nomes, no caso de nós terminais.

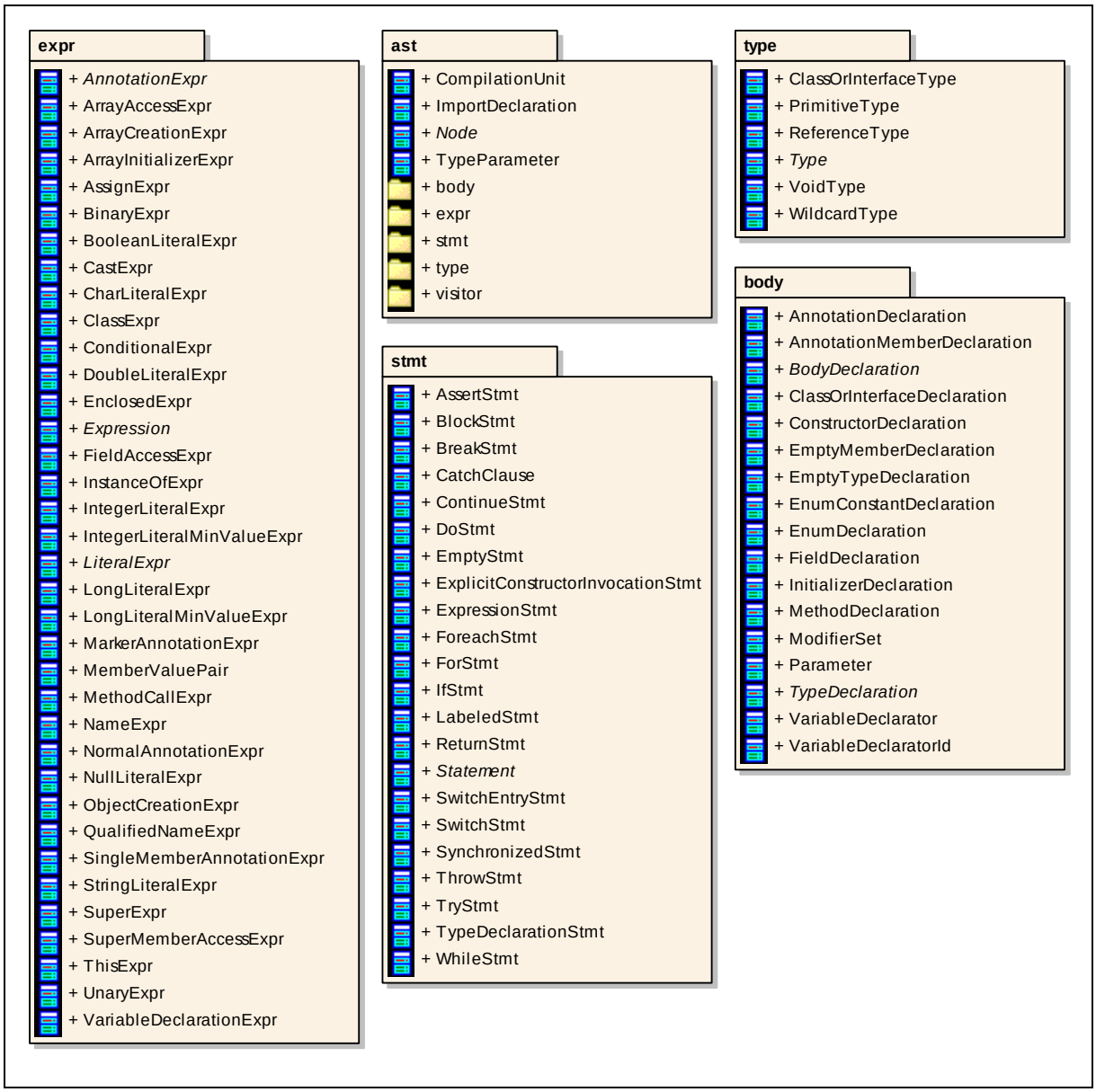


Figura 9 – Diagrama de pacotes dos nós da AST Java

Cada pacote representa um tipo específico de nó, ou seja, um agrupamento de nós com características similares. O Quadro 10 explica o que representa cada um destes pacotes.

Pacote	Descrição
`ast`	Possui a classe base de todas as demais classes da AST (`Node`) e as classes referentes aos nós que não se enquadram em nenhum outro pacote.
`body`	Classes referentes aos nós de metadados, ou seja, tudo o que caracteriza a definição de tipos (classes, interfaces, enumerações, anotações) e membros de tipos (métodos, atributos), entre outros.
`type`	Engloba as classes que representam os tipos do Java, como os tipos primitivos (`int`, `float`, etc.), tipo referência, entre outros.
`stmt`	Inclui as classes que denotam as instruções de controle de fluxo do Java, tais como `if`, `while`, etc.
`expr`	Classes que representam todas as expressões da sintaxe do Java, tais como expressões aritméticas, valores literais e chamadas de métodos.

Quadro 10 – Pacotes de classes da AST

3.2.3 Analisador semântico

Devido a grande quantidade de classes utilizadas para especificar o analisador semântico, vários diagramas foram necessários para simplificar o seu entendimento. O primeiro diagrama mostrado na Figura 10 apresenta as principais classes do analisador semântico. O Quadro 11 explica a responsabilidade de cada.

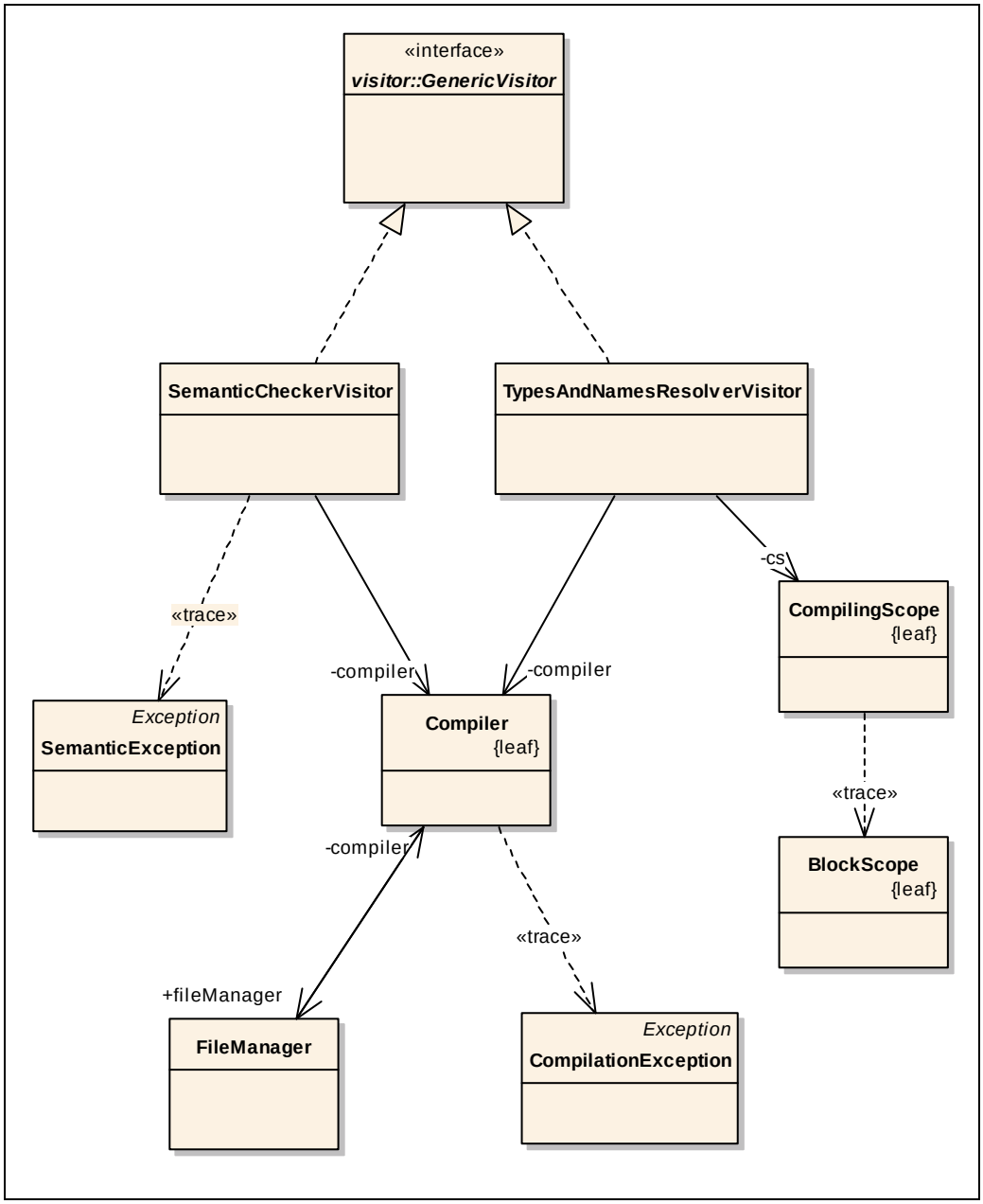


Figura 10 – Diagrama de classes do analisador semântico

Classe	Descrição
`GenericVisitor`	Interface base para todas as classes que precisam visitar todos os nós da AST. Implementa o *design pattern Visitor*.
`TypesAndNamesResolverVisitor`	Classe responsável por percorrer toda a AST e resolver todos os nomes e tipos encontrados. As informações obtidas na resolução são armazenadas no próprio nó resolvido. Nomes de tipos e variáveis são armazenados também na classe `BlockScope`, para verificar se algum nome duplicado foi utilizado e para recuperá-los mais facilmente.
`SemanticCheckerVisitor`	Classe responsável por percorrer toda a AST e aplicar as regras semânticas de cada construção do Java.
`Compiler`	Classe responsável por centralizar e comandar todo o processo de compilação. Ela percorre todos os arquivos fonte Java de entrada e executa para cada arquivo as análises léxica, sintática e semântica e a geração de código.
`CompilingScope`	Classe responsável por manter uma pilha de `BlockScope`, e garantir que um tipo ou variável não foi declarado em dois blocos subseqüentes.
`BlockScope`	Classe responsável por guardar informações sobre as variáveis e tipos declarados dentro de um bloco de código, bem como garantir que tipos e variáveis com o mesmo nome não sejam declarados.
`FileManager`	Classe responsável por todos os acessos a arquivos. A classe resolvedora de nomes faz uma chamada ao `FileManager` sempre que um nome ou tipo que precise ser resolvido represente uma classe que ainda não foi carregada. Ele pode carregar classes de arquivos `.java`, arquivos `.class` ou arquivos `.jar`. Todas as classes por ele carregadas são armazenadas em um *cache* para otimizar acessos subseqüentes.
`SemanticException`	Classe de exceção lançada pelas classes `TypesAndNamesResolverVisitor` e `SemanticCheckerVisitor` quando um erro semântico é detectado.
`CompilationException`	Classe de exceção lançada pela classe `Compiler` sempre que um erro léxico, sintático ou semântico é encontrado. A mensagem de erro desta exceção traz a linha e a coluna do arquivo fonte que apresentou erro.

Quadro 11 – Classes do analisador semântico

A resolução de nomes é uma importante parte da análise semântica, sendo responsável por designar o significado dos nomes no contexto em que ele está inserido. A Figura 11 mostra o diagrama de classes para resolução de nomes.

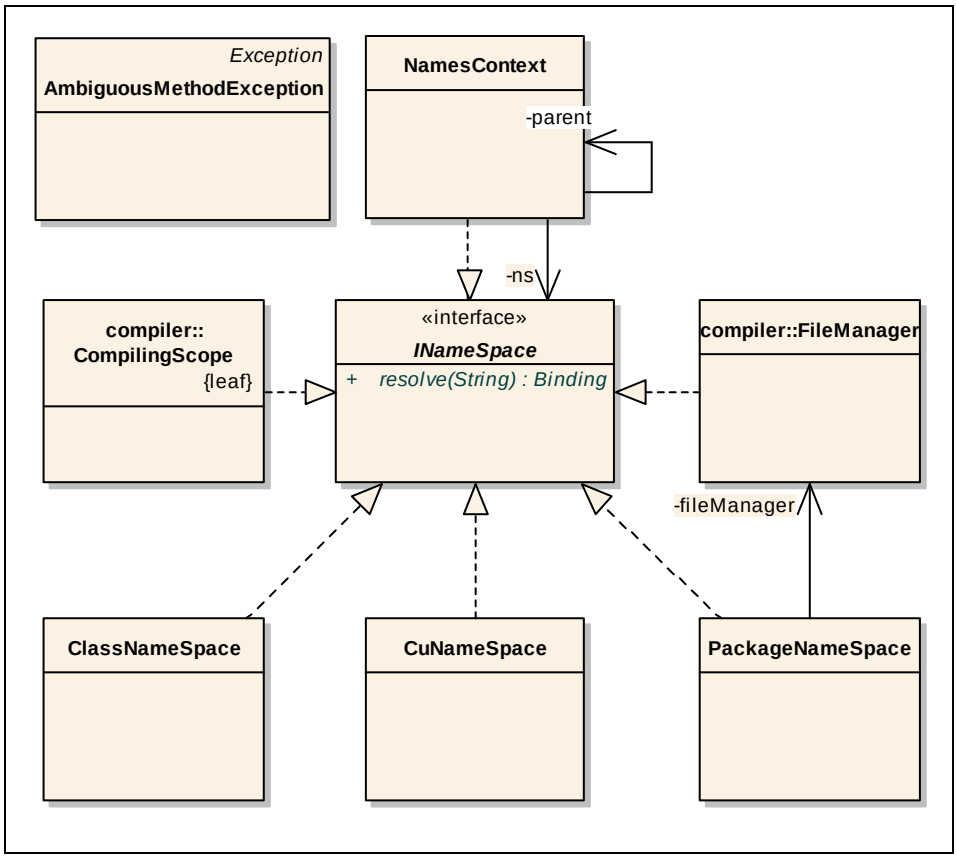


Figura 11 – Diagrama de classes dos resolvedores de nomes

A principal classe do conjunto que resolve nomes é a `NamesContext`. Ela e as demais são descritas no Quadro 12.

Classe	Descrição
`INameSpace`	Interface que todas as classes resolvedoras de nomes devem implementar. Ela possui dois métodos, um para resolver nomes simples e outro que recebe uma lista de tipos para resolver nomes de métodos.
`NamesContext`	Classe responsável por manter uma pilha de resolvedores de nomes. Ela por si só não resolve nomes, porém tenta resolver nomes no resolvedor que estiver mais no topo da pilha, e enquanto não conseguir resolver, tenta no próximo seguindo a seqüência da pilha, até que o nome seja resolvido ou que a pilha esteja vazia.
`CuNameSpace`	Classe responsável por resolver nomes de tipos dentro de um `CompilationUnit`, que representa um arquivo Java.
`ClassNameSpace`	Classe responsável por resolver nomes de tipos e membros dentro de uma classe.
`PackageNameSpace`	Classe responsável por resolver nomes de classes dentro de pacotes. Para carregar uma classe ela utiliza o `FileManager`.
`CompilingScope`	Classe responsável por resolver nomes de tipos e variáveis dentro do escopo do bloco atual.
`FileManager`	Classe responsável por resolver nomes de classes e pacotes, levando em consideração que uma classe pode estar dentro de um arquivo `.java` que vai ser compilado ou ainda de um arquivo `.class` ou `.jar` de uma biblioteca.
`AmbiguousMethodException`	Classe de exceção lançada quando o nome a ser resolvido representa um método sobrecarregado cujos tipos dos valores passados de parâmetro são ambíguos.

Quadro 12 – Classes resolvedoras de nomes

A lógica utilizada para resolver nomes é mostrada no diagrama de seqüência da Figura 12.

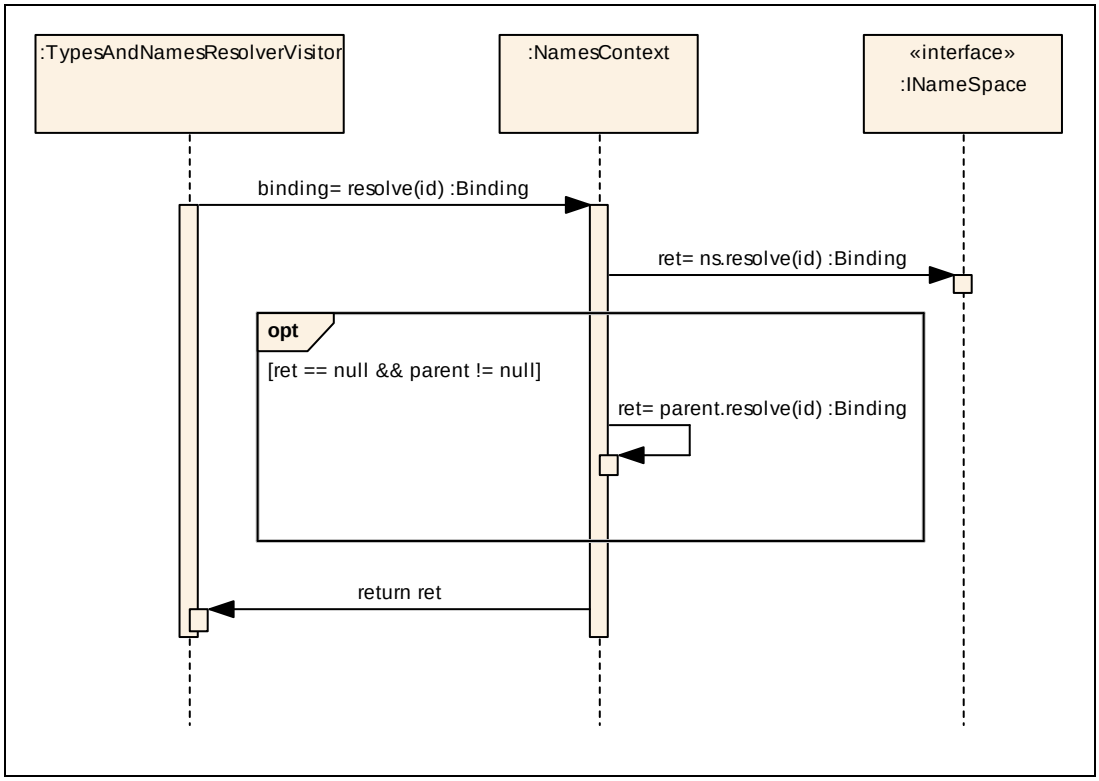


Figura 12 – Diagrama de seqüência da resolução de nomes

O processo de resolução de nomes inicia com a classe `TypesAndNamesResolverVisitor`, que percorre toda a AST, requisitando ao `NamesContext` a resolução de um nome encontrado em algum nó durante a varredura. O `NamesContext` mantém uma pilha de `INameSpace` que resolvem nomes de tipos, atributos e variáveis visíveis no contexto atual da varredura. Deste modo, o `NamesContext` requisita ao `INameSpace` do topo da pilha a resolução do nome. Caso o nome não seja resolvido, requisições são feitas aos demais resolvedores empilhados até que o nome seja resolvido ou até que não haja mais resolvedores na pilha. A não resolução do nome caracteriza um erro semântico.

Quando um nome é resolvido, ele passa a ser representado por um objeto denominado *binding*. Para cada tipo de nome existe uma definição de *binding*. Os *bindings* que representam tipos ou membros de tipos são definidos de forma abstrata, uma vez que um tipo pode ser encontrado dentro de um arquivo fonte (`.java`) ou um arquivo já compilado (`.class`). Deste modo, existem duas implementações para este tipo de *binding*. Os *bindings* básicos são mostrados na Figura 13.

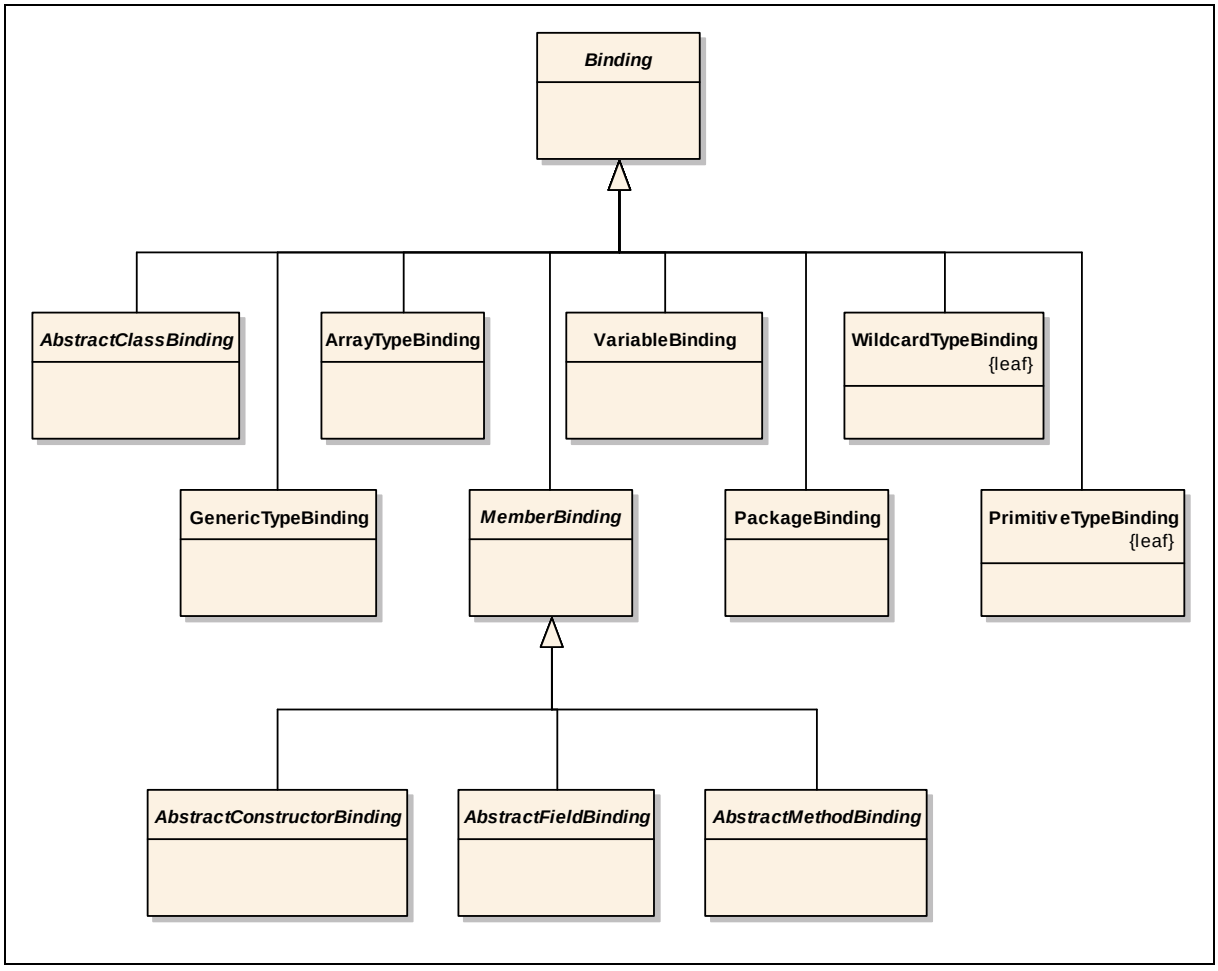


Figura 13 – Diagrama de classes dos *bindings* básicos

O Quadro 13 explica o que cada classe do diagrama representa.

Classe	Descrição
`Binding`	Classe abstrata base de todos os *bindings*.
`ArrayTypeBinding`	Representa um tipo `array`.
`PackageBinding`	Representa um pacote.
`PrimitiveTypeBinding`	Representa um tipo primitivo do Java.
`VariableBinding`	Representa uma variável.
`GenericTypeBinding`	Representa um tipo genérico do Java.
`WildcardTypeBinding`	Representa qualquer tipo, representado pelo caractere “?” na sintaxe do Java.
`AbstractClassBinding`	Classe abstrata para o *binding* que representa uma classe.
`MemberBinding`	Classe abstrata base dos *binding* que representam membros de classes.
`AbstractConstrutorBinding`	Classe abstrata para o *binding* que representa um construtor.
`AbstractMethodBinding`	Classe abstrata para o *binding* que representa um método.
`AbstractFieldBinding`	Classe abstrata para o *binding* que representa um atributo.

Quadro 13 – Classes dos *bindings* básicos

Os *bindings* de arquivos compilados são apresentados na Figura 14 e os *bindings* de arquivos fonte na Figura 15.

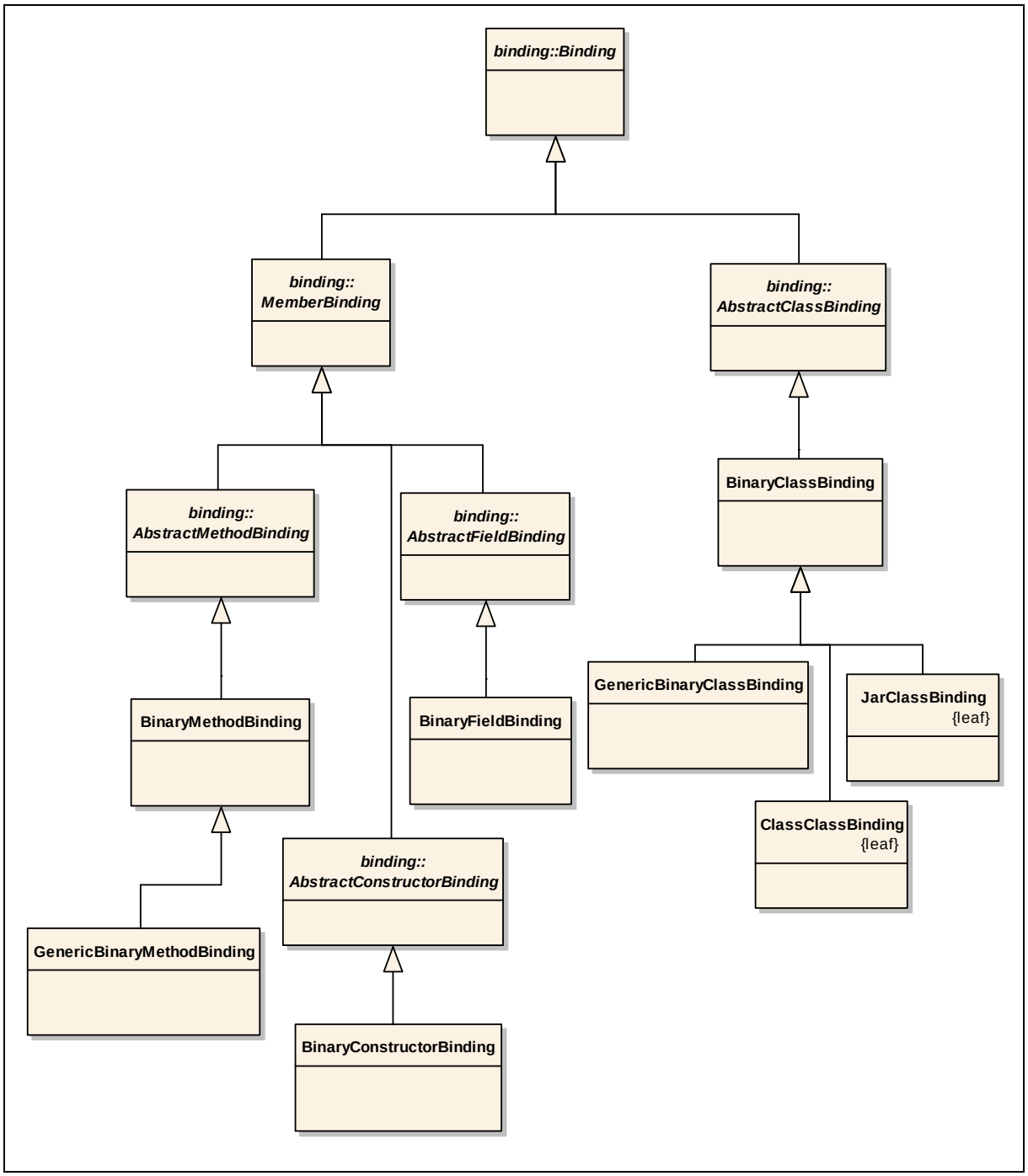


Figura 14 – Diagrama de classes dos *bindings* de arquivos compilados

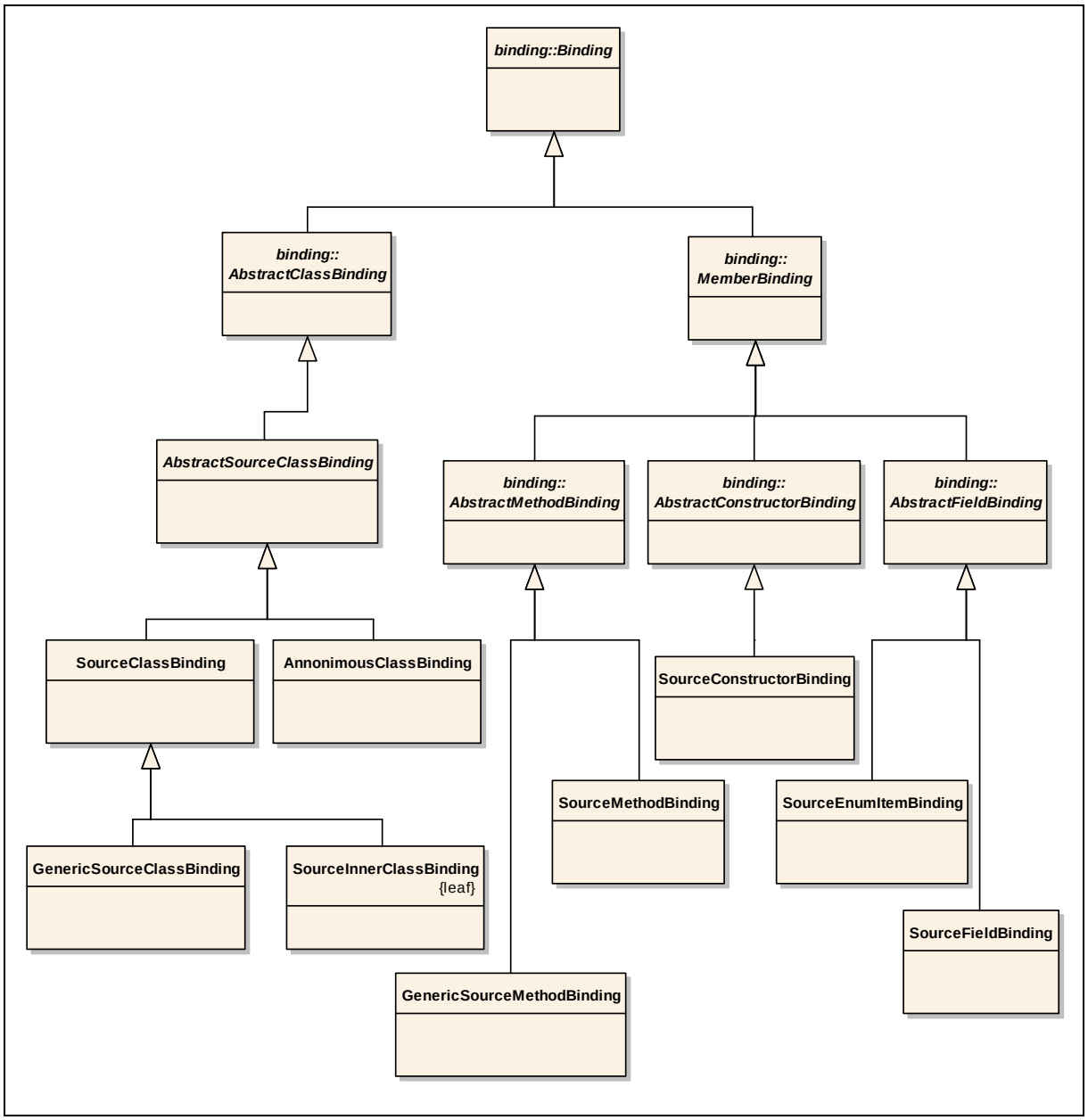


Figura 15 – Diagrama de classes dos *bindings* de arquivos fonte

3.2.4 Gerador de código C++

A geração de código C++ ocorre depois da análise semântica. A principal estrutura de dados utilizada nesta etapa é a AST obtida na análise sintática e anotada na análise semântica. Para cada classe Java são gerados dois arquivos, um arquivo `.h` com a definição da classe e um arquivo `.cpp` com a implementação.

A seção seguinte descreve o mapeamento entre as construções das LPs Java e C++

utilizado na geração de código e em seguida são apresentados os diagramas de classes.

3.2.4.1 Mapeamento de construções Java para C++

A sintaxe da linguagem Java é muito parecida com a da linguagem C++. Isto é justificado pelo fato do Java ser derivado do C++. Esta similaridade facilita a geração de código para as construções básicas do C++ a partir de Java. Porém, o Java possui vários recursos que não existem no C++, como facilitadores para trabalhar com `Arrays`, interfaces, classes anônimas, entre outros. Por isto, faz-se necessário definir um mapeamento de quais construções do C++ devem ser utilizadas para reproduzir cada construção do Java. O Quadro 14 mostra o mapeamento de classes e interfaces, o Quadro 15 mostra o mapeamento de métodos e o Quadro 16 o mapeamento de atributos.

Java	C++
classe não-final ou abstrata	classe com todos os métodos virtuais (exceto os que são finais no Java)
classe final	classe com todos os métodos não virtuais
interface	classe com todos os métodos abstratos e atributos estáticos
classe interna não estática ou anônima	classe em arquivo próprio que recebe como parâmetro no construtor uma referência para classe externa
classe interna estática	classe em arquivo próprio
herança de classe	herança de classe
implementação de interfaces	herança de classes

Quadro 14 – Mapeamento de classes e interfaces Java para C++

Java	C++
método não final	método virtual
método final	método não virtual
método abstrato	método virtual abstrato
método estático	método estático

Quadro 15 – Mapeamento de métodos de classes Java para C++

Java	C++
atributo não final	atributo virtual
atributo final	atributo constante
atributo estático	atributo estático

Quadro 16 – Mapeamento de atributos de classes Java para C++

O acesso a membros de objetos no Java é feito através do caractere "`.`", no C++ os mesmos são acessados utilizando os caracteres "`->`". Os comandos do Java (`for`, `while`, etc.) são equivalentes no C++, pois possuem a mesma sintaxe, exceto os comandos:

a) `synchronized`: não é suportado, pois o C++ não possui comando equivalente e

não pretende-se dar suporte a *threads*;

b) `for` estendido: deve ser traduzido para um `for` convencional do C++. Caso o `for` percorra uma `Collection`, o `Iterator` deve ser utilizado, como mostrado no Quadro 17.

Java	C++

`for (Object obj : list)`
`{`
`    ...`
`}`
	`for (Iterator iter = list->iterator(); iter->hasNext(); )`
	`{`
	`    Object obj = iter->next();`
	`    ...`
	`}`

Quadro 17 – Mapeamento do comando `for` melhorado Java para C++

A visibilidade de métodos e atributos Java foi respeitada no C++, ou seja, um método ou atributo `public`, `protected` ou `private` vai gerar na classe C++, respectivamente, um método ou atributo `public`, `protected` ou `private`. O Java possui ainda a visibilidade *default* (representado pela ausência do modificador de visibilidade), que foi mapeada para `protected` no C++. Além disso, no Java a visibilidade `protected` possui uma semântica que não está presente no C++, onde classes podem acessar métodos ou atributos `protected` de classes que estiverem no mesmo pacote. Para reproduzir isto no C++ foi utilizado o recurso da linguagem chamado `friend class`. Este recurso permite que uma classe acesse métodos ou atributos `protected` ou `private` de outra classe.

3.2.4.2 Diagrama de classes

O diagrama com as classes responsáveis pela geração de código é apresentado na Figura 16.

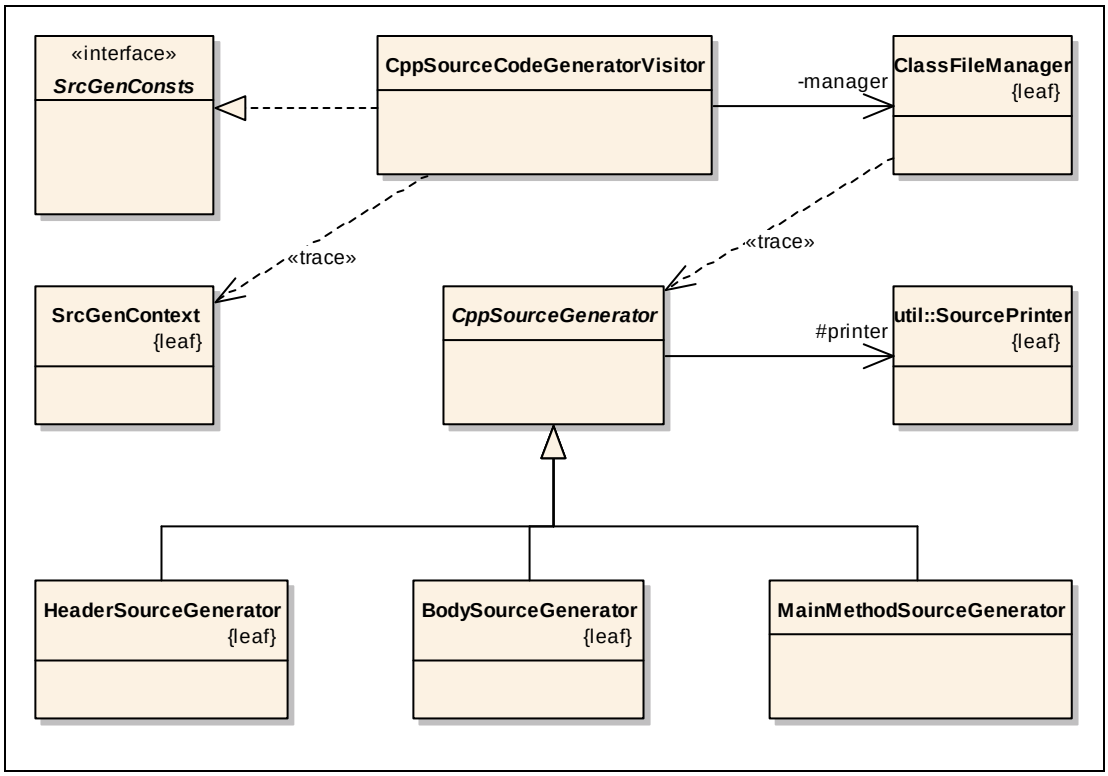


Figura 16 – Diagrama de classes do gerador de código C++

A principal classe é a `CppSourceCodeGeneratorVisitor`, que percorre todos os nós da AST e, deste modo, comanda o fluxo da geração de código. O Quadro 18 explica a responsabilidade de cada classe.

Classe	Descrição
`CppSourceCodeGeneratorVisitor`	Principal classe do gerador de código C++. Responsável por percorrer todos os nós da AST e gerar código específico para cada nó, obedecendo ao mapeamento de construções do Java para C++.
`SrcGenConsts`	Interface utilitária com várias constantes utilizadas na geração de código.
`ClassFileManager`	Classe responsável por gerenciar o sistema de arquivos, criando um novo arquivo sempre que for necessário.
`SrcGenContext`	Classe responsável por manter o contexto de geração de código. Isto é importante porque um nó pode gerar código de mais de uma maneira, de acordo com o contexto.
`CppSourceGenerator`	Classe responsável por agrupar funções comuns entre a geração de código do cabeçalho (`.h`) e da implementação (`.cpp`).
`HeaderSourceGenerator`	Classe com a responsabilidade de gerar efetivamente o arquivo `.h`.
`BodySourceGenerator`	Classe com a responsabilidade de gerar efetivamente o arquivo `.cpp`.
`MainMethodSourceGenerator`	Classe responsável por gerar o arquivo com o método `main()`, ponto de entrada dos aplicativos C++.
`SourcePrinter`	Classe que possui facilitadores para escrever código fonte em arquivos, como gerenciador de identação.

Quadro 18 – Classes da geração de código C++

Durante a varredura da AST, pode-se descobrir que certa parte do código deveria ter sido gerada em uma seção anterior, que já foi gerada e concluída com o início de uma nova seção. Por exemplo, durante a varredura de uma classe pode ser necessário adicionar um `include` no fonte gerado. Para evitar este problema, nenhum código é efetivamente escrito em arquivo até que toda a AST tenha sido totalmente percorrida. Esta solução foi utilizada também para evitar mais de uma passada na árvore. Assim, durante a única varredura, ao invés de escrever código diretamente em arquivo, um objeto com as características de uma classe C++ é alimentado, e, ao final da varredura, são gerados dois arquivos a partir desta classe: o `.h` e o `.cpp`. Deste modo, as classes `HeaderSourceGenerator` e `BodySourceGenerator` podem utilizar a mesma estrutura para gerar os arquivos C++, evitando redundância e várias passadas na AST. As classes utilizadas para representar uma classe C++ são apresentadas na Figura 17.

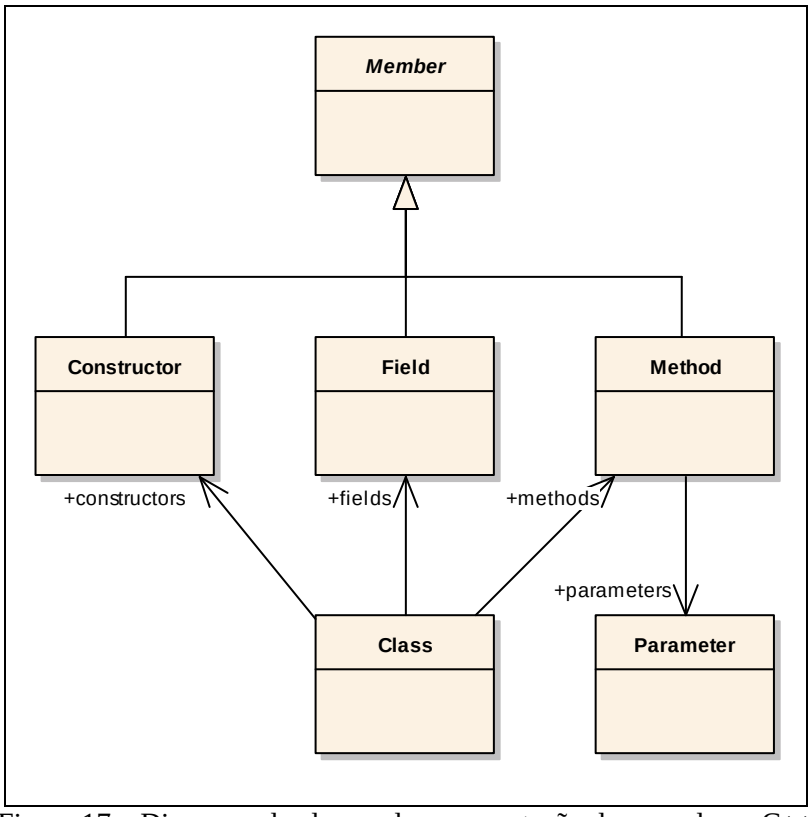


Figura 17 – Diagrama de classes da representação de uma classe C++

O nome de uma classe C++ gerada possui o mesmo nome da classe Java que a originou, porém com a concatenação do nome do pacote da classe Java e os "." substituídos por "_". O Quadro 19 mostra um exemplo.

Java	C++
`org/furb/HelloWorld.java`	`org_furb_HelloWorld.h` `org_furb_HelloWorld.cpp`

Quadro 19 – Exemplo de conversão de nome de classes Java para C++

3.2.5 Diagrama de seqüência

As principais classes que formam o compilador já foram apresentadas separadas por etapa. Sendo assim, a Figura 18 apresenta uma visão geral de todo o processo de compilação com as principais classes de cada etapa.

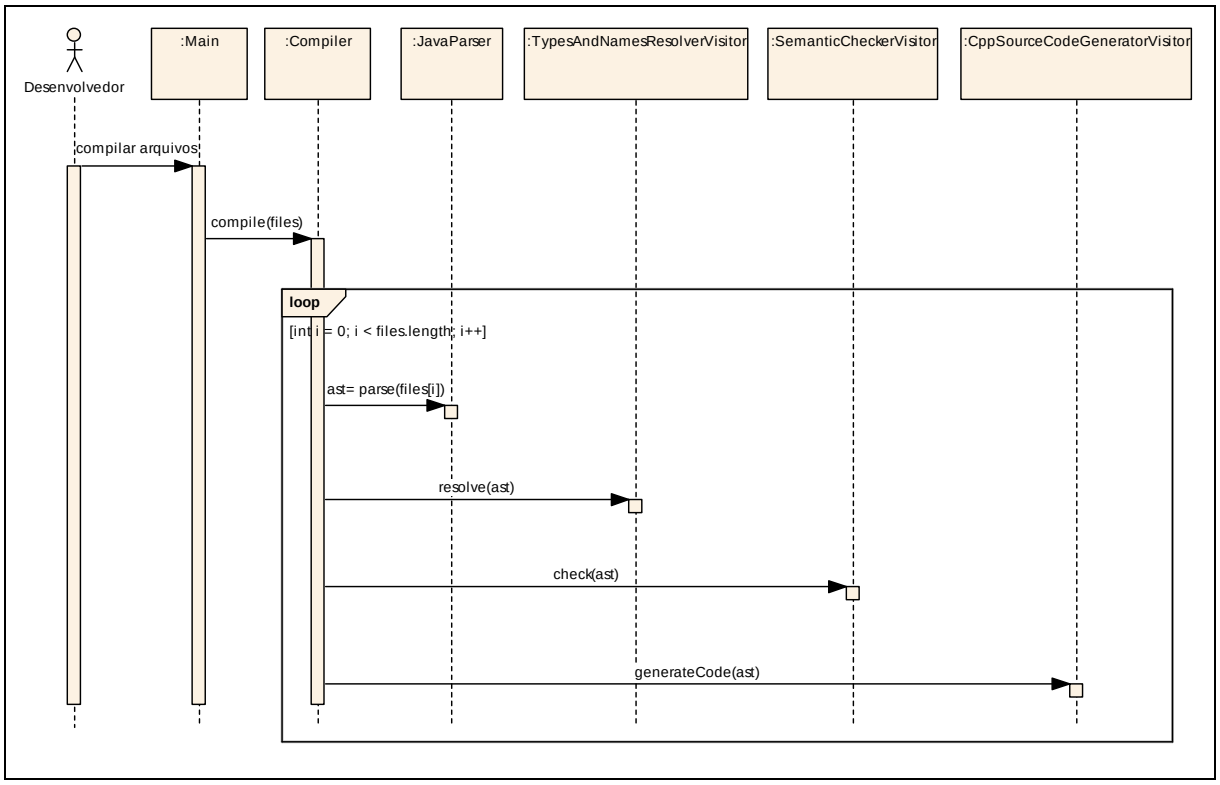


Figura 18 – Diagrama de atividades do processo de compilação

3.3 IMPLEMENTAÇÃO

Devido ao tamanho e complexidade da especificação da LP Java 5.0, não foi possível implementá-la totalmente. A regra utilizada para delimitar o que seria implementado foi garantir que no mínimo a semântica e a geração de código das construções utilizadas no estudo de caso desenvolvido para testar o compilador fossem respeitadas. Então, para documentar as limitações do analisador semântico, pode-se agrupar a semântica em quatro categorias:

a) metadados: engloba tudo o que caracteriza a definição de tipos, tais como classes, interfaces, anotações ou enumerações, e membros de tipos, incluindo métodos e

atributos, entre outros. Não é feita nenhuma verificação semântica nos seguintes casos:

- anotações: a definição ou utilização de tipos anotados é ignorada pela análise semântica e pela geração de código,
- enumerações: os tipos enumerados não são verificados pela análise semântica e não é gerado código para este tipo,
- blocos estáticos: não é verificada a semântica de blocos estáticos de código definidos no corpo de tipos, do mesmo modo que não é gerado fonte para esta estrutura;

b) tipos: são os tipos de dados do Java, tais como tipos primitivos (`int`, `double`, etc.) e referências. Todas as verificações semânticas para compatibilidade de tipos são efetuadas, inclusive a compatibilidade com o novo recurso da versão 5.0 do Java, o *autoboxing*;

c) comandos: incluem as instruções para controle de fluxo, tais como `if`, `for`, `while`, entre outros. Não são feitas as verificações semânticas e geração de código para os seguintes comandos: `assert`, rótulos, `do/while`, `while`, `break`, `continue`, `throw/throws`, `try/catch/finally`, `synchronized` e chamadas explícitas ao construtor da classe pai: (`super`);

d) expressões: são as instruções que denotam valores, tais como variáveis, acesso a atributos, constantes literais, expressões aritméticas, etc. As expressões de acesso à membros da classe pai (`super.<membro>`) e expressões relacionadas a anotações não são verificadas semanticamente, nem é gerado código para estas expressões.

Em conseqüência das limitações da análise semântica, o gerador de código possui as mesmas limitações e algumas outras. Ou seja, é gerado código apenas para todas as estruturas que são verificadas pelo analisador semântico, exceto: tipos genéricos e *loop* `for` estendido.

O Quadro 20 apresenta um trecho de código correspondente a análise semântica do comando `if`.

```
    public Object visit(IfStmt n, NamesContext arg) {
        n.condition.accept(this, arg);
        Binding b = (Binding) n.condition.data;
        if (!PrimitiveTypeBinding.BooleanType.isAssignableFrom(b)) {
            throw new RuntimeSemanticException("Cannot conver from {0} to boolean",
                                    new Object[] { b.getName() }, n.condition, file);
        }
        n.thenStmt.accept(this, arg);
        if (n.elseStmt != null) {
            n.elseStmt.accept(this, arg);
        }
        return null;
    }
```

Quadro 20 – Análise semântica do comando `if`

O Quadro 21 mostra um exemplo de código gerado pelo compilador a partir de um código em Java.

Java

```
//HelloWorld.java

public class HelloWorld {

    public static void main(String[]
args) {
        HelloWorld helloWorld = new
HelloWorld("Hello World!");

System.out.println(helloWorld.getStr());
    }

    private final String str;

    public HelloWorld(String str) {
        this.str = str;
    }

    public String getStr() {
        return str;
    }
}
```

C++

```
//HelloWorld.h
#ifndef HELLOWORLD_H_
#define HELLOWORLD_H_

#include "java.lang.Array.h"
#include "java.lang.String.h"
#include "java.lang.System.h"
#include "japa.lang.GC.h"

class HelloWorld {
public:
    HelloWorld(java_lang_StringPtr str);

    static void main(java_lang_ArrayPtr<
java_lang_StringPtr > args);

    virtual java_lang_StringPtr getStr();

private:
    const java_lang_StringPtr str;
};

typedef gc::ptr<HelloWorld> HelloWorldPtr;

#endif //HELLOWORLD_H_
```

```
//HelloWorld.cpp
#include "HelloWorld.h"

HelloWorld::HelloWorld(java_lang_StringPtr
str): str(str) {
}

void HelloWorld::main(java_lang_ArrayPtr<
java_lang_StringPtr > args) {
    HelloWorldPtr helloWorld =
HelloWorldPtr(new
HelloWorld(java_lang_StringPtr("Hello
World!")));
    java_lang_System::out->
println(helloWorld->getStr());
}

java_lang_StringPtr HelloWorld::getStr() {
    return str;
}
```

Quadro 21 – Exemplo de código gerado

3.3.1 Bibliotecas Java implementadas em C++

Java possui uma API repleta de classes para diversas aplicações. É muito difícil fazer qualquer programa em Java sem utilizar alguma delas. Deste modo, algumas delas foram implementadas em C++ para que possam ser utilizadas nos programas Java compilados com o compilador desenvolvido. O critério utilizado para definir quais classes da API seriam implementadas, foi o mesmo critério utilizado no desenvolvimento do analisador semântico e

do gerador de código, ou seja, garantir que o estudo de caso pudesse ser inteiramente compilado. Deste modo, as classes implementadas e os métodos suportados de cada classe da API são apresentados no Apêndice B.

Uma pequena adaptação teve que ser feita para dar suporte aos menus do Java. A API para criação de menus do Palm OS não fornece mecanismos para definir menus ou itens de menu dinamicamente, ela permite apenas adicionar itens a um menu existente definido estaticamente em um *resource*. Deste modo, foi desenvolvida uma solução para contornar este problema, que consiste em planificar os menus e seus itens em um único menu e adicioná-los a um menu definido em um *resource*. O Quadro 22 mostra um comparativo entre uma estrutura de menu em Java e a adaptação feita em Palm OS.

Java	Palm OS
Arquivo Editar +- Novo +- Copiar +- Salvar +- Recortar +- Sair +- Colar	Menu +- Arquivo - Novo +- Arquivo - Salvar +- Arquivo - Sair +- Editar – Copiar +- Editar – Recortar +- Editar – Colar

Quadro 22 – Comparativos entre menus do Java e a solução feita para Palm OS

3.3.2 Técnicas e ferramentas utilizadas

O compilador foi escrito na linguagem Java utilizando o ambiente de programação Eclipse. Os analisadores léxico e sintático foram gerados utilizando o gerador de *parser* JavaCC. Para efetuar a análise semântica e a geração de código foi utilizado o *design pattern* para a varredura da AST conhecido como *Visitor*. Para compilar o código C++ gerado pelo compilador e as bibliotecas do Java implementadas em C++ foi utilizado inicialmente o ambiente de programação PODS. Porém, devido ao grande número de classes C++ que foram definidas durante o desenvolvimento do compilador e a limitação do Palm OS de *jumps* relativos de no máximo 32K, o PODS não conseguiu mais compilar as bibliotecas e os fontes gerados. O problema só foi resolvido utilizando o modelo de código *smart* no CodeWarrior for Palm OS. Deste modo, as últimas bibliotecas implementadas e o estudo de caso foram compilados utilizando o ambiente de programação CodeWarrior.

3.3.2.1 Coletor de lixo

Não existe biblioteca padrão do C++ que simule o comportamento do coletor de lixo do Java. Deste modo, para simulá-lo foram utilizadas duas técnicas conhecidas como *smart pointer* e *reference counting*. *Smart pointer* pode ser considerado um tipo de dado abstrato que encapsula um ponteiro e proporciona a adição de outras habilidades (SMART..., 2007). *Reference counting* é uma técnica de armazenar o número de referências para um objeto (REFERENCE..., 2007). Com isto, foi implementado um *smart pointer* em C++ utilizando uma classe `template` que armazena o número de referências para o objeto que o ponteiro encapsulado referencia. Em todo o código gerado, toda referência para um objeto é um *smart pointer*. Sempre que o contador de referências de um objeto chegar a zero, este objeto é destruído ou coletado. Ou seja, assim como no Java, enquanto um objeto for referenciado por outro objeto não coletável, ele não é coletado. Porém no Java quando ele não for mais referenciado, ele passa a ser candidato à coleta, diferente da técnica implementada onde o objeto é coletado instantaneamente.

A técnica utilizada possui a desvantagem de não detectar ciclos, ou seja, se um objeto referenciar outro, e este outro referenciar o primeiro, mesmo que não haja nenhuma outra referência para ambos, eles não serão coletados. Isto porque ambos possuem uma referência cada no seu contador de referências.

Para cada classe gerada em C++, é definido no mesmo arquivo uma segunda classe, cujo tipo é o *smart pointer* com a primeira classe como parâmetro. O nome desta segunda classe é o mesmo nome da primeira mais o sufixo `Ptr`. O Quadro 23 mostra a declaração de um *smart pointer*.

```
...
class java_awt_Color : public java_lang_Object {
   ...
};

typedef gc::ptr<java_awt_Color> java_awt_ColorPtr;
...
```

Quadro 23 – Declaração de um *smart pointer*

Conforme o exemplo do Quadro 23, sempre que se for referenciar um objeto de `java_awt_Color`, deve-se referenciar o seu *smart pointer* `java_awt_ColorPtr`.

3.3.2.2 *Design patterns*

Design patterns, em linhas gerais, podem ser considerados como formas convenientes de reutilizar código orientado a objetos entre projetos e entre desenvolvedores (COOPER, 1998, p. 10). Cada *design pattern* possui um nome e se resume em descrever um problema, uma solução e, por fim, as conseqüências de sua utilização (GAMMA et al, 1998, p. 17). Existem diversos *design patterns*, para resolver diferentes situações cotidianas no desenvolvimento de software. A implementação do presente trabalho foi feita utilizando *Iterator* e *Visitor*.

O *pattern Iterator* provê uma alternativa para acessar os elementos de uma coleção sem expor a sua representação interna (GAMMA et al, 1998, p. 219). A API de coleções do Java utiliza este *pattern*. Ele foi utilizado no *binding* de classe para dar acesso às classes bases de uma classe.

O *pattern Visitor* representa uma operação que deve ser executada nos elementos de uma estrutura de objetos. Ele permite definir novas operações sem ter que alterar as classes dos elementos em que ele opera (GAMMA et al, 1998, p. 279). Este *pattern* foi utilizado em todas as rotinas que necessitam percorrer a AST. Deste modo, as rotinas de análise semântica e geração de código, que necessitam percorrer a AST, puderam ser escritas sem a necessidade de alterar nó algum da árvore, pois este *pattern* fornece um modo padrão para percorrer esta estrutura.

3.3.3 Operacionalidade da implementação

A ferramenta desenvolvida é um aplicativo Java 5.0 de linha de comandos, assim como o compilador do Java. Para executá-la o desenvolvedor deve informar quais arquivos devem ser compilados. Por padrão, o compilador gera os fontes C++ no mesmo diretório de execução, mas este diretório pode ser alterado via parâmetro. O Quadro 24 apresenta todos os parâmetros opcionais suportados pelo compilador.

Parâmetro	Descrição
`-d <diretório>`	Diretório onde os arquivos C++ serão gerados. Se omitido, os arquivos serão gerados no diretório corrente.
`-verbose`	Imprime os passos durante a compilação.
`-time`	Imprime o tempo de compilação.
`-sourcepath <diretórios separador por ;>`	Define caminhos adicionais com fontes que participam da compilação.
`-cp <diretórios ou arquivos zip/jar separador por ;>`	Define caminhos adicionais com classes que participam da compilação.
`-classpath <diretórios ou arquivos zip/jar separador por ;>`	Define caminhos adicionais com classes que participam da compilação.
`-version`	Imprime a versão do compilador.

Quadro 24 – Parâmetros opcionais suportados pelo compilador

Para executar o compilador primeiro devem ser informados os parâmetros opcionais, se houver algum, seguido pelos arquivos ou diretórios com arquivos que devem ser compilados. A Figura 19 mostra a execução do compilador.

Figura 19 – Execução do compilador

3.3.3.1 Estudo de caso

Para testar o compilador durante o seu desenvolvimento, foi implementado um pequeno jogo em Java, conhecido como ColorJunction. O objetivo do jogo é limpar a tela, apagando os blocos que estão unidos a outros de mesma cor (GESSER, 2007). A Figura 20

mostra a tela principal do jogo.

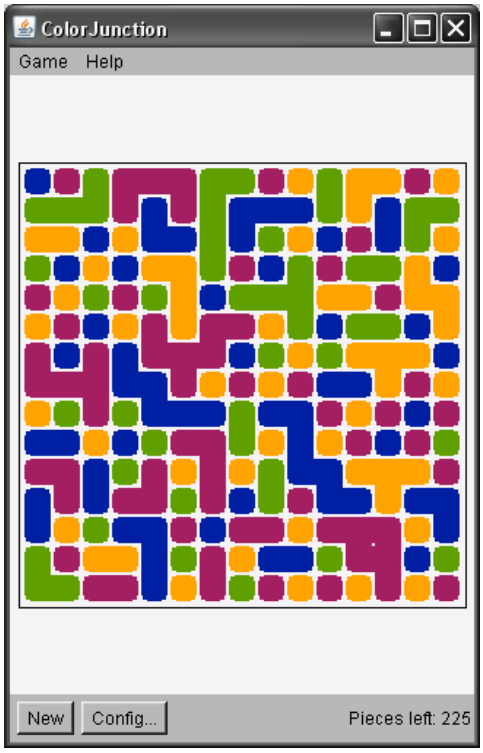


Figura 20 – Tela principal do jogo ColorJunction em Java

Apesar da aparente simplicidade do jogo, ele possui três telas diferentes, que utilizam vários componentes do AWT, e três gerenciadores de *layouts* diferentes, além de monitorar o evento de *click* do *mouse*. O diagrama de classes implementadas em Java para o jogo ColorJunction é apresentado na Figura 21.

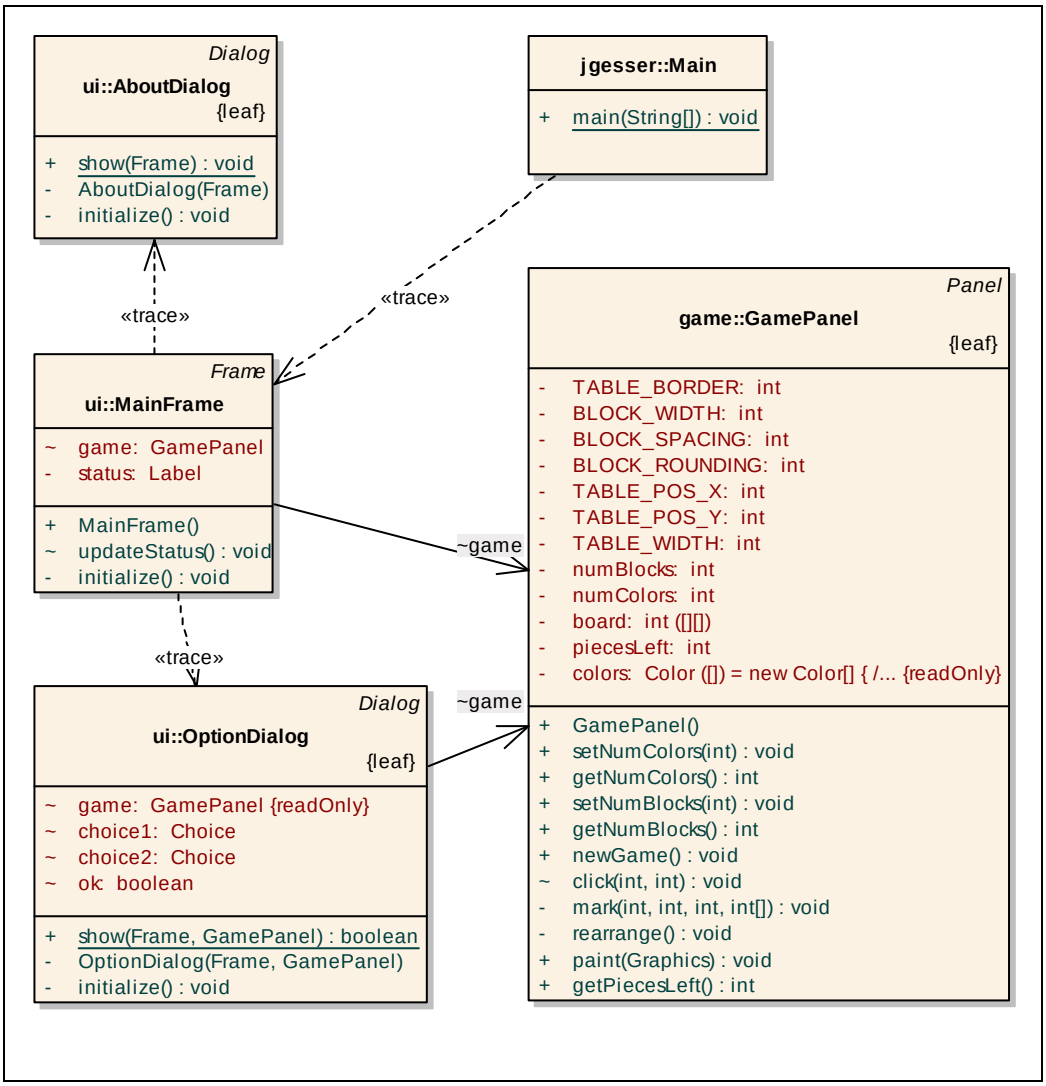
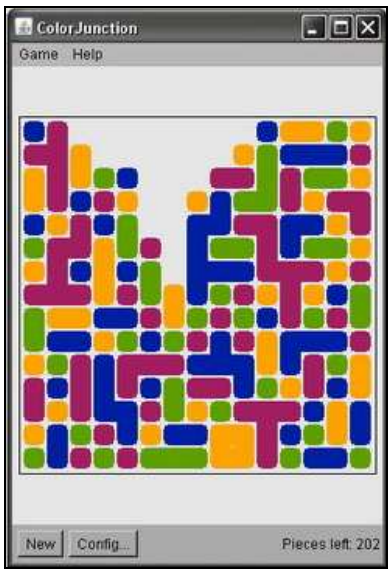
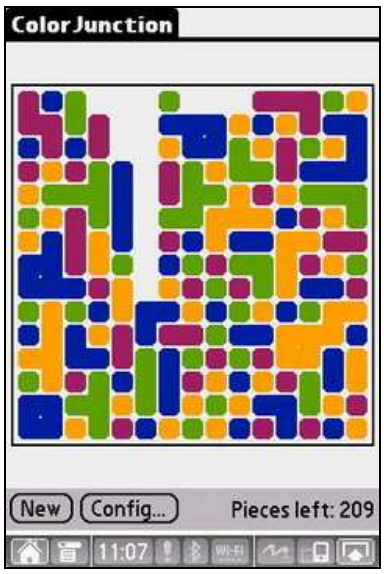


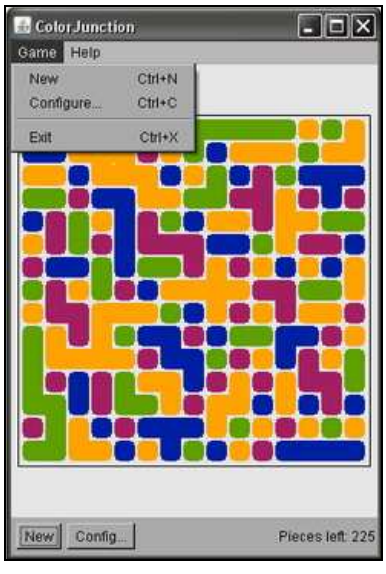
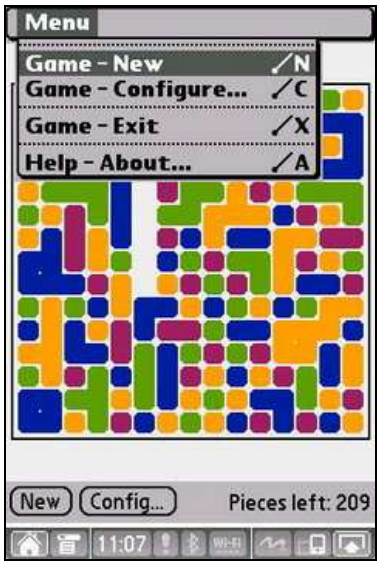
Figura 21 – Diagrama de classes do jogo ColorJunction

As telas do jogo no Palm OS ficaram muito parecidas com as telas do jogo original em Java. O Quadro 25 mostra um comparativo da tela principal do jogo executando em Java e no Palm OS. Esta tela é formada por um `Frame` com `BorderLayout` e um `Panel` com dois `Buttons` e um `Label`.

Java	Palm OS
	

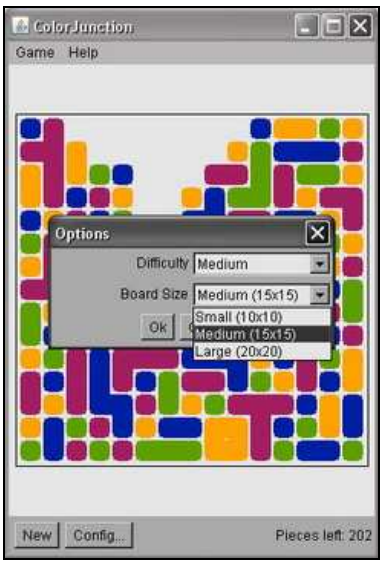
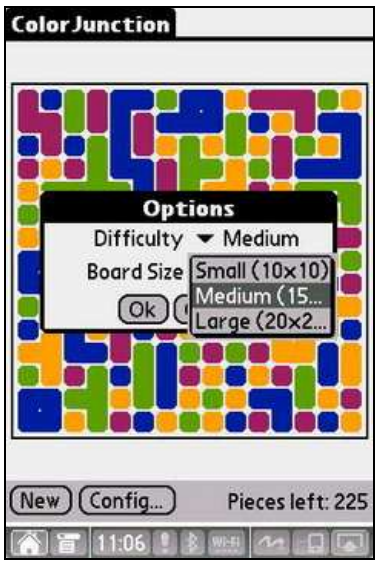
Quadro 25 – Comparativo da tela principal do jogo ColorJunction

O Quadro 26 ilustra as diferenças no menu principal do jogo em Java e no Palm OS.

Java	Palm OS
	

Quadro 26 – Comparativo do menu do jogo ColorJunction

O Quadro 27 mostra um comparativo da tela de opções. Esta tela é composta por um `Dialog` com `GridLayout` e os componentes `Label`, `Choice` e `Button`.

Java	Palm OS
	

Quadro 27 – Comparativo da tela de opções do jogo ColorJunction

Por fim, o Quadro 28 compara a tela com informações sobre o jogo, que é um `Dialog` com `FlowLayout` e quatro `Labels`.

Java	Palm OS
	

Quadro 28 – Comparativo da tela *About* do jogo ColorJunction

3.4 RESULTADOS E DISCUSSÃO

O estudo de caso pode ser considerado simples se for implementado em Java, uma vez que Java possui uma API que simplifica a criação de interfaces gráficas e tratamentos de

eventos de interação com o usuário. Como Java é fortemente orientado a objetos, é muito difícil programar sem ter em mente a reusabilidade que programação orientada a objetos proporciona. No entanto, para implementar o ColorJunction em C++ para Palm OS é necessário muito mais código, já que a API não é orientada a objetos, o que acaba influenciando a não utilização de classes durante o desenvolvimento de um aplicativo.

Um ponto forte do Java, que o diferencia de outras linguagens, é a existência do GC. O GC simplifica consideravelmente o desenvolvimento e ainda evita que muitos erros de programação aconteçam, pois é comum um desenvolvedor alocar memória para um objeto e esquecer de desalocá-la. Em Java não existe esta preocupação, e isto foi alcançado neste compilador. A utilização de *smart pointers* e *reference counting* se mostrou muito eficiente para simular o GC. No estudo de caso não foi detectado nenhum ciclo de referências, tornando a técnica totalmente eficiente para esta implementação.

Desenvolver aplicativos em Java para Palm OS pode ser muito mais produtivo do que desenvolver nativamente em C ou C++. Isto pode ser comprovado no desenvolvimento do estudo de caso utilizado para testar o compilador. A comparação do `HelloWorld` em Java e o `HelloWorld` em C para Palm OS também pode ajudar a comprovar este fato, o primeiro é relativamente mais simples que o segundo e apresenta o mesmo resultado.

O compilador implementado mostrou-se bastante eficiente na detecção de erros léxicos, sintáticos e semânticos. Porém, a compilação é abortada no primeiro erro encontrado, ou seja, um erro é mostrado de cada vez. A geração de código C++ também pode ser considerada eficiente, pois foi capaz de gerar código legível e de fácil entendimento.

As bibliotecas do Java implementadas em C++ foram capazes de reproduzir com perfeição o comportamento das mesmas em Java. O comparativo entre as telas do estudo de caso executando em Java e executando no Palm OS pode deixar isto claro. A única diferença perceptível foi o tamanho da fonte, que no Palm OS é maior.

4 CONCLUSÕES

Desenvolver um compilador para uma linguagem orientada a objetos não é uma tarefa simples. Mesmo utilizando ferramentas que auxiliam o desenvolvimento dos analisadores léxico e sintático, como o gerador de *parser* JavaCC, a implementação da análise semântica e da geração de código, entre outras tarefas, exige bastante dedicação.

O objetivo principal deste trabalho foi alcançado com sucesso. O compilador desenvolvido mostrou-se capaz de detectar os erros léxicos, sintáticos e semânticos nos programas nele compilados. Os erros léxicos e sintáticos são detectados pelo *parser* gerado pela ferramenta JavaCC. Apesar de não ter sido possível desenvolver toda a análise semântica do Java, boa parte foi implementada. Tem-se que uma das metas da análise semântica seria avisar o desenvolvedor sobre bibliotecas utilizadas no programa Java que não foram implementadas em C++, objetivo este que não foi alcançado. No entanto, as técnicas utilizadas no desenvolvimento do analisador semântico podem servir de base para trabalhos futuros na área de compiladores.

A publicação da gramática e da AST implementadas para a comunidade de desenvolvedores foi um fator importante para este trabalho. Isto permitiu melhorias na qualidade da gramática e ajudou outros desenvolvedores com necessidades similares.

O gerador de código C++ é capaz de gerar código formatado e sem perder as características do código original. O suporte ao GC foi de grande importância para este trabalho, pois sem ele não seria possível transformar código Java em código C++ funcionalmente equivalente.

As bibliotecas da API do Java escritas em C++ conseguiram obter êxito no seu propósito, permitindo execução de interfaces gráficas do AWT no Palm OS sem perder muitas características em relação ao Java. Isto permite afirmar que é possível desenvolver outros componentes gráficos sem grandes dificuldades.

Por fim, pode-se concluir que o desenvolvimento de um compilador para a linguagem Java que gere código C++ nativo para Palm OS é viável. Este compilador foi implementado visando facilitar o desenvolvimento de software para Palm OS. E, de fato, este trabalho pode comprovar que em partes isto é possível, pois desenvolver aplicativos em Java é bem mais simples do que codificar o aplicativo nativo similar para Palm OS em C ou C++.

4.1 EXTENSÕES

Não foi possível concluir o desenvolvimento de toda a especificação do Java neste trabalho, e apenas algumas bibliotecas da API foram implementadas. Deste modo, algumas melhorias são propostas, tais como:

a) concluir o analisador semântico e o gerador de código, dando suporte a todas as construções do Java 5.0;
b) transcrever os comentários das classes Java para as classes C++ geradas, permitindo uma melhor leitura dos fontes gerados;
c) implementar em C++ o restante das classes da biblioteca AWT, possibilitando a compilação de programas mais complexos;
d) codificar outras classes da API do Java em C++, tais como: `Collections`, `Sockets`, `Remote Method Invocation` (RMI), JDBC, `Swing`, entre outras;
e) melhorar o GC desenvolvido para que detecte e colete ciclos;
f) suportar a utilização de reflexão do Java, permitindo carregamento e chamadas dinâmicas de classes;
g) escrever as classes da API do Java para outras plataformas, permitindo que o compilador desenvolvido gere executável nativo para outras plataformas.

REFERÊNCIAS BIBLIOGRÁFICAS

ACCESS. **Palm OS Developer Suite overview**. [S.l.], 2005. Disponível em: <http://www.access-company.com/developers/documents/docs/dev_suite/PalmOSDevSuite/Tools_Overview.html>. Acesso em: 02 maio 2007.

_____. **Why Garnet OS?** [S.l.], 2007. Disponível em: <http://www.access-company.com/products/garnet/whygarnetos/index.html>. Acesso em: 30 abr. 2007.

AHO, A. V.; SETHI, R.; ULLMAN, J. D. **Compiladores**: princípios, técnicas e ferramentas. Tradução Daniel de Ariosto Pinto. Rio de Janeiro: LTC, 1995.

ANTLR. **ANTLR parser generator**. [S.l.], 2007. Disponível em: <http://www.antlr.org>. Acesso em: 24 jul. 2007.

AUSTIN, C. **J2SE 5.0 in a nutshell**. [S.l.], 2004. Disponível em: <http://java.sun.com/developer/technicalArticles/releases/j2se15/>. Acesso em: 30 abr. 2007.

BACHMANN, G.; FOSTER, L. R. **Professional Palm OS programming**. Indianapolis: Wiley Publishing, 2005.

COOPER, J. W. **Design patterns Java companion**. Boston: Addison-Wesley, 1998.

DEITEL, H. M.; DEITEL, P. J. **C++**: how to program. 2. ed. Nova Jersey: Prentice Hall, 1997.

_____. **Java**: como programar. 4. ed. Tradução Carlos Arthur Lang Lisbôa. Porto Alegre: Bookman, 2003.

GAMMA, E. et al. **Design pattern CD**: elements of reusable object-oriented software. Boston: Addison-Wesley, 1998.

GESSER, J. V. **Gerando programas nativos Palm OS com Java SE 5.0**. [S.l.], 2007. Disponível em: < http://www.pdaexpert.net/artigos/palm-os/gerando-programas-nativos-palm-os-com-java-se-50/>. Acesso em: 27 maio 2007.

GOSLING, J. et al. **The Java language specification**. 3rd. ed. Boston: Addison-Wesley, 2005.

GOSLING, J.; MCGILTON, H. **The Java language environment**: contents. [S.l.], 1996. Disponível em: <http://java.sun.com/docs/white/langenv/>. Acesso em: 30 abr. 2007.

GRUNE, D. et al. **Projeto moderno de compiladores**: implementação e aplicações. Tradução Vandenberg D. de Souza. Rio de Janeiro: Campus, 2001.

HEWGILL, G. **Jump**. [S.l.], 1997. Disponível em: <http://hewgill.com/pilot/jump/index.html>. Acesso em: 08 maio 2007.

HILDEBRANDT, E. **Estudo do processo de desenvolvimento em palm usando C++ na plataforma Linux**. 2003. 50 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) - Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

JAVACC. **JavaCC features**. [S.l.], 2007. Disponível em: <https://javacc.dev.java.net/doc/features.html>. Acesso em: 30 abr. 2007.

JJTREE. **JJTree reference**. [S.l.], 2007. Disponível em: <https://javacc.dev.java.net/doc/JJTree.html>. Acesso em: 24 jul. 2007.

KLEBERHOFF, R.; DICKERSON P. **Jump**. [S.l.], 2004. Disponível em: <http://jump.sourceforge.net/>. Acesso em: 08 maio 2007.

LEYENDECKER, G. Z. **Especificação e compilação de uma linguagem de programação orientada a objetos para a plataforma Microsoft .NET**. 2005. 89 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) - Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

LOUDEN, K. **Compiladores:** princípios e práticas. Tradução Flávio Soares Corrêa da Silva. São Paulo: Pioneira Thomson Learning, 2004.

MEMORY leak. In: WIKIPEDIA, a enciclopédia livre. [S.l.]: Wikimedia Foundation, 2007. Disponível em: <http://pt.wikipedia.org/wiki/Memory_leak>. Acesso em: 23 maio 2007.

PROGRAMICS.COM. **Java to C++ source translator**. [S.l.], 2003. Disponível em: <http://www.programics.com/?page=java2cpp>. Acesso em: 08 maio 2007.

REFERENCE counting. In: WIKIPEDIA, a enciclopédia livre. [S.l.]: Wikimedia Foundation, 2007. Disponível em: < http://en.wikipedia.org/wiki/Reference_counting>. Acesso em: 25 maio 2007.

SABLECC. **SableCC**. [S.l.], 2007. Disponível em: <http://sablecc.org>. Acesso em: 24 jul. 2007.

SEBESTA, R. W. **Conceitos de linguagens de programação**. 4. ed. Tradução José Carlos Barbosa dos Santos. Porto Alegre: Bookman, 2000.

SMART pointer. In: WIKIPEDIA, a enciclopédia livre. [S.l.]: Wikimedia Foundation, 2007. Disponível em: <http://en.wikipedia.org/wiki/Smart_pointer>. Acesso em: 25 maio 2007.

SUN MICROSYSTEMS. **The Java Virtual Machine specification**. 2nd. ed. [S.l.], 1999. Disponível em: <http://java.sun.com/docs/books/jvms/second_edition/html/Introduction.doc.html>. Acesso em: 30 abr. 2007.

_____. **JDBC**. [S.l.], 2002. Disponível em: <http://java.sun.com/javase/6/docs/technotes/guides/jdbc/index.html>. Acesso em: 30 abr. 2007.

_____. **Abstract Window Toolkit (AWT)**. [S.l.], 2005a. Disponível em: <http://java.sun.com/javase/6/docs/technotes/guides/awt/index.html>. Acesso em: 30 abr. 2007.

_____. **Java programming language enhancements**. [S.l.], 2005b. Disponível em: <http://java.sun.com/javase/6/docs/technotes/guides/language/enhancements.html>. Acesso em: 30 abr. 2007.

_____. **The Java language specification**. 3rd. ed. [S.l.], 2005c. Disponível em: <http://java.sun.com/docs/books/jls/third_edition/html/j3TOC.html>. Acesso em: 12 maio 2007.

_____. **The collections framework**. [S.l.], 2006. Disponível em: <http://java.sun.com/javase/6/docs/technotes/guides/collections/index.html>. Acesso em: 30 abr. 2007.

_____. **About Java technology**. [S.l.], 2007a. Disponível em: <http://www.sun.com/java/about>. Acesso em: 30 abr. 2007.

_____**. Java SE technologies at a glance**. [S.l.], 2007b. Disponível em: <http://java.sun.com/javase/technologies/index.jsp>. Acesso em: 30 abr. 2007.

VAREJÃO, F. M. **Linguagens de programação**: conceitos e técnicas. Rio de Janeiro: Elsevier, 2004.

VISWANADHA, S. **Initial version of Java 1.5 grammar that will be eventually distributed with JavaCC**. [S.l.], 2004. Disponível em: <https://javacc.dev.java.net/servlets/ProjectDocumentView?documentID=3131&showInfo=true>. Acesso em: 16 maio 2007.

WATT, D. A.; BROWN, D. F. **Programming language processors in Java**. Harlow: Person Education Limited, 2000.

WILDING-MCBRIDE, D. **Java development on PDAs**: building applications for PocketPC and Palm devices. Boston: Addison-Wesley, 2003.

WILHELM, R; MAURER, D. **Compiler design**. Wokingham: Addison-Wesley, 1995.

APÊNDICE A – Gramática do Java 5.0

No Quadro 29 é apresentada a gramática da linguagem Java 5.0 utilizada no compilador no formato EBNF.

```
CompilationUnit ::= ( PackageDeclaration )? ( ImportDeclaration )* ( TypeDeclaration )* <EOF>
PackageDeclaration ::= "package" Name ";"
ImportDeclaration ::= "import" ( "static" )? Name ( "." "*" )? ";"
Modifiers ::= ( ( "public" | "static" | "protected" | "private" | "final" | "abstract" |
         "synchronized" | "native" | "transient" | "volatile" | "strictfp" | Annotation ) )*
TypeDeclaration ::= ";"
 | Modifiers ( ClassOrInterfaceDeclaration | EnumDeclaration | AnnotationTypeDeclaration )
ClassOrInterfaceDeclaration ::= ( "class" | "interface" ) <IDENTIFIER> ( TypeParameters )? (
         ExtendsList )? ( ImplementsList )? ClassOrInterfaceBody
ExtendsList ::= "extends" ClassOrInterfaceType ( "," ClassOrInterfaceType )*
ImplementsList ::= "implements" ClassOrInterfaceType ( "," ClassOrInterfaceType )*
EnumDeclaration ::= "enum" <IDENTIFIER> ( ImplementsList )? "{" ( EnumConstant ( ","
         EnumConstant )* )? ( "," | ( ";" ( ClassOrInterfaceBodyDeclaration )* ) )? "}"
EnumConstant ::= <IDENTIFIER> ( Arguments )? ( ClassOrInterfaceBody )?
TypeParameters ::= "<" TypeParameter ( "," TypeParameter )* ">"
TypeParameter ::= <IDENTIFIER> ( TypeBound )?
TypeBound ::= "extends" ClassOrInterfaceType ( "&" ClassOrInterfaceType )*
ClassOrInterfaceBody ::= "{" ( ClassOrInterfaceBodyDeclaration )* "}"
ClassOrInterfaceBodyDeclaration ::= ( Initializer | Modifiers ( ClassOrInterfaceDeclaration |
         EnumDeclaration | AnnotationTypeDeclaration | ConstructorDeclaration |
         FieldDeclaration | MethodDeclaration ) | ";" )
FieldDeclaration ::= Type VariableDeclarator ( "," VariableDeclarator )* ";"
VariableDeclarator ::= VariableDeclaratorId ( "=" VariableInitializer )?
VariableDeclaratorId ::= <IDENTIFIER> ( "[" "]" )*
VariableInitializer ::= ( ArrayInitializer | Expression )
ArrayInitializer ::= "{" ( VariableInitializer ( "," VariableInitializer )* )? ( "," )? "}"
MethodDeclaration ::= ( TypeParameters )? ResultType <IDENTIFIER> FormalParameters ( "[" "]"
         )* ( "throws" NameList )? ( Block | ";" )
FormalParameters ::= "(" ( FormalParameter ( "," FormalParameter )* )? ")"
FormalParameter ::= Modifiers Type ( "..." )? VariableDeclaratorId
ConstructorDeclaration ::= ( TypeParameters )? <IDENTIFIER> FormalParameters ( "throws"
         NameList )? "{" ( ExplicitConstructorInvocation )? Statements "}"
ExplicitConstructorInvocation ::= ( "this" Arguments ";" | ( PrimaryExpression "." )? "super"
         Arguments ";" )
Statements ::= ( BlockStatement )*
Initializer ::= ( "static" )? Block
Type ::= ( ReferenceType | PrimitiveType )
ReferenceType ::= ( PrimitiveType ( "[" "]" )+ | ClassOrInterfaceType ( "[" "]" )* )
ClassOrInterfaceType ::= <IDENTIFIER> ( TypeArguments )? ( "." <IDENTIFIER> ( TypeArguments )?
         )*
TypeArguments ::= "<" TypeArgument ( "," TypeArgument )* ">"
TypeArgument ::= ( ReferenceType | Wildcard )
Wildcard ::= "?" ( "extends" ReferenceType | "super" ReferenceType )?
PrimitiveType ::= ( "boolean" | "char" | "byte" | "short" | "int" | "long" | "float" |
```

```
         "double" )
ResultType ::= ( "void" | Type )
Name ::= <IDENTIFIER> ( "." <IDENTIFIER> )*
NameList ::= Name ( "," Name )*
Expression ::= ConditionalExpression ( AssignmentOperator Expression )?
AssignmentOperator ::= ( "=" | "*=" | "/=" | "%=" | "+=" | "-=" | "<<=" | ">>=" | ">>>=" |
         "&=" | "^=" | "|=" )
ConditionalExpression ::= ConditionalOrExpression ( "?" Expression ":" Expression )?
ConditionalOrExpression ::= ConditionalAndExpression ( "||" ConditionalAndExpression )*
ConditionalAndExpression ::= InclusiveOrExpression ( "&&" InclusiveOrExpression )*
InclusiveOrExpression ::= ExclusiveOrExpression ( "|" ExclusiveOrExpression )*
ExclusiveOrExpression ::= AndExpression ( "^" AndExpression )*
AndExpression ::= EqualityExpression ( "&" EqualityExpression )*
EqualityExpression ::= InstanceOfExpression ( ( "==" | "!=" ) InstanceOfExpression )*
InstanceOfExpression ::= RelationalExpression ( "instanceof" Type )?
RelationalExpression ::= ShiftExpression ( ( "<" | ">" | "<=" | ">=" ) ShiftExpression )*
ShiftExpression ::= AdditiveExpression ( ( "<<" | RSIGNEDSHIFT | RUNSIGNEDSHIFT )
         AdditiveExpression )*
AdditiveExpression ::= MultiplicativeExpression ( ( "+" | "-" ) MultiplicativeExpression )*
MultiplicativeExpression ::= UnaryExpression ( ( "*" | "/" | "%" ) UnaryExpression )*
UnaryExpression ::= ( ( "+" | "-" ) UnaryExpression | PreIncrementExpression |
         PreDecrementExpression | UnaryExpressionNotPlusMinus )
PreIncrementExpression ::= "++" PrimaryExpression
PreDecrementExpression ::= "--" PrimaryExpression
UnaryExpressionNotPlusMinus ::= ( ( "~" | "!" ) UnaryExpression | CastExpression |
         PostfixExpression )
CastLookahead ::= "(" PrimitiveType
 | "(" Type "[" "]"
 | "(" Type ")" ( "~" | "!" | "(" | <IDENTIFIER> | "this" | "super" | "new" | Literal )
PostfixExpression ::= PrimaryExpression ( ( "++" | "--" ) )?
CastExpression ::= ( "(" Type ")" UnaryExpression | "(" Type ")" UnaryExpressionNotPlusMinus )
PrimaryExpression ::= PrimaryPrefix ( PrimarySuffix )*
PrimaryPrefix ::= ( Literal | "this" | "super" "." <IDENTIFIER> ( Arguments )? | "("
         Expression ")" | AllocationExpression | ResultType "." "class" | <IDENTIFIER> (
         Arguments )? )
PrimarySuffix ::= ( "." ( "this" | "super" | AllocationExpression | ( TypeArguments )?
         <IDENTIFIER> ( Arguments )? ) | "[" Expression "]" )
Literal ::= ( <INTEGER_LITERAL> | <LONG_LITERAL> | <FLOATING_POINT_LITERAL> |
         <CHARACTER_LITERAL> | <STRING_LITERAL> | BooleanLiteral | NullLiteral )
BooleanLiteral ::= ( "true" | "false" )
NullLiteral ::= "null"
Arguments ::= "(" ( ArgumentList )? ")"
ArgumentList ::= Expression ( "," Expression )*
AllocationExpression ::= ( "new" PrimitiveType ArrayDimsAndInits | "new" ClassOrInterfaceType
         ( TypeArguments )? ( ArrayDimsAndInits | Arguments ( ClassOrInterfaceBody )? ) )
ArrayDimsAndInits ::= ( ( "[" Expression "]" )+ ( "[" "]" )* | ( "[" "]" )+ ArrayInitializer )
Statement ::= ( LabeledStatement | AssertStatement | Block | EmptyStatement |
         StatementExpression | SwitchStatement | IfStatement | WhileStatement | DoStatement |
         ForStatement | BreakStatement | ContinueStatement | ReturnStatement | ThrowStatement
         | SynchronizedStatement | TryStatement )
AssertStatement ::= "assert" Expression ( ":" Expression )? ";"
```

```
LabeledStatement ::= <IDENTIFIER> ":" Statement
Block ::= "{" Statements "}"
BlockStatement ::= ( Modifiers ClassOrInterfaceDeclaration | VariableDeclarationExpression ";"
         | Statement )
VariableDeclarationExpression ::= Modifiers Type VariableDeclarator ( "," VariableDeclarator
         )*
EmptyStatement ::= ";"
StatementExpression ::= ( PreIncrementExpression | PreDecrementExpression | PrimaryExpression
         ( "++" | "--" | AssignmentOperator Expression )? ) ";"
SwitchStatement ::= "switch" "(" Expression ")" "{" ( SwitchEntry )* "}"
SwitchEntry ::= ( "case" Expression | "default" ) ":" Statements
IfStatement ::= "if" "(" Expression ")" Statement ( "else" Statement )?
WhileStatement ::= "while" "(" Expression ")" Statement
DoStatement ::= "do" Statement "while" "(" Expression ")" ";"
ForStatement ::= "for" "(" ( VariableDeclarationExpression ":" Expression | ( ForInit )? ";" (
         Expression )? ";" ( ForUpdate )? ) ")" Statement
ForInit ::= ( VariableDeclarationExpression | ExpressionList )
ExpressionList ::= Expression ( "," Expression )*
ForUpdate ::= ExpressionList
BreakStatement ::= "break" ( <IDENTIFIER> )? ";"
ContinueStatement ::= "continue" ( <IDENTIFIER> )? ";"
ReturnStatement ::= "return" ( Expression )? ";"
ThrowStatement ::= "throw" Expression ";"
SynchronizedStatement ::= "synchronized" "(" Expression ")" Block
TryStatement ::= "try" Block ( "catch" "(" FormalParameter ")" Block )* ( "finally" Block )?
RUNSIGNEDSHIFT ::= ( ">" ">" ">" )
RSIGNEDSHIFT ::= ( ">" ">" )
Annotation ::= ( NormalAnnotation | SingleMemberAnnotation | MarkerAnnotation )
NormalAnnotation ::= "@" Name "(" ( MemberValuePairs )? ")"
MarkerAnnotation ::= "@" Name
SingleMemberAnnotation ::= "@" Name "(" MemberValue ")"
MemberValuePairs ::= MemberValuePair ( "," MemberValuePair )*
MemberValuePair ::= <IDENTIFIER> "=" MemberValue
MemberValue ::= ( Annotation | MemberValueArrayInitializer | ConditionalExpression )
MemberValueArrayInitializer ::= "{" MemberValue ( "," MemberValue )* ( "," )? "}"
AnnotationTypeDeclaration ::= "@" "interface" <IDENTIFIER> AnnotationTypeBody
AnnotationTypeBody ::= "{" ( AnnotationBodyDeclaration )* "}"
AnnotationBodyDeclaration ::= Modifiers ( ( AnnotationTypeMemberDeclaration |
         ClassOrInterfaceDeclaration | EnumDeclaration | AnnotationTypeDeclaration |
         FieldDeclaration ) | ";" )
AnnotationTypeMemberDeclaration ::= Type <IDENTIFIER> "(" ")" ( DefaultValue )? ";"
DefaultValue ::= "default" MemberValue
```

Quadro 29 – Gramática do Java 5.0

APÊNDICE B – Bibliotecas Java implementadas em C++

Os diagramas a seguir apresentam as classes C++ implementadas a partir das classes da API do Java.

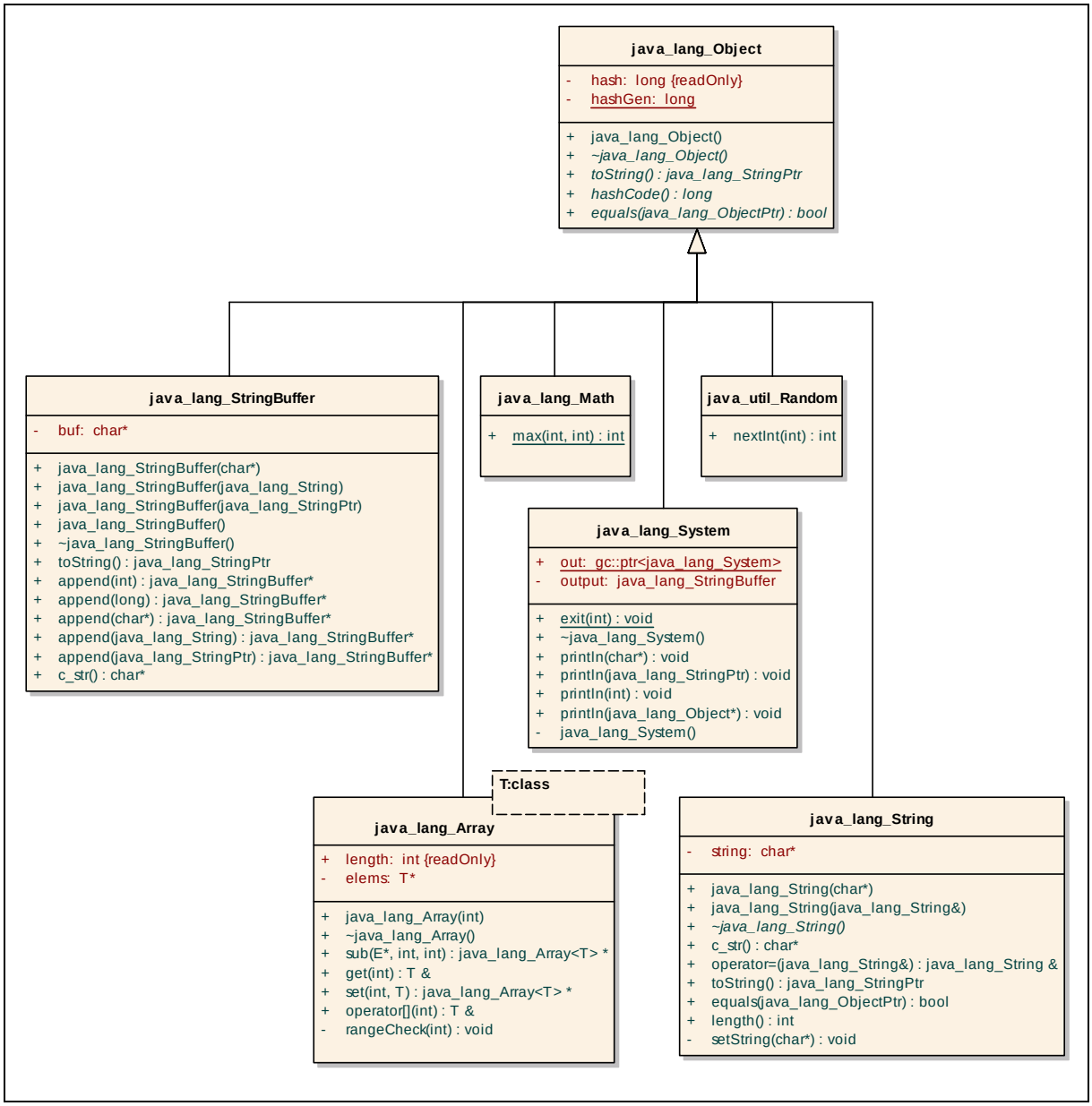


Figura 22 – Diagrama de classe dos pacotes `java.lang` e `java.util`

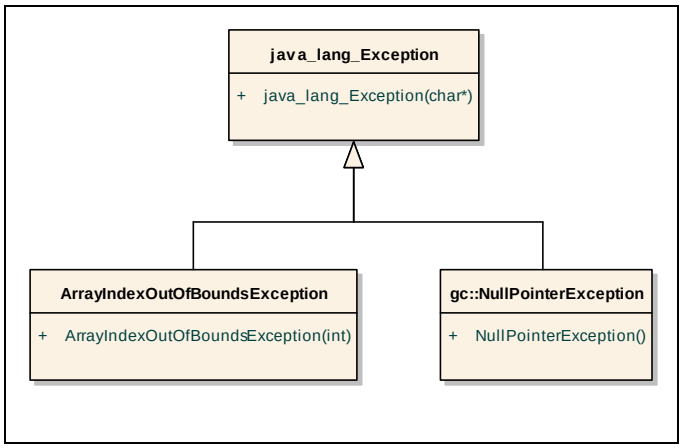


Figura 23 – Diagrama de classe de exceções

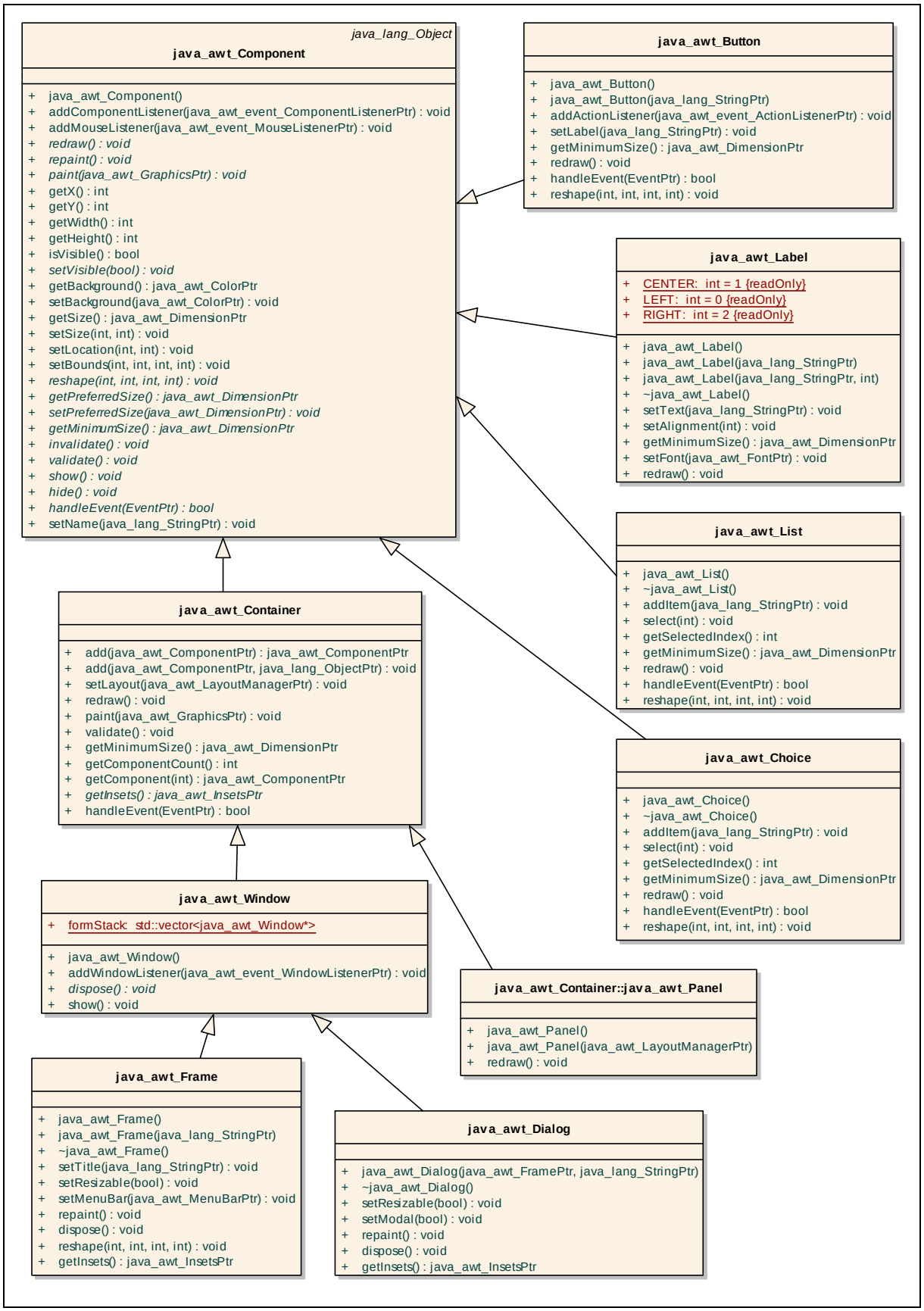


Figura 24 – Diagrama de classe de interface gráfica do AWT

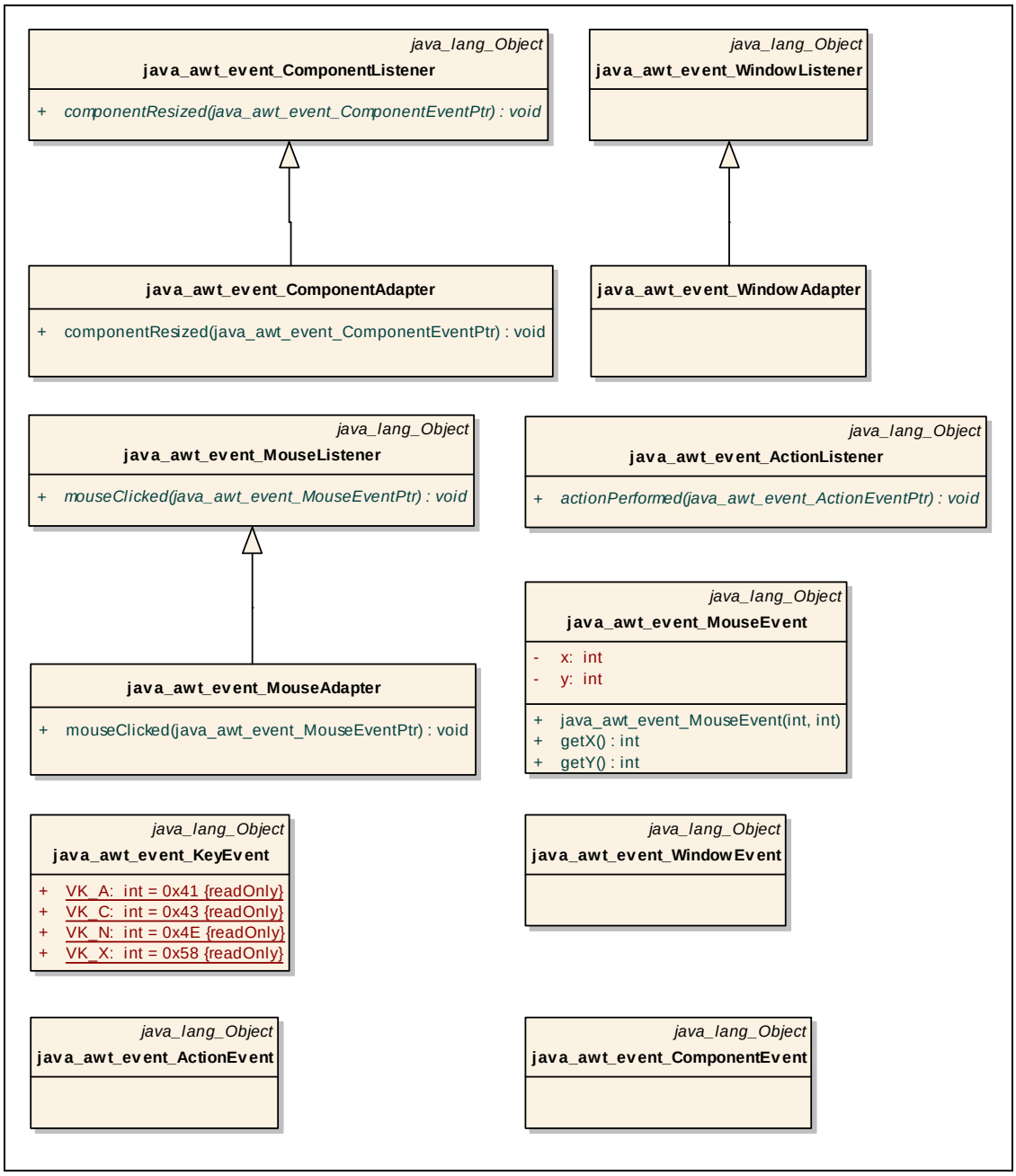


Figura 25 – Diagrama classes de tratamento de eventos do AWT

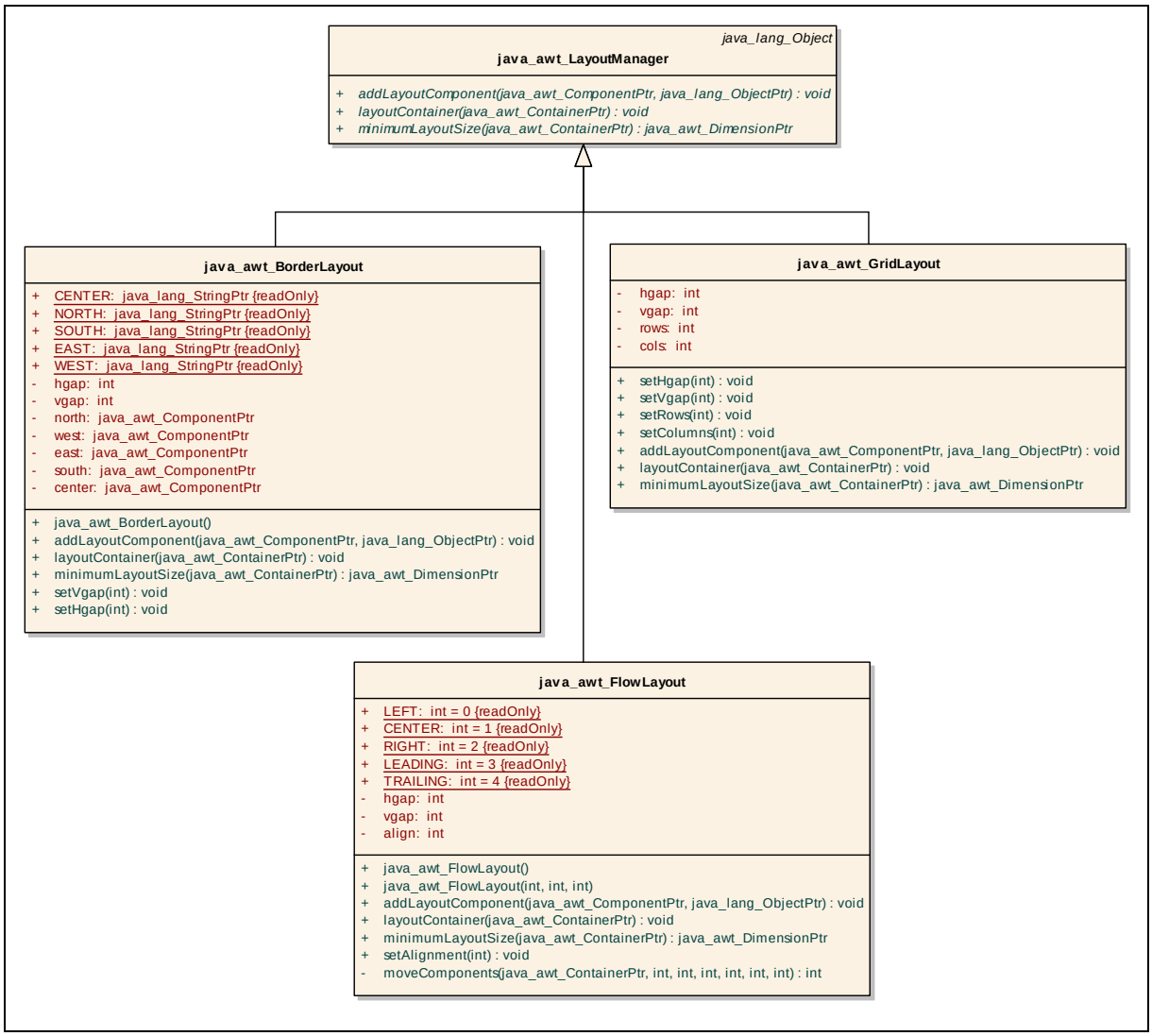


Figura 26 – Diagrama classes de gerenciamento de *layout* do AWT

Figura 27 – Diagrama classes para criação de menus do AWT

Figura 28 – Diagrama classes utilitárias do AWT