UNIVERSIDADE FEDERAL FLUMINENSE
INSTITUTO DE COMPUTAÇÃO

BERNARDO BREDER

# LINGUAGEM DE PROGRAMAÇÃO BREDER

NITERÓI
2011

BERNARDO BREDER

# LINGUAGEM DE PROGRAMAÇÃO BREDER

Trabalho de Conclusão de Curso Apresentado ao Curso de Graduação em Ciência da Computação da Universidade Federal Fluminense para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. LUIZ CARLOS CASTRO GUEDES

NITERÓI
2011

BERNARDO BREDER

LINGUAGEM DE PROGRAMAÇÃO CHAMADA BREDER LANGUAGE

Trabalho de Conclusão de Curso Apresentado ao Curso de Graduação em Ciência da Computação da Universidade Federal Fluminense para obtenção do grau de Bacharel em Ciência da Computação.

Data da Aprovação: _____

BANCA EXAMINADORA

_____

Prof. Dr. LUIZ CARLOS CASTRO GUEDES - Orientador
UFF

_____

Prof. Dr. CARLOS ROBERTO SERRA PINTO CASSINO
PUC-RIO

_____

Prof. Dr. ESTEBAN WALTER GONZALEZ CLUA
UFF

NITERÓI 2011

# AGRADECIMENTOS

Aos meus pais, Giovanni Gargano Breder e Vanda Neves Breder, e ao meu irmão Raphael Breder, pois sem eles eu não teria forças para chegar ao fim.

Ao professor Andre Leiradella, por todo conhecimento básico fornecido para a realização deste projeto.

A Renata Guedes por me ajudar em todo o processo, me incentivando nos momentos mais difíceis.

Aos professores do Instituto de Computação, em especial, Esteban Clua, Isabel Cafezeiro, Alexandre Plastino, Luiz Carlos Castro Guedes, Leonardo Murta pelos ensinamentos, convívio e amizade.

Aos profissionais do Tecgraf, em especial, Carlos Cassino, Leonardo Barros, Maria Julia, Pedro Asti, Aldo Nogueira, pelas discussões sobre o trabalho.

Aos amigos Bruno Moreira, Heraldo Borges, Marcelo Niero, Giancarlo Taveira, Felipe Rolim, Leonardo Freitas, Vitor Borner e outros por todo apoio, conselhos, ajuda e compreensão.

Agradeço também a Universidade Federal Fluminense - UFF - por todo o lazer, cultura e educação que me foi fomentada.

Meus profundos e mais sinceros agradecimentos a todas essas pessoas pela conclusão deste curso. Sem as quais seria impossível sua realização.

# RESUMO

Diferentes tipos de projetos exigem linguagens de programação especificas. Estas devem fornecer ferramentas adequadas para uma melhor implementação. Alguns projetos visam uma melhor modularização de sua arquitetura, focado em um desenvolvimento de alto nível e outros exigem um melhor desempenho de execução, usando recursos de baixo nível.

A linguagem de programação Breder foi desenvolvida visando atender a projetos que necessitam de desenvolvimento alto nível e baixo nível com a otimização dessas interligações. A monografia descreve as características da linguagem de programação Breder e como utilizá-la.

Ainda foram descritas as futuras melhorias a serem implementadas na linguagem Breder, adicionando novas funcionalidades e otimizando o tempo de processamento.

Palavras-chaves: Linguagem de programação Breder, alto nível, baixo nível, orientação a objeto.

# ABSTRACT

Projects with specific requirements require specific programming languages. These programming languages should provide tools for a better implementation. Some projects aim at improving the modularity of its architecture, focusing on high level development while others require a better runtime performance, using low level programming features.

Breder - The programming language was designed to meet the requirements of projects that require high level and low level development by optimizing the interconnections between those levels. This work describes the characteristics of the Breder programming language and how to use it.

Moreover, it describes future improvements to the implementation of the Breder language, proposing new functionalities and compile-time optimizations.



Keywords: Programming language Breder, high level, low level, object orientation.

LISTA DE ABREVIATURAS E SIGLAS

BDK: Breder Development Kit

BNI: Breder Native Interface

BVM: Breder Virtual Machine

JIT: Just In Time

API: Application Programming Interface

SUMÁRIO

# 1 INTRODUÇÃO

Muitos programadores no curso de suas faculdades tiveram poucas orientações aos estudos de linguagem de programação, por conta disso aprenderam a programar em ambientes específicos por conta própria ou em programas de treinamento de suas empresas. Tais programas focam o ensino apenas em linguagens diretamente relacionadas aos projetos atuais das empresas [1].

As linguagens que foram aprendidas por muitas vezes passam a ser obsoletas, sendo inutilizadas em alguns projetos atuais. O resultado, é que muitos programadores, quando podem escolher a linguagem para um novo projeto, optam usar aquelas com a qual estão mais familiarizados, mesmo que elas sejam inadequadas aos projetos. Se esses programadores conhecessem uma faixa mais ampla de linguagens e suas construções, estariam mais capacitados a escolher a que inclui os melhores recursos adaptados às características do problema em questão [1].

As metodologias de um projeto, ferramentas de desenvolvimento de software e linguagens de programação estão em constante evolução. Tornando o desenvolvedor de software uma profissão satisfatória, mas exigindo aprendizado contínuo. O processo de aprender uma nova linguagem de programação pode ser longo e difícil, especialmente para alguém que esteja confortável com apenas uma ou duas linguagens. Uma vez que um entendimento preciso dos conceitos fundamentais das linguagens tenham sido adquiridos, fica mais fácil ver como esses conceitos são incorporados no projeto da linguagem em questão. [1]

Normalmente encontramos dificuldade em finalizar uma tarefa devido a ausência de ferramentas que facilitam a sua implementação. Este cenário também pode ser aplicado à área de desenvolvimento de software, uma vez que a linguagem ideal pode não estar disponível, ou pode não ser de domínio da equipe de desenvolvimento, ou ainda, pode não existir. Assim, para implementar algumas tarefas somos obrigado a codificar com uma linguagem disponível, mesmo sabendo que esta não seja ideal para as necessidades do projeto em questão [3].

Para a codificação de uma aplicação complexo, é possível que tenhamos de manusear diferentes problemas de programação. Considerando que cada um destes problemas poderia ser mais facilmente resolvido com a utilização de uma determinada linguagem de

programação, idealmente poderíamos particionar o problema em segmentos, cada qual associado à linguagem mais apropriada [3].

Assim, quando a implementação da solução é feita com uma única linguagem de programação, o programador usualmente modela as soluções de diversos problemas com os construtos disponíveis na linguagem de programação. Temos neste caso, uma influência direta da linguagem na definição da solução do problema, o que usualmente pode ser constatado quando programadores que trabalham há muitos anos com uma determinada linguagem de programação, quase sempre, têm dificuldades em se adaptar à novos paradigmas de programação, uma vez que sua forma de pensar está muito atrelada ao paradigma corrente [3].

Podemos definir a programação multi-linguagem como uma técnica que permite o programador implementar o código através do uso das mais adequadas linguagens ao problema, de tal forma que a implementação da solução possa considerar diversos estilos de programação [3].

Esse trabalho tem como objetivo especificar e apresentar uma proposta de implementação de um ambiente de programação que viabilize o emprego da programação multi-linguagem, através do oferecimento de primitivas de ambiente que facilitem a interface entre os diversos segmentos de linguagens que compõem a aplicação [3].

O capítulo 2 irá abordar aspecto de uma linguagem de programação orientado a objeto, focando conceitos principais da linguagem de programação implementada por esse projeto. No capítulo 3 será discutida a proposta da linguagem de programação. No capítulo 4 será comentado características dessa linguagem de programação perante as outras do mercado. No capítulo 5 será comentada a forma de implementação adotada nesse projeto de linguagem de programação. No capítulo 6 serão discutidos os resultados obtidos. Por fim, no capítulo 7 será concluído o trabalho e também relacionado futuras implementações necessárias.

# 2 CONCEITOS DE LINGUAGEM DE PROGRAMAÇÃO

Esse capítulo irá abordar alguns aspectos de uma linguagem de programação tipicamente orientada a objeto.

## 2.1 CLASSE

Uma classe representa a definição de um objeto criado a partir dela. A definição de uma classe descreve os atributos e operações dos objetos dessa classe.

Uma classe oferece estruturas de dados ou campos e operações capazes de estruturar e organizar um conceito. Por exemplo, podemos estruturar o conceito "carro" como uma classe de nome "Carro", que poderá ter características e operações que dizem respeito a um carro, como por exemplo, os dados que representam as quatro rodas e as ações de acelerar, frear. A classe permite a criação de instâncias que têm as mesmas características conceituais, porém com valores possivelmente diferentes. Por exemplo, se a classe definir que um carro possui uma cor, será possível criar vários objetos diferentes, com cores diferentes.

Além da especificação de atributos, a definição de uma classe descreve também o comportamento dos objetos da classe, ou seja, que funcionalidades podem ser aplicadas a objetos da classe. Essas funcionalidades são descritas através de métodos [6].

Normalmente, a especificação de uma classe é composta por três regiões: o nome da classe, o conjunto de atributos dela e o conjunto de métodos [6].

O nome da classe é um identificador para a classe, que permite referenciá-la posteriormente, por exemplo, no momento da criação de um objeto.

O conjunto de atributos descreve as propriedades da classe. Cada atributo é identificado por um nome e tem um tipo associado. Em uma linguagem de programação orientada a objetos, o tipo é o nome de uma classe. Na prática, a maior parte das linguagens de programação orientada a objetos oferecem um grupo de tipos primitivos, como inteiro, real e caráter, que podem ser usados na descrição de atributos. O atributo pode ainda ter um padrão opcional, que especifica um valor inicial para o atributo [6].

Os métodos definem as funcionalidades da classe, ou seja, o que será possível fazer com objetos dessa classe. Cada método é especificado por uma assinatura, composta por

um identificador para o método, o tipo para o valor de retorno e sua lista de argumentos, sendo cada argumento identificado por seu tipo e nome [6].

Através do mecanismo de sobrecarga, dois métodos de uma classe podem ter o mesmo nome, desde que suas assinaturas sejam diferentes. Tal situação não gera conflito pois o compilador é capaz de detectar qual método deve ser escolhido a partir da análise dos tipos dos argumentos do método [6].

## 2.2 OBJETO

Um objeto é um instancia de uma classe e representa alguma entidade do domínio. O uso racional de objetos, obedecendo aos princípios associados à sua definição conforme estabelecido no paradigma de desenvolvimento orientado a objetos, é chave para o desenvolvimento de sistemas complexos [6].

No paradigma de orientação a objeto, tudo pode ser potencialmente representado como um objeto. Sob o ponto de vista da programação orientada a objeto, um objeto não é muito diferente de uma variável normal. Por exemplo, quando se define uma variável do tipo *Integer* em uma linguagem de programação como Java, essa variável tem um espaço em memória para registrar o seu estado ou valor e um conjunto de operações que podem ser aplicadas [6].

Quando se cria um objeto, adquire um espaço em memória para armazenar seu estado e um conjunto de operações que podem ser aplicadas ao objeto [6].

As técnicas de programação orientada a objetos recomendam que a estrutura de um objeto e a implementação de seus métodos devem ser tão privativos como possível. Normalmente, os atributos de um objeto não devem ser visíveis externamente. Da mesma forma, deve ser suficiente conhecer somente a especificação dos métodos, sem necessidade de saber detalhes de como foram implementados.

Encapsulamento é o princípio de projeto pelo qual cada componente de um programa deve agregar toda a informação relevante para sua manipulação como uma unidade. Aliado ao conceito de ocultamento de informação, é um poderoso mecanismo da programação orientada a objetos.

Na orientação a objeto, o uso do encapsulamento de informação é recomendado para que a representação do estado de um objeto deva ser mantida oculta. Cada objeto deve ser manipulado exclusivamente através dos métodos públicos do objeto, dos quais apenas a assinatura deve ser revelada. O conjunto de assinaturas dos métodos públicos da classe constitui sua interface operacional.

Dessa forma, detalhes internos sobre a operação do objeto não são conhecidos, permitindo que o usuário do objeto trabalhe em um nível mais alto de abstração, sem ter a preocupação com os detalhes internos da classe. Essa facilidade permite simplificar a construção de programas com funcionalidades complexas, tais como interfaces gráficas ou aplicações distribuídas.

## 2.3 HERANÇA

Herança é o processo que permite criar uma classe que herde todas as características de outra existente. A nova classe é chamada de classe derivada ou subclasse e a classe existente é chamada de classe-base ou superclasse. Uma das maiores vantagens do processo de herança é a reutilização de código. [11]

O processo de herança vai além da derivação simples. Uma classe derivada pode herdar características de mais de uma classe base. Esse processo é chamado de herança múltipla. [11]

## 2.4 INTERFACE

Algumas vezes queremos somente declarar os métodos que um objeto deve conter, mas não fornecer uma implementação deles. Desde que seu comportamento atenda critérios específicos, chamados de contrato, detalhes de implementação de métodos são irrelevantes. Estas declarações definem um tipo, e qualquer classe que implemente estes métodos é dita ser daquele tipo, sem importar como os métodos são implementados [6].

Para dar suporte a isto, você pode definir uma interface. Uma interface é como uma classe, mas contém apenas declarações vazias de seus métodos. O desenvolvedor de interface declara os métodos que devem ser oferecidos pelas classes e os que devem fazer [6].

## 2.5 MÁQUINA VIRTUAL

A máquina virtual representa uma camada de software intermediária, que isola a aplicação do contato direto com o hardware ou com o sistema operacional. Através desse mecanismo, obtemos um bom compromisso com a portabilidade, uma vez que podemos compilar uma linguagem de alto nível em um formato executável pela máquina virtual e sem a inconveniência do desenvolvedor se preocupar com a plataforma na qual o usuário executará o programa

Para atingir a questão da portabilidade, usa-se a idéia de uma máquina virtual que executa um código de máquina emulado. Essa máquina virtual traduz os comandos para a ambiente específica, ganhando independência de plataforma [12].

A Máquina Virtual Java é uma aplicativo que abstrai não só a camada de hardware, mais também a comunicação com o Sistema Operacional [12].

Uma aplicação tradicional em C, por exemplo, é escrita em uma linguagem de alto nível que abstrai as operações de hardware e é compilada para a linguagem de máquina. Nesse processo de compilação, é gerado um executável com instruções de máquina específicas para o sistema operacional e o hardware [12].

Um executável do C não é portável porque não podemos rodar-lo do Windows no Linux e vice-versa. Mas o problema de portabilidade não se restringe ao executável. As APIs são também específicas de Sistema Operacional. Assim, não basta recompilar para outra plataforma, é preciso reescrever boa parte do código [12].

Ao usar o conceito de máquina virtual, o Java elimina o problema da portabilidade do código executável. Ao invés de se gerar um executável específico, como Linux ou Windows, o compilador Java gera um executável para uma máquina genérica [12].

A JVM é uma máquina completa que roda em cima da máquina real. Ela possui suas próprias instruções de máquina e suas APIs próprias. O papel da máquina virtual é executar as instruções de máquina genéricas no Sistema Operacional e no hardware específico sob o qual está rodando [12].

É importante perceber que, por causa desse conceito da máquina virtual, o usuário final que deseja apenas rodar o programa não pode simplesmente executá-lo. Tanto os

desenvolvedores quanto os usuários precisam da máquina virtual. Mas os desenvolvedores, além da máquina virtual, precisam do compilador e de outras ferramentas [12].

# 3 LEVANTAMENTO DE REQUISITOS

Esse tópico irá enumerar os requisitos funcionais e não funcionais da linguagem Breder como todo.

## 3.1 REQUISITO DA LINGUAGEM

A linguagem deve compilar um conjunto de código fonte gerando um único arquivo que pode ser executado pela máquina virtual nos sistemas operacionais Windows, Linux e MacOs, tanto para 32 bits quanto para 64 bits.

A linguagem deve possuir o conceito de orientado a objeto, sendo fortemente tipada e de escopo estático.

A linguagem deve possuir a sintaxe similar a linguagem Java.

A linguagem pode possuir o recurso de orquestragem para a linguagem Lua.

A linguagem pode possuir o recurso de orquestragem para a linguagem Java.

A linguagem pode ter um *plugin* para o Eclipse para auxiliar o desenvolvedor a codificar.

A linguagem deve ter um site oficial que contenha todas as suas documentações e os *downloads* das versões mais recentes.

## 3.2 REQUISITO DO BDK

A linguagem deve possuir uma biblioteca padrão chamada BDK, disponível para o usuário.

A biblioteca padrão deve possuir uma especificação para classes da linguagem localizado no pacote *breder.lang* e uma implementação padrão utilizada pela máquina virtual.

A biblioteca padrão deve possuir uma especificação para classes utilitárias localizado no pacote *breder.util* e uma implementação padrão para eles.

A biblioteca padrão deve possuir uma especificação para classes de entrada e saída localizado no pacote *breder.io* e uma implementação padrão para eles.

A biblioteca padrão deve possuir uma especificação para classes de rede localizado no pacote *breder.net*.

A biblioteca padrão deve possuir uma especificação para classes de interface gráfica localizado no pacote *breder.gui*.

## 3.3 REQUISITO DO COMPILADOR

O compilador deve rodar no ambiente Windows, Linux e MacOs em 32 bits e 64 bits.

O compilador deve ser implementado na linguagem Java para garantir recursos lingüísticos, aumentando a sua produtividade, e portabilidades.

O compilador deve utilizar da ferramenta *JavaCC* para auxiliar a implementação da gramática.

O compilador não exige um tempo de resposta razoável.

## 3.4 REQUISITO DA MÁQUINA VIRTUAL

A máquina virtual deve ter um bom desempenho de execução

A máquina virtual pode ser executado em um ambiente móbile como *iPhone*.

A máquina virtual deve possuir um coletor de lixo que segue a idéia de *mark-and-sweep*.

A máquina virtual pode ter grande desempenho em operações nativas.

A máquina virtual deve rodar no ambiente Windows, Linux e MacOs em 32 bits e 64 bits.

# 4 ARQUITETURA DA LINGUAGEM BREDER

O capítulo irá descrever alguns aspectos importantes da arquitetura da linguagem Breder e explicar como o compilador e a máquina virtual foram implementados.

## 4.1 ORQUESTRAÇÃO

A orquestração é um mecanismo dessa linguagem que realiza a comunicação com outras linguagens de programação. A idéia desse mecanismo é disponibilizar ferramentas lingüísticas para o desenvolvedor executar um projeto em diferentes linguagens de programação com o menor custo de integração.

Essa linguagem foi preparada para dar suportar características de diferentes linguagens de programação, possibilitando que a distinção lingüística entre várias linguagens não atrapalhem a sintaxe de comunicação com esta linguagem. Por exemplo, o retorno múltiplo foi implantado nesta linguagem, evitando que distinções entre elas prejudiquem a sintaxe com a comunicação com outras linguagens que retornem mais de um objeto. Um outro exemplo está na herança múltipla implantada nesta linguagem, permitindo que ela possa se comunicar com outras linguagens que tenha também este recurso lingüístico.

Algumas especificações de como comunicar a linguagem Breder com outras linguagens já foram estudadas, mas ainda não foram implementadas. Porém, a comunicação mais básica entre essa linguagem com a linguagem C/C++ já foi implementada e testada. A seguir, será melhor descrito como funciona a comunicação entre essa linguagem e a linguagem C/C++.

### 4.1.1 ORQUESTRAÇÃO COM C/C++

Essa linguagem foi desenvolvida pensando em trabalhar com diversas linguagens de programação em um mesmo projeto. Assim, a sintaxe de comunicação entre a linguagem Breder e outras linguagens deve ser transparente para que a integração entre elas possa ser simples.

Para fazer a orquestração dessa linguagem com a linguagem C/C++, basta declarar um método na linguagem Breder sem uma implementação e com a palavra reservada *native* depois da palavra de nível de acesso. Quando o compilador detecta o método a ser

implementado na linguagem C/C++, ele informa à máquina virtual o nome do método que será buscado nas bibliotecas nativas, através do arquivo gerado pela compilação. Uma biblioteca nativa é um conjunto de funções implementadas por uma linguagem de programação que compila para a máquina real. As bibliotecas nativas que serão carregadas pela máquina virtual são especificadas nos argumentos de compilação. Depois do código fonte ser compilado, será gerado um arquivo *bytecode* com o nome das bibliotecas nativas. Assim, este arquivo *bytecode* conterá ter o nome das bibliotecas que serão carregadas e os nomes das funções nativas que serão buscados nas bibliotecas nativas que serão carregadas.

## 4.2 PACOTE SDK

A linguagem Breder disponibiliza, para os desenvolvedores, um pacote de classes de apoio. Esse pacote é constituído de diversas especificações na forma de interfaces e algumas das suas implementações.

A seguir, serão mostrados alguns dos principais pacotes disponíveis para os desenvolvedores.

### 4.2.1 PACOTE BREDER.LANG

O pacote *breder.lang* especifica as classes que a máquina virtual utiliza em seu núcleo. Algumas das principais classes que constituem esse pacote são classes primitivas e de exceções.

As especificações das classes não são utilizadas pelo núcleo da máquina virtual, mas são úteis para referenciar classes de outras implementações. Assim, a utilização de interfaces para os tipos primitivos possibilita a recuperação de referências primitivas criadas pela máquina virtual ou pela instanciações de classes primitivas implementadas por uma biblioteca externa. Por exemplo, a interface *breder.lang.INumber* pode ser implementada pela classe *breder.lang.standard.Number*, que é utilizada pelo núcleo da máquina virtual, ou, por exemplo, pela classe *myimpl.Number*, na qual um desenvolvedor externo implementou o número primitivo com número infinito de casas decimais.

### 4.2.2 OUTROS PACOTES

O pacote *breder.util* especifica interfaces das classes utilitárias. O *breder.io* especifica as de entrada e saída. Esse pacote tem tanto interfaces, quanto implementações padrão para elas.

O pacote *breder.gui* especifica interfaces das classes de interface gráfica. Esse pacote somente tem interfaces.

## 4.3 COMPILADOR BREDER

O compilador Breder representa a transformação de um conjunto de códigos fonte na linguagem Breder para um único arquivo binário, que pode ser executado pela máquina virtual Breder. O compilador é responsável pela interpretação e estruturação do código fonte para uma estrutura de árvore para que possa ser processada, e traduzida para uma outra linguagem alvo. [7]

Existem duas etapas na compilação de um código fonte. A etapa de análise divide o programa fonte nas partes constituintes e cria uma representação intermediária dos mesmos. A etapa de sintaxe constrói o programa alvo desejado, a partir da representação intermediária. [7]

A análise léxica é uma etapa de análise de entrada das linhas de caracteres e que produz uma seqüência de símbolos chamados símbolos léxicos que podem ser manipulados mais facilmente por um *parser*. A análise léxica analisa o alfabeto da linguagem do código fonte e verifica se os caracteres não fazem parte do alfabeto. A analise léxica é a primeira etapa de um compilador, e é seguido pela análise sintática [9].

O analisador sintático é responsável por verificar se a seqüência de símbolos contidos no código fonte compõem um programa válido [9]. A sintaxe de uma linguagem de programação pode ser descrita por uma gramática. Ferramentas como *JavaCC* recebem gramática como entrada e geram o código fonte de verificação gramatical. Assim, a partir do código fonte gerado pelo *JavaCC*, o compilador consegue organizar o programa de entrada em estrutura de dados. [10]

*JavaCC* é um gerador de analisador sintático aberto, implementado na linguagem Java. É similar ao *yacc* na medida em que gera um analisador sintático de uma gramática fornecida no formalismo de *Backus-Naur* Estendido, exceto pelo fato do programa gerado ser

em Java. Além disso, o *JavaCC* gera analisadores sintáticos descendentes, o que o limita às classes gramaticais LL(k) [10].

A análise semântica é a terceira fase da compilação, que recebe este nome pois requer o processamento da árvore gerada pela análise sintática. A informação gerada está relacionada com o significado semântico do programa traduzido. É neste analisador que se verifica os erros semânticos do código-fonte e se coleta as informações necessárias para a próxima fase da compilação, que é a geração do código fonte. A verificação realizada pelo analisador semântico deve procurar por erros como a incompatibilidade dos tipos de operadores, variáveis não declaradas e chamadas de funções ou métodos com um número incorreto de parâmetros. Para que estas verificações sejam executadas, o analisador depende de uma tabela de símbolos, onde estão armazenados informações de variáveis declaradas, funções ou métodos, tipos ou classes. Somente com a tabela de símbolos é possível realizar a validação semântica do programa. [9]

Uma vez verificado que não existem erros sintáticos e semânticos, o compilador pode realizar sua tarefa de geração de código. A geração é feita a partir das informações extraídas do analisador semântico que foram armazenadas em memória. O gerador é dividido na etapa de geração de cabeçalho e geração de código para cada método de todas as classes. A linguagem de programação desse projeto define que o compilador gere somente um único arquivo com todos os códigos de todas as classes do projeto. Assim, o resultado da compilação representa um único arquivo que pode ser carregado como argumento para a máquina virtual Breder.

Um compilador, conceitualmente, opera em fases, cada uma das quais transforma o programa fonte de uma representação para outra. Uma composição típica pode ser enumerada em 6 fases : análise léxica, análise sintática, análise semântica, geração de código intermediário, otimização de código e geração de código [7]. O compilador Breder utiliza somente as três primeiras fases e a última, as fases de geração de código intermediário e otimização de código não são utilizadas.

Não houve preocupação com o desempenho da execução do compilador Breder. A partir disso, foi escolhida uma linguagem de alto nível Java para implementar o compilador Breder, já que a principal razão foi garantir uma produtividade em desenvolver o compilador, mesmo que tenha que comprometer o desempenho da execução.

## 4.4 MÁQUINA VIRTUAL BREDER

Na ciência da computação, máquina virtual é o nome dado a uma máquina implementada através de software que executa programas simulando um computador real.

Máquina virtual Java é um programa que carrega e executa os aplicativos Java no formato *bytecode*. Ela é responsável pelo gerenciamento dos aplicativos, à medida que é executada.

Graças à máquina virtual, os programas escritos para ela podem funcionar em qualquer plataforma de hardware e software que possua a máquina virtual instalada, tornando assim a aplicação independente da plataforma onde funcionam.

Uma máquina Virtual funciona na base de um interpretador de instrução. Para cada instrução que estiver em execução, um trecho de código da máquina real será executada.

A partir disso, uma máquina virtual pode ser sintetizada em 2 partes diferentes: carga do programa na máquina virtual a partir do arquivo resultante da compilação e a interpretação de cada instrução seqüencial.

Para construir uma máquina virtual, é preciso definir todas as instruções que serão interpretadas que utilizará de uma pilha de objeto para auxiliar a execução.

A pilha de objetos é utilizada em diversas instruções da máquina virtual para armazenar, de forma temporária, os objetos que irão ser utilizados por outras instruções. Por exemplo, uma instrução de soma utiliza os dois últimos objetos do topo da pilha para realizar a operação, resultando em um único objeto que irá substituir os dois do topo da pilha.

Além da pilha de objetos, a máquina virtual irá precisar de todas as instruções seqüencialmente armazenadas em memória do aplicativo que está sendo executado. O índice da instrução é utilizado pelas instruções de desvio. Por exemplo, a instrução *jump* atualiza o contador de programa para o índice especificado no dado da instrução.

Toda operação de chamada de método gera uma mudança no valor do contador de programa. Essa mudança ocorre para desviar a próxima instrução correspondente ao método a ser executado. Porém, quando o método é finalizado, a próxima instrução anterior a chamada deve ser apontada pelo contador de programa. Para implementar esse recurso, é preciso ter

uma outra pilha, chamada de pilha de instrução, que armazena todas as próximas instruções de métodos chamados em tempo de execução.

Ao contrário do compilador Breder, a máquina virtual exige uma execução eficiente. Para isso, é preciso que ela seja desenvolvida por um compilador que gere instruções de máquina real, com o objetivo de manter o máximo de desempenho e acesso ao hardware e ao sistema operacional. Com a exigência de uma linguagem de programação eficiente e a ausência da necessidade de uma linguagem bem estruturada, a Linguagem C ANSI foi utilizada para implementar a máquina virtual Breder.

As instruções da máquina virtual Breder podem ser subdivididas em várias categorias. Dentre elas, estão as instruções aritméticas, de desvio, de empilhamento e de armazenamento. Além disso, a máquina virtual também é responsável por criar e coletar objetos em sua execução, aonde os objetos não utilizados são detectados pelo coletor de lixo e liberados.

A máquina virtual utiliza o coletor de lixo para liberar os objetos que não estão sendo mais referenciados. Para isso, o coletor utiliza da pilha de objetos da máquina virtual para detectar os que não estão mais sendo utilizados. O coletor de lixo implementado na máquina virtual Breder foi o *mark-and-sweep* [8].

# 5 DESCRIÇÃO DA LINGUAGEM BREDER

Um programa nessa linguagem é um conjunto de descrições de classes, onde algumas delas possui um método especial denominado *main*, de onde iniciará a execução. As classes são descritas em sintaxe semelhante à linguagem Java.

Nas seções seguintes serão descritas as construções da linguagem, considerando a EBNF [4] básica abaixo para definição de identificadores:

*<id> ::= <letter> { <letter> | <digit> }*

*<ids> ::= <id> { '.' <id> }*

## 5.1 DECLARAÇÕES

Essa linguagem é composta por vários tipos de declarações, sendo que a ordem não interfere no processo de compilação. A seguir, serão discutidos os tipos de declarações da linguagem.

### 5.1.1 DECLARAÇÃO DE CLASSE

Para construir uma classe é preciso criar um arquivo com o mesmo nome da classe e com a extensão "*.breder*". Por exemplo, se for criar uma classe de nome "*Teste"*, o nome do arquivo deverá ser "*Test.breder*".

Toda classe está contida dentro de algum arquivo.Além disso, a linguagem exige que um arquivo contenha somente uma classe. Para se criar mais de uma classe, é preciso criar mais de um arquivo.

Toda definição de classe exige a informação do nível de acesso. Esse nível corresponde o privilégio de acesso interno de outras classes. Uma classe declarada como protegida, não pode ser acessada por outra classe que não esteja no mesmo pacote. Uma classe declarada como publica, pode ser acessada por qualquer outra classe. Para esta versão da linguagem, uma classe só poderá ser declarada como publica, utilizando a palavra reservada *public*.

Uma classe pode ser declarada como abstrata, quando não é desejado implementar todos os métodos das classes e interfaces herdadas ou impossibilitar a criação de objetos dela. Assim, para declarar uma classe do tipo abstrato, basta utilizar a palavra reservada *abstract* depois da palavra reservada de nível de acesso.

Numa linguagem orientada a objeto, a característica lingüística de herança é muito utilizada. Nessa linguagem, uma classe pode herdar de várias outras classes e interfaces. Assim, para que uma classe herde de outras classes, basta usar a palavra reservado *extends* depois do nome da classe e escrever a lista que deseja herdar separadas por vírgula. Também, para que uma classe herde de outras interfaces, basta usar a palavra reservada *implements* depois da definição de herança de classes e escrever a lista de interfaces que deseja herdar separadas por vírgula.

Essa linguagem permite que uma classe possa não ser herdada por outras classes. Para isso, basta utilizar a palavra reservada *final* depois da palavra reservada de nível de acesso.

A seguir a demonstração da EBNF (4) de declaração de classe:

*<ClassDefinition> ::= 'public' {<ClassAtribute>} 'class' <id> [<GenericList>] [<ExtendList>] [<ImplementList>] '{' {<PropertyDefinition> ';' | <FieldDefinition> ';' | <ConstrutorDefinition> | <MethodDefinition> } '}'*

*<ClassAttribute> ::= 'abstract' | 'final'*

*<GenericList> ::= '<' <id> { ',' <id> } '>'*

*<ExtendList> ::= 'extends' <Type> { ',' <Type> }*

*<ImplementList> ::= 'implements' <Type> { ',' <Type> }*

*<Type> ::= <ids> ['<' <Type> { ',' <Type> } '>']*

### 5.1.2 DECLARAÇÃO DE CAMPO

Os campos em programação orientada a objeto são os elementos que definem a estrutura de uma classe. Esses campos também são conhecidos como variáveis de classe, e podem ser divididos em dois tipos básicos: atributos de instância e de classe. [5]

Os valores dos atributos de instância determinam o estado de cada objeto. Um atributo de classe possui um estado que é compartilhado por todos os objetos de uma classe, que também são chamados de atributos estáticos. Assim, para declarar um atributo de uma classe, basta utilizar a palavra reservada *static* depois da palavra de nível de acesso.

Uma classe pode possuir vários campos ou atributos. Todo atributo possui um nível de acesso, um tipo e um nome. Assim, para declarar um atributo de um objeto, basta escrever os três itens citados, em sua ordem.

A seguir a demonstração da EBNF (4) de declaração de campo:

*<FieldDefinition> ::= <Access> {<FieldAttribute>} ['notnull'] <Type> <id>*

*<FieldAttribute> ::= 'static'*

*<Access> ::= 'public' | 'protected' | 'private'*

### 5.1.3 DECLARAÇÃO DE MÉTODO

O objetivo de um método é gerar uma mudança de estado do objeto ou simplesmente executar um processamento, retornando alguns objetos ou não. Em um método, podem ser recebidos diversos parâmetros de diversos tipos e pode retornar diversos objetos.

Todo método é especificado por um nível de acesso, um nome, uma lista de tipos de classe de retorno e uma lista de parâmetros.

Da forma análoga ao que acontece com um campo, um método pode pertencer a um objeto ou a uma classe. Assim, quando se deseja declarar um método para uma classe, basta usar a palavra chave *static* depois da palavra do nível de acesso.

Essa linguagem permite que um método não possua uma implementação, indicando que será codificado por uma classe concerta que o herdar em alguma parte da sua hierarquia. Dessa forma, para declarar um método que não possua uma implementação, a classe deve ser do tipo abstrata e o método deve ter a palavra reservada *abstract* depois da palavra do nível de acesso.

Um método pode ser definido de forma que nunca passa ser reimplementado por outra classe que o herde. Para isso, deve-se usar a palavra chave *final* depois do nível de acesso. Essa técnica é útil quando não se deseja alterar a codificação de um método, independente de sua hierarquia de classe.

Como já descrito acima, um método pode retornar vários objetos. Em cada tipo de retorno, pode ser usada a palavra reservada *notnull* para indicar que o objeto é obrigatoriamente não nulo.

De forma análoga ao que ocorre com os valores de retornos, um método pode ter vários parâmetros. Para cada parâmetro é possível indicar que seu valor é obrigatoriamente não nulo através da palavra reservada *notnull*.

A linguagem possibilita que o tipo do retorno de um método seja o próprio tipo do objeto que foi chamado. Para implementar essa característica, quando um método de um objeto é chamado, em tempo de compilação, a classe do retorno do método será igual a classe do objeto chamado. Por exemplo, se A é uma superclasse de B e um objeto de B chamar um método definido em A cujo retorno utilize dessa técnica, a classe de retorno será B, mesmo

que a classe A não conheça a classe B. Se o mesmo objeto de B for feito um cast para a classe A e for chamado esse método, a classe de retorno será A e não B. Porém, essa técnica exige que o método retorne sempre o próprio objeto através da palavra chave *this* e o método tenha como retorno a palavra reservada *this*.

```
public class A {
  public notnull this exec () { return this; }
}
public class B extends A {
  public void test(){}
}
(...)
B b = new B();
b.exec().test();
B bb = b.exec();
A a = (A)b;
a.exec().test() // Falha na chamada de test()
A aa = a.exec();
```

Um método pode ser declarado de forma a indicar a possibilidade de lançar alguns erros. Para isso, é usada a palavra reservada *throws* para indicar a lista de erros que o método poderá lançar. Esse conceito irá ser mais bem abordado mais a frente no tópico de Tratamento de Erros.

A seguir a demonstração da EBNF (4) de declaração de método :

*<MethodDefinition> ::= <Access> {<MethodAttribute>} <MethodReturnList> <id> <ParameterList> [<MethodThrowList>] <MethodBlock>*

*<MethodAttribute> ::= 'static' | 'native' | 'abstract' | 'final'*

*<MethodReturnList> ::= 'void' | ['notnull'] 'this' | <MethodReturn> { ',' <MethodReturn> }*

*<MethodReturn> ::= ['notnull'] <Type>*

*<ParameterList> ::= '(' [ <Parameter> { ',' <Parameter> } ] ')'*

*<Parameter> ::= ['notnull'] <Type> <id>*

*<MethodThrowList> ::= 'throws' <Type> { ',' <Type> }*

*<MethodBlock> ::= (<Block> | ';')*

### 5.1.1 DECLARAÇÃO DE PROPRIEDADE

Uma implementação de uma estrutura de dados em forma de classe corresponde, basicamente, a vários campos e um método *get* e *set* para cada um deles. Esse padrão é muito utilizado e, normalmente, os métodos *get* e *set* são gerados automaticamente por alguma ferramenta de desenvolvimento. A linguagem dispõe de um mecanismo lingüístico para gerar os métodos *get* e *set* implicitamente. Para isto deve-se declarar o campo com a palavra reservada *property* depois da palavra do nível de acesso.

Em alguns casos, é necessário redefinir o método *get* ou *set* gerado pelo compilador. Para isso, basta declarar os métodos *get* ou *set* com a mesma assinatura à do método declarado implicitamente.

A seguir a demonstração da EBNF (4) de declaração de uma propriedade :

*<PropertyDefinition> ::= <Access> {<PropertyAttribute>} 'property' <PropertyType> <id>*

*<PropertyAttribute> ::= 'static'*

*<PropertyType> ::= ['notnull'] <Type>*

### 5.1.2 DECLARAÇÃO DE INTERFACE

Quando se deseja elaborar uma especificação de uma funcionalidade, seguindo a boa prática de programação dessa linguagem, deve-se dar prioridade a construção das

interfaces. Nas interfaces serão definidas todas as operações necessárias para que a funcionalidade possa ser usada de forma correta e clara.

Em sua construção, é importante que as operações não considerem as possíveis implementações. Isso porque, as interfaces são utilizadas para especificar as funcionalidades e não para especificar uma possível implementação.

Uma interface representa uma estrutura com um conjunto de métodos não implementados. O objetivo dela é somente armazenar as declarações das operações. Caso uma classe tente implementar uma interface, será necessário codificar todas as operações dela.

Deve-se reforçar que toda interface deve ser criada com a funcionalidade ideal e genérica, mesmo que esse ideal seja difícil de ser alcançado. Isso porque toda interface deve ser de alto nível, desconsiderando toda a complexidade de implementação que seja necessária para se codificar a funcionalidade.

Na linguagem, uma interface pode herdar de várias outras interfaces. Para que isto ocorra, é necessário o uso da palavra reservado *extends* depois do nome da interface e listá-las separadas por vírgula.

Como padrão dessa linguagem, toda interface deve começar o nome com a letra "I", indicando que se trata de uma interface.

A seguir, será demonstrado a EBNF (4) de declaração de uma interface :

*<InterfaceDefinition> ::= 'public' 'interface' <id> <GenericList> <ExtendList> '{' {<InterfaceConstrutorDefinition> | <InterfaceMethodDefinition> } '}'*

*<InterfaceMethodDefinition> ::= 'public' <MethodReturnList> <id> <ParameterList> [<MethodThrowList>] ';'*

*<InterfaceConstrutorDefinition> ::= 'public' <id> <ParameterList> [<MethodThrowList>] ';'*

### 5.1.3 DECLARAÇÃO DE PACOTE

Toda estrutura do tipo classe ou interface deve possuir um pacote associado. O pacote serve para determinar onde está localizada a classe ou interface declarada. Depois que

forem criadas várias classes e interfaces, será possível perceber que o pacote também tem a função de organizar e classificar.

O compilador Breder utiliza o nome do pacote para procurar a estrutura nos *classpaths* (diretórios onde busca todas as classes e interfaces). Além disso, caso queira utilizar alguma estrutura fora do pacote, será necessário informar a localização para que o compilador possa referenciá-la de forma correta. Usa-se a palavra chave *import* para importar classe referente a outro pacote.

Para declarar um pacote, o código fonte deve conter, no inicio, a palavra reservada *package* seguida do caminho do código fonte relativo ao projeto, separadas pelo caracter ".". Por exemplo, caso o projeto esteja localizado na pasta "*c:\proj*" e a classe esteja localizada na pasta "*c:\proj\com\test\Test.breder*", o pacote será "*com.test*".

A seguir, será demonstrado a EBNF (4) de declaração de um pacote :

*<PackageDefinition> ::= 'package' <ids> ';'*

*<ImportDefinition> ::= 'import' <id> { '.' ( <id> | '*' ) } ';'*

### 5.1.4 DECLARAÇÃO DE CONSTRUTOR

Para se criar um objeto de uma classe, é preciso instanciá-lo. No momento da instanciação, a linguagem sempre irá chamar o construtor dela, mesmo que ele não exista.

O construtor tem o papel de inicializar o objeto para o seu funcionamento. É falha de programação a criação de métodos de inicialização do objeto que não seja o próprio construtor. Depois que o construtor for executado, o objeto já deve ter sido inicializado por completo. Com isso, todo construtor deve exigir tudo que for necessário para que o objeto possa ser inicializado. Para se declarar um construtor de uma classe, basta declarar um método sem retorno e que tenha o mesmo nome da classe.

Uma interface pode definir um construtor padrão para todas as implementações dela. Esse recurso torna-se interessante porque o usuário da *API* só precisará saber como funciona a interface, não precisando se preocupar como e com qual construtor será implementado.

```
public interface IList {
    public IList ();
}
public class MyList implements IList {
    public MyList () {}
}
```

O exemplo acima mostra que qualquer classe que implementa a interface *IList* terá pelo menos um construtor sem nenhum parâmetro.

A seguir, será demonstrado a EBNF (4) de declaração de método:

*<ConstrutorDefinition> ::= <Access> {<ConstrutorAttribute>} <id> <ParameterList> [<MethodThrowList>] <MethodBlock>*

*<ConstrutorAttribute> ::= 'native'*

## 5.2 ESTRUTURA

Nessa seção são apresentados os comandos da Linguagem Breder.

### 5.2.1 DECLARAÇÃO DE VARIÁVEL

A linguagem possibilita a declaração de variáveis dentro de um bloco de comandos. Além disso, essas variáveis existirão enquanto o bloco declarado existir.

Para declarar uma variável é necessário especificar o tipo, o nome e opcionalmente um valor inicial.

Além disso, é possível declarar diversas variáveis dentro de uma mesma declaração. Dessa forma, a declaração será composta por variáveis de tipos diferentes ou não, com seus nomes e valores iniciais correspondentes.

A seguir, será demonstrado a EBNF (4) de declaração de variável:

*<VariableDeclaration> ::= <ParameterList> '=' <ExpressionList>*

*<ExpressionList> ::= <Expression> {',' <Expression>}*

### 5.2.2 ATRIBUIÇÃO

Essa linguagem possibilita fazer atribuição de um valor a uma variável já declarada. Além disso, a atribuição também pode ser feita de forma múltipla, na qual várias associações podem utilizar um mesmo comando de atribuição.

A seguir, será demonstrado a EBNF (4) de associação:

*<Assign> ::= <LValueList> [<ArithmeticOperator>] '=' <ExpressionList>*

*<LValueList> ::= <LValue> {',' <LValue>}*

*<ArithmeticOperator> ::= '+' | '-' | '*' | '/'*

### 5.2.3 BLOCO

Bloco é um conjuntos de instruções delimitado pelos caracteres "{" e "}". Nele, é possível declarar variáveis que serão utilizados internamente ou por outros blocos internos.

A seguir, será demonstrado a EBNF (4) de bloco:

*<Block> ::= '{' {<Command>} '}'*

*<Command> ::= <VariableDeclaration> | <Assign> | <If> | <While> | <Repeat> | <For> | <Try> | <Throw> | <Return> | <Super> | <Break> | <Continue> | <Block> | <Expression>*

### 5.2.4 IF

A linguagem possui uma estrutura de controle para desviar a execução do aplicativo em função de uma condição. Portanto, dada uma condição, é possível realizar a execução de um bloco de comandos quando ela for verdadeira. Caso a condição não seja verdadeira, é possível definir outras condições para serem testadas. As condições serão testadas seqüencialmente e caso uma delas seja verdadeira, o bloco de comandos

correspondente será executado. Porém, se todas as condições não forem verdadeiras, um bloco opcional pode ser executado.

A seguir será demonstrado a EBNF (4) de *if* :

*<If> ::= 'if' '(' <Expression> ')' <Block> {<ElseIf>} <Else>*

*<ElseIf> ::= 'elseif' '(' <Expression> ')' <Block>*

*<Else> ::= 'else' <Block>*

### 5.2.5 WHILE

A linguagem possibilita a realização de um laço de um bloco de comandos a partir de uma condição. Dada uma condição, é possível realizar a execução de um bloco de comando enquanto essa condição for verdadeira. Quando a condição for falsa, a estrutura de controle será finalizada.

A seguir será demonstrado a EBNF (4) de *while*:

*<While> ::= 'while' '(' <Expression> ')' <Block>*

### 5.2.6 REPEAT

A diferença do *repeat* para o *while* é sutil: o *while* primeiro testa a condição para depois executar o bloco de comandos e o *repeat* executa primeiro o bloco e depois faz o teste da condição.

A seguir será demonstrado a EBNF (4) de *repeat*:

*<Repeat> ::= 'repeat' <Block> 'while' '(' <Expression> ')' ';'*

### 5.2.7 FOR

A linguagem possui a estrutura de controle *for* para repetição em função de uma condição incremental. O laço *for* numérico repete um bloco de código enquanto uma variável de controle varia.

A estrutura de controle *for* é divido em três partes fundamentais. Primeira parte representa a inicialização de variáveis que podem ser usadas na segunda parte. A segunda parte representa a condição lógica que faz com que o bloco continue executando. A terceira parte representa uma associação que pode ser usada para incrementar ou decrementar uma variável. Todas as três partes são opcionais, na qual a segunda é substituído pelo valor verdadeiro caso não seja preenchida.

A seguir será demonstrado a EBNF (4) de *for*:

*<For> ::= 'for' '(' [<ForInitialize>] ';' [<ForCondition>] ';' [<ForIncrement>] ')' <Block>*

*<ForInitialize> ::= <VariableDeclaration>*

*<ForCondition> ::= <Expression>*

*<ForIncrement> ::= <Assign>*

### 5.2.8 TRATAMENTO DE ERROS

Durante a execução, a aplicação pode sofrer uns erros de vários tipos e graus de severidade. [6]

Tratamento de erro é um mecanismo de recuperação de falhas que ocorreram na execução do aplicativo. Uma exceção é lançada quando uma condição inesperada de erro é encontrada. Nesse ponto, a execução é interrompida e a pilha começa a ser percorrida, a partir do método sendo executado, em busca de um método que possua um tratador para a exceção lançada. [6]

Exceções são objetos de uma classe, os quais podem ser lançados. Todos esses objetos devem ser sub-tipos da classe *breder.lang.Throwable*.

Os erros do ambiente de execução estendem uma das classes *breder.lang. standard.RuntimeException* e *breder.lang.standard.Error*, as quais são chamadas de exceções não verificadas. Os erros da classe *RuntimeException* representam condições que refletem em falhas lógicas do programa e não podem ser, tipicamente, recuperadas durante a execução. Os da classe *Error* [6] indicam falhas na própria máquina virtual, as quais tipicamente estão além da habilidade da aplicação de controlar ou tratar.

As exceções verificadas representam condições que podem ser esperadas e, se ocorrerem, devem ser tratadas de alguma maneira. [6]

Quase todas as exceções devem estender a classe *breder.lang.Exception*, tornando-as verificadas. As novas exceções verificadas representam as condições excepcionais que podem surgir em sua biblioteca ou aplicação. [6]

Para lançar um erro, basta utilizar da palavra chave *throw* e indicar um objeto que herda da classe *Throwable*. Caso o método não trate o erro, será necessário delegar o tratamento para os métodos que o chamar. Para isso, é necessário usar a palavra reservada *throws* na declaração do método para indicar que não irá tratar tal erro. [6]

Quando um erro ocorre, o contador de programa irá ser desviado, procurando alguma clausula que irá tratar a classe do objeto lançado. Esta é declarada através da palavra chave *catch*, indicando a classe de objeto que irá tratar. A máquina virtual irá buscar por todas as clausulas declaradas, considerando a ordem seqüencial, em sua cadeia de chamadas, gradativamente desempilhando até encontrar uma que possua a classe que o herde. Quando for encontrado a primeira, o contador de programa irá apontar para o seu bloco. [6]

A seguir, será demonstrado a EBNF (4)de *try catch* :

*<Try> ::= 'try' <Block> { <Catch> }*

*<Catch> ::= 'catch' '(' <Type> <id> ')' <Block>*

## 5.3 EXPRESSÃO

Todo objeto só pode ser criado ou referenciado através de uma expressão. Existem diversos tipos de expressões capazes de referenciar objetos, classes e constantes.

Nessa seção serão demonstradas algumas das possíveis expressões que a linguagem suporta.

### 5.3.1 PRIMITIVA

A linguagem suporta primitivos do tipo *String*, *Boolean*, *Number*, *Integer*, *Natural*, *Index* que são objetos de classe definido no pacote *breder.lang*.

A classe *String* representa uma cadeia de caracteres. Essa classe suporta somente a operação de concatenação através do operador de adição.

A classe *Boolean* representa um valor lógico. Essa classe da suporte a operação lógica de *or* e *and*.

A classe *Number* representa um valor número positivo ou negativo com ponto flutuante ou não. A classe *Number* possui a operação de adição, subtração, multiplicação e divisão como operações matemáticas.

A classe *Integer* representa um valor inteiro positivo ou negativo. A classe *Integer* representa um subconjunto da classe *Number*. Essa classe suporta os mesmos operadores da classe *Number*.

A classe *Natural* representa um valor inteiro maior ou igual a zero. A classe *Natural* representa um subconjunto da classe *Integer*. Essa classe suporta os mesmo operadores da classe *Number*.

A classe *Index* representa um valor inteiro maior que zero. A classe *Index* representa um subconjunto da classe *Natural*. Essa classe suporta os mesmo operadores da classe *Number*.

Todos esses tipos primitivos são classes normais vindas do pacote SDK que podem ser instanciados pelos programas e coletados pelo coletor de lixo. Uma grande perda de desempenho está na necessidade de tratar os tipos primitivos como classes comuns e não valores primitivos da máquina virtual que não deveriam ser coletados pelo coletor de lixo. Assim, trabalhos futuros irão otimizar a memória e processamento que as classes primitivas geram.

### 5.3.2 ARITMÉTICA

A linguagem Breder possibilita que uma classe possa implementar um operador. Com isso, por exemplo, um objeto da classe *String* pode ser somado com outro objeto dessa classe. O mesmo acontece com as classes *Number*, *Integer*, *Natural* e *Integer*. Caso se deseje que uma classe qualquer ofereça suporte a uma operação aritmética, basta implementar o operador desejado.

A vantagem de implementar um operador, ao invés de escrever um método que o emule, está na prioridade de execução que alguns operadores tem sobre os outros. Assim, um operador de multiplicação será executado antes de um operador de soma. Para simular a prioridade em um operador emulado, o método que o emula deve ser chamado na ordem certa, o que precisa ser feito manualmente pelo programador. Essas chamadas em uma ordem explícita possibilita falhas de programação nas quais a ordem não é seguida corretamente. Portanto, é mais seguro implementar um operador do que tentar emulá-lo.

Para implementar os operadores de soma, subtração, multiplicação e divisão basta implementar métodos chamados de *operatorSum, operatorSub, operatorMul* e *operatorDiv* respectivamente, passando como parâmetro o objeto a ser operador e retornando um objeto resultado. Por exemplo, para implementar uma operação matemática com dois vetores, basta implementar um método chamado *operatorSum* que recebe como parâmetro um vetor e retorne um outro vetor. No exemplo abaixo é mostrado o operador citado:

```
public native notnull Vector operatorSum(notnull Vector v) {
  return new Vector(this.getX() + v.getX(), this.getY() + v.getY());
}
```

### 5.3.3 OPERAÇÕES LÓGICAS

A linguagem possui vários tipos de expressões lógicas. São expressões lógicas aquelas que utilizam os operadores *or*, *and*, ==, !=, <, <=, >, >=, na qual os operadores *and* e *or* têm o privilégio de execução a dos operadores de comparação.

Assim, toda operação binária que se utiliza de algum desses operadores, é retornado um valor lógico.

### 5.3.4 CAST

A Linguagem possibilita a utilização do recurso de *cast* para substituir o tipo de um objeto por outro. Este recurso requer que o outro tipo de objeto seja compatível a

hierarquia do tipo anterior. Caso uma incompatibilidade ocorra, será lançado um erro da classe *ClassCastRuntimeException*.

Por exemplo, é possível, de forma segura, fazer um *cast* da classe *Number* para a classe *Object*. Também é possível, de forma insegura, fazer um *cast* da classe *Object* para a classe *Number*.

O *cast* em tempo de compilação, somente determina se a operação é segura ou não. Caso seja segura, não será necessário realizar o *cast* explicito. Porém, caso seja insegura, o *cast* deve ser posto de forma explicita, na qual deverá especificar a classe.

O *cast* pode ser usado para atribuir ao tipo a característica de ser *notnull* ou não. Essa técnica é usada quando a API está mal formulada e está atrapalhando a escrita do código. Nesse caso, o *cast* pode ser feito de forma explicita colocando a palavra *notnull* antes da palavra do tipo da classe.

A seguir será demonstrado a EBNF (4) de *cast* :

*<Cast> ::= '(' ['notnull'] <Type> ')' <Expression>*

# 6 RESULTADOS OBTIDOS

A seguir serão discutidos alguns dos resultados obtidos com o desenvolvimento e uso da linguagem Breder para a primeira versão.

## 6.1 ANALISE DA API

A construção da linguagem fez com que fosse necessário criar um *plugin* para o Eclipse para ajudar o desenvolvimento de bibliotecas utilitárias em cima desta linguagem de programação. Com o auxilio do *plugin*, foi criado diversas bibliotecas utilitárias e armazenadas no pacote BDK (*Breder Development Kit*).

A API do BDK foi construída pensando em desenvolver utilitários ou componentes caixa preta na qual primeiro foi especificado um conjunto grande de interfaces de diversos tipos de funcionalidades e depois foi implementado. O BDK já possui uma implementação padrão para cada tipo de funcionalidade especificado na forma de interface. Assim, o usuário da linguagem Breder possui no BDK a implementação padrão dos pacotes utilitários, de entrada e saída de arquivo, de cálculos matemáticos e de um framework básico de interface gráfica.

As funcionalidades extras da linguagem Breder ajudou a definir um conjunto de APIs no BDK. A funcionalidade de retorno múltiplo melhorou em alguns aspectos da API em operações que era preciso retornar mais de um objeto. Em outras linguagens que não possuem o retorno múltiplo, são utilizados uma nova classe que engloba os objetos de retorno na forma de campos.

Além disso, a funcionalidade de *notnull* colaborou para manter mais confiável o uso da API. Em outras linguagens que não possui tal recurso, um valor com chance de ser nulo era passado indevidamente como parâmetro para uma função na qual era verificado somente em tempo execução e não em tempo compilação, como na linguagem Breder.

A funcionalidade de herança múltipla colaborou para manter a organização mais limpa das classes. Isso porque foram construídas algumas classes simples que eram reaproveitadas por outras classes mais complexas. Dessa forma, o código fonte de uma classe complexa acabava ficando muito menor do que o código fonte de uma classe complexa escrita em uma linguagem que não possui herança múltipla.

Por último, a técnica de horizontalidade de chamada de métodos colaborou para construir uma API mais utilizável. Isso porque a técnica permite fazer chamadas continuas de métodos com o mesmo objeto sem ser necessário repetir a instância várias vezes.

## 6.2 ANALISE DA CONFIABILIDADE

A linguagem Breder foi construída seguindo a teoria básica da linguagem Java. Assim, alguns dos resultados obtidos na linguagem Java, também foram obtidos na linguagem Breder.

O recurso diferencial da linguagem Breder que mais ajudou na confiabilidade da execução e da compilação de um código fonte foi o recurso *notnull*. Esse recurso indica, em tempo de compilação, quando uma variável pode assumir valor nulo ou não. Tal técnica obriga o desenvolvedor testar o valor nulo, antes de usar, quando o valor possuir alguma chance de ser nulo. Essa funcionalidade reduziu drasticamente os *bugs* de *NullPointerException* na medida que o compilador obrigou o desenvolvedor a tratar tal situação, fazendo com que raras situações de *NullPointerException* sejam lançadas.

## 6.3 ANALISE DA USABILIDADE

Além do sucesso de usabilidade da linguagem Java, a linguagem Breder traz o diferencial do recurso de horizontalidade de chamada de métodos. Essa funcionalidade ajuda ao desenvolvedor a fazer chamadas contínuas, ao mesmo objeto, sem precisar repetir a instância várias vezes. Para se tentar simular esta funcionalidade na linguagem Java, alguns *casts* explícitos são necessários, os quais podem trazer alguns riscos a sua execução.

## 6.4 ANALISE DA LEGIBILIDADE

A linguagem Breder agrega, praticamente, os mesmos resultados de legibilidade da linguagem Java. A linguagem Breder possui algumas funcionalidades diferenciais que melhoram a legibilidade do código como: retorno múltiplo, horizontalidade de chamada de métodos, herança múltipla e campo *property*.

O retorno múltiplo evita a criação de novas classes somente para simular a operação de retorno múltiplo. Dessa forma, o número de classes é reduzido, melhorando a legibilidade do código fonte.

A horizontalidade de chamadas de métodos colabora para o entendimento continuo das chamadas de operações em cima de um mesmo objeto. Esse recurso reduz a necessidade de repetir a carga do objeto na pilha da BVM, quando se deseja chamar vários métodos de um mesmo objeto. A redução da repetição da carga do objeto ajuda na legibilidade da leitura do código fonte.

A herança múltipla colabora na organização da construção das classes. A API do BDK segue o critério que toda classe complexa herda de classes de menor complexidade. Assim, o código fonte de uma classe complexa é reduzido em função da reutilização do código das classes de menor complexidade. As linguagens que não possuem o recurso de herança múltipla fazem com que as classes complexas tenham um código fonte extensos. Dessa forma, a herança múltipla colabora para a legibilidade do código fonte.

## 6.5 ANALISE DE DESEMPENHO

Desde o inicio do projeto foi especificado que toda operação nativa não deve ter processamento extra para que seja executado.

O trabalho da monografia constitui a primeira versão da linguagem de programação Breder. Seguindo o processo iterativo e incremental, foi especificado para esta versão os recursos básicos para a linguagem funcionar não atentando para o desempenho. Com isso, não será abordado valores numéricos detalhados de comparação de desempenho com outras linguagens do mercado que já foram desenvolvidos durante anos e em versões superiores.

Para uma análise geral de comparação de desempenho da linguagem Breder com a linguagem Java, exemplos básicos mostraram que a linguagem Breder, na primeira versão, é 3 vezes mais lento do que a linguagem Java. Os principais motivos que justificam essa diferença de desempenho está na ausência da implementação e da otimização de algumas funcionalidades.

As principais funcionalidades que tornam o desempenho reduzido na linguagem Breder se referem ao *Just-In-Time* da máquina virtual e ao tratamento de *Pool* de objetos primitivos. A funcionalidade *Just-In-Time* será implementada em trabalhos futuros com o objetivo de retirar o custo de interpretação de um *bytecode*. Além disso, para os objetos primitivos, o compilador Breder ainda não é inteligente suficiente para determinar se realmente irá criar um objeto primitivo ou um tipo primitivo internamente implementado na máquina virtual Breder. Atualmente, que serão melhoradas em trabalhos futuros, as faixas de números que não estão no *pool* de objetos primitivos são criados como um objeto normal e coletado pelo coletor de lixo quando não mais utilizado.

# 7 CONCLUSÃO

A linguagem está hospedada no site www.breder.org na qual possui o seu download e as especificações sobre as interfaces da biblioteca BDK.

A linguagem Breder foi testada em vários ambientes diferentes e com sucesso. O primeiro desafio da linguagem foi rodar em um ambiente *móbile* como *iphone*. Um paper foi publicado no congresso na Argentina pelo Bernardo Breder, orientado por Esteban Clua, cuja a referência se encontra no link http://eiffel.itba.edu.ar/wavi2010/actas_wavi_2010.pdf.

Abaixo serão discutidas algumas conclusões do projeto da linguagem Breder com expectativas para a linguagem Breder e seu futuro.

## 7.1 EXPECTATIVA PARA A LINGUAGEM

O pacote BDK disponibiliza algumas interfaces já definidas que ainda não foram implementadas. As interfaces de entrada e saída de arquivos remotos, interface gráfica e chamada de rede são exemplos de estrutura que ainda não foram totalmente implementadas.

## 7.2 TRABALHOS FUTUROS

A funcionalidade de métodos dinâmicos de um instancia será desenvolvido para o objeto poder substituir a implementação de um método definido em sua classe, em tempo de execução, simulando o comportamento de um *proxy* de um método. Além disso, um método poderá ser armazenado como uma variável em um bloco e também pode ser passado como parâmetro para uma outra função. Essa funcionalidade é muito utilizada em linguagens dinâmicas tipo Lua e Python.

A tarefa de tornar a linguagem *thread-safe* será implementada futuramente, criando assim, vários mecanismo de controle de *thread*. Essa tarefa envolve desenvolver uma API de *thread* e modificar a máquina virtual para suportar o modelo de objetos de threads.

# 8 REFERÊNCIAS BIBLIOGRÁFICAS

[1] SEBESTA R. W., 2011. Conceitos de Linguagem de Programação. 9ª Ed. Bookman. 792p

[2] KUNG F., MOREIRA G., STEPPAT N., SILVEIRA P., LOPES S., 2009. Java como Plataforma, não como Linguagem, Arquitetura e Design de Software.

[3] Freitas, A. V. e Neto, J. J. Ambiente Multilinguagem de Programação - Aspectos do Projeto e Implementação Boletim Técnico PBT/PCS/0109, ISSN 1413, 215X, Escola Politécnica, São Paulo, 2000.

[4] EBNF - O standard ISO 14977 define uma extensão de BNF designado EBNF - http://www.cl.cam.ac.uk/~mgk25/iso-ebnf.html e http://www.deetc.isel.ipl.pt/programacao/Programacao_inv_2003_2004/poo/Slides/EBNF.ppt

[5] http://pt.wikipedia.org/wiki/Atributo_(programação)

[6] Arnold K., Gosling J., Holmes D., 2007. A Linguagem de Programação Java

[7] Aho A., Sethi R., Ullman J., 1995. Compiladores Princípios, Técnicas e Ferramentas

[8] Data Structures and Algorithms with Object-Oriented Design Patterns in Java at http://www.brpreiss.com/books/opus5/html/page424.html

[9] http://pessoal.utfpr.edu.br/bertoldo/Downloads/Compilador.pdf

[10] http://javacc.java.net/

[11] http://www.univasf.edu.br/~leonardo.campos/Arquivos/Disciplinas/POO_2007_2/Aula_05.pdf

[12] Caelum, 2009. Arquitetura e Design de Software.