UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO – BACHARELADO

RUNGROOVY:

EXTENSÃO DO BLUEJ PARA EXECUÇÃO DE LINHAS DE CÓDIGO

HENRIQUE MACHADO MÜLLER

BLUMENAU
2007

2007/2-18

HENRIQUE MACHADO MÜLLER

RUNGROOVY:

EXTENSÃO DO BLUEJ PARA EXECUÇÃO DE LINHAS DE CÓDIGO

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.

Prof. Adilson Vahldick, Especialista - Orientador

BLUMENAU
2007

2007/2-18

RUNGROOVY:

EXTENSÃO DO BLUEJ PARA EXECUÇÃO DE LINHAS DE CÓDIGO

Por

HENRIQUE MACHADO MÜLLER

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

Presidente:	Prof. Adilson Vahldick, Especialista – Orientador, FURB
Membro:	Prof. Marcel Hugo, Mestre – FURB
Membro:	Prof. Wilson Pedro Carli, Mestre – FURB

Blumenau, 7 de dezembro de 2007

Dedico este trabalho a todos os amigos, especialmente aqueles que me ajudaram diretamente na realização deste e em especial a minha namorada e minha família pelo incentivo para alcançar mais este objetivo.

AGRADECIMENTOS

A Deus, pelo seu imenso amor e graça.

Ao meu coordenador, Márcio Welter, pelo apoio para a realização deste trabalho.

À minha namorada, Elizandra Bozio e meus sogros que estiveram presentes nos momentos mais difíceis

Aos meus amigos, pelos empurrões, auxílio e cobranças.

Ao meu orientador, Adilson Vahldick, por ter acreditado e me ajudado no desenvolvimento e conclusão deste trabalho.

Para realizar grandes conquistas, devemos não apenas agir, mas também sonhar; não apenas planejar, mas também acreditar.

Anatole France

RESUMO

Este trabalho apresenta o desenvolvimento de uma extensão na ferramenta BlueJ para a execução de *scripts* em linguagem Groovy. Os códigos são interceptados para a criação de objetos e execução de métodos pela biblioteca de extensão do BlueJ, interagindo com a própria IDE. Esta extensão tem como objetivo auxiliar no aprendizado da programação orientada a objetos facilitando o entendimento de instanciação e integração com os objetos. Na implementação da ferramenta foi utilizado o ambiente de desenvolvimento NetBeans IDE 5.5.

Palavras-chave:Extensão do BlueJ. Aprendizado de orientação a objetos. Groovy.

ABSTRACT

This work presents the development of a BlueJ tool extension's for execution of scripts in the Groovy language. The codes are intercepted for objects creation and methods invocation by the extension library of BlueJ, interacting with the development tool's. This extension helps to learn the instantiation and interaction with objects. The NetBeans IDE 5.5 was used to implement the extension.

Key-words: BlueJ extension. Learning object-oriented. Groovy.

LISTA DE ILUSTRAÇÕES

LISTA DE SIGLAS

ABNT – Associação Brasileira de Normas Técnicas

API – *Application Programming Interface*

BCC – Curso de Ciências da Computação – Bacharelado

DSC – Departamento de Sistemas e Computação

PHP – *Personal Home Page*

SBC – Sociedade Brasileira de Computação

XML – *eXtensible Markup Language*

SUMÁRIO

1 INTRODUÇÃO

Com o avanço do desenvolvimento de softwares, dos modelos de programação e a velocidade com que as mudanças tecnológicas vêm acontecendo, surgem novos paradigmas como, por exemplo, a Programação Orientada a Objetos (POO). Boratti (2002, p. 1) afirma que: “Este paradigma [falando da POO] procura abordar a resolução de um problema através de uma construção que represente, da melhor forma possível, como as coisas acontecem no mundo real.”.

Para facilitar o ensino da POO os professores usam muitos artifícios para simular a criação de objetos, chamada de métodos e os acessos aos atributos, como por exemplo desenhos e ferramentas de apoio ao aprendizado. Além disso, a necessidade de receber estímulos durante o processo de aprendizagem é evidente nos alunos. Estas necessidades podem surgir por diversos motivos como a falta de tempo, dificuldades na compreensão de assuntos complexos, entre outros (GALHARDO; ZAINA, 2004, p. 109).

“Os mais experientes neste paradigma [falando de POO] declaram: 'Os programas orientados ao objeto são mais fáceis de escrever'. [...] Com freqüência, quando alguns novatos fazem alterações em comandos, dizem: 'Ele funciona, mas eu não sei porquê'.” (WINBLAD, 1993 apud SOUSA, 2002, p. 1).

Börstler, Bruce e Michiels (2003, p. 84) afirmam que programação a objeto é um tópico principal na ciência da computação e em muitas universidades a POO é ensinada de forma muito básica. Muitos aspectos devem ser considerados para ajudar o aluno a pensar orientado a objeto e muitos conceitos sobre POO devem ser apresentados e fixados para que possibilite um melhor aprendizado.

“Há a necessidade de ferramentas para o apoio do aprendizado dessa técnica. Esse suporte seria dado, por exemplo, por softwares que mostrassem como e quando objetos fossem instanciados, métodos utilizados e atributos acessados.” (SOUSA, 2002, p. 10).

O BlueJ tende a facilitar esse processo de aprendizado na POO, através de recursos visuais interativos, como por exemplo classes e relacionamentos que podem ser definidos visualmente. Também é possível verificar o comportamento dos objetos em memória durante a execução, como afirma Araújo e Daibert (2006, p. 37).

Há uma limitação na ferramenta BlueJ na execução direta de código, também não existem recursos para executar arquivos com códigos que possam gerar objetos no ambiente. Com isso surgiu a idéia de uma extensão que seja capaz de executar *scripts* e criar objetos na

ferramenta.

A extensão que esse trabalho implementa usa a linguagem Groovy, pois segundo Barroso (2006, p. 50-51), “Groovy é uma linguagem simples de aprender, principalmente pela sintaxe similar ao Java.” O autor também afirma que “Groovy pode ser utilizada em instituições de ensino, nas disciplinas de Introdução a Lógica de Programação com o propósito de utilizar uma linguagem, de script ágil [...]”.

Tendo em vista o que foi apresentado acima, a extensão do BlueJ desenvolvida neste trabalho mostra quando os métodos são executados, os atributos acessados e objetos instanciados, através de linhas de código digitadas pelo usuário, escritas em Groovy.

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é desenvolver um novo recurso no BlueJ onde o usuário entra com as linhas de código para criação de objetos e interação com eles (chamada de métodos). Esta interação é refletida no ambiente de desenvolvimento.

Os objetivos específicos do trabalho são:

a) desenvolver uma nova janela para entrada de linhas de código no BlueJ;
b) implementar a interpretação e execução dessas linhas de código pelo Groovy;
c) retornar o reflexo da execução dessas linhas para o ambiente BlueJ, criando objetos e alterando seus atributos através de execução de métodos;
d) disponibilizar recursos para manter linhas digitadas, gravando e lendo *scripts*.

1.2 ESTRUTURA DO TRABALHO

A estrutura deste trabalho está dividida em capítulos que serão explicados a seguir.

O primeiro capítulo apresenta a contextualização, objetivo e justificativa para o desenvolvimento do trabalho.

O segundo capítulo apresenta a fundamentação teórica do trabalho, apresentação da ferramenta de desenvolvimento BlueJ, conceitos sobre BlueJ e extensões, a linguagem Groovy e a relação de trabalhos correlatos.

O terceiro capítulo apresenta os requisitos da ferramenta, bem como sua especificação, implementação e resultados. São mostrados diagramas de classe, modelo de casos de uso e diagrama de atividades, exemplos de código Java necessário para atender aos comandos digitados pelo usuário. Também é apresentada a operacionalidade da ferramenta com um estudo de caso.

O quarto capítulo apresenta as conclusões, limitações e sugestões para futuros trabalhos.

2 FUNDAMENTAÇÃO TEÓRICA

Na seção 2.1 é apresentada a ferramenta de ensino BlueJ. Na seção 2.2 é evidenciado BlueJ e extensões. Na seção 2.3 são apresentadas informações e características sobre a linguagem Groovy. Na seção 2.4 estão descritos trabalhos correlatos.

2.1 A FERRAMENTA DE ENSINO BLUEJ

BlueJ, que foi desenvolvido pela Monash University na Austrália, é uma ferramenta de desenvolvimento projetado para o aprendizado e introdução da POO. Segundo Barnes e Kölling (2004), "O BlueJ é um ambiente interativo de desenvolvimento com uma missão: foi projetado para ser utilizado por alunos que estão aprendendo a programar. Foi projetado por instrutores que estiveram na sala de aula defrontando-se com esse problema todos os dias.".

"O *download* do BlueJ pode ser feito em *bluej.org*. Há instaladores nativos para o Windows e Mac OS X, e um JAR executável que pode ser usado em qualquer sistema operacional com suporta a Java." (ARAÚJO E DAIBER, 2006).

A Figura 1 apresenta a interface principal do BlueJ. Conforme Araújo e Daiber (2006) a grande área central é chamada de *Class Browser*, que é a representação gráfica de classes desenvolvidas no projeto do usuário, também sendo possível criar instâncias que aparecem na parte inferior, que é chamada de *ObjectBench*.

Para a criação de novos projetos é preciso acessar o menu *Project* e depois a opção *New Project*. O nome do projeto aparecerá na parte superior da janela. Pode-se observar o nome "NovoProjeto" na mesma Figura 1.

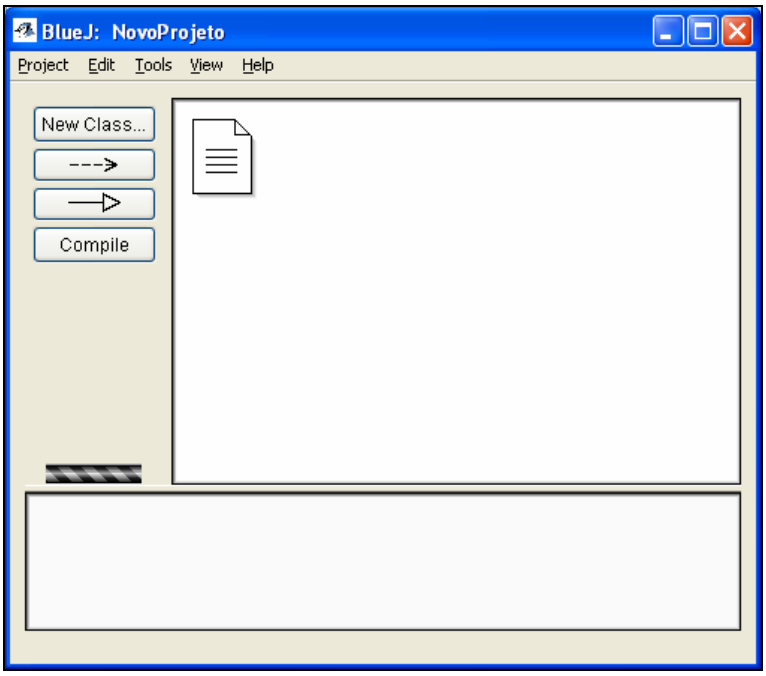


Figura 1 – Interface principal do BlueJ

Segundo Araújo e Daiber (2000, p.63), para adicionar uma nova classe ou interface no projeto, é preciso clicar no botão *New Class*. Uma janela, conforme a Figura 2, permitindo a definição do nome e o tipo de elemento será apresentada. A Figura 3 mostra cinco classes criadas, sendo elas: `Funcionário`, `FuncionarioHorista`, `FuncionarioDiarista`, `FuncionarioMensalista` e `Departamento`. Sendo que a primeira é abstrata e superclasse das três seguintes (ARAÚJO; DAIBER, 2006).

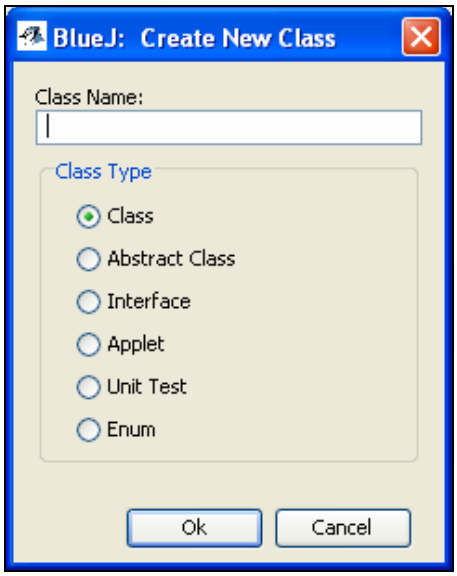
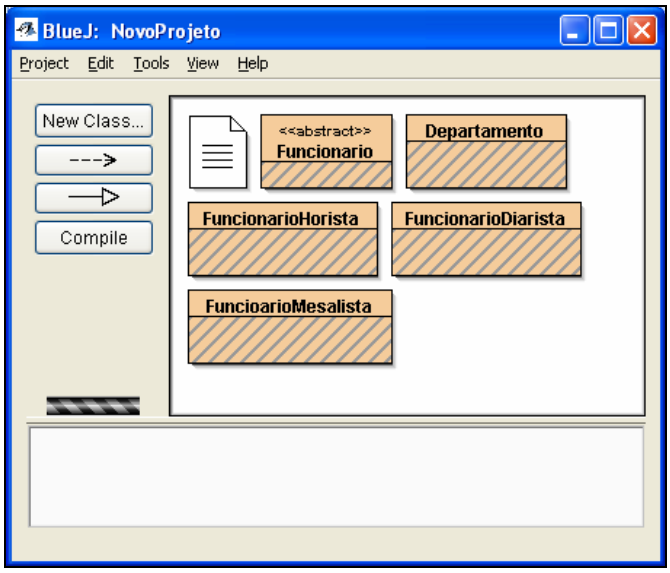


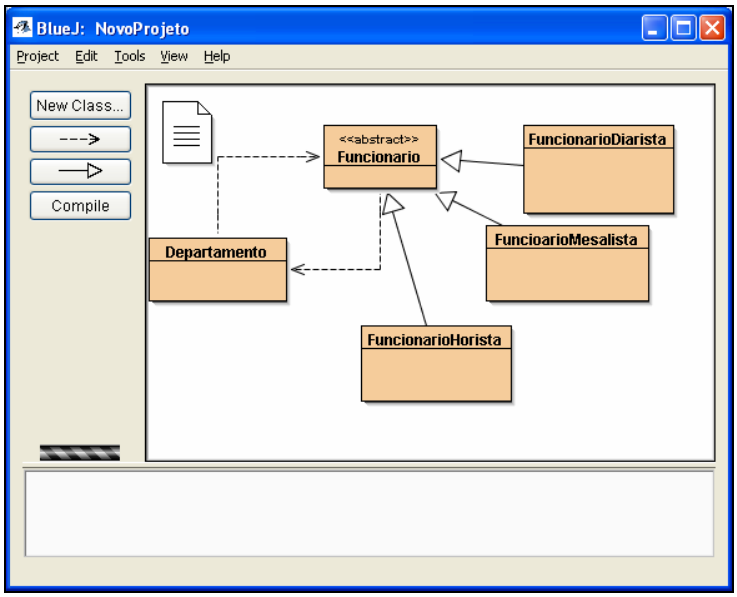
Figura 2 – Criação de nova classe



Fonte: adaptado de Araújo e Daiber (2006).

Figura 3 – Criação de classes

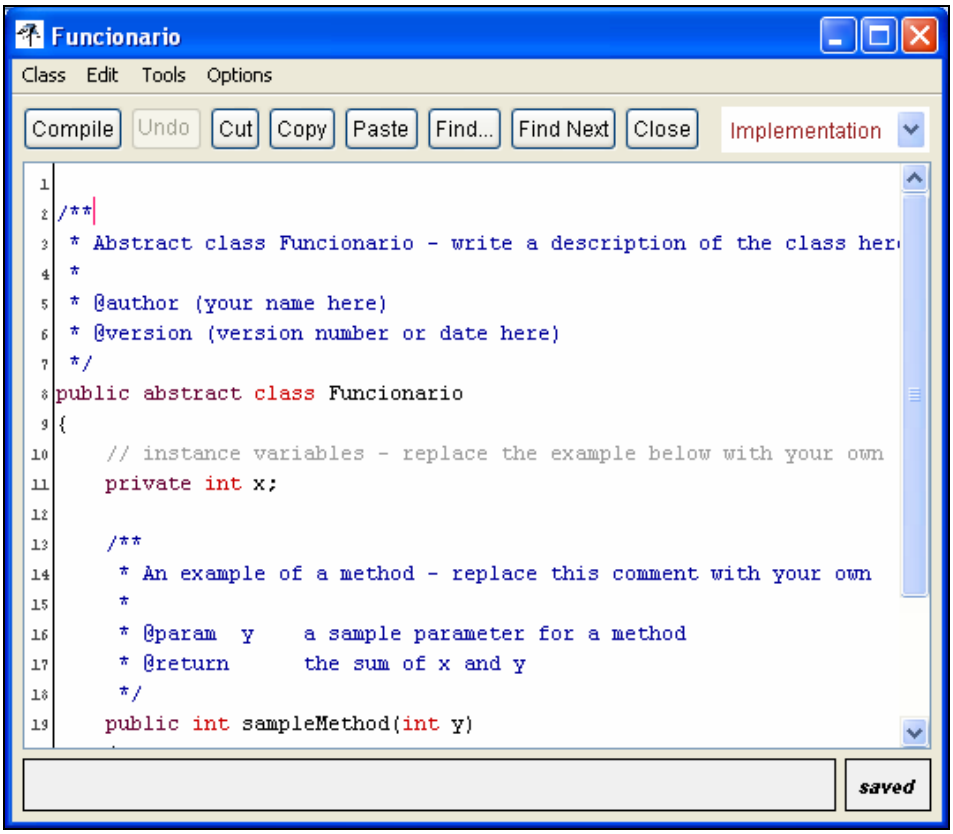
Pode-se também criar dependência entre classes. Para isso é necessário clicar no botão com uma seta tracejada, depois na classe de origem, e arrastar até a classe de destino. Para a criação de relacionamento de herança, o processo é semelhante. Deve-se pressionar o botão com a seta, selecionar a classe filha e arrastar até a classe pai. A Figura 4 mostra as classes com seus relacionamentos de herança e dependências. Para cada elemento do modelo o BlueJ cria um arquivo com código fonte, e mantém sincronizado o código com o modelo. (ARAÚJO; DAIBER, 2006).



Fonte: adaptado de Araújo e Daiber (2006).

Figura 4 – Dependências e relacionamento de herança

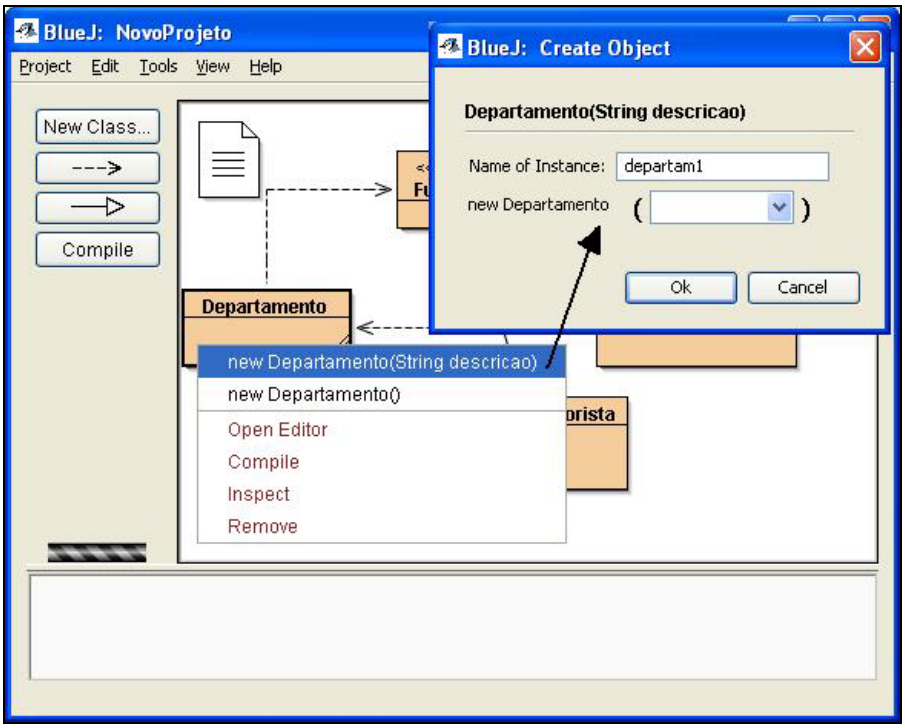
Para exibir o código fonte é preciso clicar duas vezes no elemento. Então o mesmo é aberto com *Class Browser* no qual é possível editar adicionando atributos e métodos, dentre outros. A Figura 5 mostra esse editor do Bluej.



Fonte: adaptado de Araújo e Daiber (2006).

Figura 5 – Editor do BlueJ

Assim que uma compilação é feita, é possível instanciar objetos clicando com o botão direito nos elementos visuais. Então é exibida uma janela com os campos necessários de acordo com o construtor selecionado. A Figura 6 mostra a criação de um objeto `Departamento`.



Fonte: adaptado de Araújo e Daiber (2006).

Figura 6 – Execução de construtor

Os objetos são criados no *ObjectBench* e todos os métodos, inclusive os herdados, podem ser invocados de maneira similar à invocação do construtor, deve-se clicar com o botão direto do mouse no objeto criado. Esta execução pode levar a uma janela mostrando o resultado do método, caso o mesmo seja diferente de `void`. Na Figura 7 é observada esta execução. Também é possível inspecionar os objetos ao clicar duas vezes sobre ele. Esta inspeção mostrará todos seus atributos. A Figura 8 mostra uma inspeção do objeto departamento.

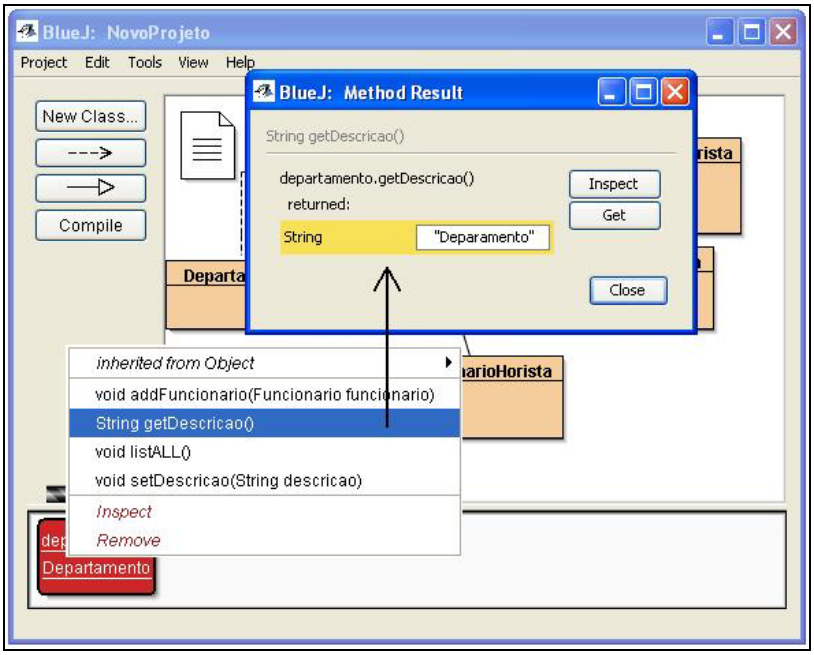
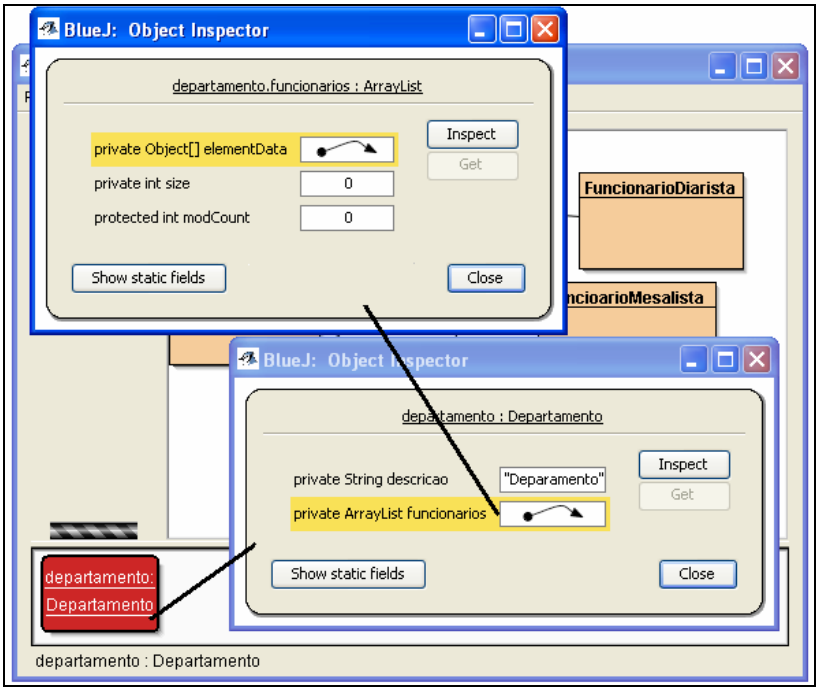


Figura 7 – Execução de método



Fonte: adaptado de Araújo e Daiber (2006).

Figura 8 – Inspeção de um objeto

2.2 BLUEJ E EXTENSÕES

Existe uma *interface* de classes na qual é possível desenvolver extensões para a ferramenta. Segundo Kölling (2007a), BlueJ oferece uma *Application Programming Interface*

(API) na qual permite que terceiros desenvolvam extensões ao ambiente. As extensões oferecem a funcionalidade adicional não incluída no núcleo do sistema.

As API do BlueJ oferecem o acesso ao menu *Tools* do ambiente, e aos menus de classe e de objeto, ao painel de *Tools/Preferences/Extensions* e podem interagir com o editor de classes para recuperar e modificar código fonte (KÖLLING, 2007b).

O acesso para extensões à aplicação se dá através do pacote `bluej.extensions` e suas classes tem a função de instanciar e manipular os objetos criados na ferramenta BlueJ. Uma extensão no BlueJ sempre começa pela extensão da classe abstrata `Extension`. Esta classe contém uma instância de `BlueJ` e que por sua vez contém um conjunto de classes invólucros (*Wrapper classes*) para a criação de objetos dentro de qualquer pacote da aplicação do usuário, como também invocação de métodos. A Figura 9 mostra o diagrama de classes que interagem com objetosdo *ObjectBench* deste pacote.

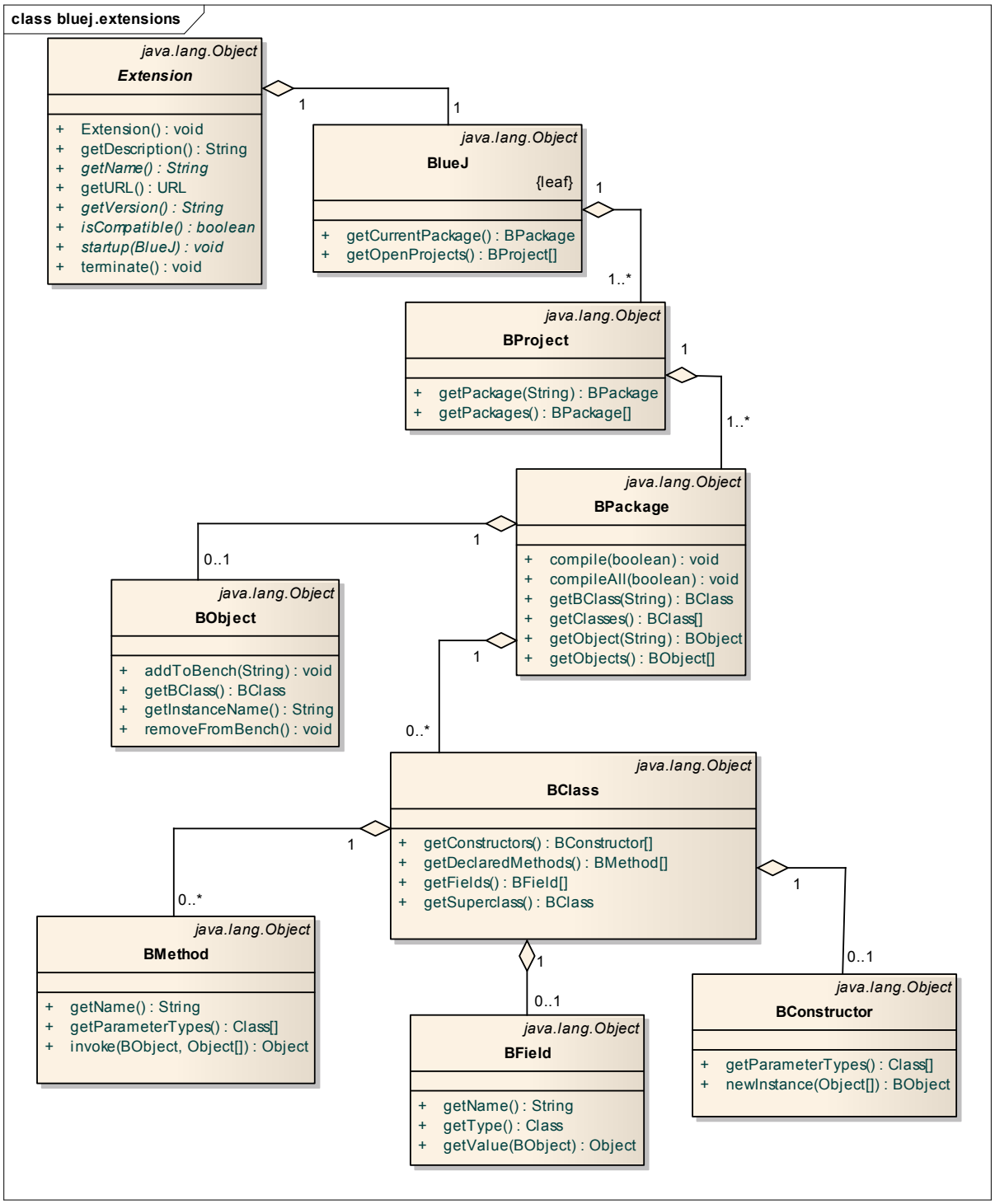


Figura 9 – Diagrama de classes do pacote `bluej.extensions`

A Quadro 1 mostra as classes e responsabilidades do pacote `bluej.extensions`, com descrição de seus principais métodos (KÖLLING, 2007c).

`Extension`	Classe principal onde começa uma comunicação entre o BlueJ e uma extensão. O primeiro método a ser chamado é o `startup(BlueJ)`.
`BlueJ`	É um *proxy object* que provê informações do BlueJ para a

	extensão. Com esta classe é possível obter todos os projetos e pacotes que estão visíveis no projeto do usuário. Assim como as classes e objetos que estão contidos neles. Também é possível adicionar menus e preferências no painel de extensão. Este painel é encontrado em *Preferences* do menu *Tools* na aba *Extensions*. Seus principais métodos são: g`etOpenProjects()`, `setMenuGenerator(MenuGenerator)`, `getCurrentPackage()`.
`BProject`	Esta classes contém operações sobre um projeto no BlueJ. Principais métodos: `getPackages()` que retorna todos os pacotes do projeto; `getDir()` retorna o diretório onde o projeto é salvo; `newPackage(String)` cria novos pacotes através da extensão.
`BPackage`	Contém operações sobre um pacote do BlueJ. Seus principais métodos são: `compileAll(boolean)` que compila todos as classes; `getClasses()` retorna todas as classes do pacote; `getBObjects()` retorna todos os objetos criados no *ObjectBench*.
`BObject`	É responsável por conter informações sobre um objeto, que pode ser adicionado e removido do *ObjectBench*. Principais métodos: `addToBench(String)` acidiona o objeto para o O*bjectBench*; `removeFromBench()` remove o objeto; `getBClass()` retorna a classe do objeto; `getInstanceName()` retorna o nome do objeto.
`BClass`	Contém informações de uma classe criada no BlueJ. Com esta classe é possível capturar todos os métodos, atributos e construtores públicos disponíveis. Sua utilização é similar a API *reflection* do Java. Seus principais métodos são: `getConstructors()`, `getDeclaredMethods()`, `getFields()`, `getMethods()`.
`BMethod`	Com uso similar à API *reflection* do Java esta classe contém informações sobre um método da classe. Este método pode ser de classe ou de instância. Os principais métodos para utilização são: `getName()` retorna o nome do método; `getParameterTypes()` retorna um `Class` contendo o tipo de cada parâmetro do método; `invoke(BObject, Object[])` faz a invocação de um método.
`BField`	Com uso similar a API *reflection*, esta classe contém informações

	sobre um atributo (*field*) de uma classe.
`BConstructor`	Contém informações sobre os construtores, sua utilização é similar a API *reflaction* e seus principais métodos são: `getParameterTypes()`, `newInstance(Object[])`. O primeiro retorna a classes dos parâmetros, e o segundo cria instâncias de `BObject`.
`MenuGenerator`	Classe responsável por adicionar item ao menu do BlueJ. Este menu pode ser o menu *Tools*, o menu de um elemento gráfico que representa uma classe ou um menu da representação de um objeto pelo *ObjectBench*. Os principais métodos desta classe são: `getToolsMenuItem()`, `getClassMenuItem(BClass bc)`, `getObjectMenuItem()`.
`PreferencesGenerator`	Única interface do pacote. É responsável por adicionar preferências no item `Preferences` do menu `Tools` a aba `Extension`. Para adicionar é preciso retornar um `JPanel` na implementação do método `getPanel()`. Para gravar e retornar valores existem os métodos: `loadValues()` e `saveValues()`.

Quadro 1 – Funcionabilidades das classes do pacote `bluej.extensions`

No mesmo pacote ainda existe as classes de exceção (Quadro 2).

`ClassNotFoundException`	Exceção é disparada quando a classe não é mais válida, a razão mais provável é uma classe ser excluída.
`CompilationNotStartedException`	Exceção quando uma compilação não é iniciada. A razão mais provável é que o BlueJ esteja com uma execução de código de classes.
`ExtensionUnloadedException`	Esta exceção é lançada quando uma extensão é terminada, pelo método `Extension.terminate()` e após outro método é chamado.
`InvocationArgumentException`	Esta excecão é lançada quando os parâmetros de uma invocação não correspondem à lista de parâmetros.
`InvocationErrorException`	Exceção disparada quando ocorre uma exceção durante a invocação de um método ou construtor. A causa mais provável é o usuário cancelar uma invocação longa do construtor ou método.
`MissingJavaFileException`	Esta excepção será lançada quando uma nova classe é criada e um arquivo fonte Java não é fornecido.
`PackageAlreadyExistsException`	Exceção disparada quando há um pedido para criar um novo pacote, mas o pacote já existe no BlueJ.
`PackageNotFoundException`	Esta exceção será lançada quando uma referência a um pacote não é mais válida. A razão mais provável é que o usuário tenha excluída o mesmo
`ProjectNotOpenException`	Esta exceção será disparada quando uma referência a um projeto não é mais válida. A razão mais provável é que o usuário tenha fechado o projeto.
`ExtensionException`	Interface, sem nenhum método, que todas as exceções a implementam.

Quadro 2 – Classes de exceção do pacote `bluej.extensions`

Outro pacote existente é o `bluej.extensions.event` que se encarrega de gerar eventos quando o usuário executa ações significativas dentro de BlueJ, como por exemplo, quando a compilação do projeto foi iniciada, quando for gerado algum erro durante a compilação e quando a aplicação do projeto for iniciada. Estes eventos são capturados por classes *listener* do pacote.

Kölling (2007a) explica que para instalar as extensões no ambiente basta colocar o arquivo `.jar` em um diretório específico. O BlueJ tem três opções separadas, onde cada uma delas dá à extensão um escopo diferente. As opções são:

a) para um único projeto: na subpasta `extensions` do diretório onde o projeto está implementado;
b) para um único usuário do sistema: dentro do diretório de trabalho do usuário, na pasta `bluej` e depois na subpasta `extensions`;
c) para todos os usuários do sistema: dentro do diretório de instalação do BlueJ, na pasta `lib` e em seguida `extensions`.

2.3 GROOVY

Groovy é uma linguagem de *scripts* com os códigos sendo interpretados quando os mesmos são executados, assim como PHP ou JavaScript. Também a possibilidade de um *script* ser compilado gerando um `.class` totalmente compatível com a *Java Virtual Machine* (JVM).

Segundo König (2007, p.32), Groovy faz importação automática dos pacotes: `groovy.lang.*`, `groovy.util.*`, `java.lang.*`, `java.util.*`, `java.net.*` e `java.io.*` também das classes `java.math.BigInteger` e `BigDecimal`, diferenciando do Java que faz importação apenas do pacote java.lang.*, assim facilitando o uso dessas classes.

"Um ponto que deve ficar claro é que Groovy não substitui a linguagem Java e sim complementa adicionando uma linguagem de desenvolvimento ágil para escrever *scripts* e aplicações com interoperabilidade com a plataforma J2SE e J2EE." (BARROSO, 2006, p. 50).

Conforme Barroso (2006, p. 52-53), Doederlein (2006, p. 38-39), König (2007, p.32), Codehaus Foundation (2007a), Groovy tenta ser tão natural quanto possível para os desenvolvedores Java, seguindo o princípio do menor esforço para o processo de aprendizado. Groovy é uma linguagem com sintaxe semelhante ao Java, mas possui suas diferenças. Algumas delas são:

a) igualdade: o símbolo `==` significa igualdade para tudo, diferente de Java aonde é usado `==` e `===` para comparação de objetos. Ou seja em Groovy `==` é igual a função `equals()` em Java;
b) final de comando: em Groovy o símbolo `;` é opcional, desde que a linha só tenha um comando, no caso de mais é obrigatório;
c) matriz: Para declaração de matriz não é possível fazer igual ao Java o Quadro 3 mostra como deverá ser declara;
d) assinatura do método: A cláusula `throws` na assinatura de um método não é controlado pelo compilador Groovy, porque não há nenhuma diferença entre verificada e não verificada;
e) comandos de repetição: O uso de comandos `while` é igual ao Java, porém o uso de `for` tem a sintaxe diferenciada. O Quadro 4 mostra as opções para o uso de `for`;
f) *string*: permite o uso de aspas simples e aspas duplas combinadas em uma mesma

string como pode ser visto na Figura 10. As *strings* em Groovy podem conter as expressões `${}` para acessar atributos e métodos da classe, similar ao recurso encontrado em Java Server Pages (JSP);

g) tipos dinâmicos: Groovy não requer tipagem forte, como por exemplo Java, ao invés disso o tipo do objeto é descoberto dinamicamente em tempo de execução. Com isso a quantidade de código também é reduzida. Um exemplo deste recurso é mostrado na Figura 11a;

h) propriedades de uma classe: existe um recurso chamado Groovy Beans, similar ao JavaBean para criação de JSP, utilizando a anotação `@Property` no código da classe e já são gerados os métodos *get* e *set* do atributo. Um exemplo que utiliza os métodos com a notação apresentada também é visto na Figura 11b;

i) sobrecarga de operadores: cada operador primitivo pode ser redefinido através de um método. Por exemplo, o operador `+` pode ser redefinido através do método `plus(b)`. Uma implementação é mostrada na Figura 12. "[...] imagine uma classe **Orcamento** com um operador +, recebendo um novo **Projeto** que é associado ao orçamento." (DOEDERLEIN, 2006, p. 39, grifo do autor). É possível redefinir também operadores de comparação.

```
-- Declaração de Matriz em Java
Int[] uma = (1,2,3);
-- Declaração de Matriz em Groovy
Int[] uma = [1,2,3]
```

Quadro 3 – Declaração de matriz em Groovy

```
-- Comando for em Java.
for ( int i=0; i < len; i++) {
  ...
}
-- Groovy existe três alternativas para isso que são
for (i in 0..len-1) {
  ...
}
for (i in 0..<len) {
  ...
}
len.times {
  ...
}
```

Quadro 4 – Uso de comandos de repetição for

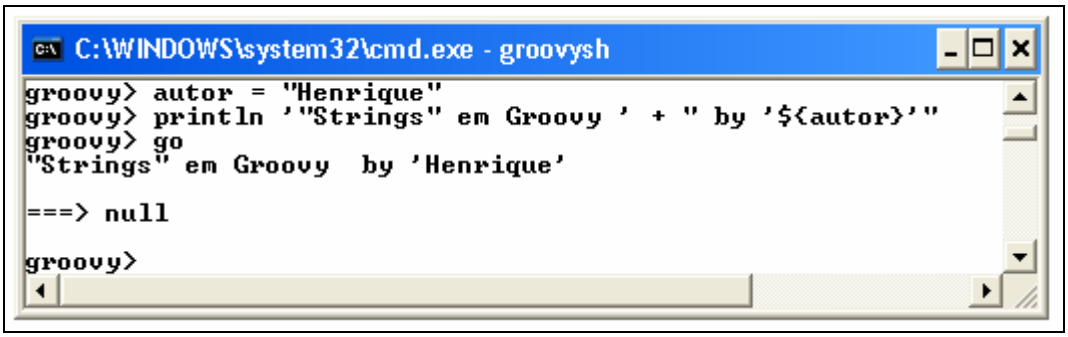


Figura 10 – *Strings* em Groovy

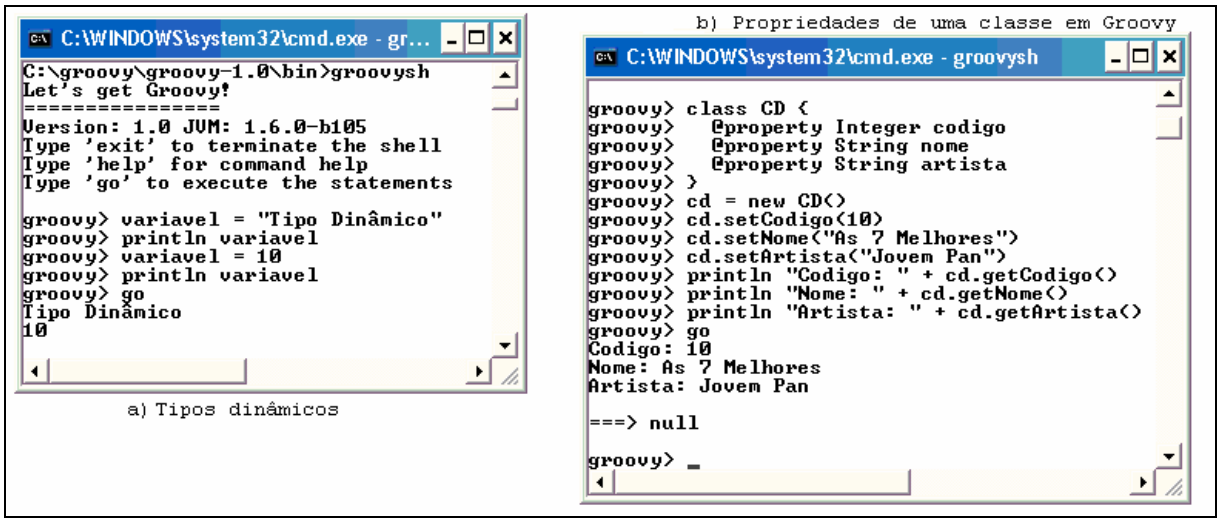


Figura 11 – Tipos dinâmicos e propriedades de uma classe com Groovy

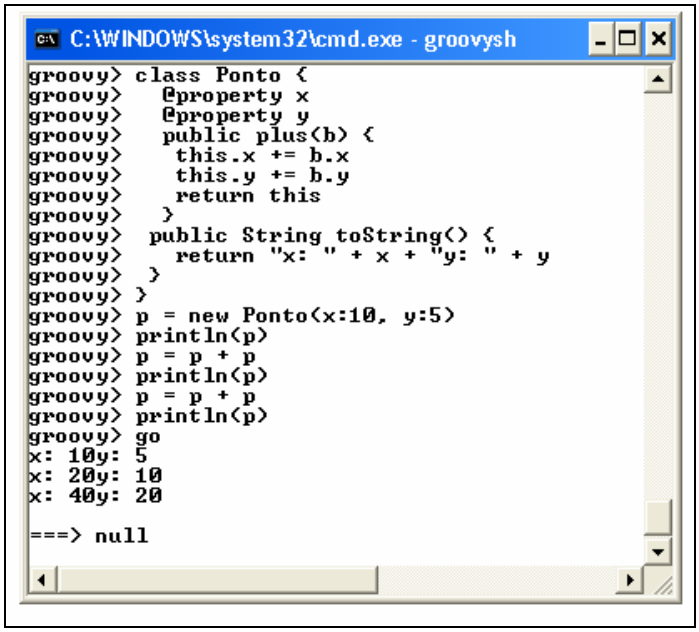


Figura 12 – Sobrecarga de operadores em Groovy

Segundo Bliki (2007), para integrar o Groovy a um programa Java é possível utilizar a classe `groovy.lang.GroovyShell`. O Quadro 5 mostra um exemplo de utilização. Para interagir com o *script* utiliza-se um `groovy.lang.Binding`, que é uma classe mensageira,

trazendo e levando informações entre Groovy e Java. Sua utilização é observada no Quadro 6.

```
private static void exemplo6() throws CompilationFailedException {
            GroovyShell shell = new GroovyShell();
            Object resultado = shell.evaluate(
               "println 'teste' ; return 'groovy'");
            System.out.println("Resultado é "+resultado);
}
```

Fonte: adaptado de Bliki (2007).

Quadro 5 – Exemplo para utilização da classe GroovyShell

```
private static void exemplo7() throws CompilationFailedException {
      Date dataDeDoje = new Date();
      Binding binding = new Binding();
      binding.setVariable("hoje",
java.text.SimpleDateFormat.getInstance().format(dataDeDoje););

      GroovyShell shell = new GroovyShell(binding);

      shell.evaluate (
           "def a='variavel definida dentro do script' ; " +
           " println 'hoje é: '+ hoje ");

      Object variavelDoScript = binding.getVariable("a");
      System.out.println(variavelDoScript);
}
```

Fonte: adaptado de Bliki (2007).

Quadro 6 – Exemplo para utilização da classe Binding

Conforme König (2007, p.373-374) na lista de construtores da classe `GroovyShell` é notado um parâmetro `CompilerConfiguration`. Esta classe serve para personalizar diversas opções de compilação do processo. A Quadro 7 mostra alguns métodos da classes `CompilerConfiguration` que podem fazer estas personalizações.

`setClasspath(String path)`	Possibilita definir seus próprios classpath usado para procurar as classes, permitindo restringir a aplicação e/ou reforçar – lá com outras bibliotecas.
`setDebug(boolean debug)`	Caso for `true` não será filtrado quando ocorrer exceções no stacktraces
`setOutput(PrintWriter writer)`	Defina o `PrintWriter` que serão impressas os erros.
`setScriptBaseClass (String clazz)`	Possibilita definir uma subclasse de Script como a classe base para o script
`setSourceEncoding(String enc)`	Definir a codificação dos scripts ao analisar arquivos pode ser usado caso a codificação for diferente do que a plataforma padrão.

Fonte: adaptado de König (2007, p. 374).

Quadro 7 – Métodos da classe `CompilerConfiguration`

Com Groovy, também é possível interceptar métodos invocados com um conjunto das classes `ProxyMetaClass` e `TracingInterceptor`. O Quadro 8 mostra uma simples utilização. Podem-se observar que o construtor e o método foram interceptados pela classe `TracingInterceptor`, isso após o uso do `ProxyMetaClass`. Esta intercepção mostrada apenas cria um buffer dos eventos ocorridos. Caso seja criada uma classe que herda de

`TracingInterceptor` podem-se interceptar os métodos e fazê-los retornar outros objetos (KÖNIG, 2007, p. 222-223).

```
import org.codehaus.groovy.runtime.StringBufferWriter
class Whatever {
  int outer(){
    return inner()
  }
  int inner(){
    return 1
  }
}
def log = new StringBuffer("\n")
def tracer = new TracingInterceptor()
tracer.writer = new StringBufferWriter(log)
def proxy = ProxyMetaClass.getInstance(Whatever.class)
proxy.interceptor = tracer
proxy.use {
  Int a = new Whatever().outer()
}

print log.toString()
//------- resultado ------
before Whatever.ctor()
after  Whatever.ctor()
before Whatever.outer()
after  Whatever.outer()
```

Fonte: adaptado de König (2007, p. 223)

Quadro 8 – Intercepção com as classes `ProxyMetaClass` e `TracingInterceptor`

Conforme a Codehaus Foundation (2007b) na sua documentação da classe `TracingInterceptor` ela é composta de três principais métodos que são:

a) `beforeInvoke(Object object, String methodName, Object[] arguments)`: este código é executado antes da invocação do método ou construtor que foi interceptado. O argumento `object` contém o objeto no qual o método é chamado. No caso de método de classe, este parâmetro contém o `Class`. O nome do método a ser executado é obtido pelo parâmetro `methodName` e os argumentos pelo parâmetro `arguments`. O retorno deste método poderá ser usado no `afterInvoke` no caso do método interceptado não ser chamado;
b) `doInvoke()`: o retorno deste código decide se o método será invocado. Caso seja falso, nenhum método é executado;
c) `afterInvoke(Object object, String methodName, Object[] arguments, Object result)`: este código é executado apos a invocação do método ou construtor que foi interceptado. Os três primeiros parâmetros são idênticos ao `befereInvoke` e o parâmetro `result` é o resultado do código `beforeInvoke` ou do método interceptado, isto é definido pelo retorno do evento `doInvoke()`. O retorno deste método será o retorno do método interceptado.

O Quadro 9 mostra um exemplo de utilização de intercepção com o uso de herança da

classe `TracingInterceptor`.

```
import org.codehaus.groovy.runtime.StringBufferWriter
class Whatever {
  int outer(){
    return 1
  }
}
class MyInterceptor extends TracingInterceptor {
  int qtd;
  String method;
  public Object beforeInvoke (Object object, String methodName, Object[]
arguments) {
    method = methodName
    return null
  }
  public Object afterInvoke (Object object, String methodName, Object[]
arguments, Object result) {
    if (methodName == "ctor")
      return result
    qtd = qtd + 1;
    return qtd;
  }
  public boolean doInvoke () {
   if (method == "ctor")
      return true
    return false;
  }
}
def log = new StringBuffer("\n")
def tracer = new MyInterceptor()
tracer.writer = new StringBufferWriter(log)
def proxy = ProxyMetaClass.getInstance(Whatever.class)
proxy.interceptor = tracer
int a
proxy.use {
  whatever = new Whatever()
  a = whatever.outer()
  println a
  a = whatever.metodoNaoDeclarado()
  println a
  a = new Whatever().outer()
  println a
}
print log.toString()
//------- resultado ------
1
2
3
```

Quadro 9 – Intercepção com herança da classe `TracingInterceptor`

É observado que o método `beforeInvoke` apenas atribuiu o nome do método para a variável `método`. No método `doInvoke` ela é verificada e caso seja um construtor da classe o retorno será `true` o que fará com que o construtor seja executado e caso não seja, o método interceptado não é executado, assim não é gerado exceção quando executa-se um método não declarado na classe. Em seguida o código `afterInvoke()` onde caso não seja um construtor o retorno deste método será a quantidade de vezes que foi feito uma invocação.

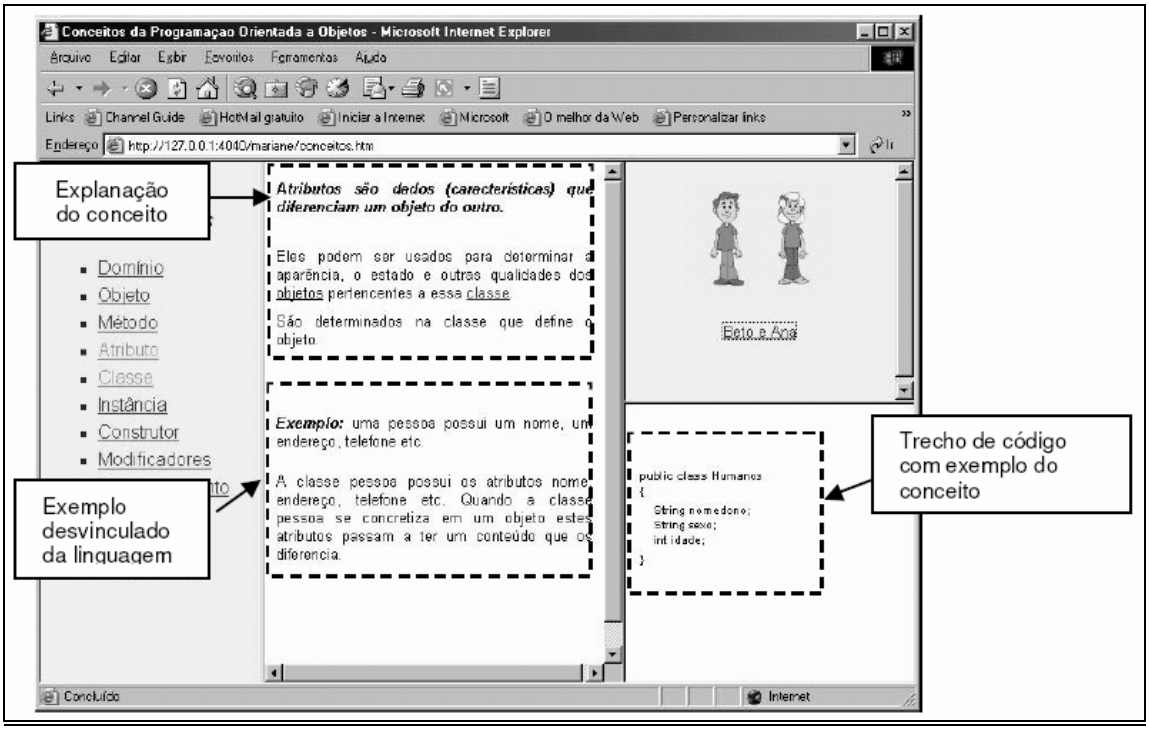
2.4 TRABALHOS CORRELATOS

Na seção 2.4.1 é demonstrada uma ferramenta de simulação para ensino que foi apresentada no XIII Seminário de Computação, em setembro de 2004. Na seção 2.4.2 é apresentada uma funcionalidade da ferramenta BlueJ chamada *Code Pad*.

2.4.1 Simulação para Ensino de Conceitos da Orientação a Objetos

O objetivo desta ferramenta de simulação é estimular o aluno no processo de aprendizado da POO, que executa no ambiente Web e possibilita ao aluno visualizar como se constrói uma classe usando técnicas da POO. Galhardo e Zaina (2004, p. 109) afirmam que: "Através de algumas interações, o aluno informará os dados solicitados pelo simulador, o sistema transportará estes dados traduzindo-os para um formato de uma linguagem orientada a objetos: a linguagem Java.".

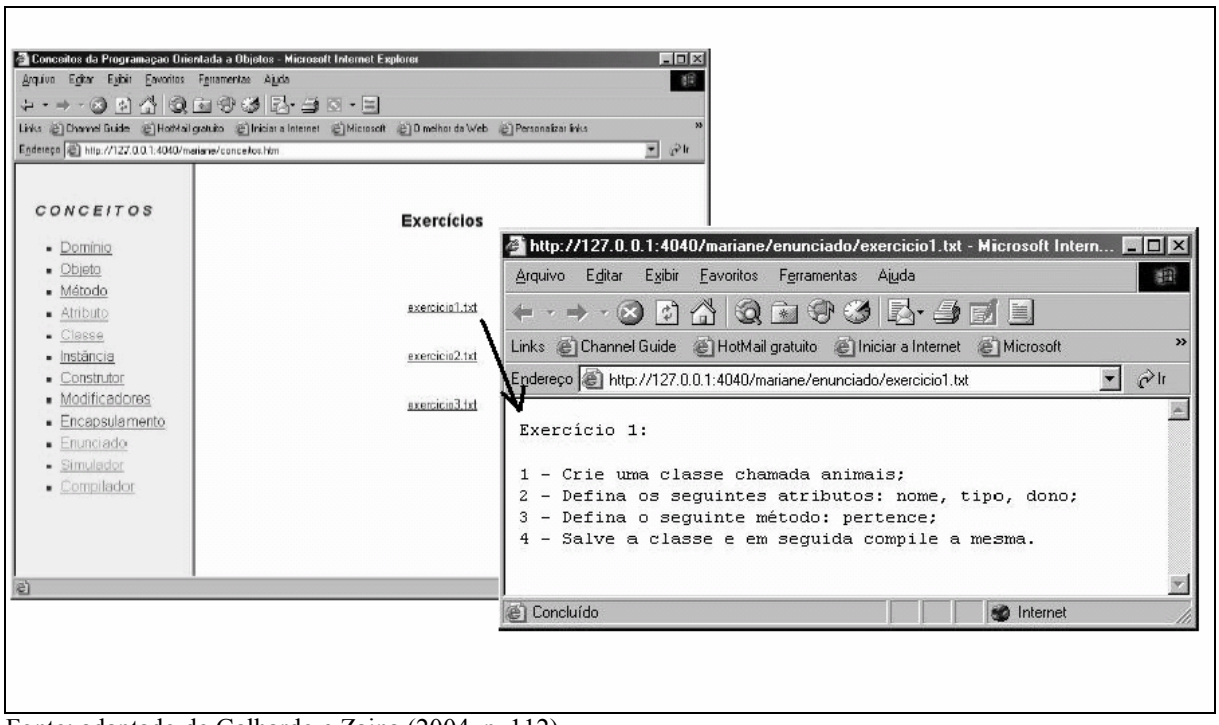
A ferramenta tem como função explicar teoricamente os conceitos da POO, exemplificando de maneira simples e de fácil entendimento estes mesmos conceitos ao longo das páginas *HyperText Markup Language* (HTML) (GALHARDO E ZAINA, 2004). Na Figura 13 é apresentada uma *interface* do simulador onde os conceitos são explicados. Estes conceitos dão suporte para o aluno interagir com a ferramenta.



Fonte: Galhardo e Zaina (2004, p. 111).

Figura 13 - Apresenta o conceito de atributo e seus respectivos exemplos

O simulador recebe um enunciado do professor, que pode ser salvo de forma a poder utilizá-lo mais vezes. O aluno, por sua vez busca o enunciado proposto e salvo pelo professor, conforme demonstrado na Figura 14.



Fonte: adaptado de Galhardo e Zaina (2004, p. 112).

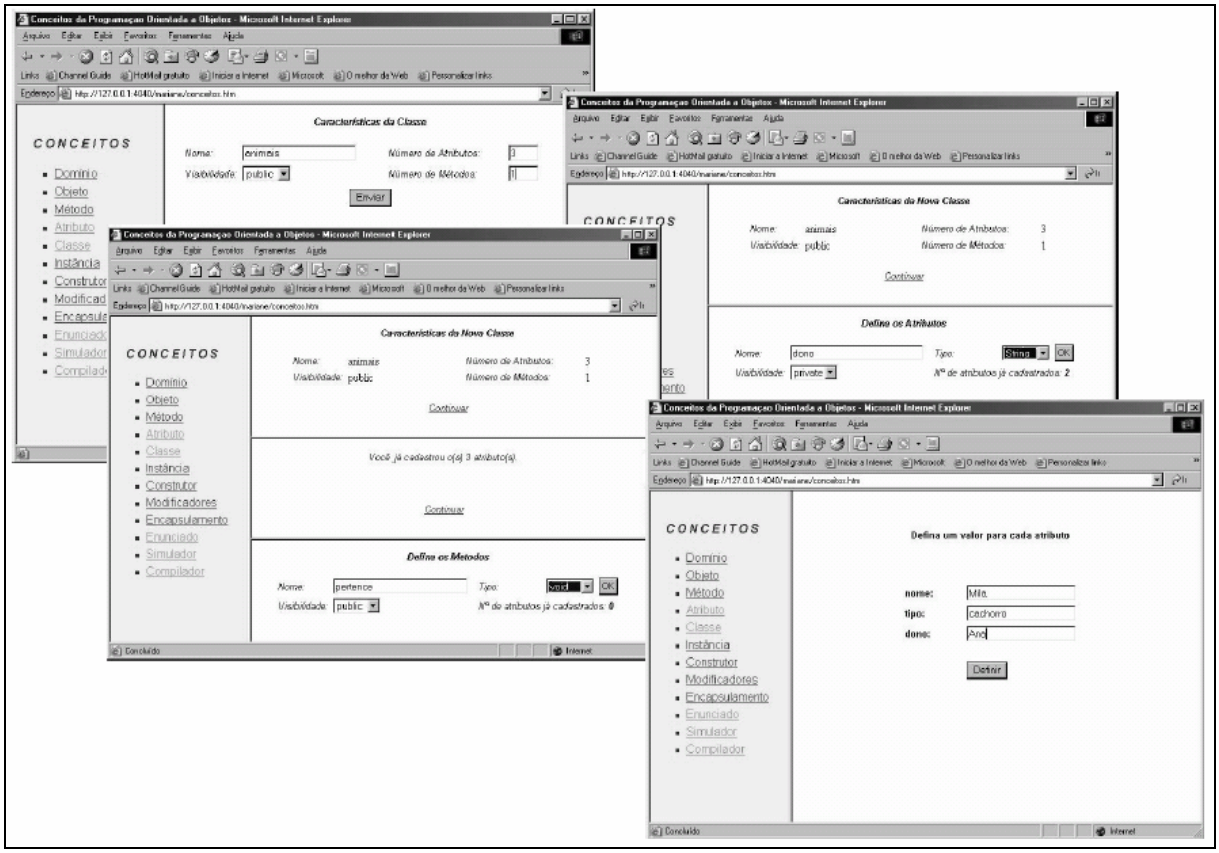
Figura 14 - Enunciados dos exercícios disponíveis aos alunos

Durante o processo de resolução do exercício o aluno fornecerá, gradativamente,

> todos os dados necessários para construção do código-solução, que partem das características gerais da classe, passam pela definição dos atributos e métodos e terminam com a definição dos valores iniciais dos atributos para serem utilizados pelo construtor da classe. (GALHARDO; ZAINA, 2004, p. 112).

O aluno resolve os exercícios propostos e constrói seus próprios códigos de solução do exercício. Todo o percurso deste processo é apresentado pelas telas da ferramenta como mostra a Figura 15.

Terminado o processo de resolução do exercício é gerado um código fonte em Java. Com isso a ferramenta permite ajudar no processo de aprendizado, passando de algo abstrato (conceito) para algo concreto (código-fonte). Também é possível alterar o resultado gerado e compilar. A ferramenta informa se houve sucesso neste processo ou mostrará os erros para o aluno, possibilitando o mesmo alterá-lo e corrigí-lo.



Fonte: Galhardo e Zaina (2004, p. 113).

Figura 15 – Construção da solução do exercício passo a passo

2.4.2 *Code Pad* do BlueJ

No ambiente de desenvolvimento BlueJ existe uma tela de console para execução de

linhas de código. Esta permite a execução de um comando por linha e a interação com a ferramenta em tempo de depuração, quando se faz invocação de métodos. Também é permitida a criação de objetos a partir de classes definidas no projeto aberto. Ainda é permitido a execução de comando básico, como por exemplo: 4 + 5 ou 33 % 4.

Existem duas maneiras de criar objetos nesta interface: a primeira é sem a integração do ambiente de objetos do BlueJ. Um exemplo de código é apresentado na Figura 16; a outra forma já integra-se com o ambiente de depuração da ferramenta, demonstrado na Figura 17. Para que isto ocorra é necessário que o aluno após a execução do comando arraste para criar o objeto.

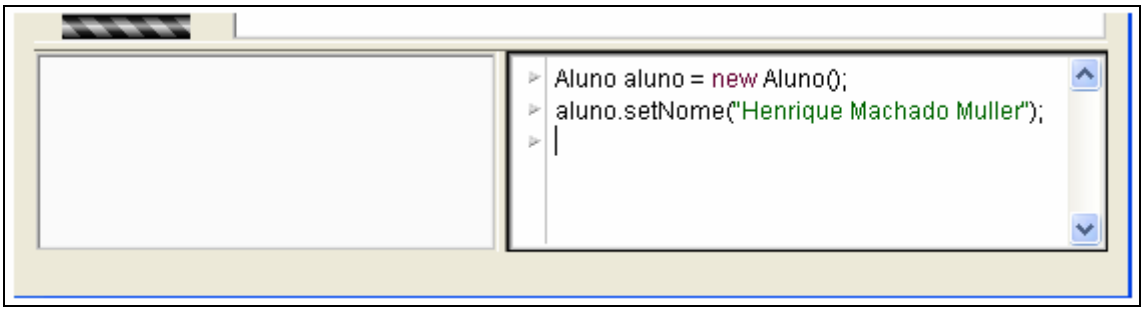


Figura 16 – Código sem integração com o ambiente

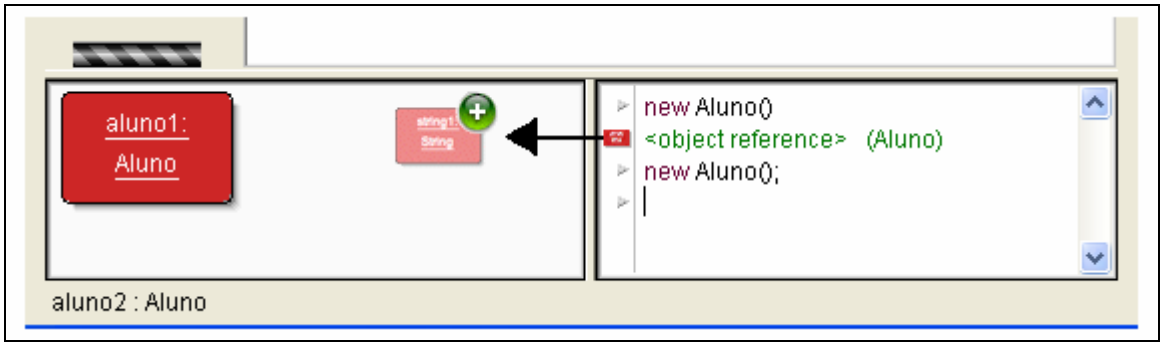


Figura 17 – Código com integração com o ambiente

É possível fazer comandos de repetição nesta ferramenta, mas uma limitação nestes comandos é que a criação de objetos não pode ser redirecionada para o *ObjectBench*, já que o *enter* do teclado executa o comando. A Figura 18 mostra esta execução.

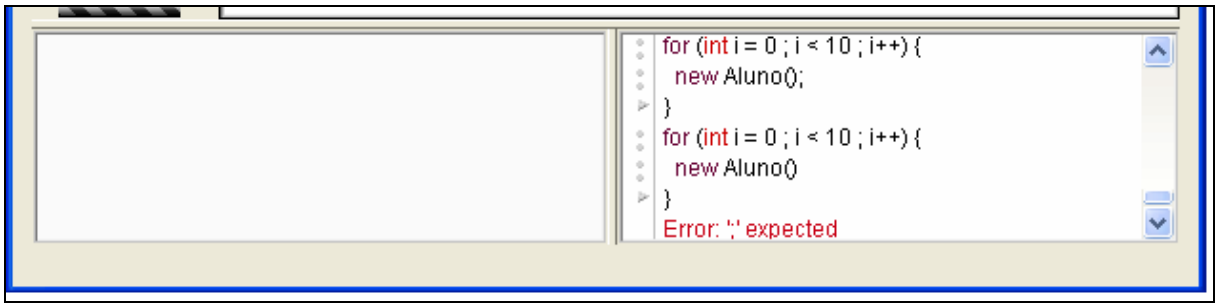


Figura 18 – Comandos de repetição

3 DESENVOLVIMENTO DA EXTENSÃO

Este capítulo apresenta os requisitos da ferramenta, bem como sua especificação, implementação e resultados obtidos. São mostrados diagramas de classe, modelo de caso de uso e diagrama de atividades, um exemplo de código Java e Groovy necessário para atender aos comandos digitados. Também são apresentadas explicações sobre o desenvolvimento da aplicação apresentada.

3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

Abaixo estão apresentados os principais requisitos funcionais e não-funcionais da extensão:

a) possuir uma janela de console para execução de comandos que criem objetos e façam chamadas de métodos (Requisitos Funcionais – RF);
b) criar e alterar o estados dos objetos criados pela ferramenta BlueJ (RF);
c) permitir nessa janela que se salve, carregue e execute arquivos contendo linhas de código (RF);
d) utilizar a linguagem Groovy para execução e instrução dessas linhas de código (Requisitos não Funcionais – RNF);
e) ser compatível com qualquer sistema operacional que tenha suporte a JVM (RNF).

3.2 DESCRIÇÃO DA EXTENSÃO

A extensão desenvolvida adiciona o opção *RunGroovy* ao menu *tools* da ferramenta BlueJ. Esta opção, ao ser selecionada, abrirá uma nova janela de console na qual é permitida a digitação de linhas de códigos. Esses comandos podem ser salvos em arquivo no disco e recuperados posteriormente para edição na tela e execução. Toda execução de comando nesta janela será refletida para o ambiente BlueJ com criação de objetos no *ObjectBench*, assim como uma execução de método também será refletida. Nesta janela é possível acessar

qualquer objeto que estiver no *ObjectBench* pelo seu nome. Por fim também é possível a abertura de várias janelas *RunGroovy* cada uma com seu *script* para a execução.

3.3 ESPECIFICAÇÃO

Para a especificação da ferramenta foram efetuados diagramas pertencentes à linguagem de modelagem UML, utilizando a aplicação Enterprise Architect. Tem-se nesta seção o modelo de casos de uso, o diagrama de atividades, o diagrama de pacotes, os diagramas de classes dos pacotes pertencentes ao modelo da ferramenta, um exemplo de código Java e Groovy necessários para atender os comandos digitados. E por fim um diagrama de seqüência que demonstra a interação com o usuário e criação dos objetos.

3.3.1 Modelo de caso de uso

Para Bezerra (2002, p. 45) o modelo de casos de uso é de extrema importância, pois força os desenvolvedores a modelarem o sistema de acordo com o usuário, e não o usuário de acordo com o sistema. O Quadro 10 apresenta a descrição do caso de uso “executar comando” que é o único caso de uso da ferramenta.

UC01 – Executar comando

Sumário: O usuário digita linha(s) de código(s) para serem executadas pela ferramenta.

Ator primário: Usuário.

Pré-condições: Nenhuma.

Fluxo Principal {Principal}.
1. O usuário digita um script.
2. O usuário seleciona operação de executar.
3. Sistema executa o(s) comando(s) digitado(s).
3.1 Sistema altera e/ou cria objetos do BlueJ conforme o(s) comando(s) digitado(s).
3.2 Sistema retorna mensagens de cada comando digitado, caso existir.

Salvar arquivo { Alternativo }.
1. No passo 1 do Fluxo Principal o usuário pode salvar o arquivo.
2. O usuário seleciona operação de salvar arquivo.
3. O Sistema mostra tela de pastas do sistema.
4. O usuário seleciona a pasta para salvar o arquivo.
5. O usuário informa o nome do arquivo para salvar.
6. O Sistema grava arquivo no disco.
7. Retorna para o passo 1 do Fluxo Principal.

Cancelamento de operação Salvar Arquivo {Alternativo}.
1. Nos passos 3 e 4 do Salvar arquivo o usuário pode cancelar a operação.

Abrir arquivo {Alternativo}.
1. O usuário seleciona a operação de abrir arquivo.
2. O sistema mostra uma tela de seleção de arquivo.
3. O usuário seleciona o arquivo que deseja abrir.
4. O sistema carrega conteúdo do arquivo na tela.
5. Retorna para o passo 1 do Fluxo Principal.

Cancelamento de operação Abrir Arquivo {Alternativo}.
1. No passo 3 do Abrir arquivo o usuário pode cancelar a operação.

Fechar arquivo {Alternativo}.
1. No passo 1 do Fluxo Principal se o arquivo foi aberto ou salvo o usuário pode fechar arquivo.
2. O usuário seleciona operação de Fechar arquivo.
3. O Sistema verificar se ouve alteração no arquivo
3.1 Caso houver alteração segue fluxo de Salvar arquivo.
4. O Sistema fecha o arquivo aberto e limpa tela de comandos.
5. Retorna para o passo 1 do Fluxo Principal.

Pós-condições: Comandos executados pela ferramenta e métodos e/ou invocados no ambiente BlueJ.

Quadro 10 – Descrição do caso de uso `Executar comando`.

3.3.2 Modelo de atividades

Para demonstrar o fluxo do processo efetuado pela ferramenta é utilizado o diagrama de atividades. Para Bezerra (2002, p. 228) o diagrama de atividades é considerado um tipo especial de diagrama de estados, sendo orientado pelo fluxo de controle.

Na Figura 19 é apresentado o diagrama de atividades da aplicação.

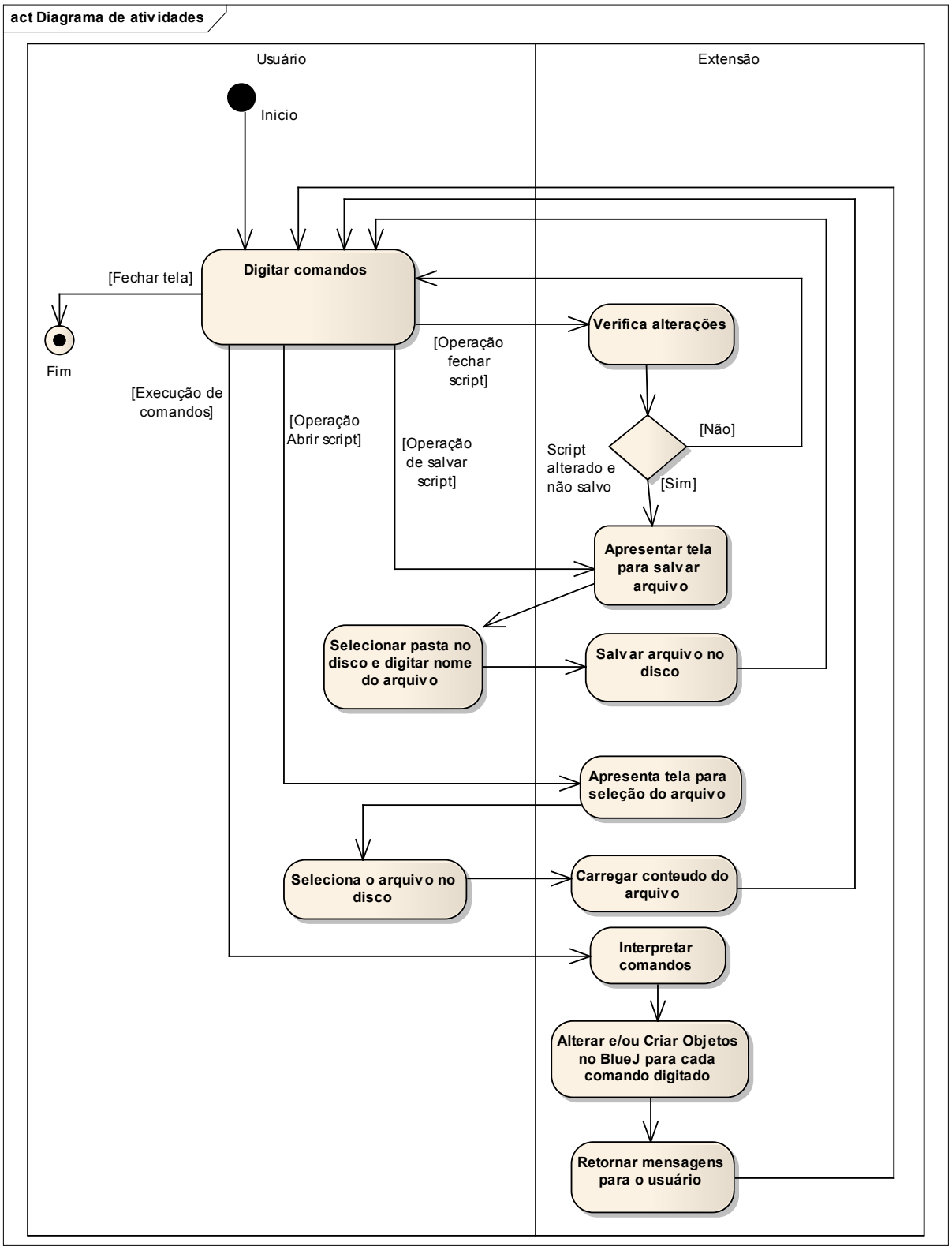


Figura 19 – Diagrama de atividades.

Como mostrado no diagrama, a interação com o usuário começa na abertura da tela para digitação de comandos. Logo após poderá escolher utilizar um arquivo de *script* já salvo. Caso desejar é aberto uma tela para escolha do arquivo e assim que escolhido é carregado seu conteúdo na ferramenta.

Após ser digitado algo na tela estarão disponíveis quatro operações que o usuário pode realizar que são: executar comandos, abrir arquivo (que já foi demonstrada anteriormente), salvar arquivo e fechar um arquivo que já foi salvo ou que foi aberto. Também é permitido o fechamento da tela, finalizando a interação com o usuário.

Na operação de execução de comandos o sistema fará a interpretação dos mesmos. Essa interpretação deverá alterar e/ou criar objetos no ambiente BlueJ e retornar uma mensagem se o comando foi executado com sucesso ou obteve falha para o usuário.

Na operação salvar arquivo, uma tela abrirá para que o usuário selecione uma pasta do disco e possa digitar um nome para o arquivo. Após o arquivo é salvo em disco pelo sistema e seu nome é apresentado no título da tela.

Na última operação, fechar arquivo, que só estará disponível para o usuário no caso de um arquivo já tenha sido salvo ou aberto. O sistema verifica se existem alterações que não foram salvas. Assim caso existir, a operação de salvar arquivo é realizada pelo sistema. Caso não existir nenhuma alteração, o arquivo é fechado seu nome é retirado na tela e o conteúdo de comandos é limpo.

3.3.3 Modelo de pacotes e classes

Para Bezerra (2002, p. 95) a modelagem de classes e de pacotes representa o aspecto estrutural estático, permitindo compreender como o sistema está estruturado internamente. A Figura 20 apresenta o diagrama de pacotes da extensão desenvolvida junto com os pacotes de *interface* do BlueJ e da biblioteca Groovy.

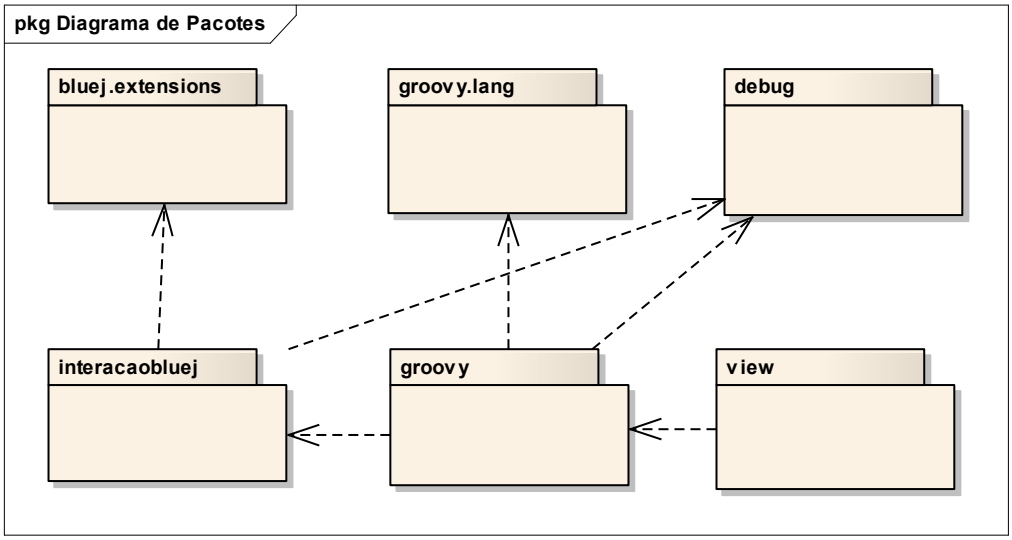


Figura 20 – Diagrama de pacotes

A extensão está dividida em quatro pacotes distintos: `interacaobluej`, `groovy`, *`debug`* e *`view`*. São utilizados também os pacotes `bluej.extensions` e `groovy.lang`. O primeiro faz parte da API de extensão da ferramenta BlueJ e é utilizado para a execução de métodos e criação de objetos e o segundo faz parte da biblioteca Groovy e é utilizado criar classes que executem linhas de código e façam uma intercepção desses comandos. Todas as classes externas foram abordadas na seção de fundamentação teórica.

Na extensão o pacote `interacaobluej` agrega as classes responsáveis para as invocações das funcionalidades da *interface* da ferramenta Bluej. Por sua vez, o pacote `groovy` contém as dependências das classes provindas da biblioteca Groovy sendo responsável pela interpretação do código digitado pelo usuário. O pacote *`view`* contém uma classe responsável pela interface do usuário. Por fim, o pacote *`debug`* tem a finalidade de gravar informações em arquivo para facilitar a depuração.

O pacote `interacaobluej` contém as classes responsáveis pela interação com a *interface* do BlueJ e são apresentados no diagrama de classes na Figura 21 e o Quadro 11 mostra cada classes com sua responsabilidade.

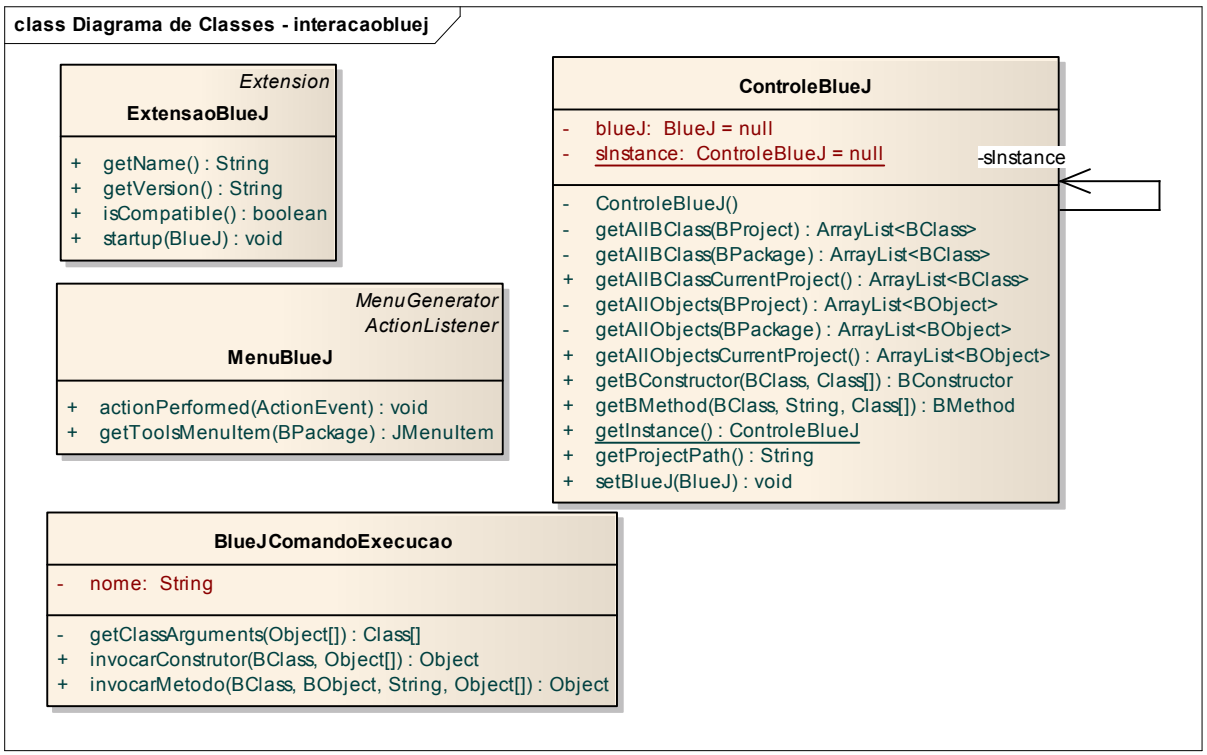


Figura 21 – Diagrama de classes `interacaobluej`

Classes do pacote `interacaobluej`	
Classe / *Interface*	**Descrição**
`ExtensaoBlueJ`	Classe responsável pelo início da extensão, registrando um novo item no menu *Tools* da ferramenta.
`MenuBlueJ`	Responsável pela exibição de um item chamado RunGroovy no menu *Tools* e por abrir a tela de execução de comandos.
`BlueJComandoExecucao`	Classe responsável pela execução de comandos no ambiente BlueJ, ou seja pela reflexão da extensão.
`ControleBlueJ`	Classe com um conjunto de métodos para manipular informações provindas do ambiente BlueJ.

Quadro 11 – Classes do pacote interacaobluej

As classes do pacote `groovy` responsabilizam-se em interpretar o comando digitado pelo usuário e interceptá-lo para a reflexão com o projeto BlueJ. Elas são apresentados na Figura 22 e a Quadro 12 suas responsabilidades.

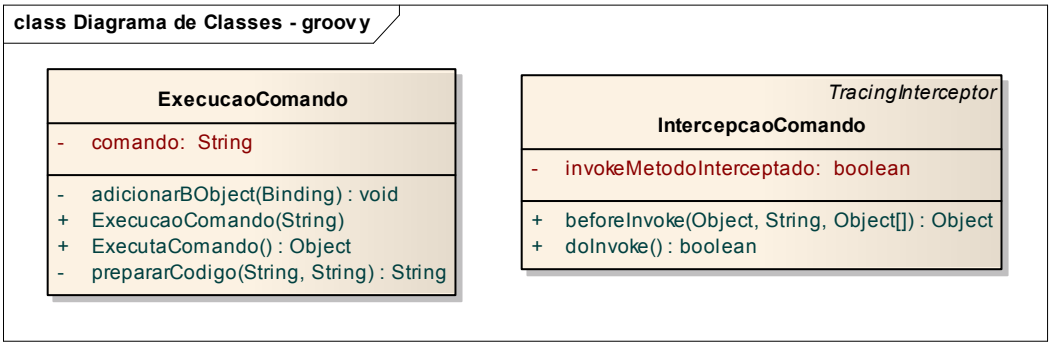


Figura 22 – Diagrama de classes do pacote `groovy`

Classes do pacote `groovy`	
Classe / *Interface*	**Descrição**
`ExecucaoComando`	Classe responsável pela execução do comando digitado pelo usuário, bem como sua preparação para ser interceptado.
`IntercapcaoComando`	Responsável pela interceptação do comando preparado da classe anterior. Esta interceptação caso necessário fará a chamada da classe `BlueJComandoExecucao` para a reflexão no ambiente BlueJ.

Quadro 12 – Classes do pacote `groovy`

O pacote debug contém a classe `ArquivoLog` responsável por gravar informações para depuração no desenvolvimento.

Por fim o pacote *`view`* contém apenas uma classe chamada `TelaComandos` que é responsável por apresentar uma interface para o usuário digitar os comandos.

3.3.4 Diagrama de seqüência

A Figura 23 mostra o diagrama de seqüência do caso de uso “executar comando” da ferramenta. Inicialmente é criado um objeto da classe `ExecutarComando` que contém todo o código digitado pelo usuário. Neste objeto é criado um `binding` que é responsável por “transmitir” os objetos criados pela ferramenta BlueJ para a classe `GroovyShell`, isto é realizado pelo método `adicionarBObject(Binding)`.

Em seguida é criado um `CompilerConfiguration` que será configurado com o `classpath` do projeto do usuário para que a classe `GroovyShell` possa executar comandos das classes criadas pelo usuário no ambiente BlueJ.

Depois o método `evaluate(String)` da classe `GroovyShell` é chamado, então o método privado `prepararComando()` prepara o código Groovy a ser executado para que seja possível a intercepção de todas as classe criadas pelo usuário no BlueJ.

Assim que o código é executado o método `beforeInvoke()` é invocado em cada comando interceptado. Neste é feita a verificação se o objeto é de alguma classe do ambiente BlueJ e feita a invocação do construtor ou método pelo objeto da classe `BlueJComandoExecucao`. Caso esta invocação não gerar nenhuma exceção então o retorno do método `doInvoke()` é `true`, do contrário será `false`.

Por fim o retorno do método `evaluate(String)` é transmitido para a tela do usuário e caso for nulo, é exibida a mensagem “Comando executado com sucesso”.

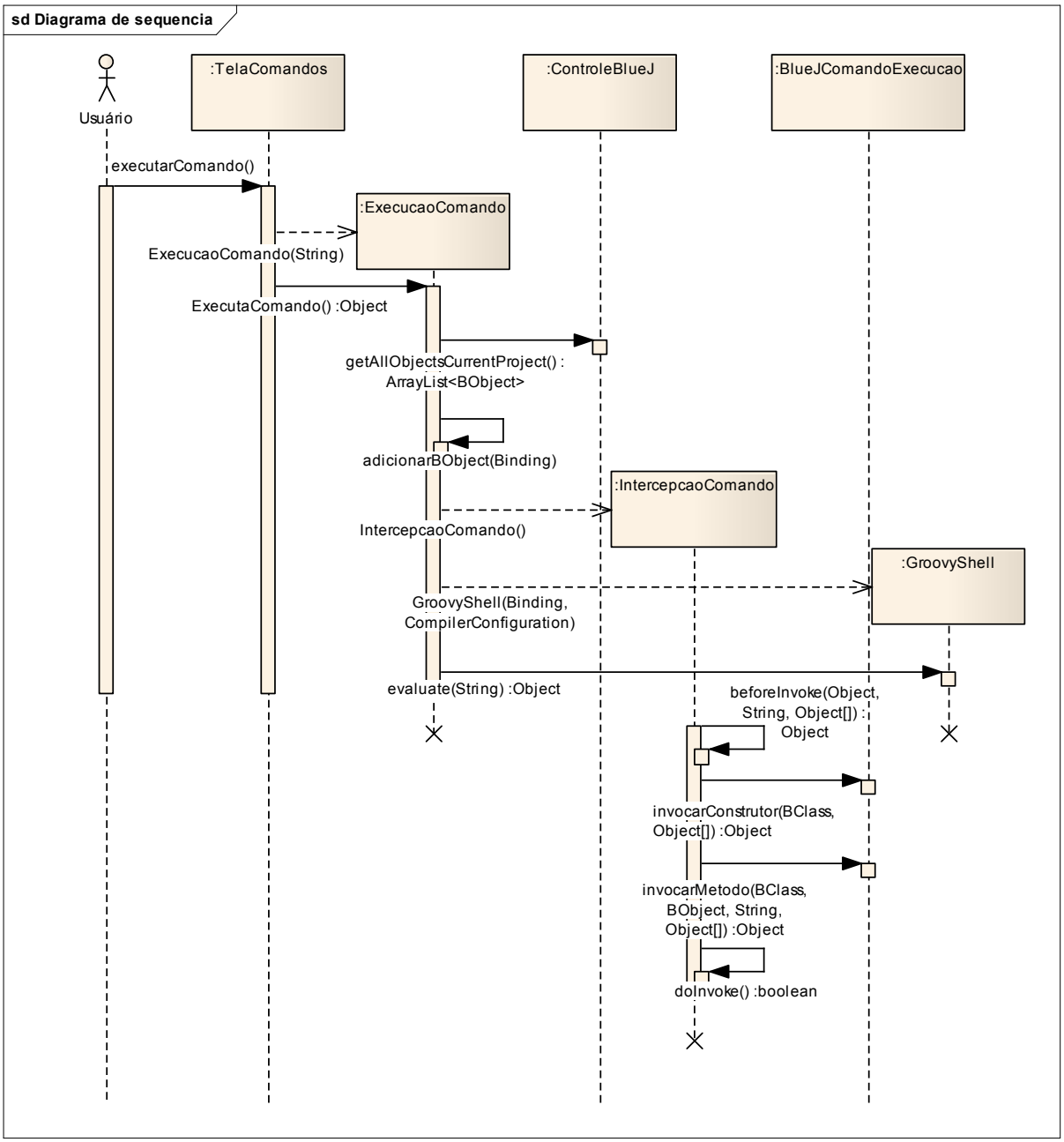


Figura 23 – Diagrama de seqüência Executar Comando

3.4 IMPLEMENTAÇÃO

A seguir são mostradas as técnicas e ferramentas utilizadas, o desenvolvimento da extensão e a operacionalidade da implementação.

3.4.1 Técnicas e ferramentas utilizadas

A extensão foi desenvolvida utilizando linguagem de programação Java em conjunto com a linguagem Groovy. Foi utilizado o ambiente de desenvolvimento NetBeans IDE 5.5, juntamente com a biblioteca gráfica SWT e o JDK versão 1.5.0. Foi utilizado a biblioteca da linguagem Groovy, `groovy-all-1.0.jar`, e a biblioteca de extensão do BlueJ, `bluejext.jar`.

3.4.2 Implementação da extensão

Esta seção apresenta a implementação das classes que compõem a ferramenta, mostrando trechos de código fonte e explicação sobre os mesmos. Está dividida entre implementações do pacotes `interacaobluej`, `groovy`, *`view`* e *`debug`*.

3.4.2.1 Implementação do pacote `interacaobluej`

A classe `ExtensaoBlueJ` é a classe que representa a extensão no BlueJ. Ela é responsável por criar a opção no menu que abrirá a janela de digitação de comandos. O Quadro 13 mostra seu código parcial. O método `startup` se encarrega de setar a instância `BlueJ` para a classe `ControleBlueJ` que tem um conjunto de métodos para manipular informações provindas da *interface* do BlueJ. No mesmo método é adicionado o menu pela classe `MenuBlueJ`.

```
package interacaobluej;
import bluej.extensions.Extension;

...

public class ExtensaoBlueJ extends Extension {

   /**
    *  Verifica se a versão do BlueJ é compatível com a extensão criada.
    *  @return verdadeiro se a extensão é compatível com a versão do BlueJ
    *  @see bluej.extensions.Extension#isCompatible()
    */
   public boolean isCompatible () {
      return true;
   }

   /**
    *  Inicio da extensão com o Bluej
    *  @param aBlueJ Intância para interação com o BlueJ.
    *  @see bluej.extensions.Extension#startup(bluej.extensions.BlueJ)
    */
   public void startup (BlueJ aBlueJ) {
      // Classe para manipulação do BlueJ
      ControleBlueJ.getInstance ().setBlueJ (aBlueJ);
      // cria e adiciona o MenuGenerator
      MenuBlueJ mMenu = new MenuBlueJ ();
      aBlueJ.setMenuGenerator (mMenu);
   }
   ...
}
```

Quadro 13 – Código parcial da classe `ExtensaoBlueJ`

Para a criação de um menu foi criada a classe `MenuBlueJ`, seu código principal é mostrado no Quadro 14. A classe herda de `bluej.extensions.MenuGenerator` para a criação de um menu no ambiente BlueJ. O método `getToolsMenuItem(BPackage)` é responsável por criar um novo `JMenuItem` dentro do menu Tools do ambiente BlueJ. O método `actionPerformed` é invocado quando este `JMenuItem` é pressionado.

```
package interacaobluej;
import bluej.extensions.MenuGenerator;

...
public class MenuBlueJ extends MenuGenerator implements ActionListener {
   /**
    *  Cria um novo JMenuItem para o menu Tools do BlueJ
    * @param bp BPackage da tela atual do BlueJ
    * @return Novo JMenuItem para o menu Tools do Bluej
    */
   public JMenuItem getToolsMenuItem (BPackage bp) {
      JMenuItem item = new JMenuItem ("Groovy");
      item.addActionListener (this);
      return item;
   }
   /**
    *  Método chamado quando o item é pressionado.
    * @param e ActionEvent
    */
   public void actionPerformed (ActionEvent e) {
      TelaComandos.main (new String[0]);
   }
}
```

Quadro 14 – Código parcial da classe `MenuBlueJ`

No Quadro 15 são mostrados os métodos públicos da classe `ControleBlueJ`. Esta classe agrega métodos para verificar a existência de um método ou construtor das classes criadas pelo usuário no ambiente de desenvolvimento BlueJ. São eles `getBMethod (BClass,`

`String, Class[])` e `getBConstructor (BClass bClass, Class[])`. Existem outros métodos chamados `getAllObjectsCurrentProject()` e `getAllBClassCurrentProject()` que são responsáveis por retornar todos os `BObject` e `BClass` respectivamente.

```
...
public class ControleBlueJ {
  private static ControleBlueJ sInstance = null;
  private BlueJ blueJ = null;
  ...
  public static ControleBlueJ getInstance () {
    ...
  }

  /**
   * Captura o caminho completo do projeto do usuário
   * @return  Path do projeto do Usuário
   */
  public String getProjectPath () {
    try {
      return blueJ.getCurrentPackage ().getProject ().getDir ().getAbsolutePath();
    } catch (ProjectNotOpenException ex) {
      ...
    }
  }



  /**
   * Verifica a existência do método na classe ou nas SuperClasses
   * @param BClass Class do BlueJ no qual quer se procurar o método
   * @param methodName String nome do Método
   * @param params Array de Class que representa os parâmetros do Método
   * @return Retorna o BMethod que representa o método ou nulo caso não existir
   */
  public BMethod getBMethod (BClass bClass, String methodName, Class[] params) {
    try {
      BMethod aRetorno = null;
      aRetorno = bClass.getDeclaredMethod (methodName,params);
      if (aRetorno == null) {
        if (bClass.getSuperclass () == null)
          return null;
        else
          return this.getBMethod (bClass.getSuperclass (),methodName,params);
      } else {
        return aRetorno;
      }
    } catch (Exception ex) {
      ...
    }
  }



  /**
   * Verifica a existência do Construtor na classe passado como parâmetro
   * @param BClass Class do BlueJ no qual quer se procurar o construtor
   * @param signature Array de Class que representa os parâmetros do construtor
   * @return Retorna o BConstructor que representa o construtor
   *          ou nulo caso não existir
   */
  public BConstructor getBConstructor (BClass bClass, Class[] signature) {
    try {
      BConstructor aRetorno = null;
      aRetorno = bClass.getConstructor (signature);
      return aRetorno;
    } catch (Exception ex) {
      ...
    }
  }



  /**
   *  Busca todos os BObject do Projeto Corrente
   * @return ArrayList<BObject> contendo os BObjects do Projeto
   */
  public ArrayList<BObject> getAllObjectsCurrentProject () {
     ArrayList result = new ArrayList ();
     try {
        result.addAll (getAllObjects (blueJ.getCurrentPackage ().getProject ()));
     } catch (PackageNotFoundException ex) {
```

```
         ...
      }
      return result;
   }
   ...
   /**
    *  Busca todos os BClass do Projeto Corrente
    * @return ArrayList<BClass> contendo os BClass do Projeto
    */
   public ArrayList<BClass> getAllBClassCurrentProject () {
      ArrayList result = new ArrayList ();
      try {
         result.addAll (getAllBClass (blueJ.getCurrentPackage ().getProject ()));
      } catch (PackageNotFoundException ex) {
         ...
      }
      return result;
   }
   ...
}
```

Quadro 15 – Principais métodos públicos da classe `ControleBlueJ`

Para retornar todos os `BObject` e os `BClass` são utilizados métodos privados da classe. Os principais métodos privados da classe `ControleBlueJ` estão no Quadro 16.

```
...
public class ControleBlueJ {
   ...
   private ArrayList<BObject> getAllObjects (BProject aProject )
           throws ProjectNotOpenException, PackageNotFoundException {
     ArrayList result = new ArrayList ();
     BPackage[] packages = aProject.getPackages ();
     for (int i = 0; i < packages.length; i++) {
        result.addAll (getAllObjects (packages[i]));
     }
     return result;
   }
   ...
   private ArrayList<BObject> getAllObjects (BPackage aPackage)
          throws ProjectNotOpenException, PackageNotFoundException {
     ArrayList result = new ArrayList ();
     BObject[] bObjects = aPackage.getObjects ();
     for (int i = 0; i < bObjects.length; i++) {
        result.add (bObjects[i]);
     }
     return result;
   }
   ...
   private ArrayList<BClass> getAllBClass (BProject aProject)
          throws ProjectNotOpenException, PackageNotFoundException {
     ArrayList result = new ArrayList ();
     BPackage[] packages = aProject.getPackages ();
     for (int i = 0; i < packages.length; i++) {
        result.addAll (getAllBClass (packages[i]));
     }
     return result;
   }
   ...
   private ArrayList<BClass> getAllBClass (BPackage aPackage)
          throws ProjectNotOpenException, PackageNotFoundException {
     BClass[] classes = aPackage.getClasses ();
     ArrayList result = new ArrayList ();
     for (int i = 0; i < classes.length; i++) {
        if (classes[i].isCompiled ())
           result.add (classes[i]);
     }
     return result;
   }
   ...
}
```

Quadro 16 – Principais métodos privados classe `ControleBlueJ`

Por fim a classe `BlueJComandoExecucao` é responsável por executar qualquer método

ou construtor através da *interface* do BlueJ. Ela é dividida em dois métodos públicos: `invocarConstrutor` e `invocarMetodo`. O primeiro é responsável por criar instâncias no ambiente BlueJ através da invocação de um construtor. E o segundo é responsável por alterações nestes objetos pela invocação de métodos, ou a execução de métodos de classes. Ambos os métodos fazem a chamada para o `getClassArguments` que é responsável por descobrir as classes dos argumentos (parâmetro `pArguments`). O Quadro 19 mostra o código destes métodos.

```
...
public class BlueJComandoExecucao  {
  ...
  public Object invocarConstrutor (BClass pBClass, Object[] pArguments) {
    BObject newInstance = null;
    Class[] clArguments = getClassArguments (pArguments);
    BConstructor bConstructor =
         ControleBlueJ.getInstance ().getBConstructor (pBClass, clArguments);
    try {
      if (bConstructor == null)
        throw new Exception("Construtor não localisado");
      newInstance= bConstructor.newInstance (pArguments);
      (newInstance).addToBench (newInstance.getInstanceName ());
    } catch (Exception ex) {
      return ex;
    }
    return newInstance;
  }
  ...
  public Object invocarMetodo (
      BClass pBClass, BObject pObject, String pMethodName, Object[] pArguments) {
    Class[] clArguments = getClassArguments (pArguments);
    BMethod metodo =
        ControleBlueJ.getInstance ().getBMethod (pBClass,pMethodName, clArguments);
    try {
      if (metodo == null)
        throw new Exception("Metodo não localisado");
      return metodo.invoke (pObject, pArguments);
    } catch (Exception ex) {
      return = ex;
    }
  }
}
```

Quadro 17 – Métodos `invocarConstrutor` e `invocarMetodo` da classe `BlueJComandoExecucao`

3.4.2.2 Implementação do pacote `groovy`

No pacote `groovy` existem as classes `IntercepcaoComando` que tem a responsabilidade de interceptar o comando digitado pelo usuário e a classe `ExecucaoComando` responsável por fazer a execução das linhas digitadas pelo usuário.

A classe `IntercepcaoComando` intercepta o código digitado através da herança da classe `TracingInterceptor`, capturando o nome do método, ou construtor, seus argumentos e parâmetros através da herança de `beforeInvoke()`. Em seguida, caso necessário enviá-los

para a classe `BlueJComandoExecucao` no pacote `interacaobluej` para a execução do método ou construtor pela na *interface* do BlueJ. Seu código é mostrado no Quadro 18.

```
...
public class IntercepcaoComando extends TracingInterceptor {
  private boolean metodoInterceptado;
  /**
   * Verifica se o método a ser executado deve ser interceptado
   * Caso verdadeiro o retorno é a execução do metodo no ambiente BlueJ
   *   Esta execução é feita pela classe BlueJComandoExecucao
   */
  public Object beforeInvoke (
      Object pObject, String pMethodName, Object[] pArguments) {
    metodoInterceptado = false;
    Object retorno = null;
    BlueJComandoExecucao bjComando = new BlueJComandoExecucao ();
    // Método de instância
    if (pObject.getClass () == BObject.class) {
      try {
         BClass bClass = ((BObject)pObject).getBClass ();
         retorno = bjComando.invocarMetodo (
                      bClass, (BObject) pObject, pMethodName, pArguments);
         if (retorno.getClass () != Exception.class)
           metodoInterceptado = true;
         return retorno;
      } catch (Exception ex) {
        ArquivoLog.escrever (ex.getMessage ());
      }
    }
    // Método é um construtor ou método estático
    if (pObject.getClass () == Class.class) {
      ArrayList<BClass> allBClass =
          ControleBlueJ.getInstance ().getAllBClassCurrentProject ();
      for (BClass bClass : allBClass) {
        try {
          if (bClass.getJavaClass ().getName () == ((Class) pObject).getName ()) {
            if (pMethodName.equals ("ctor")) {
              retorno = bjComando.invocarConstrutor (bClass, pArguments);
            } else {
              retorno = bjComando.invocarMetodo (
                   bClass, null, pMethodName, pArguments);
            }
            if (retorno.getClass () != Exception.class)
              metodoInterceptado = true;
            return retorno;
          }
        } catch (Exception ex) {
          ArquivoLog.escrever (ex.getMessage ());
        }
      }
    }
    return super.beforeInvoke (pObject,pMethodName,pArguments);
  }

  /**
   * @return Retorna true se a execução do medoto interceptado for permitida
   */
  public boolean doInvoke () {
    return ! metodoInterceptado;
  }
}
```

Quadro 18 – Métodos da classe `IntercepcaoComando`

A outra classe `ExecucaoComando` é responsável por interagir com a linguagem Groovy e permitir a execução de linhas de código Groovy através da classe `GroovyShell`. Seu principal método e construtor são mostrados no Quadro 19.

```
package groovy;
import groovy.lang.*;

...
public class ExecucaoComando {
  private String comando;
  /**
   * Cria uma nova Instância com o comando a ser executado.
   * @param comando Comando Groovy a ser executado.
   */
  public ExecucaoComando (String comando) {
     this.comando = comando;
     ArquivoLog.escrever ("Instância de ExecucaoComando com o comando: ");
     ArquivoLog.escrever (comando);
  }

  ...

  /**
   *  Método responsável pela execução do comando da instância da classe
   * @throws Exception
   * @return Retorno da execução do comando
   */
  public Object executaComando () throws Exception {
    // (1)
    Binding binding = new Binding ();
    adicionarBObject (binding);
    // (2)
    IntercepcaoComando gi = new IntercepcaoComando ();
    ProxyMetaClass proxy = ProxyMetaClass.getInstance (BObject.class);
    proxy.setInterceptor (gi);
    binding.setVariable ("proxyBObject", proxy);
    binding.setVariable ("gi", gi);
    // (3)
    CompilerConfiguration config = new CompilerConfiguration ();
    config.setClasspath (ControleBlueJ.getInstance ().getProjectPath ());
    GroovyShell groovyShell = new GroovyShell (binding, config);
    return = groovyShell.evaluate (this.prepararCodigo("proxyBObject","gi"));
  }
}
```

Quadro 19 – Principal método na classe `InteracaoGrovvy`

O método `executaComando` é responsável pela execução das linhas de código escrito em linguagem Groovy. Na parte 1 é criado um `binding`, responsável adicionar variáveis e capturar variáveis no script do `GroovyShell`. A adição de todos os objetos criados no ambiente BlueJ é realizada pelo método `adicionarBObject(binding)` e seu código pode ser visto no Quadro 20. A parte 2 é responsável por criar uma instância de `IntercepcaoComando` e instanciar um `ProxyMetaClass` da classe `BObject` para intercepção dos objetos que foram transmitidos para o `GroovyShell`. Estas variáveis são adicionadas também ao `binding`. Por fim, na terceira parte do código é feita a configuração do `GroovyShell` no que diz respeito ao *classpath* e a instância `binding` contendo as variáveis associadas anteriormente. Antes da execução do código é necessário prepará-lo com os `ProxyMetaClass` de cada classe criada pelo usuário no ambiente BlueJ. Este procedimento é realizado pelo método `prepararCodigo` que é mostrado no Quadro 21.

```
...
public class ExecucaoComando {
  ...
  /**
  *  Adiciona todos os BOjects criados na aplicação BlueJ para o Binding
  * @param aBinding  Binding aonde será adicionado os BObjects
  */
  private void adicionarBObject(Binding aBinding) {
    ArrayList<BObject> ObjectArray = null;
    ObjectArray = ControleBlueJ.getInstance().getAllObjectsCurrentProject ();
    for (BObject elem : ObjectArray) {
      aBinding.setVariable (elem.getInstanceName (),elem);
    }
  }
  ...
}
```

Quadro 20 – Método `adicionarBObject` da classe `ExecucaoComando`

```
...
public class ExecucaoComando {
  ...
  /**
   *  Retorna o comando a ser executado.
   *  O comando já é preparado com todos os ProxyMetaClass nessecários
   *
   * @param proxBObject   Nome da intância do proxMetaClass
   *                       responsável pela intercepção da Classe BOBject
   * @param IntercepcaoComando  Nome da instância do Interceptor
   * @return O codigo alterado e pronto para ser executado.
   */
  private String prepararCodigo (String proxBObject, String IntercepcaoComando) {
    String aComando = comando;
    ArrayList<BClass> ArrayBClass = null;
    ArrayBClass = ControleBlueJ.getInstance ().getAllBClassCurrentProject ();
    for (BClass bClass : ArrayBClass) {
      String Aux;
      String nomeClasse = "";
      try {
        nomeClasse = bClass.getJavaClass ().getName ().replaceAll (".","_");
      } catch (bluej.extensions.ClassNotFoundException ex) {
        ex.printStackTrace ();
      } catch (ProjectNotOpenException ex) {
        ex.printStackTrace ();
      }
      Aux = "def proxy" + nomeClasse
              + " = ProxyMetaClass.getInstance ("+ nomeClasse + ".class)\n";
      Aux += "proxy" + nomeClasse + ".setInterceptor("
                                          + IntercepcaoComando + ")\n";
      Aux += "proxy" + nomeClasse + ".use {\n" + aComando + "\n}";
      aComando = Aux;
    }
    aComando = proxBObject + ".use {\n" + aComando + "\n}";
    ArquivoLog.escrever (aComando);
    return aComando;
  }
  ...
}
```

Quadro 21 – Método `prepararCodigo` da classe `ExecucaoComando`

3.4.2.3 Implementação do pacote *debug* e *view*

O pacote *debug* é de simples implementação. O mesmo contém uma classe chamada `ArquivoLog` com um método estático para gravar informações em um arquivo que é salvo na pasta onde a extensão é colocada. Seu código pode ser observado no Quadro 22.

```
package debug;
import java.io.FileWriter;
import java.io.IOException;
...
public class ArquivoLog {
  public static String path = "C:";
  public static void setPath (String aPath) { path = aPath; }
  public static void escrever (String pLine) {
    try {
      FileWriter aFile = new FileWriter (path + "\\bluej-Log.txt",true);
      aFile.write (pLine + "\n");
      aFile.close ();
    } catch (IOException ex) {
      ex.printStackTrace ();
    }
  }
}}
```

Quadro 22 – Código da classe `ArquivoLog`

O pacote *`view`* contém uma tela de interface com o usuário que foi criada com a IDE NetBeans. Esta tela contém dois `JTextArea`: um para a digitação de comandos e outro para as mensagens de execução. Também contém quatro `JButton`, para as funções abrir, salvar, fechar arquivo e executar o código. O diagrama de dependências da *view* é mostrado na Figura 24.

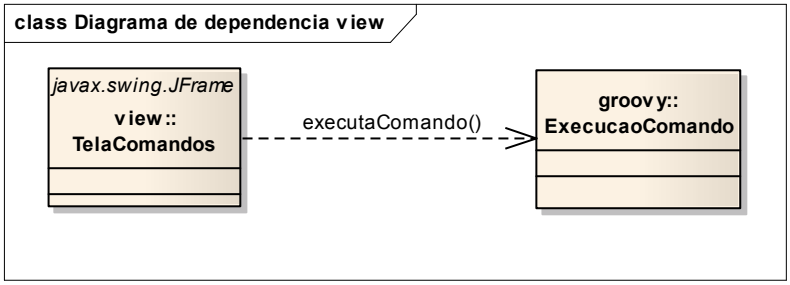


Figura 24 – Dependências da *view*

3.4.3 Operacionalidade da implementação

Nesta seção é apresentada a operacionalidade da extensão através de um estudo de caso. Para instalar a extensão no Bluej é preciso colocar o arquivo `RunGroovy.jar` na pasta `lib\extensions` no diretório onde o BlueJ foi instalado.

No ambiente BlueJ é criado um projeto com as classes `Pessoa`, `Aluno`, `Professor` e `Sala`. A Figura 25 mostra o diagrama com essas classes.

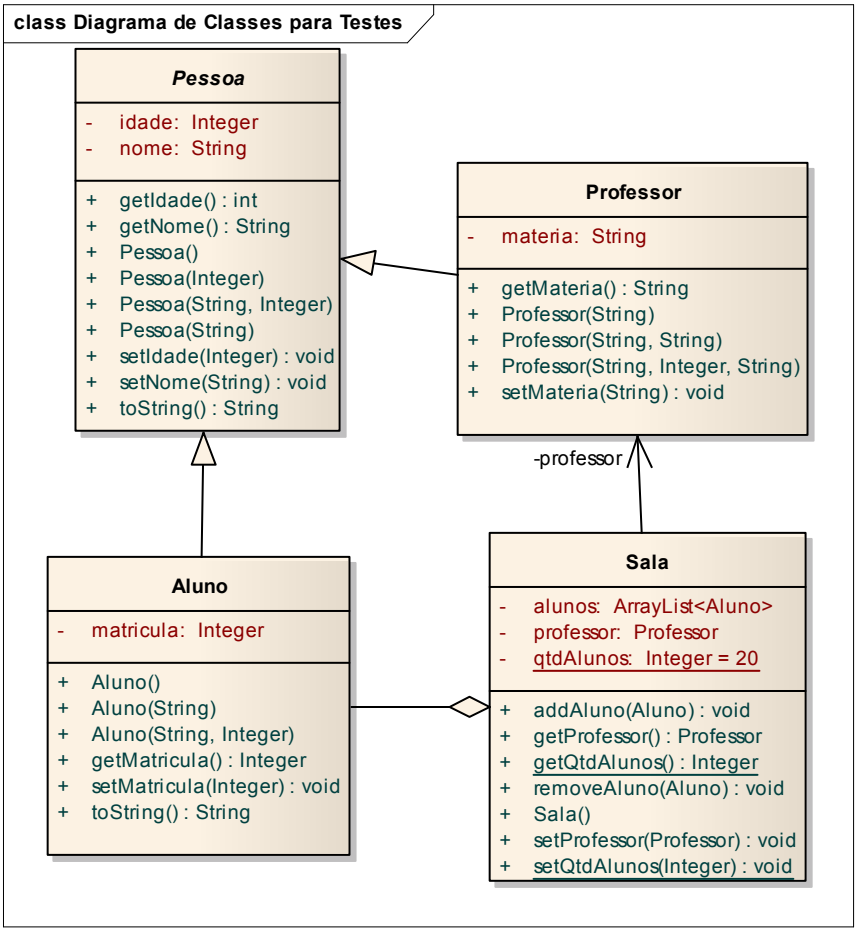


Figura 25 – Diagrama de classes para testes

A classe `Pessoa` é uma classe abstrata e superclasse das classes `Aluno` e `Professor`. A classe `Sala` por sua vez é composta de um `Professor` e um `Array` de `Alunos`. Também existe um método estático que é responsável por definir a quantidade máxima de `Alunos` na classe. A existência deste método se dar para permitir o teste de métodos estáticos na ferramenta.

Para demonstrar a execução de scripts é preciso abrir a tela de execução de comando, que se encontra no menu *Tools* e depois em *RunGroovy* A Figura 26 mostra a posição do menu e a tela aberta.

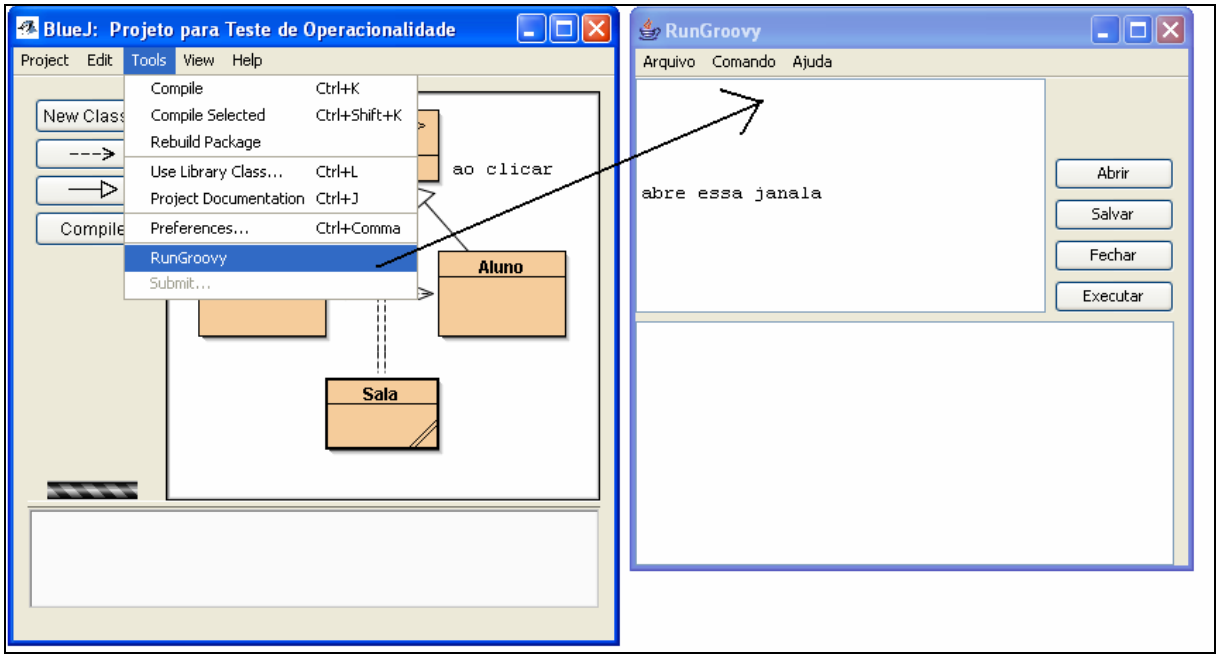


Figura 26 – Abertura da tela de execução de comando no BlueJ

Na tela de execução de comando o usuário poderá digitar comandos para interação com as classes criadas. Os comandos devem seguir a sintaxe da linguagem Groovy. Após a digitação destes comandos o usuário deve clicar em [Executar]. A Figura 27 mostra alguns comandos que foram executados no projeto de exemplo.

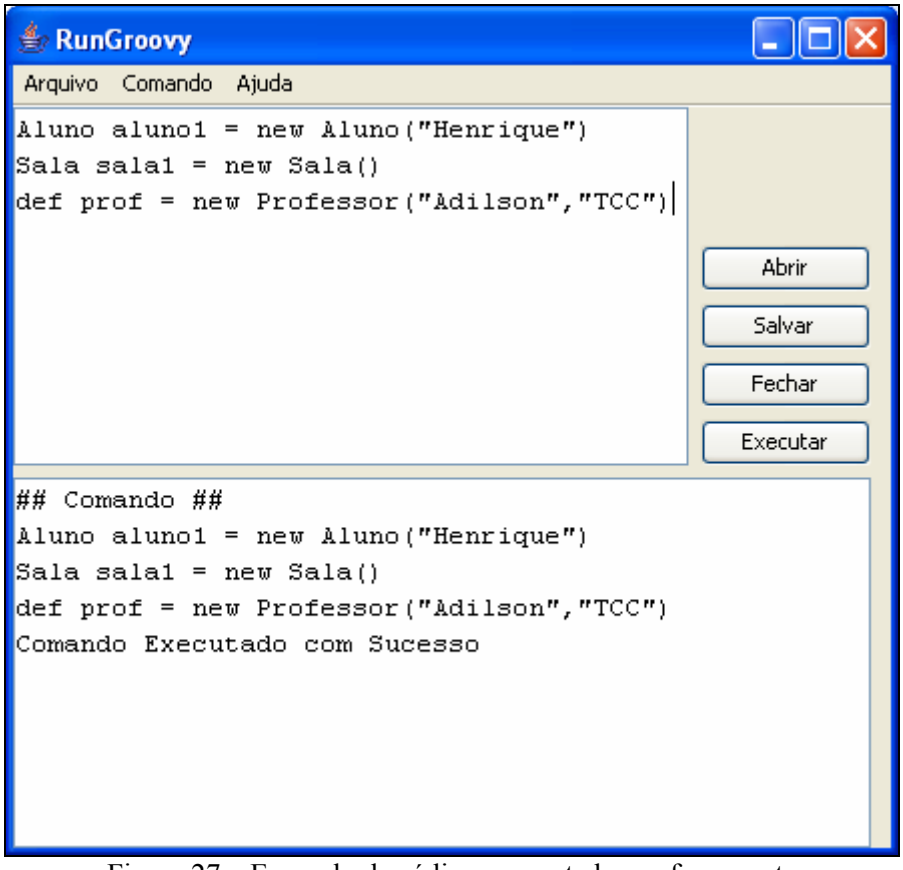


Figura 27 – Exemplo de códigos executados na ferramenta

Em seguida o usuário pode verificar que os objetos foram criados no ambiente BlueJ, assim como os parâmetros que foram enviados no construtor. A Figura 28 mostra esta verificação na ferramenta BlueJ, na qual se pode verificar que o nome do objeto `prof` definido no script não refletiu no ambiente BlueJ. Esta é uma limitação da ferramenta por não realizar a captura do nome dado nas linhas de código digitadas pelo usuário. Com isso a não é possível manter o estado da execução anterior e acessar o objeto `prof` criado anteriormente.

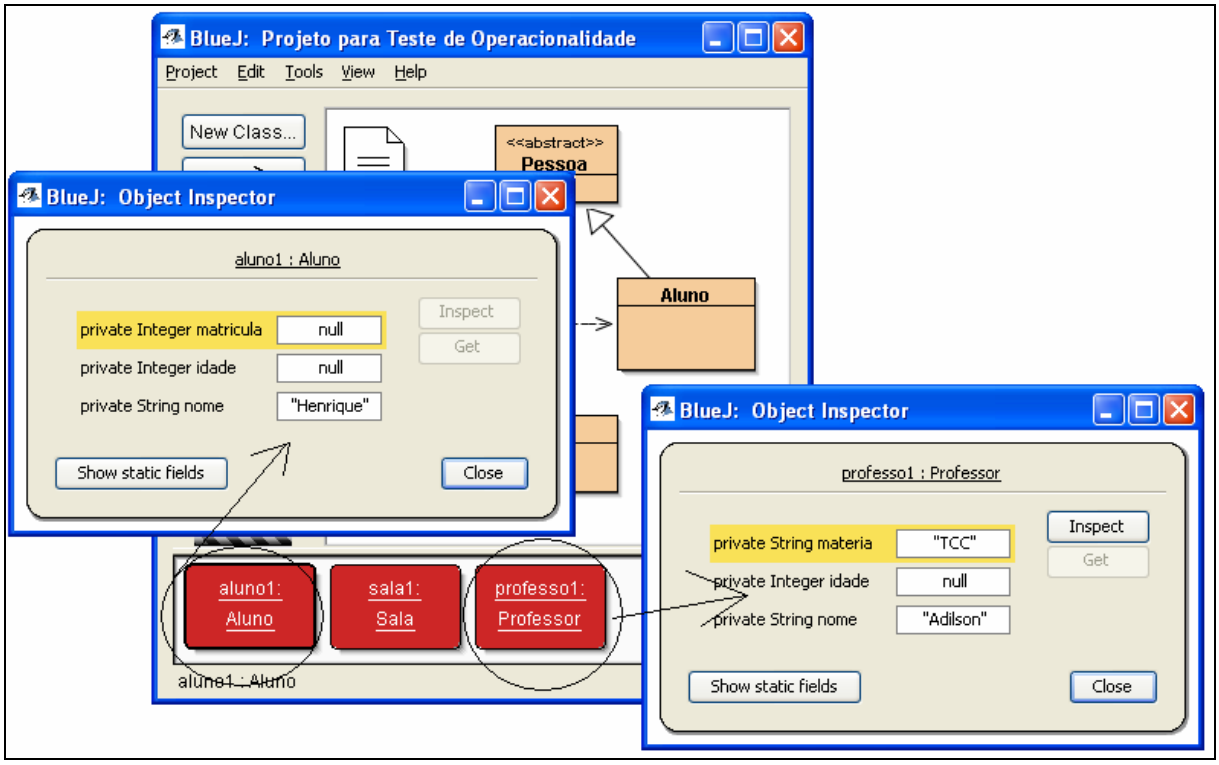


Figura 28 – Verificação dos objetos

Após a criação de objetos também é possível a manipulação deles através da tela de comando. O usuário pode criar novos objetos pela ferramenta BlueJ e depois digitar, na janela de extensão, linhas de código para interagirem com estes objetos. No exemplo foi criado mais um objeto `Aluno` com o nome `novoAluno` no próprio BlueJ e em seguida executado os comandos da Figura 29. Na Figura 30 é observado o ambiente BlueJ após a execução desses comandos.

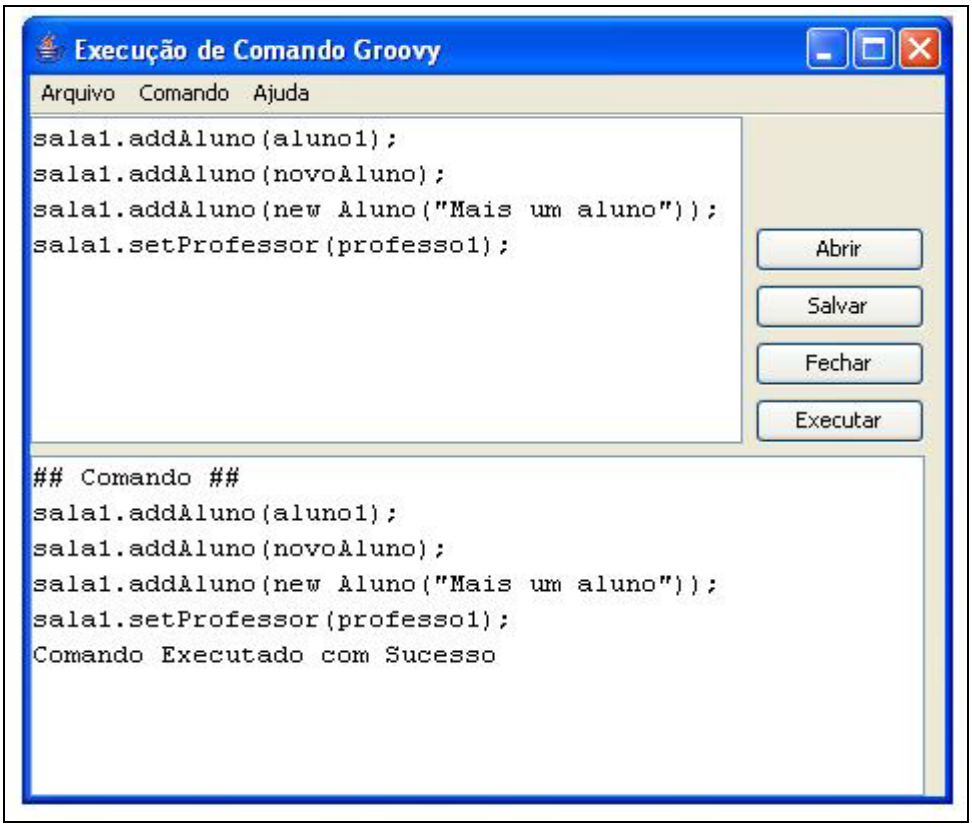


Figura 29 – Comandos com interação de objetos criados no BlueJ

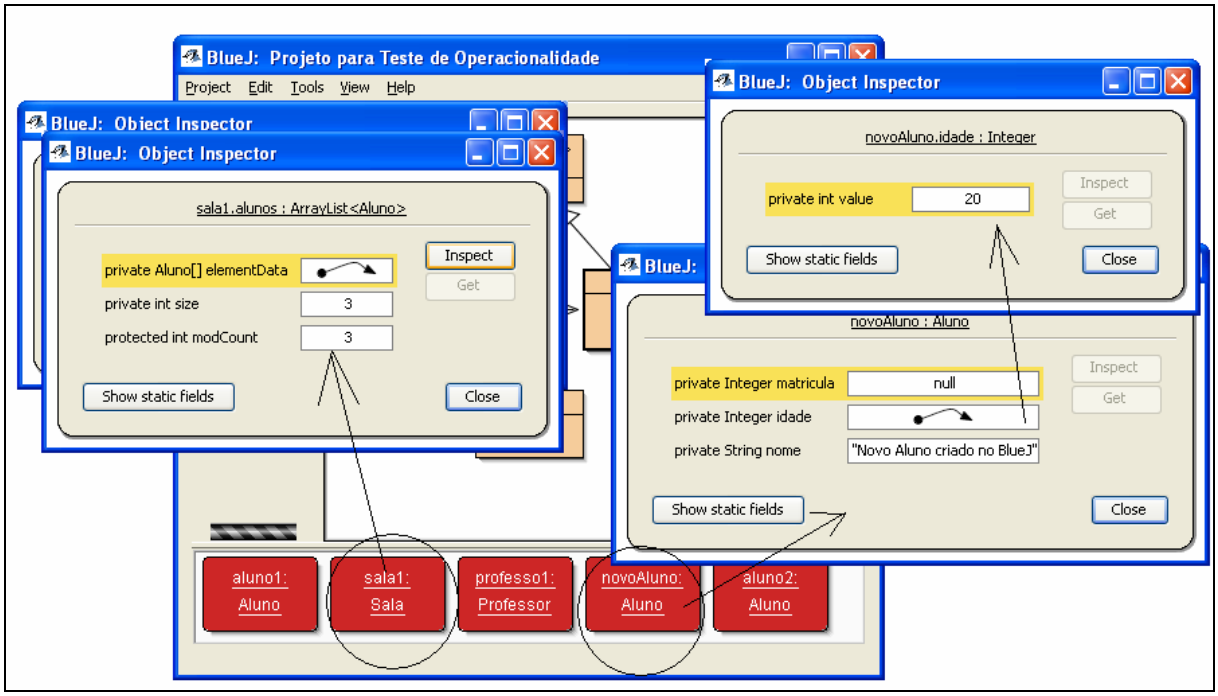


Figura 30 – Verificação dos objetos criados e atributos alterados no BlueJ

Pode-se observar que ao se executar estes últimos comandos um objeto anônimo foi criado na linha `sala1.addAluno(new Aluno("Mais um aluno"));` porém observa-se que foi criado mais um objeto aluno no ambiente BlueJ, o `aluno2` que é a representação deste objetos anônimo no script. A Figura 31 mostra a inspeção deste objeto.

Figura 31 – Inspeção do objeto `aluno2` criado pelas linhas de comandos

O usuário também tem a opção de salvar o script em um arquivo com os comandos digitados. Após salvar algum arquivo o usuário pode também carregar o conteúdo do arquivo. A tela da Figura 32 mostra um arquivo chamado teste.rgs carregado pelo usuário. A extensão `rgs` vem do nome RunGroovy Script.

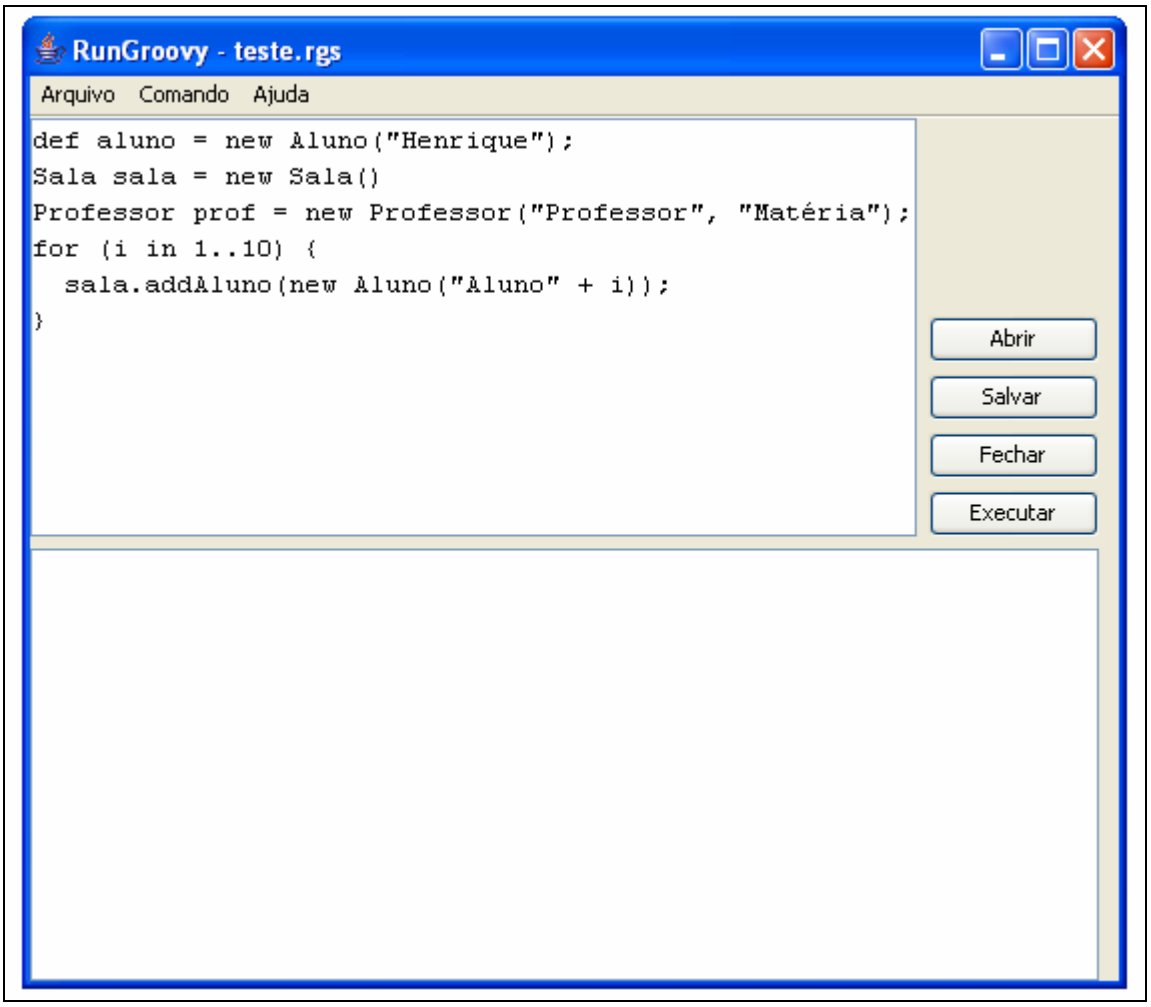


Figura 32 – Arquivo teste.rgs carregado na ferramenta

3.5 RESULTADOS E DISCUSSÃO

O trabalho desenvolvido é mais uma opção para o auxílio na aprendizagem da POO ao acrescentar recursos de simulação com objetos.

Com relação aos trabalhos correlatos é possível verificar uma ferramenta com as mesmas características do trabalho que é a ferramenta *Code Pad* do próprio ambiente BlueJ. Porém, a mesma não gera objetos de forma intuitiva porque é necessário mais interação para o usuário poder criar objetos, como é possível verificar na seção de fundamentação teórica, diferente da extensão proposta que cria objetos e acessa seu métodos apenas com o digitação de códigos.

É apresentado na Quadro 23 uma comparação das principais características entre a ferramenta desenvolvida (*RunGroovy*) e a ferramenta *Code Pad* do BlueJ.

	RunGroovy	***Code Pad* do BlueJ**
Permite a interação com o usuário para criação de objetos	X	X
Permite a execução de códigos	Linguagem Groovy	Linguagem Java
Salva e carrega arquivos de scripts	X	
Permite a execução de estruturas de repetição	X	X
Criação de objetos no *ObjectBench*	Apenas com os códigos digitados	Necessita de mais interações com o Usuário (arrastar a linha para o *ObjectBench* após a execução do comando)

Quadro 23 – Comparação entre a ferramenta desenvolvida, Code Pad do BlueJ

4 CONCLUSÕES

O ambiente BlueJ é utilizado por inúmeras instituições de ensino superior nos cursos de computação e informática. Ao mesmo tempo em que ela é considerada uma ferramenta com recursos didáticos úteis na aprendizagem de POO como a criação e interação de objetos no próprio ambiente, ela é considerada falha em alguns aspectos na autoria (um editor com poucos recursos em relação à produtividade e auxílio na codificação) e representação das classes (o diagrama de classes considera as relações de herança e dependência, além de omitir informações sobre atributos e métodos).

Porém, a ferramenta permite sua extensibilidade, isto é, implementar novos recursos no ambiente. No site do desenvolvedor existem publicados 17 extensões desenvolvidas por diversos autores (KÖLLING, 2007a).

O desenvolvimento desse trabalho visou o estudo e compreensão na produção de extensões para a ferramenta BlueJ, gerando uma documentação introdutória e *expertise* sobre o assunto.

A escolha de linguagem Groovy para criação destas linhas de código se deu por ser uma linguagem ágil, de *script* e também por sua familiaridade com Java diminuindo a sobrecarga cognitiva na aprendizagem de mais uma linguagem. No decorrer do desenvolvimento surgiram muitas dificuldades para a integração de Java, Groovy e o ambiente do BlueJ. A documentação de extensão é pouca e muito restrita.

O resultado no desenvolvimento da extensão se deu em uma ferramenta de simples utilização e acesso alcançando todos os objetivos propostos. A flexibilidade na digitação de código também é um ponto forte na sua utilização, pois qualquer script Groovy pode ser executado na ferramenta. Outras características importantes que foram observadas na linguagem Groovy é que ela é totalmente compatível com a JVM e se integra facilmente com Java.

Em relação às limitações com a ferramenta é possível observar que:

a) há restrição com o acesso direto aos atributos públicos, pois não foi implementado este acesso, visto que isso não é uma boa prática de POO;
b) os nomes dos objetos no *script* não refletem o nome dos objetos no ambiente. Não se conseguiu realizar esta intercepção nas linhas de comandos digitadas;
c) os objetos criados e nomeados no *script* são validos na própria execução e não conseguem ser acessados em execuções posteriores.

4.1 EXTENSÕES

Como norteador para trabalhos futuros pode-se melhorar a própria interface de janela para que mostre *syntax highlight* e tenha código auto completado assim como repositório de código digitado, montando assim um histórico.

Novas ferramentas ou TCCs existentes podem gerar scripts em Groovy que podem ser abertos e executados pelo RunGroovy, visto que é possível até definir classes nesses scripts. Entretanto, essas classes não são refletidas no ambiente, o que representa mais uma melhoria a ser feita. Pode-se também melhorar a extensão para se integrar com outras extensões através do ambiente, como por exemplo, o JUnit.

Também foi possível observar no desenvolvimento do trabalho a falta de segurança nos códigos digitados, sendo possível fazer a execução de comandos que, por exemplo, desliguem a máquina, façam abertura de unidades de CD entre outros, que não são nada convenientes. Então, outra sugestão é a implementação de regras de segurança na ferramenta que afetariam a criação do objeto `GroovyShell` da classe `groovy.ExecucaoComando`.

Outras ferramentas já foram produzidas como trabalhos de conclusão de curso na FURB para o auxilio na aprendizagem de OO, e espera-se que em trabalhos futuros possam ser desenvolvidos integrações entre esses trabalhos, ou mesmo novos inspirados neles, e com o ambiente BlueJ.

Por fim, o estudo e código produzido nesse trabalho podem servir como base e inspiração para novas extensões do BlueJ.

REFERÊNCIAS BIBLIOGRÁFICAS

ARAÚJO, Marcos A. P.; DAIBERT, Marcelo S. Introdução ao BlueJ: aprenda visualmente programação OO e Java. **Java Magazine**, São Paulo, ano V, n. 37, p. 62-67, 2006.

BARNES, David J.; KÖLLING, Michael. **Programação orientada a objeto com Java:** uma introdução prática utilizando o BlueJ. São Paulo: Pearson Prentice Hall, 2004.

BARROSO, Isaías C. Groovy: uma breve introdução. **Mundo Java**, São Paulo, ano III, n. 15, p. 50-57, jan. 2006.

BEZERRA, E. **Princípios de análise e projeto de sistemas com UM**L. Rio de Janeiro: Campus, 2002.

BLIKI, Fragmental. **Groovy** – linguagem de script para Java. [S.l.], 2007. Disponível em <http://www.fragmental.com.br/wiki/index.php/Groovy:_Linguagem_de_Script_para_Java>. Acesso em: 01 nov. 2007.

BORATTI, Isaias C. **Programação orientada a objetos [usando Delphi]**. 2. ed. Florianópolis: Visual Books, 2002.

BÖRSTLER, Jürgen; BRUCE, Kim; MICHIELS, Isabel. Sixth workshop on pedagogies and tools for learning object oriented concepts. In: ECOOP, 17., 2003, Darmstadt. **Anais...** [Darnastadt]: [s.n.], [2003]. p. 84-87.

CODEHAUS FOUNDATION. **Groovy** – differences from Java. [S.l.], 2007a. Disponível em: <http://groovy.codehaus.org/Differences+from+Java>. Acesso em: 01 nov. 2007.

_____. **TracingInterceptor** – Groovy 1.5.0. [S.l.], 2007b. Disponível em: <http://groovy.codehaus.org/api/groovy/lang/TracingInterceptor.html>. Acesso em: 07 dez. 2007.

DOEDERLEIN, Osvaldo P. Aprendendo Groovy. **Java Magazine**, São Paulo, ano IV, n. 32, p. 30-44, jan. 2006.

GALHARDO, Mariane F.; ZAINA, Luciana A. M. Simulação para ensino de conceitos da orientação a objetos. In: Seminário de Computação - SEMINCO, 13.,2004, Blumenau. **Anais...** Blumenau: Furb/DSC, 2004. p. 109-116.

KÖLLING, Michael. **BueJ –** extensions. [S.l.], 2007a. Disponível em: <http://www.bluej.org/extensions/extensions.html >. Acesso em: 23 maio 2007.

_____. **BlueJ –** how to write extensions. [S.l.], 2007b. Disponível em: <http://www.bluej.org/doc/writingextensions.html>. Acesso em: 30 maio 2007.

_____. **BlueJ extensions API** – bluej.extensions. [S.l.], 2007c. Disponível em: <http://www.bluej.org/doc/extensionsAPI/bluej/extensions/package-frame.html>. Acesso em: 01 abr. 2007.

KÖNIG, Dierk. **Groovy in action**. New York: Manning Publications, 2007.

SOUSA, Fábio C. de. **Utilização da reflexão computacional para implementação de um monitor de software orientado a objetos em Java,** 2002. 53 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) – Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

WINBLAD, Ann L.; EDWARDS, Samuel D.; HING, David R. **Software Orientado ao Objeto**. São Paulo: Makron Books, 1993.