

**UNIVERSIDADE FEDERAL DE SANTA CATARINA  
CURSO DE BACHARELADO EM CIÊNCIAS DA  
COMPUTAÇÃO**

**Fabio Cechinel Veronez**

**Uma Aplicação para WEB para o Apoio ao Ensino de  
Estruturas de Dados**

Trabalho de Conclusão de Curso submetido à Universidade Federal de Santa Catarina como parte dos requisitos para a obtenção do grau de Bacharel em Ciências da Computação.

Leandro J. Komosinski

Florianópolis, Junho de 2004

# **Uma Aplicação para WEB para o Apoio ao Ensino de Estruturas de Dados**

Fabio Cechinel Veronez

Este Trabalho de Conclusão de Curso foi julgado adequado para a obtenção do título de Bacharel em Ciências da Computação, e aprovado em sua forma final pela Coordenadoria do Curso de Bacharelado em Ciências da Computação.

---

José Mazzuco Junior

Banca Examinadora

---

Leandro J. Komosinski

---

Ronaldo Santos Mello

---

José Eduardo De Lucca

# Sumário

<b>Lista de Figuras</b>	<b>v</b>
<b>Lista de Tabelas</b>	<b>vii</b>
<b>Resumo</b>	<b>viii</b>
<b>Abstract</b>	<b>ix</b>
<b>1 Introdução</b>	<b>1</b>
1.1 Cenário . . . . .	1
1.1.1 Visualizadores . . . . .	2
1.1.2 Verificadores . . . . .	4
<b>2 Objetivos</b>	<b>7</b>
2.1 Objetivo Geral . . . . .	7
2.2 Objetivos Específicos . . . . .	7
<b>3 Contextualização</b>	<b>9</b>
3.1 Evolução das aplicações para <i>Web</i> com foco na tecnologia <i>Java</i> . . . . .	9
3.1.1 Common Gateway Interface . . . . .	10
3.1.2 Servlets . . . . .	10
3.1.3 JavaServer Pages . . . . .	12
3.1.4 Model 1, Model 2 e MVC . . . . .	13
3.1.5 Framework Jakarta Struts . . . . .	17
3.1.6 Funcionamento Básico . . . . .	17

<b>4</b>	<b>Desenvolvimento</b>	<b>20</b>
4.1	Definindo o Problema . . . . .	20
4.1.1	Introdução . . . . .	20
4.1.2	Definindo os parâmetros a serem analisados . . . . .	21
4.2	Definindo o funcionamento da aplicação . . . . .	22
4.3	Modelagem . . . . .	23
4.3.1	Definição dos Atores . . . . .	23
4.3.2	Casos de Uso . . . . .	23
4.3.3	Diagrama de Classes . . . . .	30
4.4	Especificações Técnicas . . . . .	34
<b>5</b>	<b>Exemplo do Funcionamento da Aplicação</b>	<b>36</b>
<b>6</b>	<b>Considerações Finais</b>	<b>48</b>
6.1	Trabalhos Futuros . . . . .	48
6.2	Conclusão . . . . .	49
<b>A</b>	<b>Protocolo HTTP</b>	<b>50</b>
A.1	Introdução . . . . .	50
A.2	Funcionamento . . . . .	51
<b>B</b>	<b>Class Loader</b>	<b>54</b>
	<b>Referências Bibliográficas</b>	<b>56</b>
	Referências Bibliográficas . . . . .	56

# Lista de Figuras

1.1	AVL Tree. . . . .	3
1.2	Interactive Data Structure Visualizations. . . . .	4
1.3	Lista ligada no Astral. . . . .	5
3.1	Passos da requisição de um arquivo JSP. . . . .	13
3.2	Esquema MVC. . . . .	15
3.3	Esquema MVC modificado. . . . .	16
3.4	Esquema Struts. . . . .	19
4.1	Pacote <i>fcv.projeto</i> . . . . .	32
4.2	Pacote <i>fcv.projeto.loader</i> . . . . .	33
4.3	Pacote <i>fcv.projeto.executer</i> . . . . .	33
4.4	Pacote <i>fcv.projeto.structures</i> . . . . .	34
4.5	Pacote <i>fcv.projeto.report</i> . . . . .	35
4.6	Pacote <i>fcv.projeto.helper</i> . . . . .	35
5.1	Tela de <i>upload</i> de classes secundárias. . . . .	37
5.2	Tela de <i>upload</i> das classes principais. . . . .	38
5.3	Tela de exibição das interfaces implementadas pelas classes analisada e padrão. . . . .	39
5.4	Tela de exibição dos construtores. . . . .	40
5.5	Tela de exibição dos métodos. . . . .	41
5.6	Tela de criação do parâmetro <i>java.lang.Object</i> . . . . .	42
5.7	Tela de criação do parâmetro <i>java.lang.Integer</i> . . . . .	43

5.8	Tela de criação do tipo primitivo <i>int</i> . . . . .	44
5.9	Tela de relatório do método <i>add(java.lang.Object)</i> . . . . .	45
5.10	Tela de relatório do método <i>getSize()</i> . . . . .	46
5.11	Tela de relatório do método <i>get()</i> . . . . .	47

# Lista de Tabelas

4.1	Caso de uso: <i>Upload</i> de classes secundárias . . . . .	24
4.2	Caso de uso: <i>Upload</i> das classes principais . . . . .	25
4.3	Caso de uso: Definição da interface analisada . . . . .	26
4.4	Caso de uso: Definição da interface analisada . . . . .	27
4.5	Caso de uso: Definição do método a ser executado . . . . .	28
4.6	Caso de uso: Definição de parâmetros . . . . .	29
A.1	Métodos da requisição . . . . .	52
A.2	Valores de retorno do servidor no campo CODIGO . . . . .	52

# Resumo

Este trabalho tem como objetivo a criação de uma aplicação para *Web* para o auxílio do ensino da disciplina Estruturas de Dados no curso de Bacharelado em Ciências da Computação. Esse aplicativo auxiliará na avaliação da implementação das estruturas básicas de dados (fila, pilha, árvore, etc) feitas pelo aluno exibindo uma comparação entre seus resultados e os resultados obtidos pela implementação padrão implementada pelo professor.

**Palavras Chave:** Estruturas de Dados, Java, Reflexão, Class Loader



# Abstract

This project objectifies the creation of a Web Application to auxiliate the teaching of the Data Structures discipline in the Computer Sciences B.Sc. course. This application will help in the evaluation of the student's implementation of the fundamental Data Structures (Stack, Queue, Tree, etc) by exhibiting a comparison between the student's implementation results and results of the standart implementation by the teacher.

**Keywords:** Data Structures, Java, Reflection, Class Loader

# Capítulo 1

## Introdução

### 1.1 Cenário

No curso de Ciências da Computação, a disciplina de Estruturas de Dados se afirma como uma das mais importantes do curso. Essa importância se dá principalmente ao fato de que os conceitos servem como base para grande parte das disciplinas ministradas ao longo do curso. Outra característica desta disciplina é a dificuldade dos alunos de absorverem os conceitos por ela passados. O principal motivo desta dificuldade se dá ao fato de um certo grau de abstração utilizado e também pela dificuldade do aluno de vincular a abstração com a realidade (a aplicação, a implementação e utilização da estrutura).

Esse fato já vem sendo estudado e muitas pesquisas tanto na área pedagógica como na área tecnológica foram feitas a fim de tentar amenizar as dificuldades encontradas por alunos e mestres. Nos últimos quinze anos, o número de trabalhos relacionados ao assunto cresceu consideravelmente. Assim como cresceu também o número de aplicativos de cunho pedagógico que têm como finalidade auxiliar os alunos no aprendizado de estruturas de dados.

Uma boa maneira de dividir esses tipos de aplicativos foi utilizada por [1]: que separou-os em dois grupos: os *Visualizadores* e os *Verificadores*.

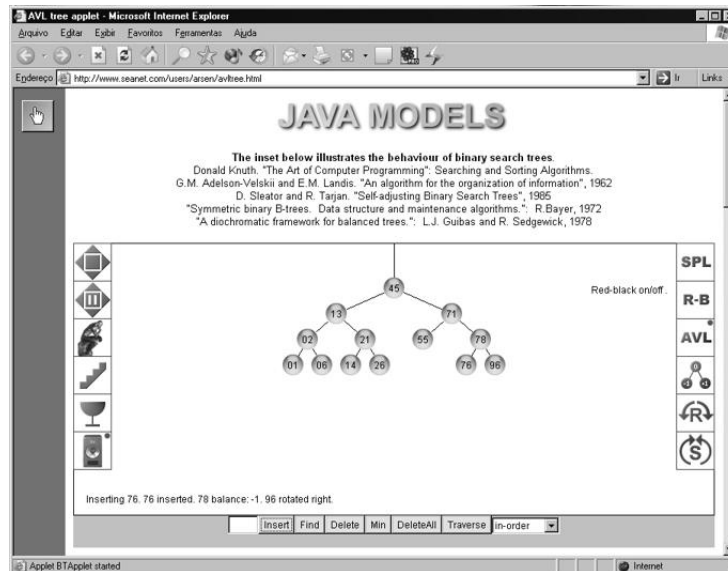
### 1.1.1 Visualizadores

Os *Visualizadores* são os aplicativos que, como a própria definição diz, possibilitam a visualização do comportamento das estruturas de dados mediante estímulos externos. Esse grupo de aplicativos é de grande utilidade para o ensino da disciplina de Estrutura de Dados, pois possibilitam uma melhor compreensão do funcionamento das abstrações por ela utilizadas. Nos últimos tempos, um número considerável de pesquisas tem sido realizadas com o intuito de avaliar a eficiência da utilização desses visualizadores no ensino prático da disciplina.

Um exemplo desse tipo de pesquisa é apresentada em [2], no qual um grupo de 60 estudantes foi dividido em 3 subgrupos. Para cada subgrupo foi utilizado um método de ensino. Para o primeiro grupo utilizou-se o método textual; para o segundo, o método de visualização, que permite ao usuário verificar a progressão dos passos; para o terceiro, os dois métodos combinados. Feito isso, os estudantes foram submetidos a dois testes para verificar a retenção da informação: um logo após as aulas e outro 15 dias depois. Os resultados obtidos indicaram que o uso das duas metodologias combinadas apresentou uma melhor performance dos alunos em ambos os testes. Outro resultado interessante foi a baixa performance dos alunos submetidos somente à metodologia de visualização do segundo teste, o que pode ser explicado pelo fato de que, pelo método de visualização, os alunos apenas decoram o funcionamento das operações e não as entendem profundamente. Sendo assim, testes como esses encorajam o uso desse tipo de aplicação com as metodologias tradicionais durante o ensino de tópicos abstratos como os de Estruturas de Dados.

O número de aplicativos Visualizadores atualmente é bastante grande principalmente quando comparado ao número de aplicativos Verificadores. Característica esta que pode ser notada pela grande quantidade de aplicativos, applets disponíveis pela internet. São exemplos o *AVL tree applet* [3] (Figura 1.1) e o *Interactive Data Structure Visualizations*[5] (Figura 1.2).

Outro exemplo de visualizador é a aplicação, aparentemente sem nome, apresentada por Tao Chen e Tarek Sobh em [6]. Essa aplicação, além de possuir uma in-



**Figura 1.1:** AVL Tree.

terface na qual o usuário pode realizar as operações básicas de estruturas de dados (inserir nodo, remover nodo, ordenar), também lhe a opção de programar sua estrutura de dados através de uma linguagem desenvolvida especialmente para a aplicação chamada JavaMy. Com essa linguagem, o usuário pode instanciar estruturas de dados predefinidas ditas como ‘observáveis’. Tendo criado a estrutura, poderá utilizar os métodos disponíveis e assim visualizar seu funcionamento. A possibilidade de poder programar um algoritmo a ser visualizado lhe dá uma flexibilidade a mais, uma vez que a maioria dos visualizadores permitem apenas a utilização de métodos pré-programados. Uma boa iniciativa nacional de visualizador é o Astral, um visualizador de estruturas de dados desenvolvido no Instituto de Computação da UNICAMP entre os anos de 1995 e 1997. Ele foi desenvolvido inicialmente para plataforma Macintosh mas foi, recentemente, portado para Windows. Esse aplicativo implementa e disponibiliza algumas estruturas para visualização, entre elas Array, Lista Ligada, Árvore AVL, Árvore-B, Hash, entre outras. O Astral disponibiliza ao usuário uma série de comandos que podem ser utilizados sobre uma estrutura. Na Figura 1.3, é apresentada a seqüência do comando inserir em uma lista ligada.

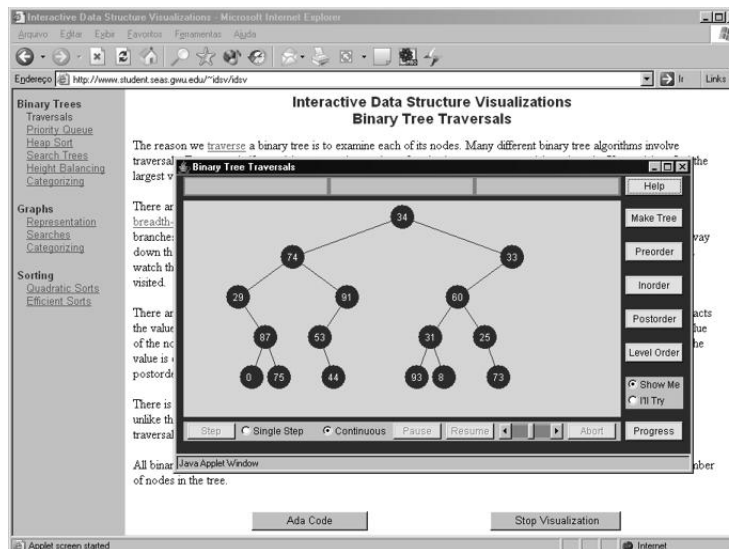
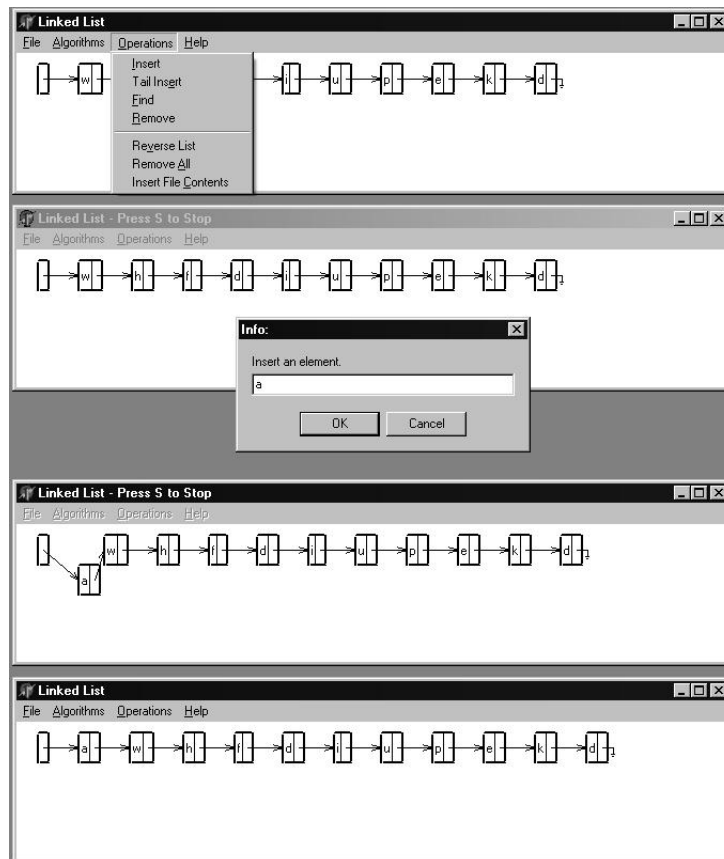


Figura 1.2: Interactive Data Structure Visualizations.

## 1.1.2 Verificadores

Os Verificadores são aplicativos que possuem um objetivo primário um pouco diferente dos Visualizadores. Seu objetivo é prover algum método para o aluno e/ou o professor validar ou analisar a sua estrutura de dados. Os Verificadores, quando comparados com os Visualizadores, são muito mais dificilmente encontrados. Alguns exemplos são o Try System [9] e o JDSL Tester [1].

O Try System, usado em plataforma Unix, funciona basicamente criando um ambiente seguro, no qual as aplicações desenvolvidas pelos alunos serão testadas. Os testes são realizados através de *scripts* editados pelo professor. Há uma variedade definida de *scripts*, cada um com sua finalidade (por exemplo, o *builder* que constrói a aplicação, o *run* que executa os testes, o *cleanup* entre outros). Quando o Try é executado cria um diretório 'scratch', para onde os arquivos a serem avaliados são copiados. Nesse diretório, o projeto (conjunto de arquivos do aluno) é construído seguindo diretrizes predefinidas pelo professor através do *script* por ele editado. A partir daí, dependendo das opções definidas ao Try, são executados métodos de segurança, ou não, ou executadas algumas funções secundárias. Os testes propriamente ditos da aplicação avaliada são executados



**Figura 1.3:** Lista ligada no Astral.

diversas vezes, com várias entradas, pelo *script* 'run', que compram as saídas para avaliar se a aplicação está correta. Outra funcionalidade do Try é a geração de um arquivo log com os resultados dos testes para análise posterior pelo professor. O aluno não tem acesso a esse arquivo.

O JDSL Tester funciona utilizando uma lógica semelhante à do Try System, a de analisar o resultado dos métodos executados. O JDSL Tester é composto por quatro componente principais: o *parser*, *tester*, *factory* e a interface *comparator*. O *tester* utiliza o *factory* para criar as estruturas de dados para então executar seus métodos, que são especificados através de *strings*, decodificadas pelo *parser*. Finalmente, a interface *comparator* disponibiliza um *framework* para comparação de duas estruturas de dados.

Exemplo:

...

```
for(int i=0;i<10;i++) {  
    execute("insert","Integer("+i+") ",  
    "Integer( "+(10-i)+")" );  
    executeCheckReturn("size","void ","int "+i);  
}
```

...

O exemplo acima ilustra um pedaço de código utilizado no JDSL Tester. Nele, o usuário, através do método *execute*, executa o método *insert* de sua estrutura. O nome desse método foi passado como uma *string* juntamente com dois *integer*'s, também passados como *string*'s, que serão utilizados como parâmetros para o método a ser executado. Em seguida é usado o método *executeCheckReturn*, que funciona semelhante ao método *execute*; recebendo *string*'s como parâmetros. O primeiro *string* define o nome do método a ser executado, a segunda *string* define seu parâmetro, a terceira *string* define o valor com o qual o retorno do método executado deve ser comparado.

A falta de pesquisas nesse grupo de aplicativos torna a realização de um trabalho nessa área algo muito mais interessante. E é nesse contexto que se situa o presente projeto.

# Capítulo 2

## Objetivos

### 2.1 Objetivo Geral

O objetivo geral deste trabalho é desenvolver uma aplicação, disponível via *Web*, cuja função auxiliar será de auxiliar a avaliação de implementações de estruturas de dados escritas na linguagem *Java*.

### 2.2 Objetivos Específicos

O trabalho tem como objetivo também, além da atividade prática da criação de uma aplicação, o estudo de certas áreas da ciência da computação. Entre elas estão:

- Estudo de sistemas distribuídos: suas arquiteturas e o funcionamento de uma aplicação disponibilizada on-line via *Web*;
- estudo de particularidades e aprofundamento dos conhecimentos da linguagem de programação *Java*. Particularidades como por exemplo a reflexão<sup>1</sup>, *classloader*, programação para *Web*, entre outras;

---

<sup>1</sup>Capacidade de explorar e manipular métodos e atributos de uma classe em tempo de execução.



- familiarização do autor do trabalho com o uso de *frameworks* e outras ferramentas usadas para a concepção de um *software*.

# Capítulo 3

## Contextualização

### 3.1 Evolução das aplicações para *Web* com foco na tecnologia *Java*

No final da década de 1980 e no início da década de 1990, a popularização da internet ganhou um grande impulso com a criação da *Web*. A possibilidade de se requisitar um documento e visualizá-lo através de um *browser* de uma maneira simples e bastante rápida foi um dos principais motivos pelo qual a internet é hoje uma grande febre em todo o mundo.

A principal razão pela qual a requisição de um arquivo pela *Web* é tão rápida é a utilização de um protocolo simples e não orientado a estados: o HTTP (para mais informações sobre HTTP consulte o apêndice A na página 50). O fato de não precisar manter uma conexão possibilita esse protocolo tratar um grande fluxo de requisições.

Todavia, a arquitetura da *Web* e do protocolo HTTP foi projetada inicialmente para a visualização de conteúdo estático. Ou seja, o cliente faz uma requisição de arquivo a um servidor que o envia ao cliente, sem realizar nenhum processamento sobre ele. Claro que esse arquivo pode mudar de tempos em tempos, quando for atualizado por um administrador por exemplo, mas, a princípio, seu conteúdo não é alterado mediante a interação do usuário.

### 3.1.1 Common Gateway Interface

Com o passar do tempo e o amadurecimento da *Web*, algumas tecnologias foram criadas na tentativa de superar essa falta de interação com o usuário. Uma das primeiras a ser utilizada amplamente foi o **Common Gateway Interface**, também conhecido como CGI. Uma aplicação CGI funciona como uma “ponte” entre o servidor *Web* e uma outra aplicação da máquina servidor utilizando características do sistema operacional (como variáveis de ambiente e dispositivos de I/O) para realizar a comunicação entre ambas as partes. O CGI também define uma série de convenções que estipulam como as variáveis de ambientes e os dispositivos de I/O devem ser utilizados.

Então, quando um servidor *Web* recebe uma requisição endereçada a um CGI, ele executa o programa CGI. Este processa as informações e retorna uma resposta ao servidor *Web* que por sua vez, envia uma resposta ao cliente.

Porém, a grande deficiência do CGI é o seu grau de processamento de que necessita. Pois para cada requisição a ele endereçada é criado um novo processo, o que acaba consumindo grande quantidade de recursos da máquina servidor.

### 3.1.2 Servlets

Foi então que em 1997, no momento em que a linguagem *Java* estava apresentando um crescimento e uma aceitação bastante grande, a tecnologia **Java Servlet** foi criada.

*Servlets* são objetos *Java* como quaisquer outros mas que tem como responsabilidade receber e tratar as requisições a eles endereçadas. Os *servlets* são utilizados em conjunto com *servlet containers* que atuam juntamente com o servidor *Web* para tratar as requisições do cliente. Cada *servlet* pode se responsabilizar por tratar um ou mais tipos de requisições. Então quando um requisição chega ao servidor, é mapeada por ele ao *servlet* responsável por tratá-la. Executada a tarefa, o *servlet* retorna uma resposta ao servidor. Ao contrário dos CGI's, várias requisições são tratadas por *threads* diferentes ao invés de processos diferentes, o que gera um consumo de recurso menor aumentando assim a eficiência do *servlet* em relação ao CGI. Outra grande vantagem dos *servlets* re-

side no fato de terem disponíveis para si todas as vantagens da linguagem *Java*, como por exemplo, portabilidade, rica API, entre outras.

Na arquitetura da tecnologia *Java Servlet* há dois principais pacotes: *javax.servlet* e *javax.servlet.http*. Esses dois pacotes possuem as interfaces e as classes implementadas e/ou estendidas por todos os *servlets*. No primeiro pacote residem as classes bases como a interface *javax.servlet.Servlet*. Já no segundo pacote estão localizadas as interfaces e classes que tratam as requisições HTTP. É esse último pacote que possui a classe abstrata *javax.servlet.http.HttpServlet*, classe que a maioria dos *servlets* estendem e onde são definidos os métodos que deverão ser implementados para tratar cada tipo de requisição. Por exemplo, para tratar uma requisição do tipo GET e POST são executados os dois seguintes métodos respectivamente:

```
protected void doGet(HttpServletRequest req, HttpServletResponse
resp) throws ServletException, IOException
```

e

```
protected void doPost(HttpServletRequest req, HttpServletResponse
resp) throws ServletException, IOException
```

Assim como a requisição do tipo HEAD é tratada pelo método *doHead* e assim por diante.

A maneira como o *servlet* responde ao servidor é que necessita de maior atenção. Como pode ser observado acima, os métodos *doGet* e *doPost* recebem como parâmetros objetos do tipo *HttpServletRequest*, que representa a requisição do cliente, e do tipo *HttpServletResponse*, que representa a resposta ao cliente. É através desse objeto que o *servlet* retorna informação ao cliente.

Abaixo, segue um trecho de código de implementação do método *doGet* que exemplifica uma maneira como um *servlet* retorna uma resposta ao cliente.

```
...
public void doGet(HttpServletRequest req, HttpServletResponse
```

```

res) throws ServletException, IOException {
    ...
    PrintWriter out = res.getWriter();
    out.println("<html>");
    out.println("<head>");
    out.println("<title>.: Meu Servlet :.</title>");
    out.println("</head>");
    out.println("<body>");
    out.println("<h1> Meu Servlet </h1>");
    out.println("</body>");
    out.println("</html>");
    ...
}
...

```

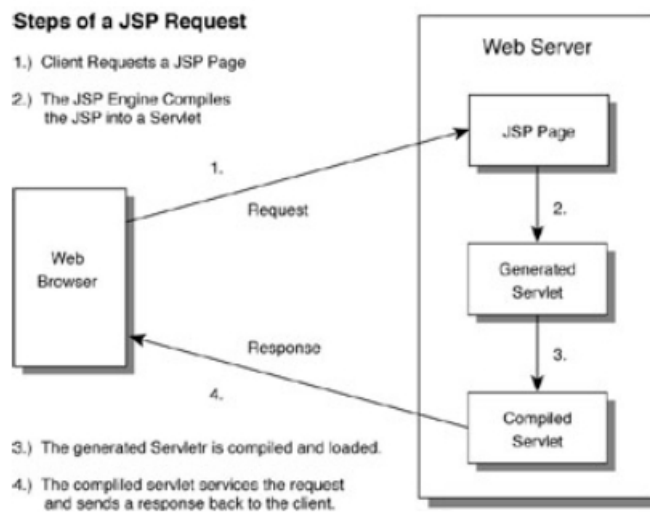
Como pode ser observado acima, a resposta ao cliente é codificada internamente, através do código HTML, ao próprio *servlet*. Essa prática gera alguns transtornos aos desenvolvedores de aplicações devido à mistura de código HTML com código *Java* e também pela necessidade de recompilação da classe para cada alteração que se faça necessária na camada de interface (código HTML).

### 3.1.3 JavaServer Pages

A fim de solucionar o problema acima mencionado, foi criada a tecnologia **JavaServer Pages**, ou JSP. O JSP funciona como uma extensão da tecnologia *Java Servlet* para facilitar a dissociação entre interface (HTML) e servlets. JSP são documentos de textos planos que contém uma mistura de HTML estático, *tags* XML e *scriptlets*. As *tags* e os *scriptlets* são os elementos responsáveis por encapsular a lógica que gera o conteúdo dinâmico da página requisitada.

Quando uma requisição é feita a uma página JSP, essa página é processada em um arquivo *Java* que é, então, compilado em uma classe para depois ser

executado pelo *container Servlet*. No final de todo o processo, a página JSP não se torna nada mais do que um *servlet*. A Figura 3.1 retirada de [19] descreve este processo.



**Figura 3.1:** Passos da requisição de um arquivo JSP.

A grande vantagem na utilização da tecnologia JSP reside na separação das funções de interface das funções lógicas da implementação de um *servlet*. Uma nítida diferenciação dos dois conceitos torna o desenvolvimento de uma aplicação um processo mais produtivo e de fácil manutenção.

### 3.1.4 Model 1, Model 2 e MVC

*Model 1* e *Model 2* são modelos para construção de aplicações *Web* apresentados nas primeiras especificações JSP. Apesar de esses modelos não aparecerem mais nas especificações, eles ainda são termos muito citados na comunidade de desenvolvedores.

A principal diferença entre os dois modelos reside no fato de como e por quem uma requisição deve ser tratada. No *Model 1* as requisições são tratadas diretamente por uma página JSP que se responsabiliza pela exibição da interface ao cliente, bem como

pela comunicação com a camada lógica da aplicação (classes *Java* ou outros). Também é a página JSP que determina qual será a próxima página a ser exibida.

Em contrapartida, na filosofia do Model 2 existe um *servlet* chamado *controller servlet*, especialmente dedicado a receber as requisições do cliente e determinar qual a página JSP a ser exibida. A utilização do Model 2 proporciona uma maior organização, e conseqüentemente uma maior extensibilidade do projeto, devido a uma clara separação das responsabilidades de visualização, tratamento de requisição e da lógica da aplicação em si. Essa divisão de responsabilidades muitas vezes é referida como sendo o paradigma de modelagem MVC.

### 3.1.4.1 Modelo MVC

O paradigma de modelagem MVC — acrônimo de **Model View Controller** — é original dos anos 80 e foi utilizado a princípio na criação de aplicações em Smalltalk. A principal filosofia desse paradigma é uma clara divisão da aplicação em três camadas: o Modelo, a Visão e o Controle.

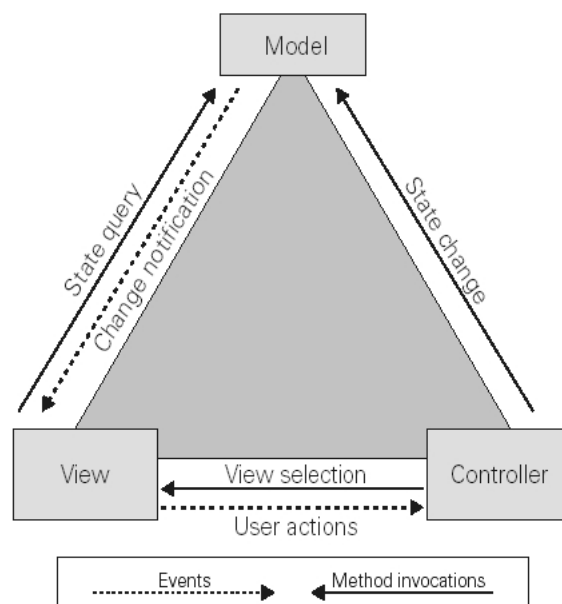
**Modelo:** A camada Modelo representa a lógica da aplicação, seus dados e suas regras de negócio. Representa a abstração da realidade modelada.

**Visão:** Camada responsável por exibir o conteúdo da camada Modelo. É sua responsabilidade definir como as informações da camada Modelo devem ser exibidas ao usuário.

**Controle:** Esta camada trata os eventos que afetam a Visão ou o Modelo. É responsável por receber interações da camada Visão e transformá-las em ações a serem tomadas pela camada Modelo.

No MVC, o Controle recebe os eventos, geralmente da camada Visão, e determina qual a ação a ser tomada pela camada Modelo, assim como o que deve ser alterado na camada Visão. A camada Modelo, atendendo ao comando da camada Controle, executa suas funções. A camada Visão, então, requisita à camada Modelo as informações

necessárias para a sua atualização. O processo pode ser observado na Figura 3.2, retirada de [21], abaixo:



**Figura 3.2:** Esquema MVC.

#### 3.1.4.2 MVC em aplicações *Web*

Em aplicações *Web*, a separação das camadas do MVC pode ser feita da seguinte maneira:

**Modelo:** A finalidade desta camada varia bastante de aplicação para aplicação. Mas incluem-se aqui as classes da aplicação, EJB, relacionamento com banco de dados, entre outro.

**Visão:** Consiste basicamente de arquivos HTML e JSP que são usados, através de um *browser*, como interface entre a aplicação e o usuário. Aqui as informações da camada Modelo devem ser exibidas ao usuário.

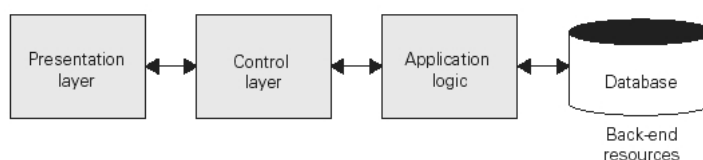
**Controle:** Nessa camada é utilizado o *controller servlet*. É tarefa desse servlet receber



requisições de um cliente, definir a operação a ser tomada para cada requisição, definir a Visão retornada ao cliente e então, para finalizar, retornar esta Visão.

Todavia, há um problema na utilização do MVC na sua forma plena em aplicações para *Web*. Na arquitetura tradicional do MVC, a Visão se atualiza automaticamente quando recebe uma notificação de alteração da camada Modelo e em uma aplicação para *Web* essa notificação por parte do Modelo é impossível de ser feita, ou pelo menos muito difícil de se conseguir. Em tais aplicações, cabe ao cliente fazer uma nova requisição ao Controle para que haja uma atualização na Visão. E é neste ponto que se diferenciam o Model 2 e o MVC. Entretanto, apesar dessa diferença, muitos autores usam o termo MVC para definir o padrão usado em aplicações *Web*.

O que é utilizado então é uma alternativa, um MVC alterado para as características de uma aplicação *Web*. O resultado da alteração é esquematizada na Figura 3.3 retirada de [21] mostrada abaixo.



**Figura 3.3:** Esquema MVC modificado.

Como pode ser observado, a principal mudança aplicada é a troca de um relacionamento triangular entre as camadas por um relacionamento linear, onde a camada Controle é posta entre as camadas Visão e Modelo. Neste novo esquema, a Visão passa a receber as informações do Modelo através do Controle, ao invés de recebê-las diretamente da camada Modelo. Apesar da mudança de comunicação entre as camadas, as responsabilidades de cada uma delas permanecem as mesmas. E é baseado nesse paradigma de modelagem, juntamente com o uso do *framework Struts*, que a aplicação deste trabalho é implementada.

### 3.1.5 Framework Jakarta Struts

O projeto Struts tem como objetivo prover um *framework open source* para o auxílio na construção de aplicações para *Web* em Java. Foi originalmente criado por Craig R. McClanahan e atualmente faz parte do *Apache Jakarta Project* [15] mantido pela *Apache Software Foundation* [14]. Esse framework será amplamente utilizado neste projeto.

A principal funcionalidade do Struts é o fornecimento de uma camada Controle baseada em tecnologias como *Java Servlets*, Java Beans, XML, sobre a qual será estruturada a aplicação desenvolvida. Todavia, esse framework não se detém só a isto. Ele também provê funcionalidades para a criação das camadas Visão e Modelo. Como pode ser notado, o *framework* Struts incentiva em muito o desenvolvedor a usar o modelo MVC de modelagem.

### 3.1.6 Funcionamento Básico

As principais classes do Struts são as relacionadas à camada de Controle, dentre as quais se pode relacionar as seguintes classes

- *org.apache.struts.action.ActionServlet*
- *org.apache.struts.action.Action*
- *org.apache.struts.action.ActionForm*
- *org.apache.struts.action.ActionMapping*
- *org.apache.struts.action.ActionForward*

A classe *ActionServlet* estende a classe *javax.servlet.http.HttpServlet* e geralmente atua como o *controller servlet* do *framework*. É ela a responsável por receber as requisições HTTP e redirecioná-las à entidade apropriada que tratará a requisição. Assim que o Controle recebe uma requisição do cliente, delega seu tratamento a uma classe que deve ser estendida da classe *Action*. Essa classe atua como uma ponte entre as

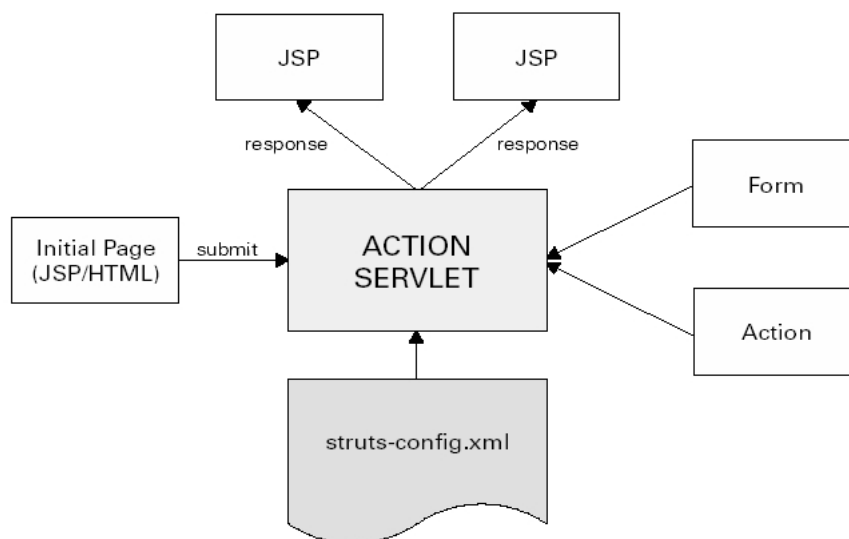
requisições do cliente e a camada Modelo. O principal método da classe *Action* é o que possui a seguinte assinatura:

```
public ActionForward execute(ActionMapping mapping, ActionForm  
form, javax.servlet.http.HttpServletRequest request,  
javax.servlet.http.HttpServletResponse response) throws  
java.lang.Exception
```

Todas as vezes que o *ActionServlet* delega o tratamento de uma requisição a um *Action* ele o faz executando esse método. Para validar e receber as informações submetidas pelo usuário, o *ActionServlet* cria e envia como parâmetro ao *Action* uma instância de uma subclasse da classe *ActionForm*. A classe *ActionServlet* pode determinar qual *Action* e *ActionForm* utilizar através de um arquivo XML, geralmente chamado *struts-config.xml*, que mapeia os respectivos *Action's* e *ActionForm's* para cada requisição, além de possuir outros dados de configuração.

Após o tratamento da requisição, é necessário o envio de uma resposta. O Struts por si só não define a resposta, mas a mapeia através do arquivo XML de configuração (normalmente *struts-config.xml*) e encaminha uma requisição para outra fonte, geralmente outro arquivo JSP ou HTML. Para tal, o Struts possui uma classe chamada *ActionForward* que possui este tipo de informação. Um objeto *ActionForward* é retornado ao *servlet* após a execução do método *execute* da classe *Action*. Veja na Figura 3.4, retirada de [21], um esquema simplificado do framework Struts.

Partindo deste ponto, com todas estas tecnologias bastante amadurecidas, o desenvolvimento de aplicações *Web* tornou-se um processo muito mais simples, rápido e confiável. Fato que está colaborando com o grande crescimento da procura de aplicações do tipo atualmente.



**Figura 3.4:** Esquema Struts.

# Capítulo 4

## Desenvolvimento

### 4.1 Definindo o Problema

#### 4.1.1 Introdução

Como dito anteriormente, este trabalho tem como objetivo a criação de uma aplicação que, via *Web*, auxilie a validação de implementações de estruturas de dados escritas em *Java*. Mas o que significa dizer que uma implementação está correta? O que é preciso analisar para se chegar a tal conclusão?

A validação de uma implementação pode ser dada levando-se em consideração vários parâmetros, dependendo do objetivo e/ou do público final do trabalho. Por exemplo, se no uso de uma implementação o tempo de resposta for essencial para o bom funcionamento do sistema então o tempo de resposta deve ser um dos parâmetros avaliados para a validação dessa implementação. Caso o tempo de resposta não seja tão importante, pode-se levar em consideração somente o resultado obtido pela implementação, ou então o método utilizado para a obtenção desse resultado, ou ainda os dois. As possibilidades são várias.

### 4.1.2 Definindo os parâmetros a serem analisados

Uma estrutura de dados é, sob certo ponto de vista, um objeto que possui uma interface para o mundo externo. Através dessa interface são aplicadas sobre a estrutura operações que alteram seu estado e/ou retornam um valor. Sendo assim, os parâmetros fundamentais para se analisar o resultado de uma operação executada sobre uma estrutura de dados são o retorno da operação e a maneira pela qual a operação executada afetou essa estrutura. Mas então chegamos a outra pergunta: como analisar o estado de uma estrutura?

Uma estrutura de dados é, também, uma portadora de informações. A diferença entre as estruturas de dados dá-se pelo método como essas informações são armazenadas em cada uma. O método pode variar não somente de acordo com o tipo da estrutura mas também de implementação para implementação. Variando conforme as práticas e técnicas do programador. Frequentemente estruturas de dados implementadas por programadores diferentes possuem representação interna totalmente distintas, mas comportamentos corretos. Dada essa possível diferença de implementação, a atitude mais correta é delegar ao próprio programador da estrutura a tarefa de implementar um método que represente fielmente a sua estrutura a fim de que possa ser analisada.

A possível diferença de estruturação interna e a quase certa distinção na implementação levantam um outro parâmetro importante na implementação de uma estrutura de dados: o tempo de resposta do método. Esse parâmetro não está relacionado tão diretamente com o fato de uma estrutura estar correta ou não, mas sim, com a sua eficiência. Todavia, é um parâmetro interessante a ser avaliado pelo usuário que deseja melhorar a performance da sua implementação.

Outro parâmetro que não está relacionado a estruturas de dados em geral mas a uma particularidade de linguagem de programação é a exceção lançada pelo método executado. Em *Java*, e também em outras linguagens de programação modernas, o fluxo do código pode levar a situações que não podem ou não devem ser tratadas pelo próprio método. Nesse caso, são lançadas exceções que poderão ser (ou não) tratadas em outros pontos da implementação. Então, a exceção lançada por um método de uma estrutura de

dados é um parâmetro importante na análise do seu funcionamento.

Foram definidos acima, quatro principais parâmetros utilizados na análise do funcionamento de um método de uma estrutura de dados:

- Retorno do método;
- alteração sofrida pela estrutura;
- tempo de execução e
- a exceção lançada.

Esses serão, então, os parâmetros utilizados pela aplicação no presente trabalho para exibir ao usuário os resultados do método a ser analisado.

## 4.2 Definindo o funcionamento da aplicação

Já foi mencionado anteriormente que o objetivo deste trabalho de conclusão de curso é desenvolver uma aplicação para *Web* para o auxílio do ensino de Estrutura de Dados. Mas até o momento não foi definido como será o funcionamento dessa aplicação nem como poderá auxiliar os alunos na disciplina.

A aplicação desenvolvida no trabalho deve disponibilizar ao estudante a possibilidade de acessá-la via *internet* e fazer *upload* da classe a ser analisada e da classe padrão, que por definição funciona corretamente, juntamente com as classes necessárias para a execução dos métodos das classes analisada e padrão. Feito o *upload* das classes, essas serão carregadas para a Máquina Virtual da máquina remota, onde a aplicação está sendo executada<sup>1</sup>.

Feito isso, o usuário pode escolher uma interface que é implementada pelas duas classes (a ser analisada e a padrão) e que define os métodos da estrutura de dados. Deverá escolher também qual construtor deseja utilizar para criar a estrutura. Para

---

<sup>1</sup>Esse processo é realizado através de um *Class Loader* desenvolvido para esta aplicação. Para mais informações sobre *Class Loaders* veja a seção B na página 54.

esta aplicação definiu-se que o usuário deverá utilizar, para as duas classes, tanto para a classe analisada quanto para a padrão, construtores com a mesma assinatura, ou seja, que possuam o mesmo número de parâmetros e dos mesmos tipos. Escolhido o construtor, a estrutura de dados é criada.

Com as estruturas de dados criadas, o usuário poderá escolher qual método da estrutura ele deseja executar. Após cada método executado, um relatório com os parâmetros definidos anteriormente é exibido para que este possa analisar os resultados da sua implementação, comparando-os ao resultados da implementação padrão<sup>2</sup>.

Caso o usuário deseje também, é possível emitir um conjunto de relatórios composto pelos relatórios gerados pelos métodos executados até então.

Desta maneira, o usuário poderá visualizar a evolução da sua estrutura de dados à medida que executa os métodos, facilitando a identificação de erros na sua implementação.

## 4.3 Modelagem

### 4.3.1 Definição dos Atores

Ao analisar o funcionamento da aplicação, pode-se perceber que a única entidade que interage com o sistema é o próprio usuário. Então, definiu-se como sendo o único ator da nossa aplicação, o ator *usuário*.

### 4.3.2 Casos de Uso

Continuando a análise do funcionamento da aplicação, ficam evidentes seis principais casos de uso:

1. *Upload* de classes secundárias;
2. *upload* das classe principais;

---

<sup>2</sup>O processo de invocar métodos, construtores, pesquisar interfaces de uma classe pode ser realizado através de reflexão, uma característica da linguagem Java. Para mais informações veja [27].



3. definição da interface analisada;
4. definição do construtor utilizado;
5. definição do método a ser executado e
6. definição de parâmetros.

A seguir serão definidos os casos de uso.

Caso de uso:	<b><i>Upload de classes secundárias</i></b>
Atores:	Usuário.
Finalidade:	Enviar as classes necessárias para o funcionamento da classe analisada e da classe padrão.
Visão Geral:	O usuário determina ao sistema qual a classe deseja carregar. Essa é então enviada ao servidor e carregada na Máquina Virtual do computador remoto.
Tipo:	Principal e essencial.
Referências cruzadas:	

**Tabela 4.1:** Caso de uso: *Upload* de classes secundárias

- Sequência Típica dos Eventos

1. O sistema exibe ao usuário a tela onde este informará a classe a ser enviada. O usuário poderá escolher a classe escrevendo o nome do arquivo ou através de uma caixa de diálogo.
2. O usuário escolhe a classe e aperta o botão de envio.
3. O sistema recebe o arquivo classe do usuário e o carrega na Máquina Virtual Java.

4. O sistema exibe ao usuário a mensagem de sucesso e disponibiliza a este a possibilidade de enviar outra classe secundária ou finalizar o processo e escolher as classes principais.

- Sequências alternativas

1. O arquivo enviado não é um arquivo *class* válido: o sistema retorna à tela inicial do caso de uso e exibe ao usuário uma mensagem de erro.

Caso de uso:	<b><i>Upload das classes principais</i></b>
Atores:	Usuário.
Finalidade:	Enviar a classe que implementada pelo aluno e classe padrão implementada pelo professor.
Visão Geral:	O usuário informa ao sistema qual é o arquivo <i>class</i> que representa a sua implementação e também o arquivo <i>class</i> que representa a implementação do professor.
Tipo:	Principal e essencial.
Referências cruzadas:	<i>Casos de Uso:</i> o usuário deve ter completado o caso de uso <i>Upload de classes secundárias</i>

**Tabela 4.2:** Caso de uso: *Upload das classes principais*

- Sequência Típica dos Eventos

1. O sistema exibe ao usuário a tela onde este informará os arquivos a serem enviados, os arquivos do tipo *class* que representam a sua implementação e a implementação do professor. O usuário poderá escolher as classes escrevendo o nome dos arquivos ou através de uma caixa de diálogo.
2. O usuário escolhe as classes e aperta o botão de envio.

3. O sistema recebe os arquivos *class* do usuário e os carrega na Máquina Virtual Java.

- Sequências alternativas

1. Algum dos arquivos enviados não é um arquivo *class* válido: o sistema retorna à tela inicial do caso de uso e exibe ao usuário uma mensagem de erro.

Caso de uso:	Definição da interface analisada
Atores:	Usuário.
Finalidade:	Definir qual será a interface implementada pela classe do aluno que será analisada.
Visão Geral:	O usuário informa ao sistema quais dentre as interfaces implementadas por ambas as classes principais ele deseja que seja avaliada.
Tipo:	Principal e essencial.
Referências cruzadas:	<i>Casos de Uso</i> : o usuário deve ter completado o caso de uso <i>Upload das classes principais</i>

**Tabela 4.3:** Caso de uso: Definição da interface analisada

- Sequência Típica dos Eventos

1. O sistema exibe ao usuário a tela onde este informará os arquivos a serem enviados, os arquivos do tipo *class* que representam a sua implementação e a implementação do professor. O usuário poderá escolher as classes escrevendo o nome dos arquivos ou através de uma caixa de diálogo.

2. O usuário escolhe as classes e aperta o botão de envio.

3. O sistema recebe os arquivos *class* do usuário e os carrega na Máquina Virtual Java.

4. O sistema encaminha o usuário para a tela inicial do caso de uso *Definição do construtor utilizado*.

- Sequências alternativas

1. Algum dos arquivos enviados não é um arquivo *class* válido: o sistema retorna à tela inicial do caso de uso e exibe ao usuário uma mensagem de erro.

Caso de uso:	<b>Definição do construtor utilizado</b>
Atores:	Usuário.
Finalidade:	Definir quais construtores das classes principais serão utilizados.
Visão Geral:	O usuário informa ao sistema quais serão os construtores de cada classe utilizados para a criação das duas estruturas de dados, a implementada pelo aluno e a implementada pelo professor.
Tipo:	Principal e essencial.
Referências cruzadas:	<i>Casos de Uso</i> : o usuário deve ter completado o caso de uso <i>Definição da interface analisada</i>

**Tabela 4.4:** Caso de uso: Definição da interface analisada

- Sequência Típica dos Eventos

1. O sistema apresenta ao usuário uma lista em que cada item é composto pelo construtor de cada classe que possui as assinaturas iguais.
2. O usuário escolhe uma dupla de construtores.
3. Caso o construtor possua parâmetros, o sistema exibe ao usuário a tela para definição dos valores desses parâmetros. Ver caso de uso *Definição de parâmetros* na tabela 4.6.

4. Caso o construtor não possua parâmetros ou os mesmos já tiverem sido corretamente definidos, o construtor é selecionado e a estrutura de dados é criada.
5. O sistema encaminha o usuário para a tela inicial do caso de uso *Definição do método a ser executado*.

- Sequências alternativas

1. Caso o usuário não defina nenhum construtor, o sistema retorna à tela inicial deste caso de uso.

Caso de uso:	<b>Definição do método a ser executado</b>
Atores:	Usuário.
Finalidade:	Definir qual será o método a ser analisado.
Visão Geral:	O usuário informa ao sistema qual será o método da interface escolhida que será executado.
Tipo:	Principal e essencial.
Referências cruzadas:	<i>Casos de Uso</i> : o usuário deve ter completado o caso de uso <i>Definição do construtor utilizado</i>

**Tabela 4.5:** Caso de uso: Definição do método a ser executado

- Sequência Típica dos Eventos

1. O sistema apresenta ao usuário a lista dos métodos da interface escolhida pelo usuário.
2. O usuário escolhe um dos métodos e submete a sua escolha.
3. Caso o método possua parâmetros, o sistema exibe ao usuário a tela para definição dos valores desses parâmetros. Ver caso de uso *Definição de parâmetros* na tabela 4.6.

4. Caso o método não possua parâmetros ou os mesmos já tiverem sido corretamente definidos o construtor é selecionado e a estrutura de dados é criada.
  5. O sistema exibe ao usuário um relatório contendo as informações recolhidas da execução de cada método.
  6. O sistema disponibiliza ao usuário a opção de retornar à tela inicial deste caso de uso
  7. O sistema disponibiliza ao usuário a opção de visualizar o conjunto de relatórios de todos os métodos executados até o momento
- Sequências alternativas
    1. Caso o usuário não defina nenhum método o sistema retorna a tela inicial deste caso de uso.

Caso de uso:	Definição de parâmetros
Atores:	Usuário.
Finalidade:	Definir os parâmetros necessários para a execução de um método ou construtor.
Visão Geral:	O usuário informa ao sistema os valores necessários para a criação dos parâmetros utilizados por um construtor ou por um método.
Tipo:	Principal e essencial.
Referências cruzadas:	

**Tabela 4.6:** Caso de uso: Definição de parâmetros

- Sequência Típica dos Eventos

1. O sistema exibe ao usuário a tela onde este deve informar os dados necessários para a criação dos parâmetros de determinado método ou construtor. O sistema deverá exibir ao usuário o formulário para o preenchimento dos valores dos parâmetros da seguinte forma:
    - Caso o parâmetro seja do tipo primitivo, o usuário deverá informar diretamente, através de um campo de formulário, o valor do parâmetro.
    - Caso o parâmetro seja um objeto, o sistema deverá oferecer ao usuário as opções de construtores para sua escolha juntamente com a opção de escolher uma subclasse desse objeto, ou então, definir o parâmetro como *null*.
  2. O usuário define a suas opções e submete suas informações ao sistema.
  3. Caso seja escolhida a opção de subclasse, o sistema retorna ao início do caso de uso atualizando o tipo do objeto a ser criado.
  4. Se não houver mais pendências e parâmetros a serem criados, o sistema retorna ao estado anterior ao caso de uso (execução do método ou do construtor).
- Sequências alternativas
    1. Caso o usuário tente definir um valor a um tipo primitivo que não corresponda a esse tipo ou que não possa ser convertido, o sistema deverá retornar à tela inicial e exibir uma mensagem de erro.
    2. Caso o usuário tente definir uma subclasse inválida a um objeto, o sistema deverá retornar à tela inicial e exibir uma mensagem de erro.

### 4.3.3 Diagrama de Classes

O próximo passo após definidos os casos de uso é definir a estruturação das classes, suas responsabilidades e sua disposição em pacotes para uma implementação mais organizada.

As classes da aplicação desenvolvida neste trabalho foram divididas em seis pacotes:

1. *fcv.projeto*
2. *fcv.projeto.loader*
3. *fcv.projeto.executer*
4. *fcv.projeto.structures*
5. *fcv.projeto.report*
6. *fcv.projeto.helpers*

O pacote *fcv.projeto* (Figura 4.1), além de conter os sub-pacotes acima listados, contém as classes que são derivadas da classe *org.apache.struts.action.Action* que possuem a responsabilidade de controladores da aplicação. São elas que, juntamente com as classes derivadas de *org.apache.struts.action.ActionForm*<sup>3</sup>, recebem as requisições do usuário, definem que ação tomar e qual será a resposta retornada. Também faz parte desse pacote a classe *fcv.projeto.ClassBean*, cuja a responsabilidade é guardar as informações referentes as classes analisadas. Informações como as próprias classes (a classe do aluno e a classe padrão), a interface analisada, os construtores utilizados, o método a ser executado entre, outros. Outra classe do pacote é a classe *fcv.projeto.Dispatcher*, que tem por função de criar os objetos, invocar os métodos e gerar os relatórios.

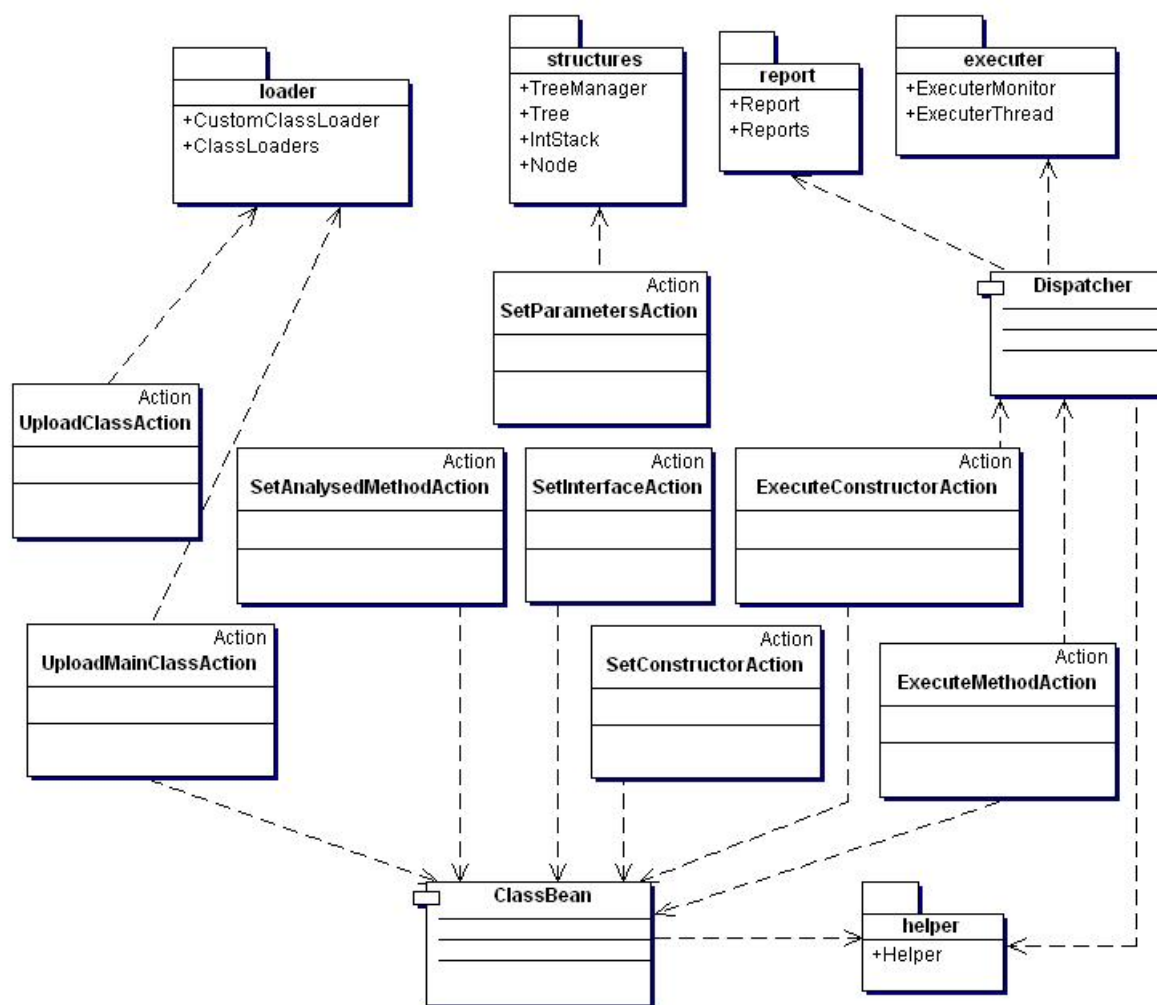
O pacote *fcv.projeto.loader* (Figura 4.2) contém as classes utilizadas para carregar aquelas classes enviadas pelos usuários para a Máquina Virtual Java. A classe *fcv.projeto.loader.CustomClassLoader* é o class loader utilizado pela aplicação. A classe *fcv.projeto.loader.ClassLoaders* é a responsável por armazenar as várias instâncias de *ClassLoader*'s utilizados por cada sessão.

O pacote *fcv.projeto.executer* (Figura 4.3) possui as classes necessárias para executar os métodos a serem analisados. Nesse pacote estão presentes as classes

---

<sup>3</sup>Estas classes foram omitidas do diagrama por não exercerem papel fundamental na compreensão do sistema e para simplificá-lo facilitando a visualização.



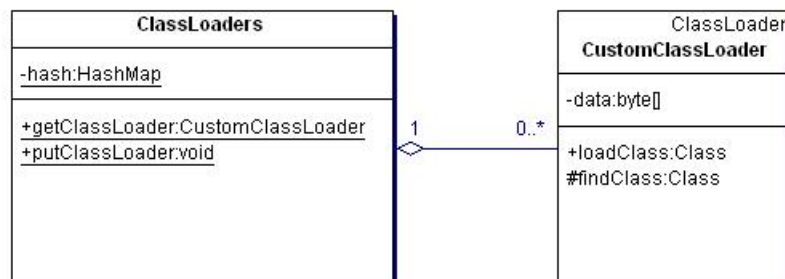


**Figura 4.1:** Pacote *fcv.projeto*.

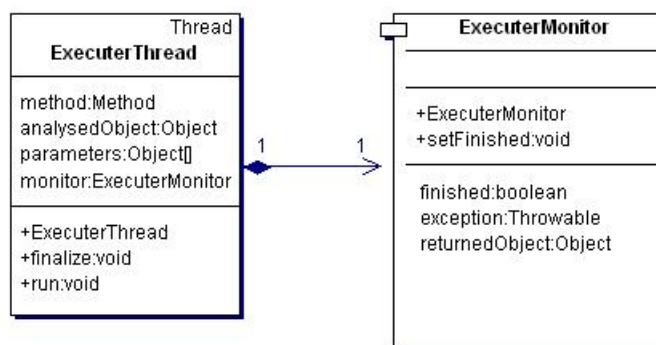
*fcv.projeto.executer.ExecuterThread* e *fcv.projeto.executer.ExecuterMonitor*. Para possibilitar a finalização de um método implementado por um aluno caso este venha a exceder demasiadamente o tempo utilizado pelo método implementado pelo professor, faz-se uso de um *thread*<sup>4</sup>

O pacote *fcv.projeto.structures* (Figura 4.4) contém as classes utilizadas para a definição dos parâmetros de um método ou construtor. Nessa aplicação os parâmetros são armazenados na forma de uma árvore. Daí a necessidade da criação desse pacote.

<sup>4</sup>Para mais informações sobre *threads*, consulte [28].



**Figura 4.2:** Pacote *fcv.projeto.loader*.



**Figura 4.3:** Pacote *fcv.projeto.executer*.

O pacote *fcv.projeto.report* (Figura 4.5) contém as classes utilizadas para guardar as informações dos relatórios gerados.

O pacote *fcv.projeto.helper* (Figura 4.6) foi criado com a intenção de conter classes auxiliares que efetuassem métodos secundários, que não afetassem profundamente o funcionamento da aplicação e que pudessem ser usadas por várias outras classes da aplicação.

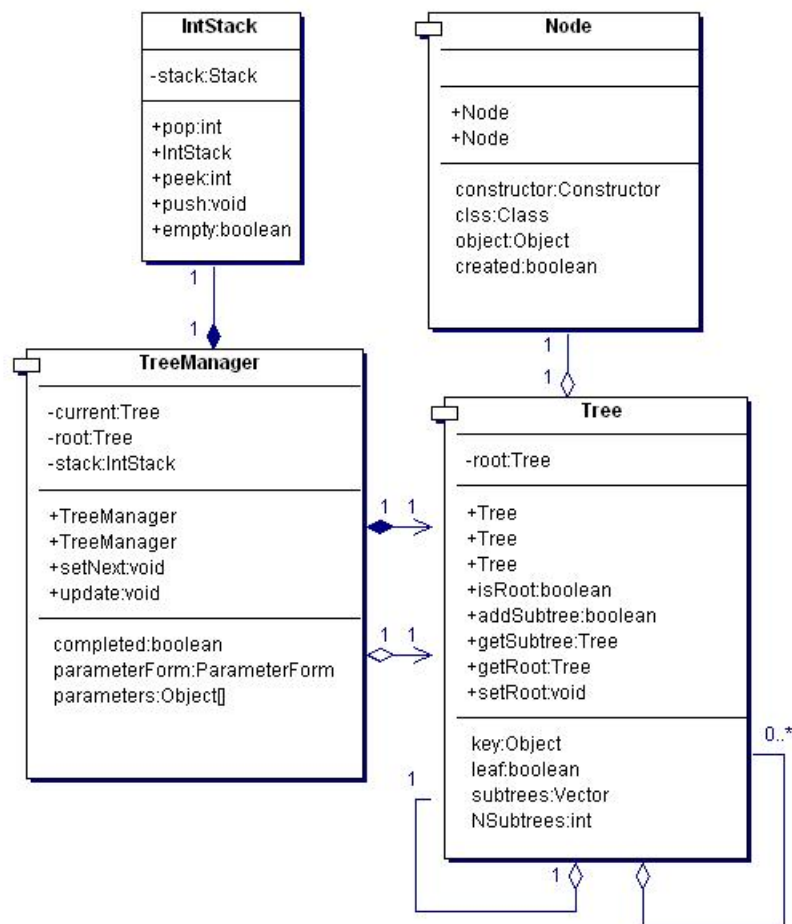


Figura 4.4: Pacote *fcv.projeto.structures*.

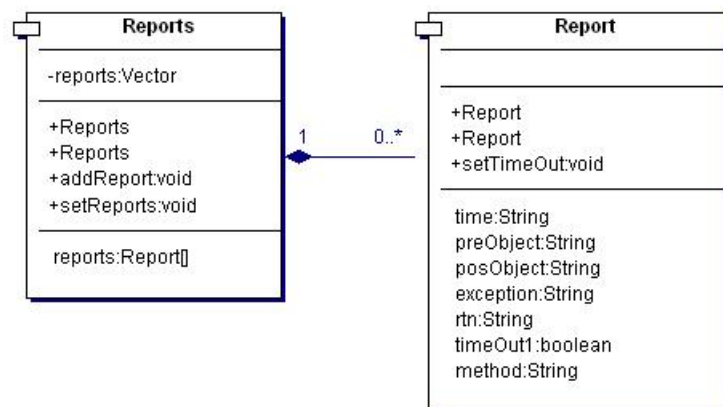
## 4.4 Especificações Técnicas

Para o desenvolvimento da aplicação foram utilizadas as seguintes tecnologias:

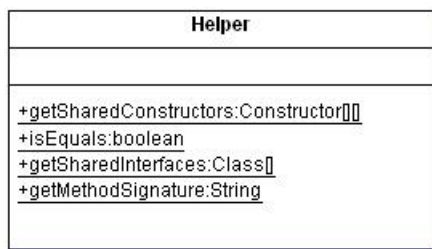
- Java™2 Platform, Standard Edition, v 1.4.2<sup>5</sup>.
- JavaServer Pages.
- Apache Struts Framework v 1.1<sup>6</sup>.

<sup>5</sup>Disponível em <http://java.sun.com>.

<sup>6</sup>Disponível em <http://jakarta.apache.org/struts>.



**Figura 4.5:** Pacote *fcv.projeto.report*.



**Figura 4.6:** Pacote *fcv.projeto.helper*.

- Apache Ant 1.5.1<sup>7</sup>, utilizado para automação do processo de compilação.
- Together ControlCenter Version: 6.0.1<sup>8</sup>, utilizado para a criação dos digramas de classes.

Para a realização de testes foi utilizado o servidor de aplicações Tomcat 4.1<sup>9</sup>.

<sup>7</sup>Disponível em <http://ant.apache.org>.

<sup>8</sup>Para maiores informações consulte <http://www.togethersoft.com/>

<sup>9</sup>Disponível em <http://jakarta.apache.org/tomcat>

# Capítulo 5

## Exemplo do Funcionamento da Aplicação

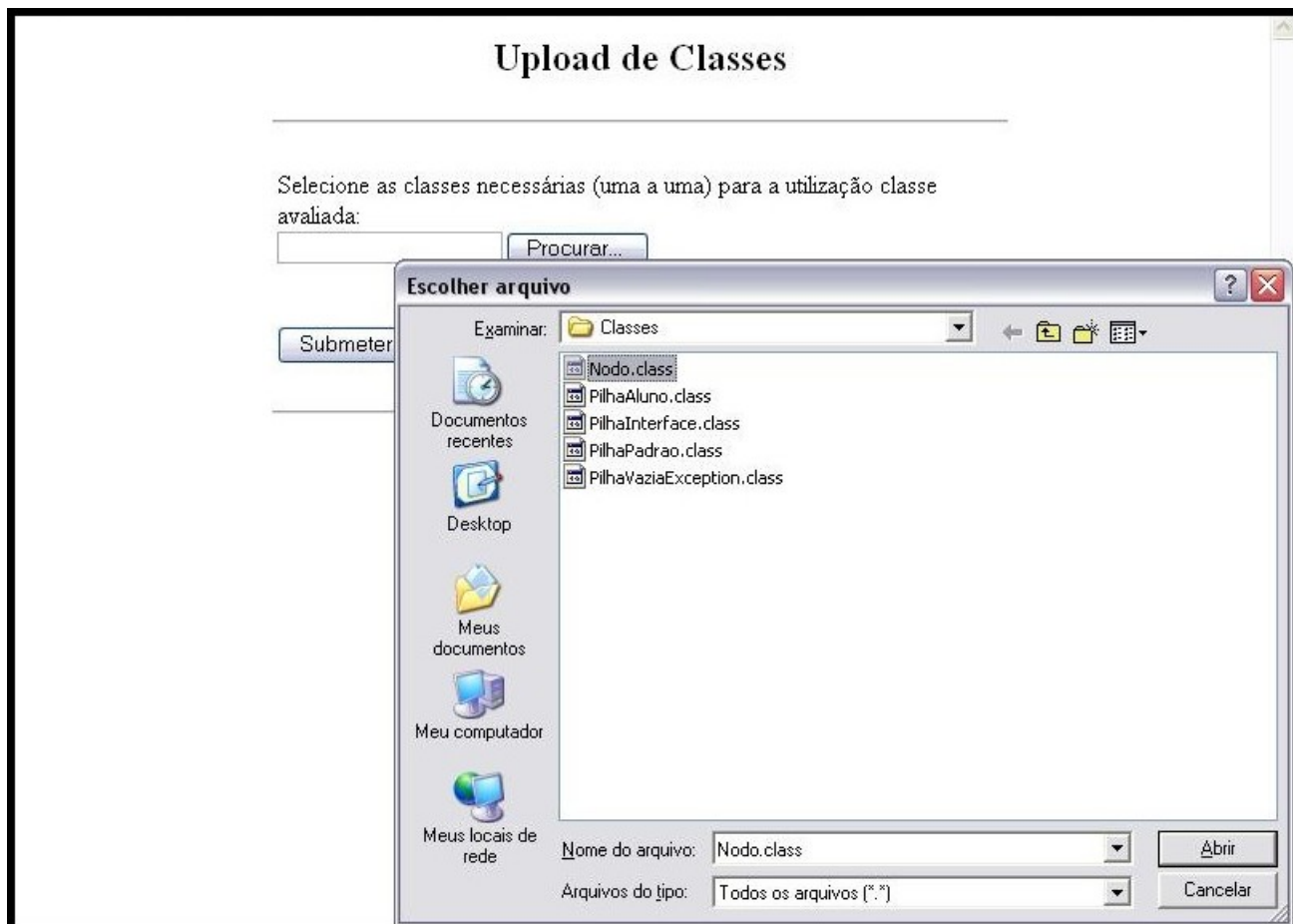
A seguir, é demonstrado o funcionamento da aplicação desenvolvida através de imagens capturadas da tela de um computador. Para a realização dessa demonstração, foi utilizada a implementação de uma *Pilha*.

Na primeira tela apresentada (Figura 5.1) o usuário deve enviar as classes necessárias para o funcionamento, tanto da classe a ser analisada quanto da classe padrão. Essas classes devem ser enviadas uma a uma, ou seja, o usuário envia uma classe, caso seja necessário o envio de outra classe esse processo se repete, caso contrário o usuário deverá escolher a opção de enviar as classes principais (padrão e analisada).

Tendo enviado todas as classes secundárias, o próximo passo é enviar a classe a ser analisada e a classe padrão. Esse processo se dá através da tela apresentada na Figura 5.2 onde o usuário seleciona as classes desejadas e as envia ao servidor.

Finalizado este processo, é exibida ao usuário (Figura 5.3) uma lista com todas as interfaces implementadas por ambas as classes principais. O usuário deverá então escolher qual interface define o funcionamento da estrutura de dados que ele deseja testar.

Escolhida a interface, é necessária a criação das estruturas. Para isso é apresentado ao usuário uma lista com os construtores (Figura 5.5) que possuem a mesma

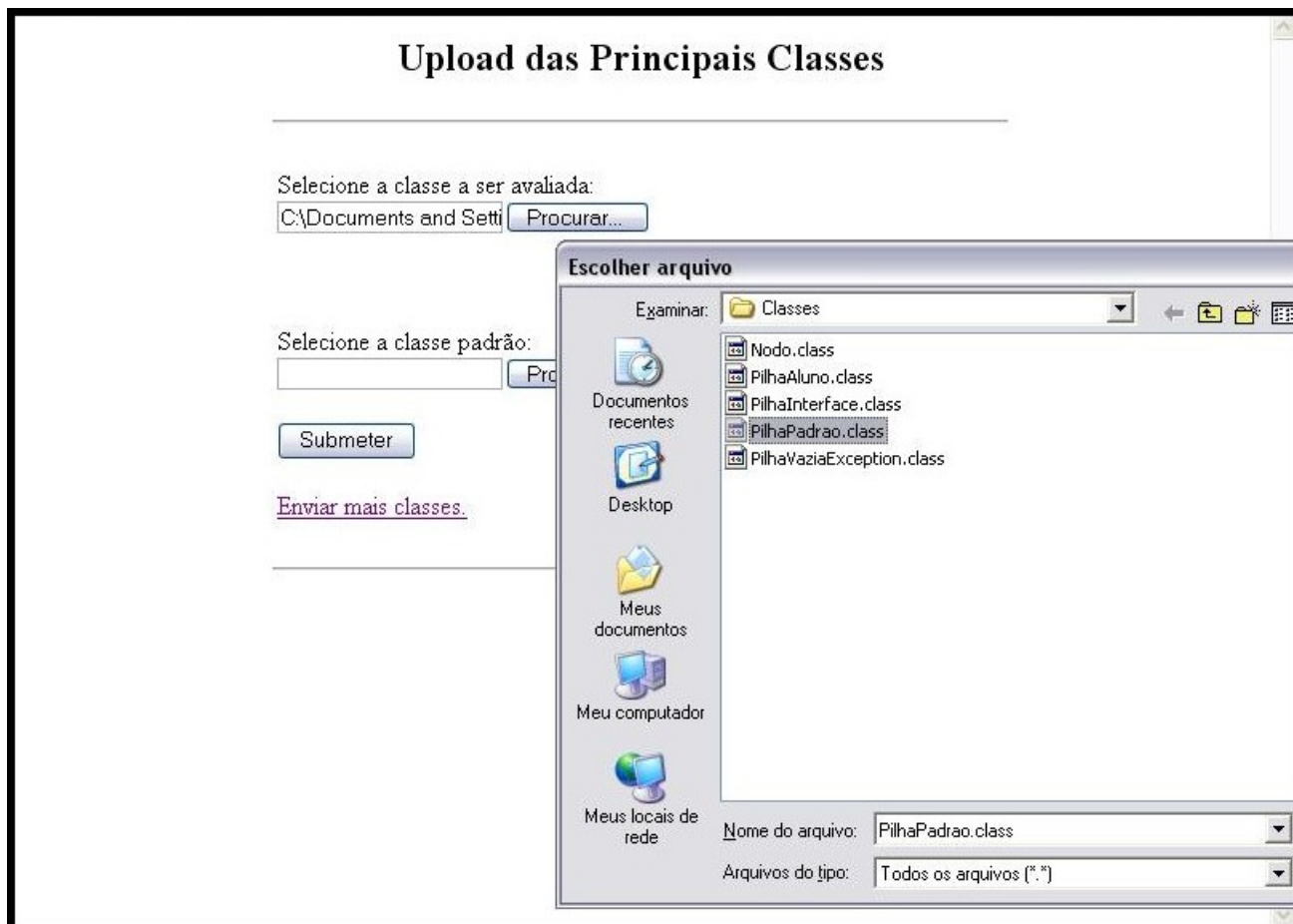


**Figura 5.1:** Tela de *upload* de classes secundárias.

assinatura (mesmo número e tipos de parâmetros) de ambas as classes. O usuário deve escolher qual construtor deseja utilizar para a criação das estruturas.

Após criada a estrutura, é finalmente exibida ao usuário uma lista com todos os métodos definidos pela interface escolhida. O usuário poderá, então, escolher qual o método a ser executado. No presente exemplo primeiramente será executado o método `add(java.lang.Object)`.

Quando um método possuir algum parâmetro, como nesse caso, o usuário será encaminhado para a tela onde deverá fornecer as informações necessárias para a criação desses parâmetros. O método `add(java.lang.Object)` possui um objeto como parâmetro, nesse caso é fornecida ao usuário a possibilidade de escolher um dos con-



**Figura 5.2:** Tela de *upload* das classes principais.

strutores desse objeto, ou utilizar uma subclasse desse objeto ou então passar o valor *null* (Figura 5.6). Neste exemplo foi escolhida a opção de utilizar uma subclasse: *java.lang.Integer*.

O usuário deverá agora definir o método de criação do objeto *java.lang.Integer* (Figura 5.7). Novamente o usuário tem a possibilidade de escolher ou um dos construtores dessa classe, ou uma subclasse ou o valor *null*. Neste caso foi escolhido o construtor que possui como parâmetro o tipo primitivo *int*.

O próximo passo é a definição do valor do tipo primitivo *int*. Quando se faz necessária a criação de um tipo primitivo a única opção disponível ao usuário é um campo onde esse deve informar o valor do tipo primitivo (Figura 5.8). No exemplo o usuário escolheu o valor 51.



**Figura 5.3:** Tela de exibição das interfaces implementadas pelas classes analisada e padrão.

Com todas as informações necessárias para a criação do parâmetro, o método `add(java.lang.Object)` é então executado. Após a execução de um método é exibido ao usuário um relatório com as informações recolhidas dessa execução (Figura 5.9). A primeira informação é a assinatura do método. Também é exibido o tempo de execução de cada método, nesse exemplo pode-se notar que o tempo de execução do método da classe analisada foi maior que o da classe padrão. Como o método executado retorna `void` e não houve nenhuma exceção lançada, os valores dos campos 'Retorno' e 'Exceção' são `null`. As outras informações recolhidas são o estado da estrutura antes e depois da execução do método. Essa informação é adquirida através do método `toString()` implementado por cada estrutura.





**Figura 5.4:** Tela de exibição dos construtores.

Para melhor ilustrar esse exemplo será realizado o teste dos outros dois métodos definidos na interface. O próximo será o método *getSize()* que retorna o tamanho da estrutura. Como esse método não possui parâmetros ele é executado logo após a sua escolha e o seu relatório é exibido (Figura 5.10). Neste relatório pode-se notar que o valor do campo 'Retorno' agora possui um valor, 1, o tamanho da estrutura. Como no relatório anterior, nenhuma informação recolhida da execução desse método evidência um mau funcionamento das estruturas de dados.

O último método executado nesse exemplo é o *get()*. Como esse método também não possui parâmetros ele é executado logo após a sua escolha e o seu relatório é exibido (Figura 5.11). No relatório deste método pode-se notar uma disparidade entre as

**Escolha o método a ser executado:**

---

`public abstract void teste.PilhaInterface.add(java.lang.Object)`  
 `public abstract java.lang.Object teste.PilhaInterface.get() throws teste.PilhaVaziaException`  
 `public abstract int teste.PilhaInterface.getSize()`

---

**Figura 5.5:** Tela de exibição dos métodos.

informação colhidas da execução dos métodos das duas classes. Enquanto a execução do método da classe padrão foi finalizada normalmente, a execução da classe avaliada lançou uma exceção do tipo *java.lang.NullPointerException*, o que evidencia um provável erro de programação no código da classe analisada.

## Definição de Parâmetros

Método: `public abstract void teste.PilhaInterface.add(java.lang.Object)`

---

Objeto `java.lang.Object`. Escolha um construtor:

`public java.lang.Object()`

Usar subclasse:

null

---

**Figura 5.6:** Tela de criação do parâmetro `java.lang.Object`.

## Definição de Parâmetros

**Método:** `public abstract void teste.PilhaInterface.add(java.lang.Object)`

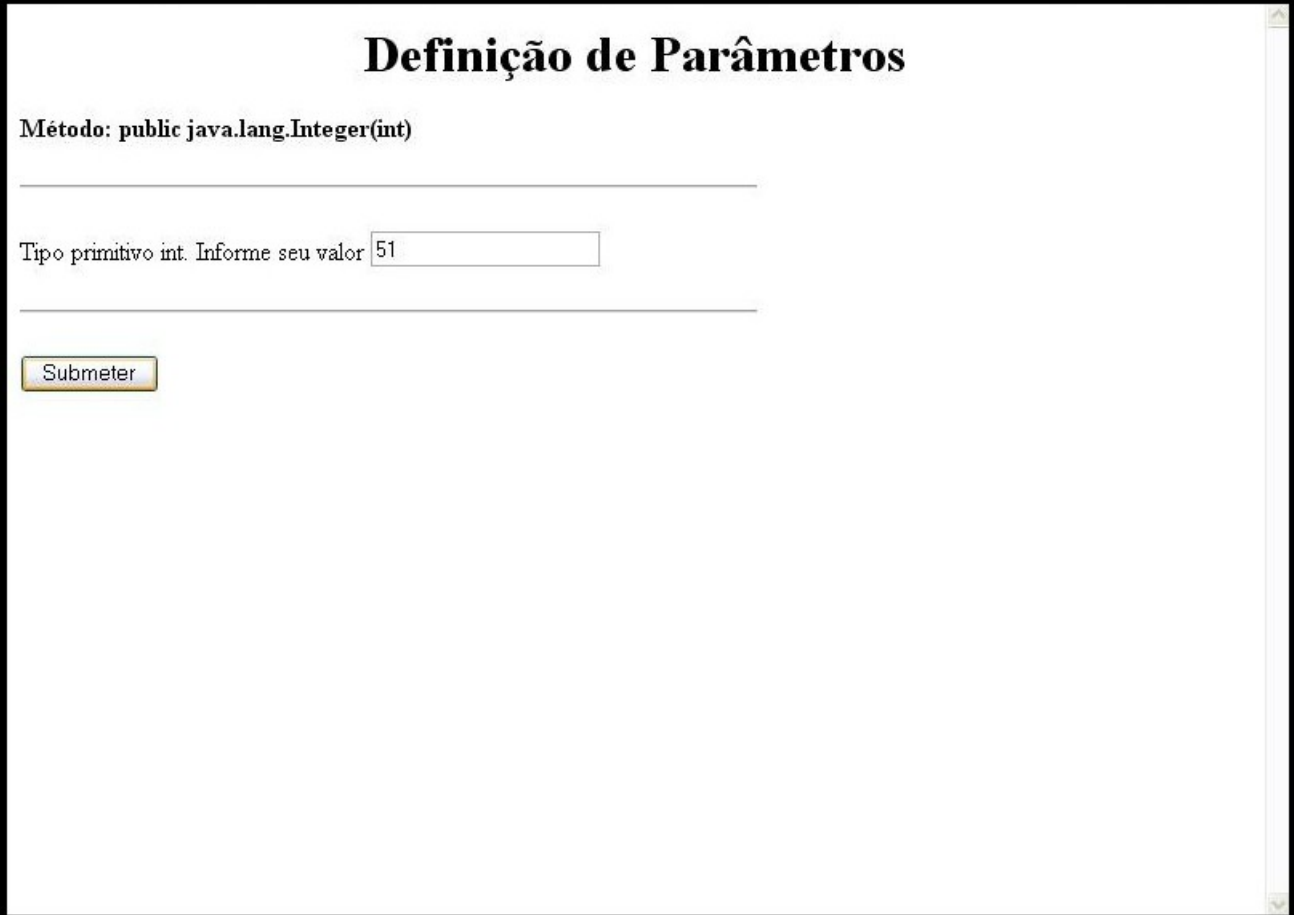
---

Objeto `java.lang.Integer`. Escolha um construtor:

- `public java.lang.Integer(int)`
- `public java.lang.Integer(java.lang.String) throws java.lang.NumberFormatException`
- Usar subclasse:
- null

---

**Figura 5.7:** Tela de criação do parâmetro `java.lang.Integer`.



**Definição de Parâmetros**

Método: `public java.lang.Integer(int)`

---

Tipo primitivo int. Informe seu valor

---

**Figura 5.8:** Tela de criação do tipo primitivo *int*.

### Relatório Parcial

---

	Resultado Da Classe Padrão	Resultado Da Classe Avaliada
<b>Método:</b>	public void add([java.lang.Integer]:51)	public void add([java.lang.Integer]:51)
<b>Tempo:</b>	1 ms	47 ms
<b>Retorno:</b>	null	null
<b>Exceção:</b>	null	null
<b>Objeto pré-execução:</b>	()	()
<b>Objeto pós-execução:</b>	(51)	(51)

---

[Executar outro método](#)  
[Ver relatório completo](#)

**Figura 5.9:** Tela de relatório do método *add(java.lang.Object)*.

**Relatório Parcial**

---

	Resultado Da Classe Padrão	Resultado Da Classe Avaliada
<b>Método:</b>	public int getSize()	public int getSize()
<b>Tempo:</b>	63 ms	31 ms
<b>Retorno:</b>	1	1
<b>Exceção:</b>	null	null
<b>Objeto pré-execução:</b>	(51)	(51)
<b>Objeto pós-execução:</b>	(51)	(51)

---

[Executar outro método](#)  
[Ver relatório completo](#)

**Figura 5.10:** Tela de relatório do método `getSize()`.

### Relatório Parcial

---

	Resultado Da Classe Padrão	Resultado Da Classe Avaliada
<b>Método:</b>	public java.lang.Object get()	public java.lang.Object get()
<b>Tempo:</b>	62 ms	31 ms
<b>Retorno:</b>	51	null
<b>Exceção:</b>	null	java.lang.NullPointerException
<b>Objeto pré-execução:</b>	(51)	(51)
<b>Objeto pós-execução:</b>	()	<Erro no método toString()> java.lang.NullPointerException

---

[Executar outro método](#)  
[Ver relatório completo](#)

**Figura 5.11:** Tela de relatório do método *get()*.



# Capítulo 6

## Considerações Finais

### 6.1 Trabalhos Futuros

Como possibilidades de trabalhos futuros pode-se citar:

**Segurança:** Realização de um estudo sobre uma maneira de garantir a segurança do sistema, visto que são executados métodos de classes sem nenhuma distinção, podendo ser um método malicioso ou não.

**Melhor métrica de tempo:** Realização de um estudo para garantir uma maneira mais precisa de medição do tempo utilizado na execução de um método. Atualmente, devido à utilização de uma *thread* para execução de método, o tempo calculado não é precisamente o tempo utilizado pelo método, pois essa *thread* ‘disputa’ tempo de processamento com outros fluxos de código. E a maneira pela qual o tempo de processamento é dividido entre os processos depende do sistema operacional.

**Automação dos testes:** Atualmente cada método a ser executado deve ser escolhido manualmente, um a um, assim como os parâmetros necessários para a sua execução. Uma forma de automatizar o processo poderia ser a utilização de um arquivo que definisse as informações necessárias — a utilização de XML<sup>1</sup> é uma possível solução.

---

<sup>1</sup>*Extensible Markup Language*. Para mais informações consulte <http://www.w3.org/XML/>.

## 6.2 Conclusão

A área de desenvolvimento de aplicativos para *Web* implementadas em *Java* é um assunto relativamente novo. Novas tecnologias foram e estão sendo lançadas em um período de tempo bastante curto, o que torna essa área ainda mais interessante pois propicia um campo amplo para novos estudos.

A realização do presente trabalho proporcionou ao autor a sua inclusão nesta área até então por ele desconhecida e não antes abordada no curso de Bacharelado em Ciências da Computação da Universidade Federal de Santa Catarina, através do desenvolvimento de programas aplicativos e do estudo de suas fundamentações teóricas.

Foi de grande valor também a possibilidade de um estudo mais aprofundado de tecnologias já conhecidas pelo autor. Exemplo disso são certas particularidades da linguagem *Java*, tais como a utilização de *class loader*, *Java Servlets*, reflexão, assim como a utilização de *frameworks* como o Struts e de ambientes de desenvolvimento e outras tecnologias utilizadas em atividades periféricas, como por exemplo, a utilização de  $\LaTeX$  na concepção deste relatório.

Para finalizar, é importante ressaltar que a aplicação desenvolvida nesse trabalho vem auxiliar a preencher uma lacuna que existe na área de desenvolvimento de aplicação *Verificadoras* (como citado na seção 1.1.2, página 4). Definitivamente essa aplicação não alcançou seu estado final de amadurecimento, podendo ser expandida com novas funcionalidades, ou então ter aprimoradas a suas funcionalidades já implementadas.

# Apêndice A

## Protocolo HTTP

O Protocolo HTTP — Hypertext Transfer Protocol — é, em uma visão geral, o protocolo utilizado para a comunicação entre o browser (entidade cliente) e o servidor web (entidade servidor). É um protocolo projetado para ser robusto e tolerante a falhas, visando maior *throughput* da conexão. Mas na verdade, o HTTP é muito mais do que isso. Segundo a própria W3C [10]:

*“The Hypertext Transfer Protocol (HTTP) is an application-level protocol for distributed, collaborative, hypermedia information systems. It is a generic, stateless, protocol which can be used for many tasks beyond its use for hypertext, such as name servers and distributed object management systems, through extension of its request methods, error codes and headers”*

W3C

### A.1 Introdução

A função do HTTP é especificar como o cliente e o servidor devem ‘conversar’. Entretanto, é importante observar que esse protocolo só define o que deve ser dito e não como deve ser dito. Para isto é utilizado o protocolo TCP/IP. Uma conversa no HTTP dura apenas uma transação, ou seja, o cliente pede conexão ao servidor fazendo a

sua requisição, o servidor responde com a negação ou com a resposta da requisição e a conexão é terminada. Caso o cliente queira fazer uma nova requisição ao provedor, deve estabelecer a conexão novamente.

## A.2 Funcionamento

Inicialmente o usuário digita o URL<sup>1</sup> desejado. O URL possui o formato básico “protocolo://servidor/arquivo”, no qual *protocolo* especifica o protocolo a ser utilizado na comunicação — nesse caso o HTTP; *servidor* especifica o endereço do servidor e *arquivo*, o arquivo a ser aberto no servidor. Após a confirmação do URL pelo usuário o programa cliente (geralmente um browser) envia uma requisição ao servidor. Essa requisição possui basicamente o seguinte formato:

```
<METODO> <ARQUIVO> HTTP/<VERSAO>  
<nomedocampo1>: <valordocampo1>  
<nomedocampo2>: <valordocampo2>  
  
<corpodarequisicao, se houver>
```

O campo `METODO` informa o método de requisição utilizado. Geralmente utiliza-se o GET, mas existem outros, como pode ser observado na Tabela A.1. O campo `ARQUIVO` define o arquivo a ser buscado no servidor, como por exemplo, “index.html”. O campo `VERSAO` informa a versão do protocolo HTTP a ser utilizada. Nas linhas seguintes são especificados os campos de informações e seus respectivos valores. A linha em branco define o fim do cabeçalho. As informações seguintes são opcionais, dependendo do método utilizado. É comum nesse espaço serem postas informações mandadas pelo cliente ao servidor. O método utilizado nesse caso é, geralmente, o método POST.

---

<sup>1</sup>URL: *Uniform Resource Locator*

GET	Requisição do cliente para ver um documento.
POST	Envio de dados do cliente para o servidor.
HEAD	Requisição para somente o cabeçalho do documento.
PUT	Requisição para gravar um documento no URL.
DELETE	Requisição para remover um documento do URL.

**Tabela A.1:** Métodos da requisição

Após a requisição ser enviada, o servidor deve responder com as informações solicitadas pelo cliente. A resposta do servidor segue o seguinte formato:

```
<CODIGO> <TEXTO> HTTP/<VERSAO>
<nomedocampo1>: <valordocampo1>
<nomedocampo2>: <valordocampo2>

<corpodaresposta>
```

O campo `VERSAO` informa a versão do protocolo HTTP utilizado e deve ser a mesma enviada na requisição. O campo `CODIGO` informa o código da transferência e possui inúmeros valores e significados, como especificado na Tabela A.2. O campo `TEXTO` é a representação textual do campo `CODIGO`. Quando a resposta é completada com sucesso o valor desse campo é OK. A linha em branco determina o fim do cabeçalho. A partir daí inicia-se o documento em si.

100 a 199	Informação do cliente sobre o servidor.
200 a 299	Resposta bem sucedida.
300 a 399	Página ou servidor redirecionado.
400 a 499	Página, servidor, <i>host</i> não encontrado ou falha na requisição do cliente.
500 a 599	Erro interno do servidor.

**Tabela A.2:** Valores de retorno do servidor no campo `CODIGO`

A seguir, é apresentado um exemplo de resposta a partir de uma requisição ao URL “<http://www.inf.ufsc.br/>”:

```
200 OK HTTP/1.0 server: Apache/1.3.19 (Unix) PHP/4.0.4pl1
content-type: text/html accept-ranges: bytes date: Thu, 12 Dec
2002 20:51:08 GMT connection: close etag: "1002e-13f8-3dc00668"
content-length: 5112 last-modified: Wed, 30 Oct 2002 16:18:48 GMT
```

```
<html> <head> <meta http-equiv="Content-Type" content="text/html;
charset=iso-8859-1"> <meta name="GENERATOR" content="Microsoft
FrontPage Express 2.0"> <title>Departamento de Informatica e
Estatistica</title>
```

...

# Apêndice B

## Class Loader

A grande maioria dos estudantes de computação ou profissionais da área tem conhecimento de que a linguagem *Java* é uma linguagem interpretada. Ou seja, quando um código *Java* é compilado, não é gerado código executável de máquina, mas sim, um código intermediário. No caso de *Java*, esse código intermediário é chamado de *bytecode*. Sendo assim, quando o usuário executa uma classe *Java* o *bytecode* desta classe é carregado para a Máquina Virtual, que interpreta o código e gera resultados reais. Entretanto, muita gente não sabe como esse processo é realizado e nem quais são as entidades relacionadas a ele.

*Java* possui um tipo especial de objeto que é chamado *class loader*, responsável por carregar as classes *Java*, ou então um *array* de *bytes* que a represente, para a Máquina Virtual *Java*. Essencialmente, existem dois tipos de *class loaders*: o *bootstrap class loader* e o *user defined class loader*.

O *bootstrap class loader* ou *primordial class loader* pode ser considerado como o *class loader* base em *Java*. É único, ou seja, só existe uma instância dele e faz parte da própria implementação da JVM. É implementado usando métodos nativos. Esse *class loader* é o responsável por carregar as classes básicas necessárias para o funcionamento da JVM, incluindo as classes da API *Java*.

Já o *user defined class loader* é resultado de uma das características mais interessantes do *Java*: a possibilidade de o usuário definir seus próprios *class loaders*. Os *class loaders* definidos pelo usuário funcionam como qualquer outro objeto escrito em *Java*, compilados em *bytecode* e carregados para a Máquina Virtual. A única coisa que o usuário deve fazer para criar seu *class loader* é estender a classe abstrata *java.lang.ClassLoader* redefinindo seus

métodos, se necessário ou desejado.

A possibilidade de implementar um *class loader* dá ao usuário grande flexibilidade para escolher como suas classes serão carregadas para a JVM. Além disso, há uma particularidade na arquitetura do *class loader* que torna a criação de várias instâncias um mecanismo muito poderoso, pois para cada *class loader* a JVM cria um novo *name-space*. *Name-space* é um conjunto único de classes carregadas por um *class loader* específico e assemelha-se a um ambiente seguro, criado para cada *class loader*, pois classes carregadas em um *name-space* não “enxergam” classes fora dele, assim como as classes fora dele não conseguem enxergá-lo. Por exemplo, imagine uma aplicação que carregue classes através da internet de diversas fontes diferentes. Não seria tão incomum se duas classes, uma de cada fonte, possuíssem nomes iguais. Se cada classe for carregada em *name-spaces* separados não haverá problema algum, pois seria como se elas estivessem rodando em ambientes totalmente separados sem nenhuma ligação entre elas. Outro ponto importante é que, quando uma classe carregada por um *class loader* referencia alguma outra classe ainda não carregada, a requisição para carregar a segunda classe é enviada ao *class loader* que carregou a primeira classe, preservando assim a consistência do *name-space*. A capacidade de criação de vários *name-spaces* dentro de uma mesma aplicação possibilita a utilização de classes de diversas fontes sem que elas entrem em conflito umas com as outras.

É esta característica da arquitetura da Máquina Virtual Java a utilizada neste trabalho, pois para sua realização é necessário que várias classes sejam carregadas de fontes diferentes pela internet através do navegador.



# Referências Bibliográficas

- [1] R. Baker, M. Boilen, M. T. Goodrich, R. Tamassia, and B. A. Stibel. **Testers and Visualizers for Teaching Data Structures**, Proc. ACM Symposium on Computer Science Education (SIGCSE), 1999.
- [2] COLASO, Vikrant, et al. **Learning and Retention in Data Structures: A Comparison of Visualization, Text, and Combined Methods**.
- [3] **AVL tree applet**. Disponível em: <http://www.seanet.com/users/arsen/avltree.html>. Acesso em: 11 de novembro de 2002.
- [4] **www.binarytreesome.com**. Disponível em: <http://www2.binarytreesome.com:3000/applets/latest/applet.html>. Acesso em: 11 de novembro de 2002.
- [5] DUANE J. JARC, **Interactive Data Structure Visualizations**. Disponível em: <http://www.student.seas.gwu.edu/~idsv/idsv.html>. Acesso em: 11 de novembro de 2002.
- [6] CHEN, Tao; SOBH, Tarek. **A Tool for Data Structure Visualization and User-defined Algorithm Animation**.
- [7] RÖBLING, Guido; SCHÜLER, Markus; FREISLEBEN, Bernd: **The ANIMAL Algorithm Animation Tool**.
- [8] GARCIA, Islene C.; REZENDE, Pedro J.; CALHEIROS, Felipe C.: **Astral**: Um Ambiente para Ensino de Estruturas de Dados através de Animações de Algoritmos.
- [9] REEK, Kenneth A. The TRY System - or - How to Avoid Testing Student Programs.
- [10] W3C, **Hypertext Transfer Protocol Overview**. Disponível em: <http://www.w3.org/Protocols/>. Acessado em: 12 de novembro de 2002.

- [11] GARSHOL, Lars M. **How the web works: HTTP and CGI explained.** Disponível em: <http://www.garshol.priv.no/download/text/http-tut.html>. Acessado em: 12 de novembro de 2002.
- [12] Sun Microsystems. **JavaServer Pages(TM) Technology.** Disponível em: <http://java.sun.com/products/jsp/>. Acessado em: 8 de fevereiro de 2003.
- [13] The Apache Jakarta Project. **The Apache Struts Web Application Framework.** Disponível em: <http://jakarta.apache.org/struts/>. Acessado em 9 de dezembro de 2003.
- [14] The Apache Software Foundation. **Welcome! - The Apache Software Foundation.** Disponível em: <http://www.apache.org/>. Acessado em 9 de dezembro de 2003.
- [15] The Apache Jakarta Project. **The Jakarta Site - The Jakarta Project – Java Related Products.** Disponível em: <http://jakarta.apache.org/>. Acessado em 9 de dezembro de 2003.
- [16] eNode, Inc. **Model-View-Controller Pattern.** Disponível em: <http://www.enode.com/x/markup/tutorial/mvc.html>. Acessado em: 9 de fevereiro de 2003.
- [17] ECKEL, Bruce. **Thinking in Java.** Prentice Hall; 3rd edition; Dezembro 2002.
- [18] DEITEL, Harvey M.; DEITEL, Paul J.; DEITEL, Harvey M.; **Advanced Java 2 Platform: How to Program** Prentice Hall; Setembro 2001.
- [19] GOODWILL, James. **Mastering Jakarta Struts.** John Wiley & Sons, Inc.; Novembro 2002.
- [20] CAVANESS, Chuck. **Programing Jakarta Struts.** O'Reilly & Associates; 1st edition. Novembro 2002.
- [21] HUSTED, Ted; et al. **Struts in Action.** Manning Publications Company. November 2002.
- [22] OAKS, Scott. **Java Security.** O'Reilly & Associates; 1st edition. Maio 1998.
- [23] KAMINSKY, Alan. **Java Class Loading and Class Loaders – Lecture Notes.** Disponível em <http://www.cs.rit.edu/~ark/lectures/cl/>. Acessado em: 20 de novembro de 2003.
- [24] VENNERS, Bill. **Security and the Class Loader Architecture.** Disponível em <http://www.artima.com/underthehood/classloaders.html>. Acessado em 21 de novembro de 2003.

- [25] MCGRAW, Gary; FELTEN, Edward. **The Class Loader Architecture (Ch. 2, Sec. 7) [Securing Java]**. Disponível em <http://www.securingsjava.com/chapter-two/chapter-two-7.html>. Acessado em 20 de novembro de 2003.
- [26] Sun Microsystems. **Java BluePrints - J2EE Patterns**. Disponível em <http://java.sun.com/blueprints/patterns/MVC-detailed.html>. Acessado em: 13 de dezembro de 2003.
- [27] Sun Microsystems. **The Reflection API**. Disponível em <http://java.sun.com/docs/books/tutorial/reflect/index.html>. Acessado em: 10 de dezembro de 2003.
- [28] Sun Microsystems. **Threads: Doing Two or More Tasks At Once**. Disponível em <http://java.sun.com/docs/books/tutorial/essential/threads/index.html>. Acessado em: 10 de dezembro de 2003.
- [29] KOMOSINSKI, Leandro J. **Aplicação Cliente-Servidor via Web Usando Java**. Florianópolis, 2002.