ALLAN JONES BATISTA DE CASTRO

DANIEL ASSIS ALFENAS

DANILO PICAGLI SHIBATA

HENRIQUE TATSUYUKI SOEJIMA

SINTETIZADOR TEXTO-VOZ COM AUTÔMATOS ADAPTATIVOS

Projeto de Formatura apresentado à disciplina PCS 2050 – Laboratório de Projeto de Formatura II, da Escola Politécnica da Universidade de São Paulo.

São Paulo,

2004

ALLAN JONES BATISTA DE CASTRO

DANIEL ASSIS ALFENAS

DANILO PICAGLI SHIBATA

SINTETIZADOR TEXTO-VOZ COM AUTÔMATOS ADAPTATIVOS

Projeto de Formatura apresentado à disciplina PCS 2050 – Laboratório de Projeto de Formatura II, da Escola Politécnica da Universidade de São Paulo.

Área de Concentração: Sistemas Digitais

Orientador: Prof. Dr. Ricardo Luis de Azevedo Rocha

São Paulo,

2004

AGRADECIMENTOS

Às nossas eternas namoradas, Patrícia, Carine e Amuni, por suportar nossa ausência e pelo incentivo dados, nos momentos em que mais precisávamos.

Aos nossos pais, pela paciência de suportar filhos que nunca estão em casa para lhe dar um beijo.

Aos nossos amigos que nos acompanham sempre, mesmo que muitas vezes nós não os tenhamos acompanhado.

Ao nosso orientador e amigo Ricardo Rocha, pelo acompanhamento dado, pelos inúmeros erros de português corrigidos e pelo bom humor sempre presente.

A todos esses e muitos outros que sempre nos apoiaram neste difícil caminho da graduação, o nosso mais profundo agradecimento.

RESUMO

Neste trabalho é apresentado o projeto do software *Voicer*, um sintetizador texto-voz baseado em técnicas adaptativas e no sofware *Adaptools*. O *Adaptools* é um programa, desenvolvido na linguagem de programação Java, que descreve autômatos adaptativos, permitindo a visualização e configuração através de uma interface gráfica. O objetivo essencial deste projeto foi desenvolver um autômato capaz de realizar a síntese de voz a partir de textos escritos na língua portuguesa, do modo em que ela é falada no Brasil. Foi trabalhado tanto o mecanismo reconhecedor de texto quanto o gerador de som a partir da transcrição fonética. Neste documento são exibidos os aspectos teóricos relacionados a autômatos adaptativos e síntese de voz, assim como os aspectos de implementação e os resultados obtidos. É mostrado também um exemplo de aplicação distribuída baseada no *Voicer*.

ABSTRACT

This work describes *Voicer*, a speech synthesizer based on adaptive automata using Adaptools, a Java application that describes adaptive automata and simulates its execution, using a graphic interface. The main goal of the project was the development of an automaton which would be used with a graphic interface to read texts and map the words to its sounds. This document describes adaptive automata and voice synthesizing theory alongside implementation and results.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

1 INTRODUÇÃO

A demanda por sistemas de síntese de voz tem crescido nos últimos tempos. Estes sistemas, basicamente, tentam reproduzir a linguagem natural através de sinais de áudio. O Laboratório de Linguagens e Técnicas Adaptativas (LTA) do PCS iniciou um projeto de síntese digital de voz como aplicação da ferramenta *Adaptools* (PISTORI, 2003). Sendo o assunto de interesse dos integrantes deste grupo de projeto de formatura e a continuidade interessante para o LTA, iniciou-se o projeto Sintetizador de Voz com Autômatos Adaptativos.

1.1 Objetivos

O primeiro objetivo do projeto Sintetizador de Voz com Autômatos Adaptativos foi de desenvolver um sistema de código aberto para conversão de texto digital em sinais de áudio que reproduzam a fala, para a língua portuguesa falada no Brasil. Este sistema foi chamado inicialmente de *Vocalizador Digital de Textos*, e, posteriormente, alterado para *Voicer*.

O segundo objetivo do projeto foi complementar a ferramenta *Adaptools*, um aplicativo para PC que descreve e simula autômatos adaptativos, permitindo a configuração e visualização do seu funcionamento através de uma interface gráfica. Com esta ferramenta foi feita, por exemplo, a implementação de um mecanismo adaptativo de recuperação de erros para autômatos de estados finitos que pode ser facilmente estendida para autômatos de pilha estruturados (PISTORI, 2003). Também foi criado um autômato adaptativo que associa textos a sons (fonemas), com muitas restrições (poucas regras, universo pequeno de fonemas e fala truncada). O nosso objetivo foi, então, desenvolver este autômato do *Adaptools* para conversão de texto em voz, acrescentando as regras ortográficas e fonéticas ainda não inclusas, gravando mais fonemas e aprimorando a entonação e as pausas naturais da fala humana.

Podemos citar como um terceiro objetivo mostrar a aplicabilidade das técnicas adaptativas na área de engenharia de computação com um exemplo real e útil, a tradução texto-voz. As técnicas adaptativas são, sem dúvida alguma, de grande

utilidade para resolver inúmeros problemas, mas que ainda foram pouco utilizadas para implementação em soluções práticas.

É importante ressaltar que este projeto em nenhum momento teve como objetivo reproduzir as diversas variantes (dialeto) da língua portuguesa falada no Brasil. O objetivo foi reproduzir a fala da comunidade brasileira, em um único conjunto de regras, embora o sintetizador represente melhor a variante falada na região da cidade de São Paulo.

1.2 Justificativa

1.2.1 Motivação e Aplicações

Este projeto tem bons motivos acadêmicos, sociais, pessoais e comerciais para existir.

Motivos acadêmicos, porque as tecnologias adaptativas vêm sendo usadas nas mais diversas áreas da engenharia de computação, como na modelagem de aplicações complexas através de dispositivos ISDL-Adp (CAMOLESI; NETO, 2004); na construção de compiladores; no processamento de linguagem natural (NETO; MORAES, 2002; MENEZES, NETO, 2002 apud PISTORI, 2003); e na área de robótica e visão computacional (JUNIOR; NETO; HIRAKAWA, 2000 apud PISTORI, 2003). Também podemos citar, entre tantos outros, um interessante software que cria composições musicais barrocas em tempo real, através de redes de Markov adaptativas (BASSETO; NETO, 1999). Mesmo na conversão de texto em voz a tecnologia adaptativa vem sendo utilizada, através do *Adaptools* (PISTORI, 2003; ZUFFO; PISTORI, 2004), o que fortemente motivou o grupo a iniciar o projeto.

Motivos sociais, porque permite a pessoas portadoras de problemas físicos ou, até mesmo, com ausência de voz comunicar-se com outras pessoas através de voz sintetizada a partir de texto digitado em teclados de computadores. Se combinado com softwares que transformem voz em texto, será possível criar aplicações digitais interativas para portadores de deficiência visual.

Pessoais, por ser um projeto completo e complexo, que requer grande dedicação à pesquisa e a estudos variados, relacionados à fonética, à natureza do som, às tecnologias disponíveis e às técnicas adaptativas, sem deixar de lado a engenharia de software.

Motivos comerciais, porque a fala é uma forma mais natural de comunicação, trazendo mais conforto e praticidade para usuários que poderiam, então, escutar os textos ao invés de lê-los. Exemplos de aplicações comerciais como "leitura de e-mails e documentos", para pessoas sem tempo de parar na frente do computador e ler; "leitura de textos para crianças", para crianças dormirem; "leitura de textos em um *chat* convencional", onde as pessoas poderiam ouvir e falar através da própria digitação dos textos.

A utilização de autômatos adaptativos justifica-se pelas dificuldades encontradas na tradução texto-voz. Uma mesma sílaba pode corresponder a fonemas diferentes, como a sílaba *xa* nas palavras *fixa* e *caixa*; também uma mesma palavra pode ter leitura diferente, como *seca* (substantivo) e *seca* (conjugação do verbo secar). Estes problemas estão intimamente ligados com a dependência de contexto, que não pode ser resolvida utilizando-se autômatos de estado finitos ou autômatos de pilha, mas pode ser resolvido utilizando-se autômatos adaptativos.

1.2.2 Possibilidades de Evolução

Existem várias possibilidades para a evolução deste projeto. Podemos citar:

- Reproduzir diferentes dialetos da língua portuguesa, através da regravação de alguns fonemas e provavelmente alterando algumas regras do autômato. A razão destas possíveis alterações são melhor explicadas nos capítulos seguintes;
- Criar um módulo para conversão da voz em texto, também utilizando técnicas adaptativas, e com metodologia para reconhecimento de padrão de voz;
- Ampliação do pacote JSAPI através da inclusão de adaptatividade na linguagem de marcação JSML;
- Adaptação para outras línguas.

1.2.3 Outros Sintetizadores

Existem, atualmente, muitas soluções para tradução texto-voz. Para a língua portuguesa falada no Brasil, no entanto, não são muitas as opções. Alguns exemplos serão exibidos aqui, embora sem profundidade.

Uma das soluções é o *OddCast* (www.oddcast.com), que é capaz de sintetizar muito bem diversas línguas, como o inglês e o português. Todo o conteúdo acessível na *home page* do projeto pode ser ouvido através do sintetizador. Não temos informações relativas à implementação do *OddCast*.

A necessidade de tornar mais natural a interação do homem com o computador motivou o surgimento de um novo paradigma, denominado computação ubíqua (WEISSER, 1991). A computação ubíqua propõe o desenvolvimento de aplicações que não sejam centradas no conjunto *mouse*-teclado-*display*, de modo que as pessoas não precisem se adaptar aos métodos de entrada e saída dos computadores. Neste sentido, tanto a síntese quanto o reconhecimento de voz são muito úteis. O Departamento de Ciências de Computação e Estatística do ICMC-USP tem investigado os problemas de computação ubíqua através do projeto InCA-SERVE e tem adotado o pacote *Java Speech API* (JSAPI) junto ao pacote comercial da IBM denominado *Via Voice* para síntese e reconhecimento de voz (PIMENTEL; INÁCIO, 2004). O pacote da IBM foi aprovado, mas a gramática, as palavras e as sentenças tiveram de ser definidas para a língua inglesa, porque a versão do *Via Voice* usada estava em inglês.

O JSAPI utiliza uma linguagem de marcação, que segue o padrão XML, denominada JSML (*Java Speech Markup Language*) para a síntese de voz, permitindo alterar propriedades como tom, ritmo e volume. Também utiliza um arquivo-texto em formato especial para definição de gramáticas (JSGF, *Java Speech Grammar Format*) (SUN MICROSYSTEMS, 1998). Este pacote não implementa a síntese, mas apenas define uma especificação. Empresas como IBM (*Via Voice*), Microsoft (*Agent*) e AT&T (*NaturalVoices*) têm implementações próprias para o sintetizador. Tanto as implementações da JSAPI citadas quanto o *Oddcast* são projetos privados, impossibilitando o acesso ao código, diferentemente do nosso projeto.

Caseiro; Trancoso; Oliveira; Viana (2002) descrevem um conversor texto-voz baseado em transdutores de estado finito (WFST, *Weighted Finite State Transducers*). Justificam o uso de WFST pela flexibilidade, elegância e eficiência da integração de múltiplas fontes de informação, como as providas por outros módulos de análise de texto. Eles comparam soluções baseadas em regras, *data-driven* e soluções híbridas, assim como outras soluções como as baseadas em redes neurais. A solução *rule-based*, batizada de DIXI, foi baseada no corpus fundamental da língua portuguesa, que foi retirado dos primeiros 10.000 parágrafos do BD-, sendo que 75% desse corpus foi utilizado para treinamento.

Comparativamente, a maior diferença do sintetizador baseado em autômatos adaptativos é a força da adaptatividade quanto a gramáticas dependentes de contexto, caso observado na conversão texto-voz e que nenhuma das soluções apresentadas parece levar em consideração.

1.3 Organização do Texto desta Monografia

Todos os capítulos desta monografia estão distribuídos, com exceção deste primeiro capítulo, em três grandes partes: *Aspectos Conceituais*, *Voicer: A Ferramenta* e *Contribuições e Conclusão*. A primeira parte é formada por três capítulos. O capítulo 2 apresenta o modelo adaptativo e define seu uso no projeto; o capítulo 3 apresenta o *Adaptools*, ferramenta utilizada como base para o *Voicer*; e o capítulo 4 apresenta as questões de fonética e fonologia e algumas idéias de síntese texto-voz existentes.

A segunda parte enfoca o desenvolvimento do projeto Sintetizador Texto-Voz com autômatos adaptativos. O capítulo 5 apresenta o autômato desenvolvido no projeto do Voicer e um detalhamento do modelo utilizado na implementação, ilustrando as mudanças planejadas no autômato. O capítulo 6 mostra a metodologia aplicada ao projeto, como as responsabilidades atribuídas aos membros de equipe, relata o projeto e a implementação real, assim como compara o custo real e o planejado e mostra e justifica diferenças entre o cronograma inicial e a execução das tarefas. O capítulo 7 apresenta a especificação do projeto e, também, as mudanças em relação ao *Adaptools* para atender as necessidades especificadas. Toda a tecnologia e ferramentas utilizadas serão expostas neste capítulo.

A última parte é composta por dois pequenos capítulos. O capítulo 8 detalha quais foram os resultados, as contribuições e possíveis aplicações do projeto. O capítulo 9 contém a conclusão e comentários gerais sobre a experiência com este projeto, mostrando as perspectivas de continuidade com o posicionamento de cada membro do grupo.

2 TECNOLOGIAS ADAPTATIVAS

Neste capítulo são discutidos, de forma resumida, os princípios da tecnologia e dos autômatos adaptativos.

O conceito de dispositivo adaptativo guiado por regras generaliza a formalização de uma série de dispositivos formais, como, por exemplo, autômatos finitos, autômatos de pilha e máquinas de Turing (PISTORI, 2003). Todos estes dispositivos são baseados em um conjunto fixo e finito de regras. As regras mapeiam cada possível configuração do dispositivo em uma nova configuração, eventualmente estimulado por alguma entrada. Quando a transição gera alguma saída, o dispositivo em questão é chamado de transdutor. Um dispositivo guiado por regras inicia em uma determinada configuração, e segue aplicando regras sucessivas até que não haja mais estímulos de entrada ou até que se atinja uma configuração à qual nenhuma regra possa ser aplicada. Com base na configuração atingida, o dispositivo aceita ou rejeita a cadeia de entrada.

Um dispositivo adaptativo contém um conjunto de regras que varia conforme os estímulos de entrada. Porém estas variações são determinadas por um outro conjunto de regras, chamado de *ações adaptativas*, que agem sobre o conjunto de regras original, modificando-o através de adições e remoções de regras; esse conjunto é chamado de *camada adaptativa*. À camada que contém as regras originais é dado o nome de *camada subjacente.* Uma definição mais formal de um dispositivo adaptativo pode ser encontrada em (NETO, 1993) e (PISTORI, 2003). É possível adicionar uma camada adaptativa a qualquer dispositivo, como por exemplo, *statecharts* (NETO; JUNIOR; SANTOS, 1998), redes de Markov (BASSETO; NETO, 1999) e autômatos de pilha estruturados. Neste último caso, o dispositivo é chamado de autômato adaptativo.

2.1 Autômatos Adaptativos

Os autômatos adaptativos se destacam como uma interessante alternativa para as máquinas de Turing (NETO, 2000 apud PISTORI, 2003), tendo o mesmo poder de

expressão que elas (ROCHA, NETO, 2000), mas tendo a desejável característica de poderem ser representados como uma simples extensão dos autômatos de pilha estruturados. Sendo assim, enquanto um autômato adaptativo é capaz de aceitar cadeias geradas por gramáticas dependentes de contexto, enquanto um autômato de pilha estruturado está limitado a gramáticas livres de contexto (NETO, 1987).

2.1.1 Funções Adaptativas

As ações adaptativas do autômato são implementadas através de funções adaptativas. Uma *função adaptativa* determina quais modificações (ações adaptativas elementares) serão realizadas na camada subjacente do dispositivo.

Uma função adaptativa pode ser descrita como uma ênupla da forma (**F**, **P**, **V**, **G**, **C**, **R**, **I**, **A**, **B**) onde:

- **F** é o nome da função adaptativa;
- **P** é uma lista ordenada dos parâmetros formais (ρ_1, ρ_2, ρ_3,..., ρ_m). Estes valores são preservados durante toda a execução de **F**;
- **V** é a lista dos identificadores de variáveis (υ_1, υ_2, υ_3,..., υ_n). São valores desconhecidos no instante da chamada de **F** e que, uma vez preenchidos com o resultado de ações de consulta, não serão mais alterados durante toda a execução da função;
- **G** é a lista (g^*_1, g^*_2,..., g^*_p) de identificadores de geradores, variáveis especiais que são preenchidas com novos valores, ainda não utilizados pelo autômato, que permanecem durante toda a execução da função;
- **C** é a lista de ações elementares de consulta, utilizadas para preencher variáveis indefinidas;
- **R** é a lista de ações elementares de remoção;
- **I** é a lista de ações elementares de inserção;

- **A** é uma ação adaptativa inicial, opcional, que deve ser executada antes de **F**;
- **B** é uma ação adaptativa final, opcional, que deve ser executada depois de **F**.

Uma ação adaptativa é dada por um par ordenado (**F**, π) onde:

- **F** é o nome da função adaptativa a ser executada;
- π é a lista de argumentos (τ_1, τ_2, τ_3,..., τ_m) que correspondem posicionalmente à lista de parâmetros definidas por **P**.

São três os tipos das ações elementares:

- *de consulta* - permite a busca por padrões nas regras presentes na configuração corrente do autômato, que resulta em uma ou mais variáveis definidas com o resultado da busca. Se nenhuma regra satisfaz as condições de consulta, as variáveis permanecem inalteradas. É identificada pelo prefixo "?";
- *de remoção* - usa o mesmo mecanismo básico de consulta. Se todas as variáveis de consulta tenham sido definidas, a regra correspondente deve ser eliminada da camada subjacente. É identificada pelo prefixo "-";
- *de adição*.- Faz a inserção de regras na camada subjacente. Sempre ocorre após a execução de todas as ações elementares de consulta e de remoção, pois todas as variáveis devem ter sido previamente definidas. Caso contrário, a ação de adição não acontecerá. Ela pode conter nomes de geradores ao invés de nomes de variáveis. No momento da inserção o nome do gerador é substituído por um novo símbolo. É identificada pelo prefixo "+".

O exemplo de função adaptativa abaixo, descrito através das suas ações adaptativas elementares, foi retirado de (PISTORI, 2003).

? [(?x, ρ_1, ?y)] ? [(?y, ρ_2, ?z)] - [(?x, ρ_1, ?y)]

A primeira ação elementar, ? [(?x, %$_1$, ?y)] , é uma ação de consulta, que procura por todas as transições partindo de um estado x que vão para um estado y, consumindo o símbolo contido em ρ_1, primeiro parâmetro da lista de parâmetros π.

Ao fim desta primeira consulta, as variáveis x e y estarão definidas com os nomes dos respectivos estados se ao menos uma transição com tais características foi encontrada. A segunda ação elementar irá procurar por transições partindo do estado y (cujo valor foi definido na primeira consulta) para o estado z consumindo o símbolo contido no parâmetro ρ_2. Por fim, já tendo todas as variáveis definidas, a última ação elementar, de remoção, é unificada com as anteriores a fim de obter os valores já definidos para y e x; então todas as transições encontradas na primeira busca são removidas do autômato.

2.1.2 Notação de Máquina

Qualquer configuração de um autômato adaptativo, em um instante k, é definida por uma quádrupla $(E_k, \lambda_k, q_k, \omega_k)$ onde:

- E_k é a máquina de estados que representa o autômato naquele instante;
- λ_k é o conteúdo completo da pilha no autômato;
- q_k é o estado corrente do autômato;
- ω_k é a parte da cadeia de entrada que ainda não foi consumida pelo autômato.

Na configuração inicial t_0 do autômato adaptativo, o conteúdo λ_0 da pilha deve ser vazio e o estado corrente do autômato deve ser o estado inicial da máquina inicial E_0. Na configuração final t_f, o conteúdo λ_f da pilha deve atender o critério de aceitação da máquina final E_f, que usualmente é a pilha vazia; a cadeia de entrada ainda não consumida também deve estar vazia, isto é, $\omega_f = \varepsilon$.

A notação das produções em um autômato adaptativo é muito semelhante à de um autômato de pilha estruturado (por este ser o mecanismo subjacente). Para uma transição interna de máquina usa-se a seguinte notação:

$(\gamma, q_{ij}, \beta) : A, \rightarrow (\gamma, q_{ik}, \alpha) : B$

Onde:

- γ representa o conteúdo da pilha;
- β representa a situação da cadeia de entrada antes da aplicação da produção;

- α representa a situação da cadeia de entrada depois da aplicação da produção;
- q_{xy} representa um estado qualquer;
- **A** é o nome da função adaptativa a ser executada antes da aplicação da produção;
- **B** é o nome da função adaptativa executada após a aplicação da produção.

Alteração semelhante acontece nas transições de chamada e de retorno de sub-máquina, que é a adição **A** e **B**. A notação completa para autômatos de pilha estruturados pode ser encontrada em (NETO, 1987).

2.1.3 Notação Gráfica

A notação para as transições adaptativas utilizadas neste documento tem a seguinte forma:

λ, q [**A** (π_A) **.** **B** (π_B)]

Onde:

- λ é o símbolo consumido da cadeia de entrada. Se nenhum símbolo é consumido, $\lambda = \varepsilon$, ou, se q for omitido, λ também pode ser omitido;
- q é o estado empilhado. Se nenhum estado for empilhado, q é representado pelo símbolo nop, que pode ser omitido;
- **A** é a função adaptativa executada antes da aplicação da transição, como definido em 2.1.2;
- **B** é a função adaptativa executada após a aplicação da transição, como definido em 2.1.2;
- π_x é um conjunto de parâmetros passados para um função adaptativa qualquer

A figura 2.1 mostra um exemplo de autômato adaptativo que utiliza esta notação.

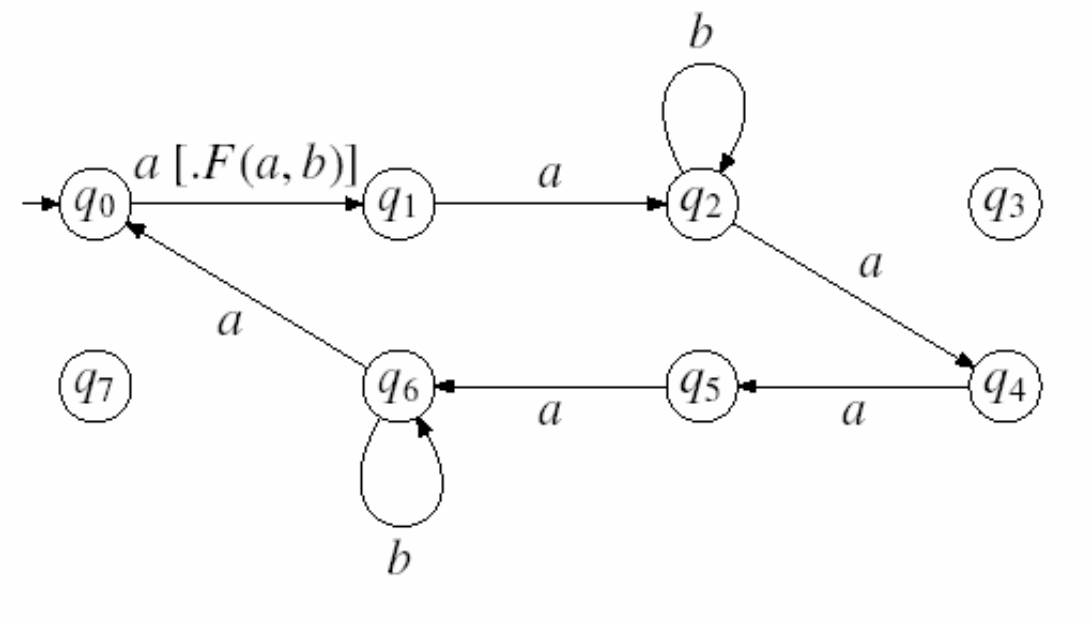

Figura 2.1 –Exemplo de autômato adaptativo que utiliza a notação exposta no tópico 2.1.3 (PISTORI,2003).

3 ADAPTOOLS

Esse capítulo tem como objetivo expor aspectos de arquitetura e implementação do software Adaptools que foi utilizado como base inicial para desenvolvimento e evolução do aplicativo voicer.

O Adaptools possui um "kernel" (núcleo) que atua como uma máquina virtual. Sua principal função é executar autômatos que podem ser autômatos de pilha estruturado ou autômatos adaptativos. O suporte do Adaptools a autômatos adaptativos é particular pois pequenas modificações no formalismo foram feitas com objetivo de estabelecer padrões na forma como transições internas, externas e ações adaptativas devem ser dispostas.

3.1 Autômatos Adaptativos

O software Adaptools visou padronizar a representação de autômatos adaptativos em formato de tabelas, onde cada coluna dessa tabela possui um significado:

- Cabeçalho (*head*): Nome da sub-máquina ou da função adaptativa. Se representar um nome da função adaptativa, esse nome será precedido por um símbolo que indicará o tipo da função adaptativa. Os símbolos podem ser ?, + ou - que significam respectivamente a ação adaptativa elementar de consulta, de adição e remoção.
- estado de origem (*origin*): Significa exatamente o que seu nome diz, o estado de origem da regra em questão.
- símbolo de entrada (*input*): Representa o símbolo de entrada da regra em questão.
- estado de destino (*dest*): Representa o estado que o autômato estará após ocorrida a transição.
- empilha (*push*): Indica o estado que será empilhado na pilha do autômato.
- símbolo de saída (*output*): O símbolo que será lançado na saída ao ocorrer

esta transição.

- ação adaptativa (*adaptive*): Representa a coluna referente a ação adaptativa elementar podendo ser ação de consulta, adição ou remoção como citado anteriormente.

Podemos dizer que as regras da tabela do Adaptools são divididas em três tipos básicos:

- Regra de transição interna: Uma transição qualquer onde o autômato que se encontra num determinado estado (origin) consome um símbolo (input) e transiciona para um determinado estado (dest) podendo ou não empilhar algum estado (push) ou lançar algum símbolo de saída (output).
- Regra de transição externa: São as chamadas de sub-máquina e retorno, onde o autômato faz uma chamada a outra máquina empilhando o estado de retorno. Os retornos de chamada de sub-máquinas são identificados pela palavra reservada *pop*.
- Regra de ação adaptativa elementar: Relacionada às ações adaptativas elementares. As ações adaptativas são representadas pelo formato *A.P,* onde *A* e *P* são opcionais e representam as ações adaptativas anterior e posterior respectivamente. Além disso, é opcional a passagem de parâmetros para as funções adaptativas. Esses parâmetros são passados dentro de parênteses e separados por vírgula. Logo, a notação das ações adaptativas pode ser algo do tipo *A.P(param1, param2, param3)*.

Alguns outros padrões foram estabelecidos com objetivo de conseguir representar toda estrutura na forma de tabela, dessa forma, não seria necessário criar outros tipos de estrutura para representação. Para isso foi criado um conjunto de palavras reservadas que possuem um significado e modo de uso no Adaptools. O conjunto de palavras reservadas é mostrado e explicado, logo abaixo:

- **nop:** Só é possível utilizá-lo nas colunas *push*, *output*, *adaptive* e indicará, respectivamente, a ausência de estado de retorno, do símbolo de saída ou de ação adaptativa
- **eps:** Palavra reservada para representação da cadeia vazia.

- **pop:** Representa uma transição de retorno de uma chamada de sub-máquinas. Logo, o próximo estado será aquele armazenado no topo da pilha e retornaremos para a máquina que efetuou a chamada.
- **fin:** Usado apenas para indicar que o estado de destino será um estado final. Essa palavra reservada é opcional e é usada somente na coluna *push*.
- **spc:** Palavra reservada que indica os caracteres especiais que indicam espaços em branco, como tabulações, saltos de linha e o próprio espaço em branco.
- **?sta:** Representa uma variável especial cujo valor é o valor do estado atual do autômato. Esta palavra reservada é utilizada somente na coluna *adaptive*.
- **?inp:** Representa uma variável especial cujo valor será o próximo símbolo a ser consumido (símbolo de entrada).
- **digit:** Palavra reservada utilizada para representar os dígitos de 0 até 9. Pode ser representada alternativamente com notação 0..9.
- **letter:** Palavra reservada utilizada para representar as letras do alfabeto, de *a* até *z* podendo ser maiúsculos ou minúsculos. Também podem ser representadas por *a..z* e *A..Z*.
- **special:** Palavra reservada para qualquer outro símbolo que não seja nem letra e nem dígito.
- **other:** Palavra reservada para símbolos que não podem ser consumidos no estado de origem da regra em questão. Geralmente essa palavra reservada é utilizada para tratamento de erro dentro de uma ação adaptativa.

3.2 Formato do arquivo de regras do Adaptools

O formato do arquivo de regras lido pelo Adaptools, que contém o chamado "código objeto", é basicamente um *dump* da tabela de regras em um arquivo-texto. As colunas da tabela são separadas por ponto-e-vírgula. Ao iniciar o Adaptools podemos selecionar o arquivo contendo o código objeto e assim carregá-lo para a tabela de

regras para ser utilizado.

3.3 Diferenças em Relação à Definição de Autômato Adaptativo Original

O formato baseado em tabela que o Adaptools utiliza provoca um efeito colateral na definição de autômatos. Autômatos que possuem não-determinismos serão executados sempre da mesma forma, sendo que a transição que se apresentar primeiro na tabela de regras será a transição a ser executada. Logo, se houver o não-determinismo, sempre a mesma regra será executada, enquanto as outras nunca serão acessadas.

3.4 Tratamento de Rotinas Semânticas

O Adaptools oferece um tratamento de rotinas semânticas que podem ser estendidas. As rotinas semânticas são introduzidas em um autômato adaptativo através da coluna de símbolo de saída (*output*). É possível através de hierarquia de classes criar novas classes que dão semânticas diferentes para o mesmo código objeto, dessa forma o Adaptools fica extensível e versátil. Porém do modo que isto é implementado, é necessário a recompilação do sistema. A rotina semântica deve analisar o símbolo de saída e retornar para a máquina virtual um *token* qualquer ou uma cadeia de símbolos qualquer que será impresso na saída *output* do Adaptools.

4 SINTETIZADORES

A voz é o meio de comunicação primário entre as pessoas. Progressos significativos têm sido feitos na área de síntese de voz, que nada mais é que a geração automática de sinais de áudio que simulam a fala. Os sintetizadores mais recentes têm uma alta inteligibilidade, e a naturalidade e qualidade da voz, se ainda não estão perfeitas, satisfazem os mais variados tipos de aplicação, como aquelas voltadas para portadores de deficiência visual ou para sistemas telefônicos (auto-atendimento, tele-marketing, etc) (PISTORI, 2004).

Os procedimentos de síntese de voz (TTS - *text-to-speech*) costumam ser divididos em duas fases (LEMMETTY, 1999). A primeira é a análise do texto, comumente chamada de síntese de baixo nível ou *low-level systhesis*, em que a cadeia de entrada é transcrita em alguma forma de representação fonética ou outra representação lingüística. A segunda consiste na geração dos sinais de áudio, usualmente chamada de síntese de alto nível ou *high-level systhesis*.

4.1 Métodos de Síntese de Voz

A primeira idéia que nos vem quando falamos de síntese de voz é tocar arquivos de som pré-gravados, contendo as palavras ou sentenças inteiras. Esse método garante alta naturalidade e qualidade da voz, mas limita-se ao poder do dicionário de palavras.

Lemmetty, 1999 mostra que os sintetizadores podem ser classificados, quanto à metodologia empregada, em três grupos. O método descrito no primeiro parágrafo se enquadra no primeiro grupo: a síntese por concatenação. Sistemas que utilizam este método trabalham com uma base de arquivos de som pré-gravada, em que cada arquivo é comumente associado a um fonema. Os sons dos arquivos vão sendo concatenados conforme os fonemas são gerados a partir da cadeia de entrada, comumente um texto já em formato digital. Esta é a abordagem que atinge a maior naturalidade e inteligibilidade do som, porém utilizando um maior espaço de memória.

Um segundo método é baseia-se no modelo do aparelho vocal humano. Mantendo um conjunto de regras que simulam a língua, os lábios e as cordas vocais, o sistema imita a voz humana ao produzir ressonância e articulação. Esse tipo de síntese se mostra muito complexo, sendo pouco utilizado atualmente.

O terceiro método é baseado em um sistema de combinação de freqüências que resulta em uma expressão vocal determinada. A entrada é processada por um conjunto de ressonadores, onde cada um gera a saída equivalente a um som vocal. No final a saída gerada por todos os ressonadores é somada, produzindo um fonema. A disposição dos ressonadores na estrutura de processamento pode variar a cada sintetizador de voz. Esse método algumas vezes é chamado de *formant synthesis* porque ele modela apenas a fonte de som e as freqüências formadoras.

4.2 História

As primeiras pesquisas de síntese de voz aconteceram na década de 60. Foi uma época marcada por dúvidas sobre a viabilidade destes sistemas e, sobretudo, de pioneirismo. Apenas a partir da década de 70 houve um certo aperfeiçoamento dos sintetizadores (CHBANE, 1994). No final desta década e no início dos anos 80, surgiram os primeiros sintetizadores comerciais e acadêmicos com vocabulário ilimitado. Esses sistemas, a "Infovox" colocou no mercado o sistema "AS 201/PC", capaz de sintetizar voz a partir de textos em Inglês, Francês, Espanhol, Alemão, Italiano, Sueco e Norueguês. De lá pra cá, muita coisa evoluiu. Sistemas como *Via Voice*, da IBM, e *Natural Voices*, da AT&T, tem auto grau de inteligibilidade (PIMENTEL, INÁCIO, 2004).

No Brasil, estudos de síntese de voz tiveram início na Escola Politécnica da Universidade de São Paulo, para um sintetizador para o idioma português (CAMPOS, 1980 apud CHBANE, 1994). Em 1984 (ESQUIVEL, 1985 apud CHBANE, 1994) apresentou um sistema de síntese em tempo real. Um dos primeiros produtos comerciais produzidos no Brasil foi o "Dosvox", um conjunto de programas como editor de texto e calculadora para auxílio a deficientes visuais, de baixo custo. Foi desenvolvido na Universidade Federal do Rio de Janeiro.

Em Portugal, as pesquisas realizadas culminaram com a criação de um laboratório especializado no processamento de linguagens naturais, o Laboratório de Línguas Faladas L^2F, do INESC-ID/IST, em Lisboa. Eles adotaram o "Speech Compiler for your Language - SCYLA" para o desenvolvimento de regras para o Português europeu. Recentemente eles portaram as regras do DIXI, o sintetizador de texto-voz por eles desenvolvidos, para regras baseadas em WFSTs (*Weighted Finite State Transducers* – Transdutores de Estados Finitos Ponderados) (TRANCOSO, CASEIRO, 2004).

4.3 Problemas na Conversão Texto-Voz

Para facilitar a exposição dos problemas comumente encontrados, vamos dividir a conversão em três fases: processamento de texto, pronúncia e prosódia, como mostrado por (LEMMETTY, 1999).

4.3.1 Processamento de Texto

Esta é uma tarefa complexa e realmente muito trabalhosa. A primeira tarefa está relacionada com a expansão de números, como inteiros, ordinais, frações, raízes, etc. Por exemplo, 1345 deve ser expandido para “mil trezentos e quarenta e cinco”. Ainda há muitos outros caracteres especiais que devem ser expandidos, como “@”. Existem problemas semelhantes que são dependentes de contexto, como, por exemplo, o numeral romano III. Em “Capítulo III” ele é lido como “três”; em “Henry III” ele é lido como “terceiro”. Algumas abreviações também devem ser expandidas para o formato exato da fala.

4.3.2 Pronúncia

Os maiores problemas de pronúncia estão relacionados aos homógrafos, palavras que têm a mesma grafia, mas tem significado e até mesmo pronúncias diferentes, como *seca* e *modelo* (substantivos) e *seca* e *modelo* (conjugações do verbo secar e modelar, respectivamente).

Existem outros problemas relacionados à assimilação, isto é, o fato de um segmento adquirir a propriedade de um segmento adjacente a este. Isto acontece, por

exemplo com a letra “s” nas palavras *casca* e *rasga*. Na primeira, o s é desvozeado; na segunda é vozeado. Interessantemente, para os falantes do português, é possível diferenciar o “s” vozeado e desvozeado em final de sílaba, mas não o “r” vozeado e desvozeado em final de sílaba, como em *carga* e *arca*. Estas diferenças acontecem porque estão adjacentes ao segmento vozeado [g] ou ao segmento desvozeado [k] (SILVA, 2003).

O último problema está associado com exceções quanto às regras ou consoantes que variam sua pronúncia conforme a origem da palavra. A letra “x”, por exemplo, pode ter o som de “s”, de “x” ou de “ks” (não é possível diferenciar, por regras gramaticais, o “x” de *lixo* e *fixo*).

4.3.3 Prosódia

Encontrar a entonação, o esforço e a duração corretos a partir do texto escrito é uma das tarefas mais problemáticas destes últimos anos no desenvolvimento da conversão texto-voz. A dependência dos aspectos prosódicos existe em relação a muitos aspectos que não podem ser observados na escrita, como a emoção e as características (sexo, idade) do falante. A diferença entre sentenças imperativas, interrogativas e neutras também é fundamental para entonação.

5 O AUTÔMATO DO VOICER

Partindo da máquina virtual do Adaptools (PISTORI, 2003), que executa uma versão ligeiramente modificada de um autômato adaptativo, e do conjunto de exemplos do Adaptools, foi desenvolvido um novo código-fonte e um novo conjunto de regras. As questões do desenvolvimento relacionadas às técnicas adaptativas serão tratadas neste capítulo.

5.1 Arquitetura

O *Voicer* é baseado em síntese por concatenação de fonemas. As duas fases comumente observadas na síntese de voz podem ser nele observadas.

O autômato funciona como um mecanismo de reconhecimento de texto, e é composto por três blocos distintos e ligados:

- Analisador Léxico: este bloco não contém funções adaptativas, e pode ser enxergado como um autômato de pilha estruturado tradicional. Aqui, a cadeia de entrada (o texto) é consumida, separando-o em sílabas, espaços e sinais de pontuação. Estes átomos são representados por estados que vão sendo empilhados durante a leitura ;
- Modificador: este bloco contém funções adaptativas que agem sobre o terceiro bloco. Os estados, relativos às silabas, são desempilhados e chamam uma função adaptativa específica que altera o transdutor, adicionando transições para todos os fonemas que podem ser gerados a partir da sílaba;
- Transdutor: responsável por escolher a transição correta, isto é, ele escolhe a transição que melhor representa o fonema para a sílaba analisada. Junto com o modificador forma um Pré-Analisador Semântico. A saída deste transdutor adaptativo é uma cadeia de saída, a transcrição fonética do texto digitado.

Ao fim do transdutor, o autômato deve estar no estado de aceitação e a pilha vazia. Neste momento é chamado o Analisador Semântico, de implementação simples, que funciona como o gerador dos sinais de áudio. O Analisador Semântico é

representado pela classe *SoundSemantics*; esta classe é mais bem explicada no capítulo 7.

O analisador sintático foi omitido neste projeto. O analisador sintático permitiria resolver problemas relacionados à dependência de contexto, como as diferenças entre *seca* (substantivo) e *seca* (do verbo secar). Ele não foi implementado para o projeto de formatura devido ao pouco tempo disponível, mas é uma das possibilidades de evolução do projeto.

Alguns padrões foram adotados na descrição das regras no arquivo salvo. A primeira é que os estados são representados por números inteiros, com poucas exceções. Por exemplo, o estado inicial é representado pelo número zero (nenhum valor lido).

Durante a leitura dos caracteres, os estados são descritos da seguinte forma:

- [1-23] – representam as letras do alfabeto de [A-Z];
- [24-33] – representam {á, â, ã, é, ê, í, ó, ô, õ, ú}.

Para os estados que são empilhados, são utilizadas as seguintes regras:

- Sílabas: é feita a concatenação dos textos que representam as letras para identificar o estado. Exemplos: {as} = 0118, pois a = 01 e s = 18; {hal} = 080111, pois h = 08, a = 01 e l = 11.
- Estados empilhados: têm a seguinte forma: 9XXYYZZ, onde XX representa a vogal, YY o sufixo e ZZ o prefixo, ambos YY e ZZ podem ser semivogais ou consoantes. “00” indica que não há sufixo ou prefixo. Esta forma de representação não admite a existência de tritongos ou outras sílabas com quatro ou mais caracteres;
- Espaços e sinais de pontuação: o estado empilhado é chamado de *SPC*, portanto, não há diferenciação rítmica entre um ponto e um espaço, assim como não há diferença na entonação de um pergunta e de uma exclamação.

No decorrer deste capítulo iremos exemplificar o funcionamento do autômato do *Voicer* utilizando uma versão simplificada, somente com regras para o reconhecimento das sílabas {na} e {ba}. A configuração inicial, com a indicação dos

blocos, pode ser observada na figura 6.1. Algumas funções adaptativas foram omitidas para manter a simplicidade da figura., assim como alguns nomes foram mudados.

5.1.1 O Analisador Léxico

O analisador léxico separa a cadeia de entrada em palavras, espaços e sinais de pontuação, e as palavras, em sílabas. Todas essas partes finais da divisão são chamadas de *tokens* (átomos).

O funcionamento deste bloco é simples. Ele poderia ter sido implementado, por exemplo, como uma sub-máquina de um autômato maior. A leitura é feita do último ao primeiro caractere, até terminar o texto de entrada. O motivo disto é que a tônica é definida a partir da última sílaba. Empiricamente, observamos também que é mais fácil separar as sílabas lendo de trás para frente.

O estado inicial é o "0". Deste momento em diante toda letra lida gera uma transição para um estado diferente do atual. Quando uma sílaba é reconhecida e devidamente empilhada, uma transição para um estado anterior, normalmente o "0", é disparada, permitindo o reconhecimento de um novo átomo. Somente depois de lida toda a cadeia é que os estados começam a ser desempilhados.

Existe um estado que tem a função de empilhar um espaço além do átomo atual. Isso ocorre para permitir a leitura de estados que podem ser considerados como "completos" ou "incompletos". Estes estados são aqueles que tem uma das seguintes consoantes precedendo a vogal: S, R, H e L. Se um espaço ou sinal de pontuação é lido quando o autômato esta em um desses estados deve-se empilhar um identificador para este estado e uma pausa. Como não podem ser empilhados os dois valores a tarefa é separada em duas transições, a primeira empilhando o identificador para o estado e a segunda empilhando *SPC*.

Um outro estado especial é o estado de erro. Quando uma seqüência de caracteres não pode mais ser relacionada com nenhuma sílaba, a cadeia é rejeitada e a execução do autômato é interrompida; nenhuma palavra será lida, pois o transdutor não será chamado.

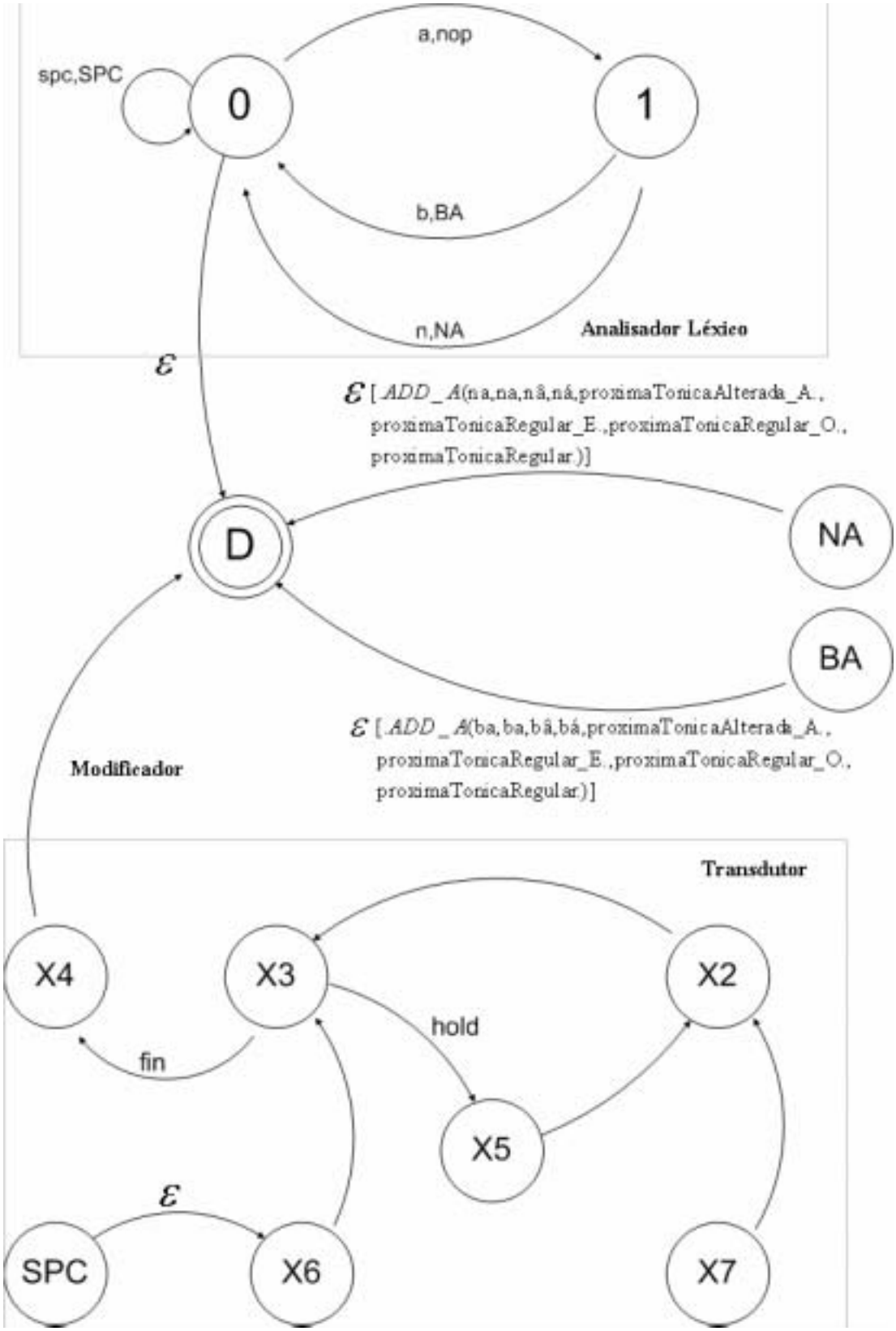

Figura 5.1 – Configuração Inicial do Autômato do Voicer

Para o exemplo simplificado, o autômato utilizado pode parecer não determinístico devido ao modelo de tabela utilizado pelo *Adaptools*. Por exemplo, existe uma transição em vazio saindo do estado "0"; porém, conforme o modelo utilizado, as transições são ordenadas: todas aquelas saindo do estado "0" são mais prioritárias do que aquela que não consome qualquer símbolo. Sendo assim, a transição em vazio só acontecerá quando não houver mais qualquer símbolo de entrada.

A tabela 5.1 mostra a execução do analisador léxico da figura 5.1 para a cadeia de entrada *banana*.

Estado Atual	Cadeia	Próximo Estado	Pilha
0	banana_	0	SPC
0	banana	1	SPC
1	banan	0	SPC,NA
0	bana	1	SPC,NA
1	ban	0	SPC,NA,NA
0	ba	1	SPC,NA,NA
1	b	0	SPC,NA,NA,BA
0	ε	D (9009900)	SPC,NA,NA,BA

Tabela 5.1 – Execução do analisador léxico da figura 5.1 para a cadeia *banana*.

5.1.2 O Modificador

O modificador passa a trabalhar quando toda a cadeia já foi lida. O autômato, então, vai para um estado de aceitação, mas a pilha está cheia; então os estados passam a ser desempilhados.

Para cada sílaba, uma única transição para o estado de aceitação é executada. Esta transição não consome nenhum símbolo da cadeia de entrada nem empilha outro estado; somente chama uma função adaptativa que modifica o transdutor. No autômato modificador original, as seguintes funções adaptativas podem ser chamadas:

- *Add_A*, *Add_E* e *Add_O*: cada uma destas funções adiciona um conjunto de estados e transições no transdutor que permitam a escolha pelo fonema correto. Recebem oito parâmetros: os nomes da sílaba, do fonema regular, do fonema aberto e do fonema fechado, e as funções adaptativas para permitir a escolha da próxima tônica alterada ou regular para as sílabas terminadas em "A", "O", "E" e outras terminações ("U", "I", "R", etc.);
- *Add2*: altera o transdutor de modo a forçar o acento tônico no último fonema;
- *Add3*: força a sílaba acentuada como tônica.

Estado Atual	Cadeia	Próximo Estado	Pilha	FUNÇÃO ADAPTATIVA (F)	Config. do Transdutor antes de *F*
9009900	ε	BA	SPC,NA,NA	-	Figura 5.2
BA	ε	9009900	SPC,NA,NA	.ADD_A(...)	Figura 5.2
9009900	ε	NA	SPC,NA	-	Figura 5.3
NA	ε	9009900	SPC,NA	.ADD_A(...)	Figura 5.3
9009900	ε	NA	SPC	-	Figura 5.4
NA	ε	9009900	SPC	.ADD_A(...)	Figura 5.4
9009900	ε	SPC	-	-	Figura 5.5

Tabela 5.2 –Execução do modificador da fig.5.1 para a cadeia *banana.*

A tabela 5.1 mostra a execução do modificador da figura 5.1 para a cadeia de entrada *banana*. As figuras 5.2, 5.3, 5.4 e 5.5 mostram as configurações do transdutor inicial e pós-chamadas de função adaptativa. Repare que, sempre que a função *Add_A* é executada, um novo estado para a sílaba é gerado e inserido antes do estado X4; também os estados para seleção da vogal tônica são inseridos para a sílaba anterior. Na figura 5.2, estes estados são alcançáveis a partir X3 pela transição que contém a função adaptativa ÚLTIMA, e o estado gerado representante da sílaba é o BA. Também os estados X5, X6 e X7 tem suas transições modificadas. A função de cada um destes estados será explicada no item 5.4.

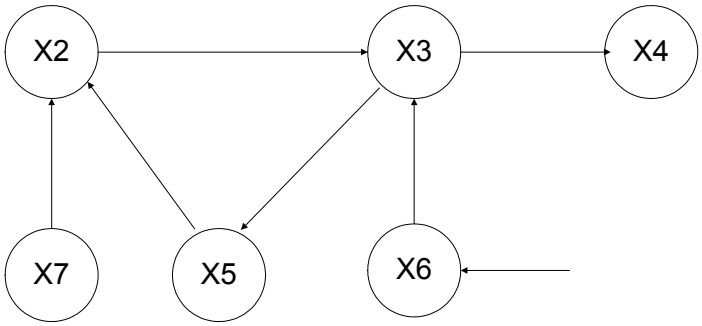

Figura 5.2 –Configuração inicial simplificada do transdutor

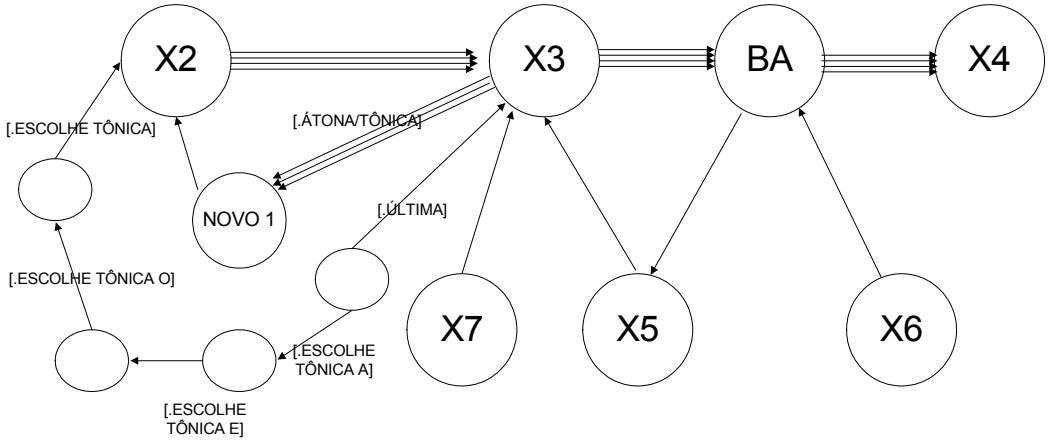

Figura 5.3 –Configuração do Transdutor após o desempilhamento da sílaba BA da palavra *banana*.

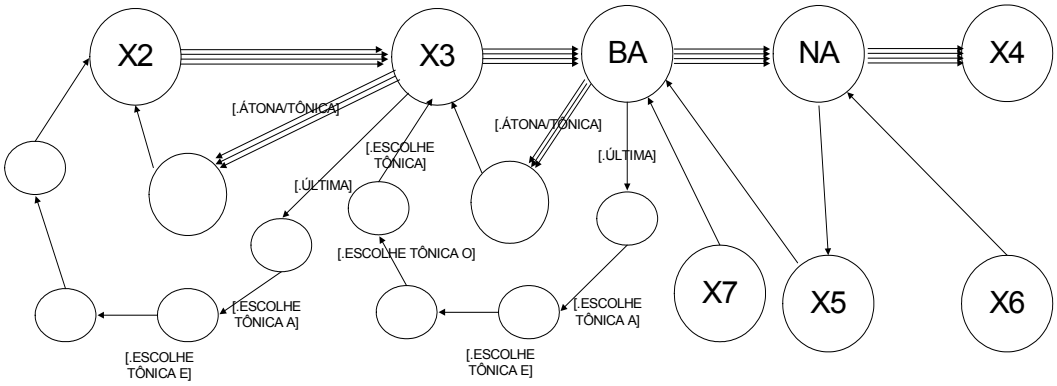

Figura 5.4 – Configuração do Transdutor após o desempilhamento da primeira

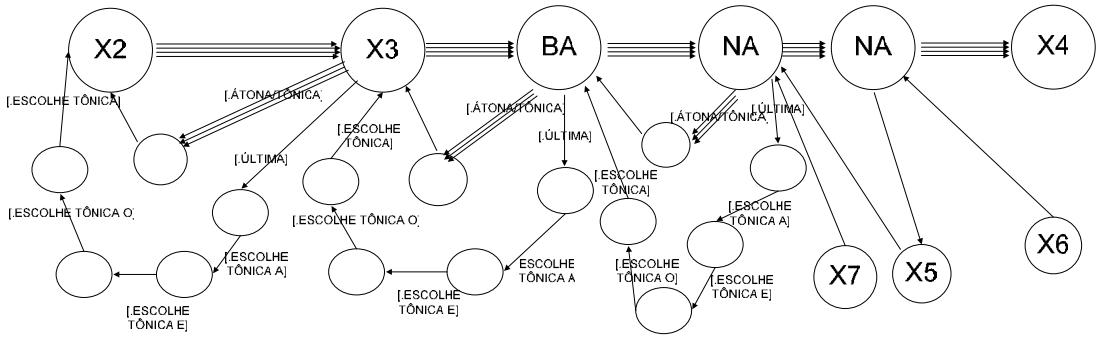

Figura 5.5 – Configuração do Transdutor após o desempilhamento da segunda sílaba NA da palavra *banana*.

5.1.3 O Transdutor

O último bloco do autômato é responsável por eliminar o não determinismo que é gerado pelo modificador quando ele adiciona os estados correspondentes a cada sílaba no autômato. Por exemplo, quando a sílaba (estado) BA é desempilhada o transdutor passa da configuração exibida na figura 5.2 para aquela exibida na figura 5.3, são adicionadas quatro transições de X3 para o novo estado BA, uma para cada um dos seguintes fonemas: bá, bâ, bã e ba. É o próprio analisador que decide qual destas transições disparar; cada uma delas irá gerar um átomo (um fonema) diferente que será inserido na cadeia de saída. Relembremos que um fonema é a unidade fonêmica indivisível (SILVA,2003) .

O transdutor passa a ser executado quando o estado SPC é desempilhado (ver figura 5.1). Uma transição em vazio leva ao estado X6. Para a cadeia de entrada *banana*, a configuração estará, neste instante, como na figura 5.5.

Antes de mostrarmos como as regras foram aplicadas na escolha do fonema correto, vamos ressaltar um importante aspecto de implementação. Seja Σ o alfabeto formado por todos os símbolos aceitos pelo analisador léxico e Q o conjunto de todos estados do autômato do *voicer*. Abstraindo, qualquer transição pode estar em um de dois estados. Uma transição *aberta* tem a forma $Q \times (\Sigma \cup \varepsilon) \rightarrow Q$. Uma transição *fechada* tem a forma $Q \times \Omega \rightarrow Q$, de tal forma que o conjunto $\Sigma \cap \Omega$ não tem nenhum elemento. Isto é, uma transição fechada existe, mas nunca vai ser disparada se utilizarmos uma cadeia de entrada formada por símbolos de Σ. Para evitar não

determinismos, que o *Adaptools* não está preparado para resolver, as funções adaptativas de algumas transições abrem ou fecham outras transições do transdutor. Somente neste bloco existem transições fechadas, assim como todas as transições abertas presente nele não consomem nenhum símbolo de entrada.

Estado Atual	Próximo Estado	Descrição da Função Adaptativa
X6	NA2	Abre a transição NA2→X5
NA2	X5	Fecha NA2→X5
X5	NA	Abre um das transições NA→DCNA2
NA	DCNA2	Abre uma das transições NA→NA2. Fecha NA→DCNA2
DCNA2	BA	Abre uma das transições BA→DCNA
BA	DCNA	Abre uma das transições BA→NA. Fecha BA→DCNA
DCNA	X3	Abre uma das transições X3→DCBA
X3	DCBA	Abre uma das transições X3→BA. Fecha X3→DCBA
DCBA	X2	Não chama nenhuma função adaptativa para tomada de decisão porque não existem mais transições não determinísticas.
X2	X3	Abre NA2→X4. A partir desta transição percorre o único caminho possível, que não tem funções adaptativas, gerando uma cadeia de saída.
X3	BA	Põe "ba" na saída
BA	NA	Põe "nâ" na saída
NA	NA2	Põe "na" na saída
NA2	X4	Não faz nada.
X4	D(9009900)	Fim da palavra. Se a pilha estiver vazia, fim do autômato.

Tabela 5.3 –Execução do transdutor da fig.5.1 para a cadeia *banana.*

Cada estado ou transição do transdutor está intimamente ligado com as regras fonéticas. O estado X6 sempre tem uma única transição para o estado da última sílaba da palavra e o estado X7 tem uma para a penúltima; eles existem para auxiliar

as ações adaptativas de consulta ou remoção. Para o exemplo (ver figura 5.5), os estados DCNA, DCNA2 e DCBA estão cercados pelas transições que fazem a escolha dos fonemas e, de fato, removem o não determinismo. As transições entre X3, BA, NA e NA2 geram as transcrições fonéticas que serão utilizadas pela classe *SoundSemantics* para abrir e tocar os arquivos de som de cada fonema.

A tabela 5.3 mostra a execução passo a passo do autômato.

5.2 O Problema de Desempenho na Unificação

Um problema de desempenho encontrado nas primeiras versões do Voicer era a lentidão. Isso acontecia porque cada query das funções adaptativas iterava com cada uma das mais de 55000 regras do arquivo de regras do *Voicer***.** Embora condizente com a definição de adaptatividade, não era a melhor solução para uma aplicação de sintetizador texto-voz.

A solução foi criar mais uma coluna no arquivo para dizer se a regra é unificável ou não. As seguintes colunas são encontradas na solução final: head, estado de origem, símbolo de entrada, estado de destino, empilha, símbolo de saída, função adaptativa e a coluna que diz se a regra é ou não unificável.

Na prática, isso quer dizer que as queries serão realizadas somente entre as regras unificáveis. Isso aumentou o desempenho drasticamente, pois no conjunto inicial de regras existem apenas sete unificáveis. Todas as regras inseridas por ações adaptativas de inserção são unificáveis.

As alterações no código-fonte estão nas seguintes classes:

- *Adapter*: métodos *executeQueryAction()* e *executeRemoveActions()*;
- *Rules*: vários métodos foram alterados para suportar uma lista de regras unificáveis. A lista de regras unificáveis é criada no construtor;
- *Rule*: adição da propriedade *Unifiable*;
- *Vmds*: alteração da função *LoadObjectCode*, que faz a leitura do arquivo e o transforma em regras.

5.3 Tratamento de Exceções nas Regras

Nem todas as palavras utilizadas pelos falantes da língua portuguesa obedecem às regras. Por exemplo, seja o texto “rebola, sacola e cebola”. As duas primeiras palavras deste texto têm a letra “o” pronunciado aberto; a última tem o “o” fechado.

Para resolver essas exceções foi criado um dicionário. Ele funciona como um pré-compilador. Antes de o texto digitado ser passado ao analisador léxico, cada palavra deste texto é procurada no dicionário. Se encontrada, ela é substituída pela forma que corresponde à pronúncia correta. O texto final, com todas substituições feitas, é, então, passado ao analisador léxico.

Para o exemplo citado, serão procuradas no dicionário entradas para as palavras *rebola*, *sacola* e *cebola*. As duas primeiras buscas resultarão em falha; a busca pela última palavra resultará na localização do par (*cebola*, *cebôla*) e na conseqüente substituição da palavra na cadeia a ser passada ao autômato. O aspecto final do texto de entrada será “rebola, sacola e cebôla”.

Uma interface gráfica para o cadastro de exceções foi adicionada a interface original do *Voicer*.

6 METODOLOGIA E PLANEJAMENTO

O planejamento foi a principal preocupação desde o início do projeto. As primeiras reuniões foram centradas em definir o cronograma e a divisão das tarefas e a metodologia adequada para comunicação dentro da equipe e organização documentos.

	Atividade do Projeto	Responsável
01	Documentação Geral do Projeto	Daniel Alfenas
02	Planejamento, Cronograma e Andamento do Projeto	Henrique Soejima
03	Estudo sobre Autômatos Adaptativos	Danilo Shibata
04	Estudo sobre Autômato de Pilha Estruturado	Danilo Shibata
05	Estudos sobre o *Adaptools*	Danilo Shibata
06	Estudo sobre o Som	Allan Jones
07	Estudos sobre tecnologias de manipulação de som	Allan Jones
08	Estudo sobre fonemas + Voz humana	Henrique Soejima
09	Implementação	Daniel Alfenas
09ª	Implementação do modelo completo baseado em *Adaptools*, a partir da geração de fonemas	Todos
10	*Home Page* para organização dos trabalhos	Henrique Soejima

Tabela 6.1 – Distribuição das Atividades do Projeto

6.1 Atividades e Divisão da Equipe

A primeira tentativa de organizar a equipe resultou na listagem de futuras atividades a serem realizadas. A responsabilidade por cada uma destas atividades foi delegada a um membro da equipe em uma das primeiras reuniões realizadas. É importante ressaltar que a distribuição apresentada na tabela 5.1 representa apenas uma forma

de centralizar a organização de atividades, assim sendo, todos da equipe terão participação efetivamente em cada um dos itens.

A distribuição das tarefas foi a mesma até o fim do projeto e não teve necessidade de alteração.

6.2 Comunicação

Para facilitar a organização e a distribuição de documentos e códigos-fonte, assim como a comunicação entre os membros da equipe e o orientador, um *website* teria de ser desenvolvido para publicação de arquivos e envio de e-mail aos membros. Desta forma, as informações estariam ao alcance de toda a equipe, do orientador e de qualquer pessoa outra pessoa interessada no projeto. A home page se encontra no seguinte endereço: http://www.amovoce.com.br/Voicer/.

6.3 Cronograma Planejado

O primeiro cronograma do ano foi apresentado no documento de especificação entregue no mês de abril e pode ser conferido na tabela 5.2. Este cronograma foi dimensionado de modo a alocar a maior parte das tarefas para o último quadrimestre de estágio, entre abril e agosto de 2004. Neste período não há obrigações acadêmicas, permitindo que muitas das tarefas pudessem ser terminadas antes que o último módulo de aula começasse. A parte relacionada à documentação final (inclusa no item documentação do cronograma) desde cedo foi planejada para ser feita nos últimos meses do ano, quando a maior parte das tarefas já deveria ter sido realizadas.

A primeira atualização completa aconteceu em setembro de 2004, logo no retorno às aulas. Com um atraso de duas semanas em relação ao cronograma apresentado inicialmente, as atualizações se deram nos seguintes pontos:

- O estudo e a aplicação (implementação) de uma solução adaptativa nas decisões não determinísticas durante a determinação dos fonemas. Essa atividade inclui um estudo maior das regras que relacionam sílabas e fonemas. Dada a complexidade, esta tarefa teve estimativa de conclusão para 05/11/2004;

- Atualização e finalização da documentação do projeto, com estimativa de conclusão para 19/11/2004;
- Implementação de um sistema de gerenciamento de documentos no site do projeto.

	Atividade	Período (ano: 2004)
01	Entrega do Documento de Especificação do Projeto e Publicação do Site da Equipe	05/Abril
02	Elaboração da Segunda Apresentação	12/Abril a 15/Abril
03	Segunda Avaliação (apresentação)	16/Abril (10h30)
04	Estudos sobre Autômatos Adaptativos e de Pilha Estruturados	05/Abril a 31/Maio
05	Estudo detalhado sobre o Adaptools, incluindo a arquitetura e a manipulação dos fonemas.	01/Abril a 31/Maio
06	Estudo sobre o Som e tecnologias de manipulação relacionadas ao projeto	01/Abril a 31/Maio
07	Estudo sobre fonemas e características principais da voz humana	01/Abril a 31/Maio
08	Gravação de fonemas e testes no Adaptools	01/Junho a 31/Julho
09	Implementação de aperfeiçoamento das características da voz digital	01/Agosto a 31/Agosto
10	Testes Preliminares (registrando problemas e bugs a serem corrigidos)	01/Setembro a 15/Setembro
11	Debugging e outras correções no projeto	15/Setembro até a entrega final.
12	Documentação de Projeto	De 01/Abril até a entrega final de acordo com as necessidades e desenvolvimento.

Tabela 6.2 – Cronograma inicial apresentado na especifica entregue em abril

A distribuição real das tarefas através do tempo e a comparação desta distribuição com os cronogramas apresentados está no capítulo 7.

6.4 Cronograma Efetivo

Após a elaboração do cronograma anterior, houve adequações no calendário de entrega de relatórios por parte do Comitê Gestor que demonstrou sensibilidade para com a carga de atividades à qual os alunos estavam sujeitos.

A avaliação neste módulo acadêmico, que seria constituída de dois relatórios de avaliação parcial mais a monografia final, teve a eliminação do segundo Relatório de Avaliação Parcial. Somou-se a isso um adiamento da entrega da cópia da versão final da monografia para a data de 06/12/2004 e da entrega da impressão oficial da monografia para a data da apresentação final.

O calendário re-planejado pela equipe no início de Setembro e reportado no tópico anterior foi parcialmente cumprido:

1) com a definição e implementação da solução adaptativa para a seleção de fonemas até a data de 05/11/2004 (planejamento cumprido com êxito);

2) atualização e finalização da documentação do projeto atingida em 04/12/2004 (com 15 dias de atraso);

3) não foi finalizado o sistema de gerenciamento on-line de documentação e versionamento do aplicativo (sistema que seria localizado no site do projeto de formatura)

6.5 Custos Planejados

A tabela de custos estimados para este projeto foi entregue com o relatório resumido 1, em setembro deste ano, e pode ser observada na tabela 5.3. Os custos reais de implementação e a comparação destes custos com os planejados estão no capítulo 7.

Os custos estão estimados em homens-hora, e foram obtidos através da simples enumeração de tempo, pessoal e infra-estrutura necessários para a realização do projeto.

Tamanho da Equipe	4 alunos + 1 Orientador (5 colaboradores)		
Natureza do custo	Homens-Hora *(h*h)*	Estimativa de Custo (homens*horas)	
Atividades:	Planejamento inicial da equipe (incluindo Orientador)	5 * 5	25 hh
(PCS2040)	Estudos (leitura e pesquisa) sobre autômatos, adaptatividade	4 * 12	48 hh
	Estudo da estrutura e funcionamento do Adaptools	2 * 8	16 hh
	Pesquisas sobre formação de fonemas e levantamento de fonemas a serem utilizados no projeto	1 * 8	8 hh
	Desenvolvimento do WebSite básico.	1 * 3	3 hh
	Documentação do Modelo do Projeto	4 * 4	16 hh
	Elaboração da 1ª Apresentação + resumo/relatório	2 * 3	6 hh
	Elaboração da 2ª Apresentação + resumo/relatório	2 * 3	6 hh
	Elaboração da 3ª Apresentação (apresentação mais completa) + resumo + relatório completo	5 * 3	15 hh

(Módulo de estágio)	Gravação de fonemas básicos	1 * 3	3 hh
	Criação das regras para reconhecimento de todos os fonemas	2 * 24	48 hh
	Implementação de alterações no Adaptools, extraindo somente o que é relevante para o Sintetizador Texto-Voz	2 * 10	20 hh
(PCS2050)	Integração e testes da nova versão do Sintetizador com os fonemas completos	2 * 3	6 hh
	Elaboração da 1ª Apresentação + resumo/relatório	2 * 2	4 hh
	Estudos e busca de solução adaptativa para reconhecimento de fonemas tônicos em textos (questão não-determinística)	4 * 12	48 hh
	Alteração nas regras para readequação à solução adaptativa	4 * 12	48 hh
	Implementação da solução no Sintetizador	2 * 8	16 hh
	Elaboração da 2ª Apresentação	2 * 3	6 hh
	Definição de cenário de testes	4 * 3	12 hh
	Revisão e fechamento da documentação final do sistema	4 * 6	24 hh
	Elaboração da 3ª Apresentação + relatório final + produto final	5 * 4	20 hh
TOTAL:			398 hh
horas/homem(4):			99,5 h/ homem

Tabela 6.3 – Tabela de custos estimados apresentada em setembro de 2004

6.6 Custo Efetivo do Projeto

Os custos reais de implementação e a comparação destes custos com os planejados apresentam-se abaixo.

Como já citado no tópico anterior, os custos estão contabilizados em homem*hora, e foram obtidos através do registro simples de tempo, pessoal e infra-estrutura necessários para a realização do projeto.

Tamanho da Equipe	4 alunos + 1 Orientador (5 colaboradores)		
Natureza do custo	Homens-Hora *(h*h)*	Estimativa de Custo (homens*horas)	
Atividades:	Planejamento inicial da equipe (incluindo Orientador)	5 * 5	25 hh
(PCS2040)	Estudos (leitura e pesquisa) sobre autômatos, adaptatividade	4 * 12	48 hh
	Estudo da estrutura e funcionamento do Adaptools	2 * 8	16 hh
	Pesquisas sobre formação de fonemas e levantamento de fonemas a serem utilizados no projeto	1 * 8	8 hh
	Desenvolvimento do WebSite básico.	1 * 3	3 hh
	Documentação do Modelo do Projeto	4 * 4	16 hh
	Elaboração da 1ª Apresentação + resumo/relatório	2 * 3	6 hh
	Elaboração da 2ª Apresentação + resumo/relatório	2 * 3	6 hh
	Elaboração da 3ª Apresentação (apresentação mais completa) + resumo + relatório completo	5 * 3	15 hh

(Módulo de estágio)	Gravação de fonemas básicos	1 * 5	5 hh
	Criação das regras para reconhecimento de todas as sílabas	1 * 120	120 hh
	Implementação de alterações no Adaptools, extraindo somente o que é relevante para o Sintetizador Texto-Voz	1 * 20	20 hh
(PCS2050)	Gravação de fonemas adicionais necessários ao funcionamento do Sintetizador	1 * 4	4 hh
	Segmentação das cadeias de fonemas gravados.	1 * 8	8 hh
	Integração e testes da nova versão do Sintetizador com os fonemas completos	2 * 5	10 hh
	Elaboração da 1ª Apresentação + resumo/relatório	2 * 2	4 hh
	Estudos e busca de solução adaptativa para reconhecimento de fonemas tônicos em textos (questão não-determinística)	1 * 120	120 hh
	Alteração nas regras para readequação à solução adaptativa	1 * 40	40 hh
	Implementação da solução no Sintetizador	2 * 50	100 hh
	Elaboração do 2º RA (Apresentação dispensada pelo Comitê Gestor)	1 * 6	6 hh
	Definição de cenário de testes	4 * 3	12 hh
	Revisão e fechamento da documentação final do projeto	4 * 40	160 hh
	Relatório final + produto final	4 * 20	80 hh

	Elaboração da Apresentação Final	2 * 2	4 hh
TOTAL:			836 hh
horas/homem(4):			209,00 h/ homem

Tabela 6.4 – Tabela de custos finais consolidados

Houve um aumento elevado na carga de trabalho da equipe, devido a:

- Atraso no cronograma planejado para o módulo de estágio (de Maio a Agosto).
- A dificuldade para a definição da solução adaptativa foi superior à esperada e exigiu uma dedicação muito maior por parte da equipe.
- Houve um atraso na elaboração da documentação e a documentação envolveu aspectos teóricos que exigiam um estudo extra sobre fonética, autômatos adaptativos, estudos sobre a forma como Adaptools trata e simula autômatos adaptativos e regras de acentuação e gramática da língua portuguesa brasileira.
- Houve extrema dedicação na tentativa de atingir uma solução de alta qualidade técnica, mesmo sem o embasamento acadêmico (de técnicas adaptativas) que a complexidade do assunto tratado pelo projeto exigiu.

Em relação aos custos de infra-estrutura e tecnologias utilizadas, a equipe concentrou-se em utilizar infra-estrutura própria de informática e soluções de software abertas e com custo mínimo (ou zero) em software – como exemplo temos a adoção da Linguagem Java, a IDE Eclipse para programação e a base na ferramenta Adaptools.

As estações de trabalho utilizadas para geração da documentação e programação do aplicativo final foram as pertencentes a cada um dos integrantes da equipe ou pertencentes aos laboratórios da Universidade de São Paulo de livre acesso aos seus alunos.

7 ESPECIFICAÇÃO

Este capítulo busca a apresentação de forma estruturada da arquitetura do sistema desenvolvido, apresentando alguns detalhes sobre o empacotamento de classes, detalhes de implantação, além de informações sobre o processo de alteração de código da versão do *Adaptools* que foi utilizada para a última versão do sistema.

7.1 Requisitos Funcionais

O sistema desenvolvido visa atender aos seguintes requisitos estabelecidos para o cumprimento do presente projeto:

- Realizar o reconhecimento (validação semântica) de frases da língua portuguesa, de forma a possibilitar o mapeamento em fonemas utilizados no Brasil, essencialmente, para a variante paulistana. Caso a frase não seja reconhecida como válida, o sistema deve rejeitá-la.
- Aceitar a seqüência de cadeias. O sistema deve efetuar a correta seleção dos fonemas a serem tocados, tratando através de técnicas adaptativas dos casos em que uma mesma sílaba apresenta duas ou mais sonoridades que dependem do contexto (da formação da palavra) onde o fonema se situa.
- Após a seleção dos fonemas quanto à tonicidade e abertura dos mesmos, o sistema deve executar a leitura (apresentar os sons do fonema) da frase fornecida ao sistema.

7.2 Requisitos Não Funcionais

A modularidade e escalabilidade do presente sistema foram utilizadas de forma concreta, ao efetuarmos a implementação de um módulo de interação com conteúdo de informação na Internet segundo atividade proposta pela disciplina PCS2057-Hipermídia e Multimídia desta escola.

O sistema apresenta informações pertinentes ao processo de reconhecimento, através dos autômatos e das regras armazenadas, demonstrando em um pequeno frame da tela principal do sistema todo o processo com suas transições de estados e chamadas de funções adaptativas (esta funcionalidade tem poderoso teor didático).

A presente versão do sistema apresenta-se estável, e os erros foram detectados e eliminados até o momento durante testes desenvolvidos com protótipos intermediários.

7.3 Plataforma de Desenvolvimento

Existem algumas peculiaridades quanto à plataforma de desenvolvimento. Por exemplo, embora tenhamos feito o sistema em Java, não podemos garantir que ele funcionaria em ambiente Unix – todo o desenvolvimento e testes foi feito no Windows XP. Neste capítulo, expomos os softwares utilizados.

7.3.1 Linguagem

O sistema foi desenvolvido utilizando-se a linguagem Java, da Sun Microsystems, na versão 1.4.2_06, mais o framework livre de interface gráfica denominado Thinlet (http://www.thinlet.com). É importante observar que o *Voicer* utiliza o pacote *Java Sound*, que não está incluído no JRE em versões anteriores a 1.3.

7.3.2 IDE

Adotou-se como padrão o Eclipse como IDE (plataforma livre – http://www.eclipse.org).

7.3.3 Adaptools

Como citado anteriormente, a equipe baseou-se na ferramenta *Adaptools* (também livre - http://www.ucdbnet.com.br/adaptools) que trata da modelagem e simulação de autômatos adaptativos (PISTORI, 2003). Deste sistema serviram como base os módulos relacionados ao *Kernel* (com os principais métodos de interpretação do arquivo que armazena as regras de reconhecimento), *Adapter* (com um modelo genérico de manipulação de autômatos adaptativos) e o módulo de execução de som

(classes do package *javax.sound.sampled.** - nativo da própria linguagem Java). As alterações realizadas sobre as classes originais do *Adaptools* estão detalhadas na seção 7.4.4.

7.4 Descrição da arquitetura de software

Neste tópico vamos fazer uma descrição sucinta dos pacotes, assim como listar as alterações. Diagramas e demais anexos poderão ser encontrados na documentação final da disciplina de projeto de formatura.

7.4.1 Arquitetura de software

O diagrama abaixo apresenta a arquitetura básica do software desenvolvido.

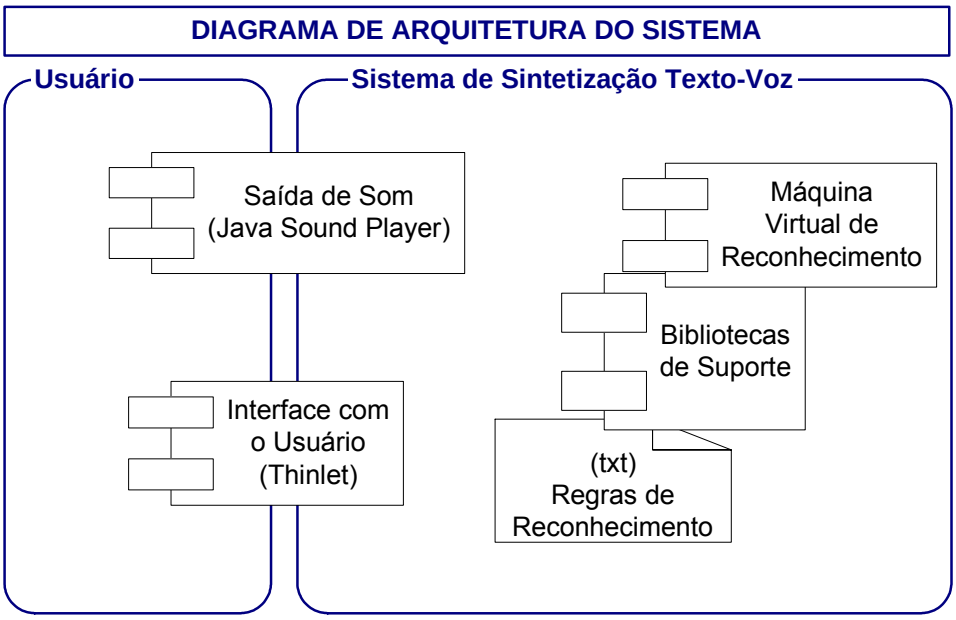

Figura 7.1 – Diagrama de Arquitetura do Sistema

7.4.2 Pacotes

O Sintetizador Texto-Voz estrutura-se em quatro packages:

a) *com.coop4.voicer.ui*: (*user interface package*) contém as classes de tratamento da interface com usuário (entrada e saída de texto), baseado no *Thinlet*.

b) *com.coop4.voicer.sound*: (*sound player package*) contém as classes responsáveis pela execução dos arquivos de som que representam cada um dos fonemas a serem "pronunciados" pelo sintetizador.

c) *com.coop4.voicer.util*: (*utilities library*) biblioteca com classes úteis para o reconhecimento e inteligência para o aprendizado do sistema.

d) *com.coop4.voicer.vm*: (*virtual machine*) contém a classe *Kernel* e todas as classes de lógica de negócios do sistema.

7.4.3 Especificação de Classes

O documento de especificação de classes encontra-se no Anexo B.

7.4.4 Modificações realizadas nas classes utilizadas do Adaptools

O software *Adaptools* de licença GPL foi utilizado como base para a implementação de nosso projeto. Inúmeras alterações foram efetuadas a fim de melhorar seu uso para nosso problema específico.

- *Kernel* – Esta classe não sofreu alteração em sua lógica principal, apenas modificamos sua implementação para remover código referente à interface gráfica com o usuário;
- *Adapter* – Poucas alterações foram feitas, seguindo a mesma linha da classe *Kernel*;
- *Vmds* – Essa classe foi totalmente alterada, sendo que no *Adaptools* essa classe representava uma interface da estrutura de dados de uma máquina virtual. Em nosso projeto essa classe realmente é uma classe concreta e ela possui toda a estrutura de dados da máquina virtual (regras, arquivo de entrada, entrada, estados finais, etc...). No *Adaptools* essa interface era implementada por duas classes chamadas *Viewer* e *VmdsImpl* que guardavam a mesma estrutura de dados, porém o *Viewer* utilizava a API gráfica Swing do Java para criar a interface gráfica da aplicação para o usuário, enquanto a classe *VmdsImpl* atuava através de comandos pelo console;

- *Rule* – Classe que representa uma Regra de transição de um autômato. Tanto no *Adaptools* quanto no *Voicer* essa classe possui os mesmos atributos. Seus métodos foram pouco modificados, porém produzem o mesmo efeito;
- *Rules* – Essa classe foi reformulada para ser um modo inteligente de se guardar todo o conjunto de regras da língua portuguesa. Pelo fato do conjunto de regras da língua portuguesa ser composto por mais de 60 mil regras, foi necessário criar uma estrutura diferente da utilizada pelo *Adaptools*, para evitar que as buscas pelas regras não demorem. Nele, tornou-se impossível utilizar nosso conjunto de regras, pois estas demoravam minutos para serem carregadas. Em nosso software isso foi reformulado de modo que as regras são carregadas em milésimos de segundo. Mais detalhes sobre este item estão no tópico 3.2;
- *Semantics* – No *Adaptools* essa classe é uma classe abstrata que possuía apenas métodos abstratos. No projeto do *Voicer* ela foi transformada em uma interface.
- *SoundPlayer* – Vários métodos nunca utilizados foram removidos e a procura pelo arquivo de som é feita através do *ClassPath* da aplicação, não mais através de arquivos no sistema de arquivos do sistema operacional. Dessa forma é possível agrupar todos os arquivos de som em um arquivo compactado (zip, por exemplo) e armazenar esse arquivo compactado de sons no *ClassPath* da aplicação. Dessa forma, economizamos espaço e apenas trocando o arquivo de sons por outro arquivo de sons podemos mudar a voz de quem fala no programa.

A interface gráfica do programa também foi totalmente reformulada, uma vez que o *Adaptools* é um programa genérico para construir autômatos adaptativos, e o projeto *Voicer* tem uma única função que é vocalizar textos digitados. A interface gráfica pode ser acoplada, sendo que podemos executar o programa na presença da interface com o usuário ou sem a mesma, ou seja, com chamadas em modo console, invisíveis ao usuário, sem a necessidade de reimplementar o núcleo inteligente do programa.

8 RESULTADOS E APLICAÇÕES

Este capítulo mostra os resultados obtidos ao fim do projeto de formatura, dando ênfase às palavras que o *Voicer* é capaz ou não de transcrever em notação fonética (síntese de baixo nível) e também aos fonemas que é ou não capaz de transformar em som (síntese de alto nível).

O primeiro resultado é que a versão entregue não é plenamente capaz de ler textos, mas sim palavras. Não é realizada qualquer análise do contexto, quer seja para a determinar a pronúncia correta da palavra, ou a correta entonação da sentença. Não foi criado um analisador sintático separado do semântico, que seria capaz de resolver problemas de dependência de contexto. Algumas funções da análise sintática foram distribuídas nos blocos nomeados transdutor e modificador.

A adaptatividade foi utilizada apenas na geração da transcrição fonética a partir das sílabas, separadas pelo analisador léxico. Porém ela poderia ser usada em todas as fases de um sintetizador, inclusive na geração de som (PISTORI, 2004).

8.1 Resultados da Síntese de Baixo Nível

No *Voicer*, em que todo o reconhecimento do texto é feito por um único autômato que recebe um texto de entrada que obedece as regras gramaticais da língua portuguesa, a saída é a transcrição fonética, que será repassada para o módulo responsável pela síntese de alto nível.

O analisador léxico é capaz de separar quase todas as sílabas corretamente, incluindo junções consonantais no início de palavra. Ele falha apenas nas junções vocálicas – ele sempre considera estas junções um hiato, nunca um ditongo ou tritongo.

O modificador gera todas as transcrições fonéticas corretamente a partir do texto recebidos. Os ditongos e tritongos são transcritos erroneamente por que o analisador léxico passa as vogais em sílabas separadas. Os casos que não estão

adequados ao conjunto de regras adotado, como nas palavras *cebola* e *fixo*, são tratados por dicionário.

8.2 Resultados da Síntese de Alto Nível

A síntese de alto nível, realizada pela classe *SoundSemantics*, transforma a transcrição fonética em som. A classe *SoundSemantics* recebe como entrada os átomos referentes ao fonema que deve ser pronunciado. Nesse procedimento, a classe delega para outra classe, *SoundPlayer* que recebe o nome da sílaba a ser tocada e faz a busca pelo arquivo em formato *wave* correspondente. Possuímos um grande número de fonemas da língua portuguesa, porém é possível que algum fonema em específico não seja encontrado, assim, um som de erro é tocado indicando a falta de tal fonema. Dessa forma, distinguem-se dois tipos de erro, pois a máquina virtual executou corretamente e compilou a entrada corretamente, porém o conjunto de fonemas gravados não comportou os fonemas requeridos para serem tocados. Algumas sílabas não triviais, por exemplo, *bem*, não possui um fonema gravado. Para essas sílabas que terminam em *r, s, m, l,* etc. é necessário fazer a junção de dois outros fonemas para produzir o som corretamente. Na fase atual do projeto essa característica não foi implementada, porém poderá ser adicionada a classe *SoundPlayer* assim que possível. O funcionamento será bastante simples, sendo que se o *player* for requisitado para pronunciar uma sílaba como *sal*, ele procurará pelo fonema gravado com o nome *sal.wav*, e não encontrando tal fonema, ele dividirá em dois outros fonemas separado pela vogal, no caso será procurado pelos fonemas *s* e *al*. Assim sendo, se o *player* encontrar ambos os fonemas, eles serão executados em seqüência e a sílaba *sal* será pronunciada pelo voicer.

Outro problema que permanece está relacionado com o corte dos fonemas gravados. Por estes cortes terem sido feitos com demasiada folga, fica a impressão de que há uma pausa demasiadamente grande entre cada sílaba de uma palavra pronunciada. Um corte mais preciso deverá resultar em uma síntese de voz muito mais natural.

8.3 Aplicações

Inicialmente, a idéia principal era utilizar o *Voicer* como uma aplicação de auxílio a portadores de deficiências visuais. Porém tal aplicação ainda não foi implementada, ficando como tarefa futura. É necessário trabalhar mais os arquivos de fonemas gravados e também a separação silábica de hiatos e tritongos para permitir que pessoas digitem livremente seus textos no *Voicer* e escutem-nos com alguma fidelidade.

Como exemplo de aplicação foi implementado um *site* infantil, onde mostramos que é possível adicionar recursos multimídia à navegação, sendo que é demonstrada a utilização de *link*s sonoros, onde a criança pode com apenas um clique ouvir a pronuncia de algum texto ou palavra que é referenciada por esse *link*. Essa funcionalidade é exposta e explicada mais detalhadamente no anexo A.

8.4 Exemplo: A Palavra “Banana”

Por fim, vamos mostrar um pouco do funcionamento do *Voicer*, exibindo algumas imagens da sua interface, para o exemplo já clássico da palavra *banana*. Por que uma palavra tão simples é tão referenciada? A única razão para isso é que ela possui duas sílabas *na* pronunciadas de forma diferente: a mais à esquerda é fechada e a mais à direita é aberta. Isto acontece porque a primeira letra “a” precede uma consoante nasal, o “n”. Nos primeiros testes do *Voicer*, antes da adição das funções adaptativas ao autômato, essa palavra era pronunciada como “banána".

A figura 8.1 mostra a interface gráfica assim que clicamos no botão “*Run*”, com a palavra *banana* já digitada, com um espaço no fim da cadeia de entrada. Então a cadeia começa a ser consumida pelo analisador léxico, primeiro o espaço final, depois a palavra seguindo os passos indicado na tabela 5.1. Tanto a pilha quanto o conjunto de regras presentes no início do modificador podem ser observadas na figura 8.2. Então o modificador é chamado, desempilhando todas as sílabas previamente identificadas pelo analisador léxico, conforme a tabela 5.2. Ao fim do modificador, temos a figura 8.3. Finalmente, o autômato passa ao bloco do transdutor, que cria a transcrição fonética, como exibido na tabela 5.3 e na figura 8.4.

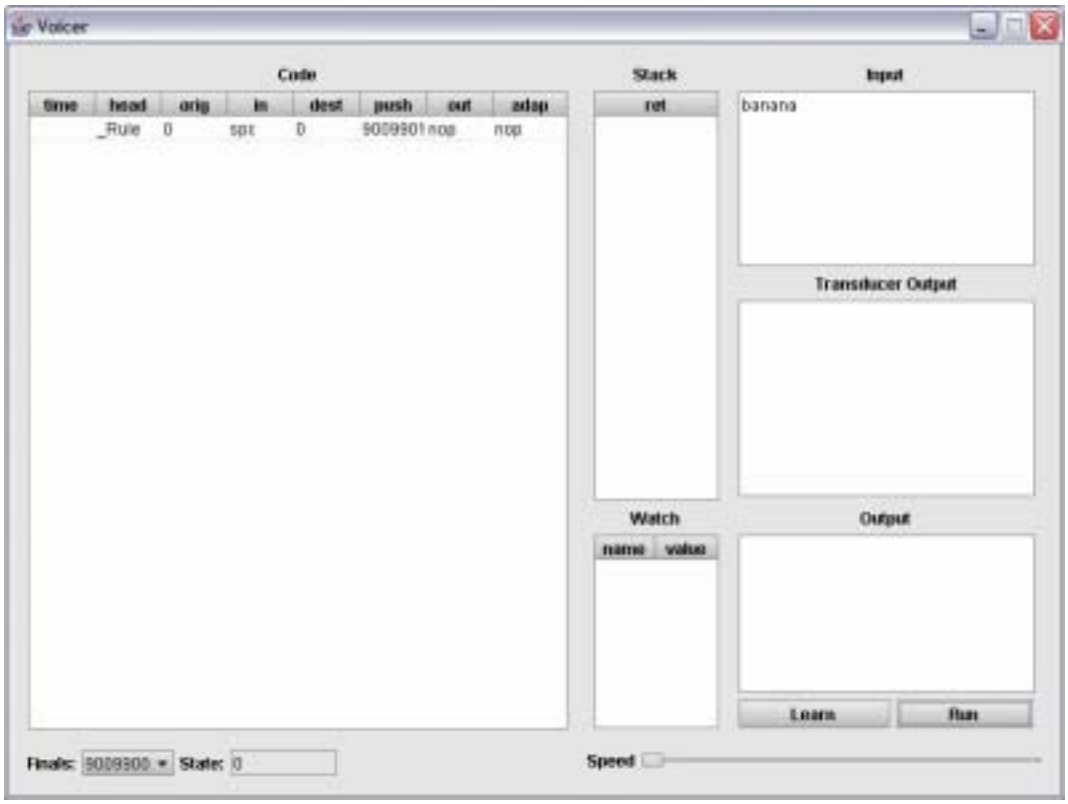

Figura 8.1 –Aparência Inicial do *Voicer* para a cadeia de entrada *banana*

Code

time	head	orig	in	dest	push	out	adap
	_Rule	0	spc	0	9009901	nop	nop
	_Rule	0	spc	0	9009901	nop	nop
	_Rule	0	a	01	nop	nop	nop
	a_Rule	01	n	0	9010008	nop	nop
	_Rule	0	a	01	nop	nop	nop
	a_Rule	01	n	0	9010008	nop	nop
	_Rule	0	a	01	nop	nop	nop
	a_Rule	01	b	0	9010001	nop	nop
	_Rule	0	eps	9009900	nop	nop	nop

Stack

ret
9010001
9010008
9010008
9009901
9009901

Figura 8.2 –Pilha e conjunto de regras após a execução do analisador léxico, para a cadeia de entrada *banana*

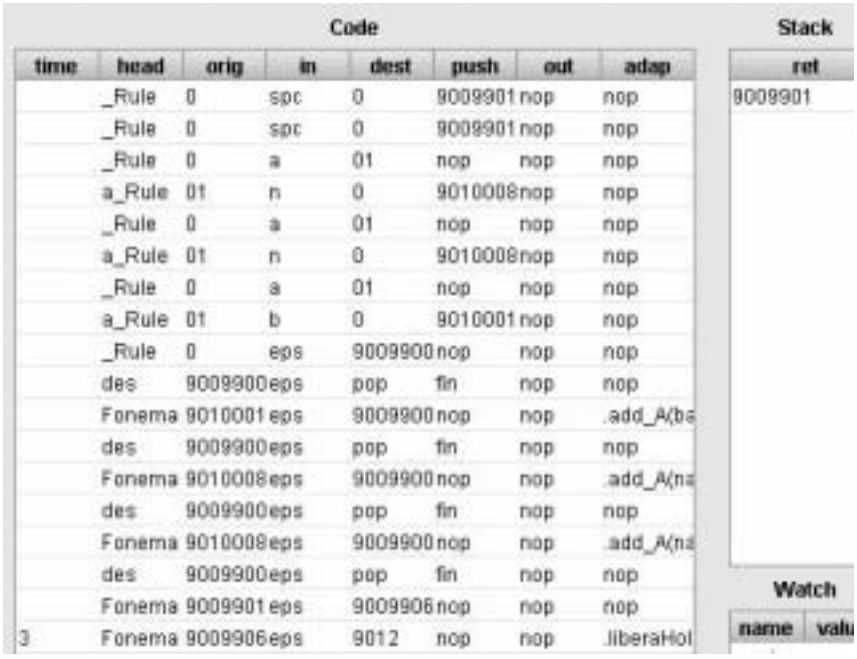

Figura 8.3 –Pilha e conjunto de regras após a execução do modificador, para a cadeia de entrada *banana*

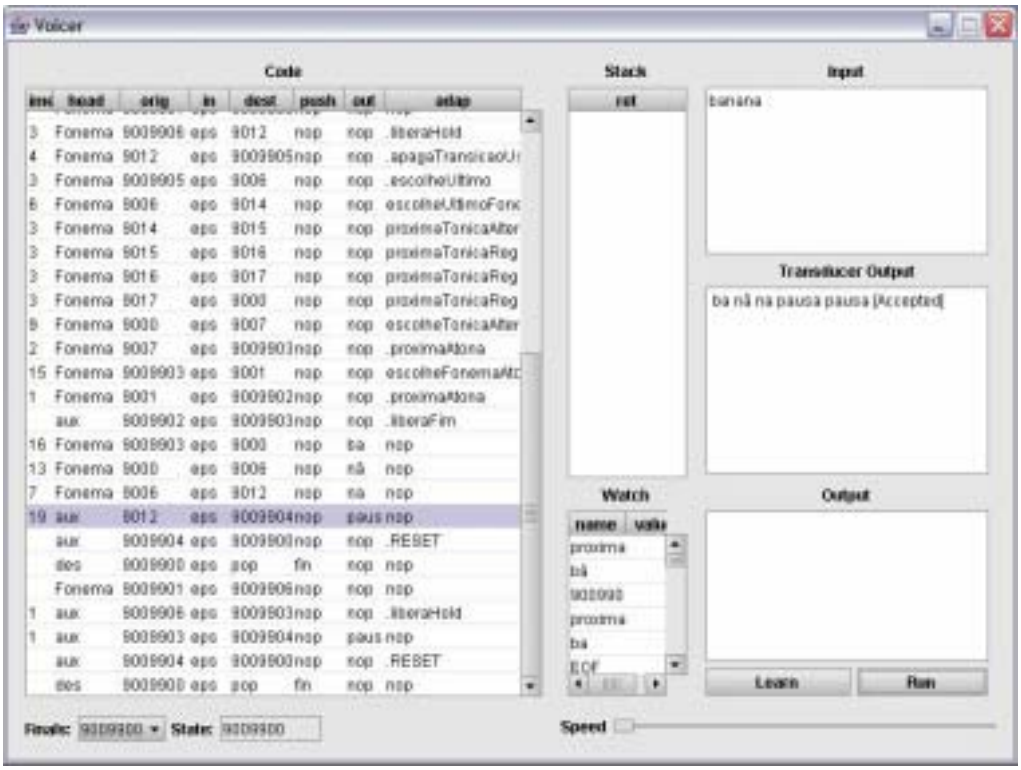

Figura 8.4 –Aparência do *Voicer* após atingir o estado de aceitação com pilha vazia, para a cadeia de entrada *banana*

9 CONCLUSÃO E COMENTÁRIOS FINAIS

A equipe envolvida neste projeto deparou-se com questões cuja solução não era trivial. O não-determinismo na escolha de fonemas representa um exemplo dessa situação.

A utilização de técnicas adaptativas para o reconhecimento de fonemas exigiu uma carga de dedicação muito maior do que a planejada inicialmente. O que como conseqüência também exigiu um estudo tecnicamente pesado para a elaboração da documentação adequada do projeto.

A vantagem utilizada como motivação para o desenvolvimento deste sintetizador foi o fato de a estrutura de um autômato adaptativo poder ser alterada em tempo de execução, possibilitando a geração de diferentes pronúncias para a mesma sílaba.

Do ponto de vista do trabalho em equipe, as qualificações diversificadas de cada membro se complementaram no cumprimento das atividades do projeto. Agregando, sobretudo, a cada membro uma nova experiência do espírito de equipe. Em particular, sobre a eficiência de reuniões realizadas, houve um nível de aproveitamento elevado do trabalho individual para o cumprimento de um objetivo comum no decorrer do projeto, com trabalho à distância de cada membro.

As qualidades e interesses de cada membro foram potencializados ao máximo:

- O aluno Allan Jones, possuindo grande qualificação e competência de desenvolvimento em Java, assumiu as tarefas relacionadas ao desenvolvimento nessa Linguagem, que foram solucionadas em um prazo menor do que seria exigido por qualquer outro membro da equipe.
- O aluno Danilo Shibata, desde o princípio demonstrou maior interesse, conhecimento e motivação em relação ao estudo sobre os autômatos

adaptativos. Dedicou-se com total afinco ao estudo da solução adaptativa que resolvesse o não-determinismo na escolha de fonemas.

- O aluno Daniel Alfenas, altamente especializado na elaboração de documentos técnicos e análise de software, foi decisivo na estruturação do documento principal deste projeto, a monografia do Projeto de Formatura.
- O aluno Henrique Soejima, atuou na documentação do código fonte e geração de artefatos UML (Engenharia de Software) do Sintetizador e foi responsável pelo planejamento e elaboração de cada uma das apresentações parciais ao longo do ano. Foi também, responsável pela gravação e segmentação dos fonemas finais utilizados no Sintetizador.

Na fase de documentação final, houve um engajamento de toda a equipe com a meta de consolidar o conhecimento obtido em forma de documentos, além de registrar os objetivos atingidos e experiência adquirida ao redor deste projeto.

Com respeito às perspectivas de continuidade do projeto, após o término do período letivo, parte da equipe tem planos de prosseguir com melhorias no projeto do Sintetizador aqui apresentado. Os alunos Daniel Assis Alfenas e Danilo Picagli Shibata, apresentaram forte motivação em dar continuidade ao trabalho e o aluno Allan Jones Batista de Castro dispôs-se a apoiar a equipe caso haja o prosseguimento do mesmo. Já o aluno Henrique Tatsuyuki Soejima possui outros planos para o período pós-graduação, mas, caso seja solicitado pode vir a auxiliar em algumas tarefas a equipe.

ANEXO A – MÓDULO MULTIMÍDIA

O presente documento visa apresentar o trabalho da presente equipe para realizar a implementação de uma pequena solução multimídia como projeto solicitado pela disciplina PCS2057 – Hipermídia e Multimídia e sua integração com o Projeto de Formatura do Sintetizador Texto-Voz Com Autômatos Adaptativos.

O documento está dividido basicamente em três partes excluindo-se esta seção introdutória. A saber:

- O Módulo Multimídia E Sua Integração Ao Sintetizador Texto-Voz: A solução multimídia escolhida pela equipe e as idéias e sistemas existentes que embasaram esta solução;
- Considerações Finais: Na última seção, apresentamos os resultados obtidos e os caminhos percorridos até o desenvolvimento do módulo final. Nesta seção efetuamos uma análise das possíveis melhorias que poderiam ser implementadas ao sistema, exigindo, porém, um maior prazo e custo agregados.

A.1 O Módulo Multimídia e sua integração ao Voicer

A idéia a ser desenvolvida foi a de disponibilizar a leitura digital de textos via protocolo HTTP, de modo que um usuário possa acessar o sistema de sintetização texto-voz via Internet.

Assim torna-se possível que um usuário navegando na Internet, tenha a opção de durante a navegação clicar em um link (adequado para tradução texto-voz) e ter o conteúdo deste link lido, para que possa manter suas mãos desocupadas para outra atividade que estiver exercendo, ou estar atentando para outros elementos enquanto ouve o conteúdo do link.

Para que este processo seja possível, é necessário que o usuário instale o Sintetizador Texto-Voz em seu desktop.

O aplicativo original, projetado para as disciplinas PCS2040 e PCS2050 - Projeto de Formatura I e II, respectivamente - foi modificado adicionando-se um módulo que efetua a execução automática do reconhecedor em uma estação cliente da WEB e realiza a síntese da voz de acordo com o texto fornecido para leitura, a partir de um link da Internet.

A motivação desta idéia básica de solução multimídia é fornecer uma forma mais cômoda de obtenção de informação através da Internet.

A.1.1 Soluções similares e alternativas existentes no mercado

Um sistema que representou uma fonte de inspiração para a implementação de nossa solução de tradução texto-voz via Internet foi o sistema de disponibilização de conteúdo via RSS, que é responsável por manter determinado conteúdo da Internet atualizado e disponibilizado em endereço fixo sob um formato XML de forma que terceiros possam utilizar este arquivo para prover conteúdo sempre atual em seus portais ou sistemas.

Como projetar um módulo capaz de efetuar a leitura de uma página qualquer na Internet seria uma tarefa inviável para o prazo fornecido para esta atividade, então buscamos uma solução utilizando um padrão para formatação do conteúdo a ser convertido em voz.

O aplicativo RSS (RDF Site Summary, onde RDF corresponde ao padrão Resource Description Framework de provisão de conteúdo) trouxe à equipe a idéia de utilizar um formato de conteúdo a ser utilizado no reconhecimento de um texto estruturado: o XML.

Como o RSS utiliza um padrão de XML para disponibilização de conteúdo que contém imagem, decidimos em criar um formato particular de XML para ser entendido pelo Voicer e seu conteúdo seria puramente texto, o qual seria lido pelo nosso aplicativo quando o usuário acessá-lo através de um link.

Documentos XML apresentam uma estrutura fácil de ser implementada e disponibilizada por qualquer fornecedor de conteúdo que deseje fornecer ao seu cliente uma forma adicional de conforto no acesso ao seu conteúdo. Para isso poderia ser gerado um módulo “publisher” que seria responsável por obter a informação que o provedor forneça e monte o XML a ser interpretado pelo Voicer instalado no cliente. A versão Publisher não correspondia ao foco do trabalho do grupo e demandaria tempo extra que a nossa equipe não possuía, assim, assumiu-se que o arquivo XML já estaria sendo gerado manualmente para demonstração da tecnologia.

Atualmente, existem alguns provedores de grande porte como o Universo On-Line (UOL) que provê a seus clientes notícias em forma de vídeo. Porém, esse tipo de comunicação é totalmente dependente de uma boa largura de banda para que o cliente possa assistir às notícias via Internet, ou seja, uma parte considerável do público (que não possui banda suficiente) não tem acesso a esse conteúdo.

Nosso módulo multimídia apresenta a lógica de leitura localizada no cliente, assim sendo, não é necessário que ele possua banda larga para que o conteúdo de determinado site seja acessado de forma multimídia. Bastaria ao aplicativo cliente, responsabilizar-se de atualizar o(s) arquivo(s) XML com conteúdo fornecido pelo provedor de informação.

A.2 Aspectos técnicos e implementação

O aplicativo *Voicer* originalmente foi projetado e implementado para comprovar a possibilidade de se construir regras da língua portuguesa através de autômatos de pilha e autômatos adaptativos. A sua arquitetura interna foi implementada de modo que este se torne bastante abrangente e de fácil reutilização, uma vez que o seu núcleo funciona como um compilador que lê regras que descrevem transições de autômatos, e estas regras podem representar qualquer objeto ou modelo desejado. No caso do aplicativo *Voicer*, foi criado um arquivo em formato texto chamado voicer.spa que contém todas as regras descritas da língua portuguesa, funcionando como um compilador de palavras da língua portuguesa brasileira em átomos que correspondem a sílabas fonéticas (por exemplo, a palavra “casa” será compilada para os átomos “ka” e “za”). Após a compilação das palavras ou frases,

todos esses átomos são empilhados na pilha da máquina principal do autômato que ao final serão desempilhados para inferir sua semântica posteriormente. A semântica dada a cada sílaba fonética poderia ser implementada de várias formas, como o uso de síntese de voz, porém como não é objetivo desse projeto englobar módulos de síntese de voz e toda sua complexidade, escolhemos um conjunto bem definido de arquivos de som previamente gravados e que correspondem a todas sílabas fonéticas que são resultado da compilação das frases e palavras na língua portuguesa. Em suma, temos um processo de compilação que gera como resultado um conjunto de átomos, e esses átomos são mapeados para arquivos de som. Esses arquivos de som são chamados pelo aplicativo *Voicer* e tocados em seqüência.

A tela principal do aplicativo *Voicer* possui componentes da interface gráfica que detalham visualmente o comportamento de toda a estrutura da máquina virtual do *Voicer* (figura 1). Temos uma tabela que mostrará todas as transições dos autômatos efetuadas num processo de compilação de uma palavra ou frase. Temos uma caixa de texto onde o usuário pode entrar com qualquer palavra ou frase. Logo abaixo dessa caixa de texto de entrada, temos uma outra caixa de entrada que indicará os átomos de saída no processo de compilação.

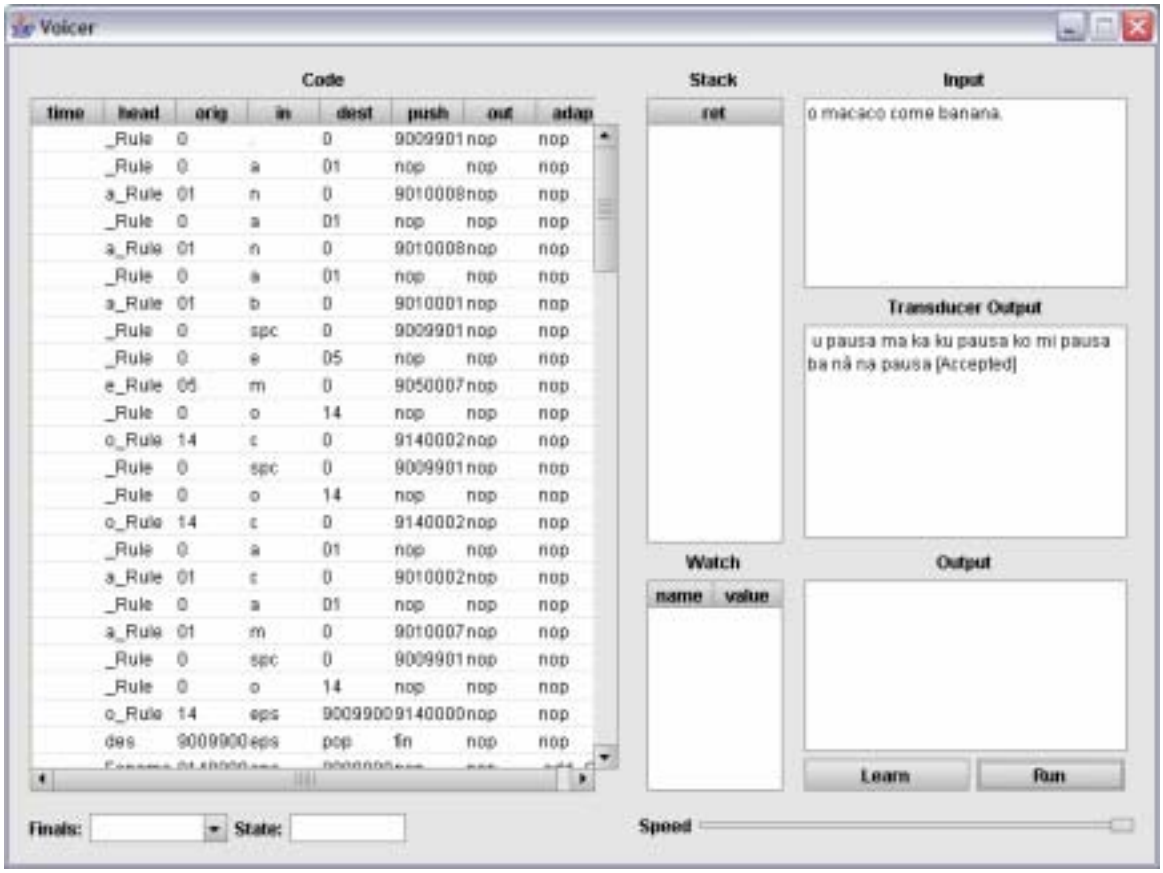

Figura A.1 – Tela principal do aplicativo *Voicer*

Para a utilização comercial ou particular do aplicativo *Voicer*, todos esses detalhes "acadêmicos" presentes na interface gráfica não possuem sentido algum. Dessa forma, existe ainda outra tela mais simplificada que o aplicativo *Voicer* apresenta apenas uma caixa de texto para entrada de palavras ou frases pelo usuário e o botão de executar ou aprender. Isto é mostrado na figura 2.

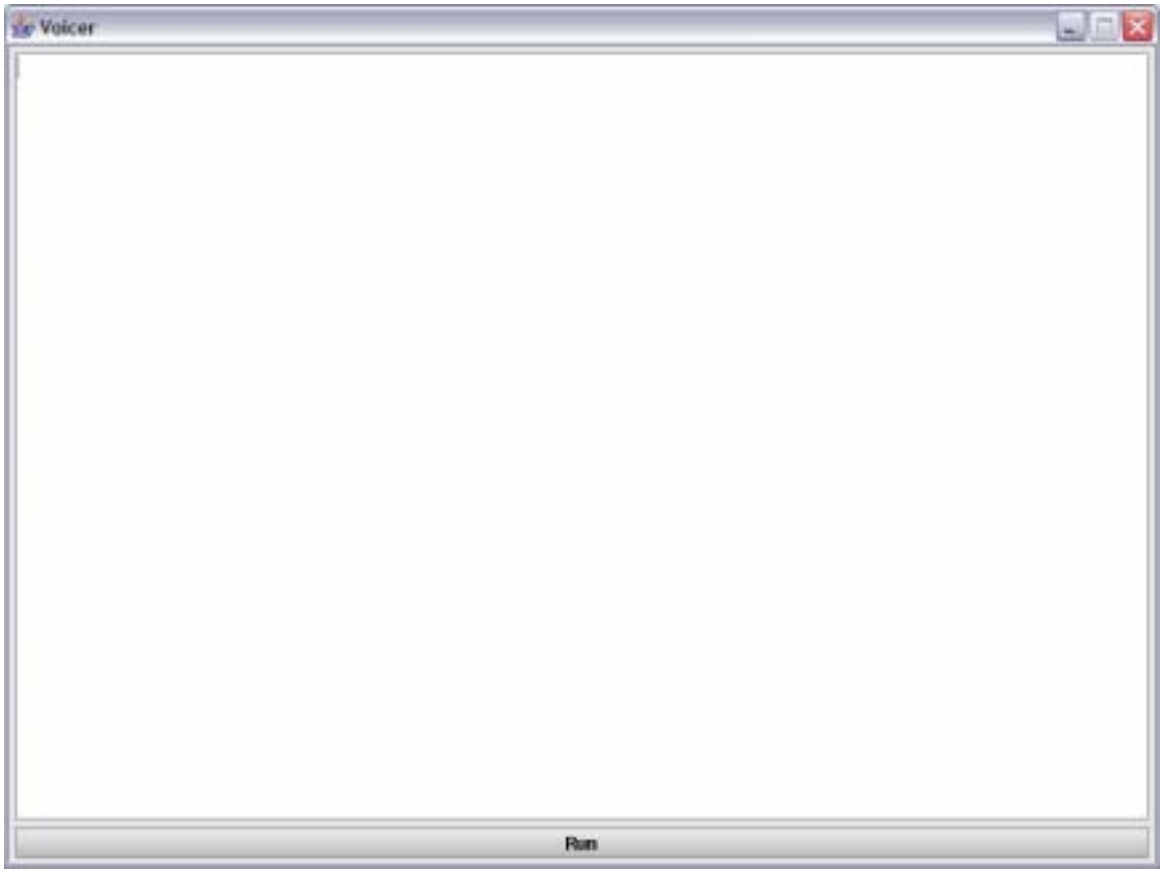

Figura A.2 – Tela simplificada onde o usuário pode digitar o texto a ser falado.

Existe ainda um outro modo de execução desse aplicativo que chega a ser mais interessante para a aplicação prática desse projeto, onde não existe interface gráfica e o programa apenas executa recebendo parâmetros de entrada. Nesse modo não gráfico, o aplicativo decodifica o parâmetro de entrada e fala para o usuário a palavra ou texto correspondente. Para tal foi criado uma extensão do aplicativo *Voicer* onde definimos um protocolo de comunicação onde passamos o conteúdo que o aplicativo deve utilizar para a fala. Esse protocolo de comunicação foi definido com nome *voicer*, e para o usuário ele funciona como um "link" comum de navegação que geralmente encontramos nas páginas da internet ou quando estamos utilizando aplicativos de mensagens instantâneas. Esse "link" aponta para um endereço URL que possui um arquivo XML com o padrão específico do *voicer*. Esse conteúdo XML possui diversos textos identificados por um nome. Um exemplo de arquivo XML encontra-se abaixo:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<voicer>
   <data id="one">
      O macaco come banana.
   </data>
   <data id="two">
      A banana está boa.
   </data>
   <data id="three">
```

Figura A.3 – XML que representa as informações que serão transmitidas via HTTP.

Nesse arquivo XML de exemplo, existem três tópicos que possuem textos identificados por um nome. O tópico identificado pelo nome “one” possui como conteúdo o texto “O macaco come banana.”, assim como o tópico identificado pelo nome “two” possui como conteúdo o texto “A banana está boa.”. Esse arquivo XML seria o arquivo provedor de conteúdo e que pode ser alterado com novos conteúdos ou notícias pela empresa provedora de conteúdo, da mesma forma que o padrão RSS é utilizado.

O “link” pode ser disponibilizado utilizando o protocolo voicer, como mostrado abaixo:

`voicer://http://www.provedorconteudo.com/esportes.xml?futebol`

No exemplo acima, o aplicativo voicer irá vocalizar o tópico de nome futebol que está contindo num arquivo disponibilizado no endereço `http://www.provedorconteudo.com/esportes.xml`.

Esses “links” podem ser disponibilizados em qualquer página através de tags básicas de html, como no exemplo abaixo:

```
<a
href=”voicer://http://www.provedorconteudo.com/esportes
.xml?futebol”> Clique aqui para ouvir as últimas
notícias do futebol</a>
```

Figura A.4 – Link HTML para o conteúdo multimídia a ser requisitado.

A criação desses links foi possível através de adição de chaves no registro do sistema operacional Windows. Ao executar o aplicativo pela primeira vez na máquina do usuário, o aplicativo adiciona ao registro do windows o protocolo *voicer* que permite que qualquer referência que é iniciada com *voicer://* se torne uma referência a um conteúdo que deve ser falado. Abaixo temos um exemplo do que será adicionado ao registro do Windows:

```
Windows Registry Editor Version 5.00

[HKEY_CLASSES_ROOT\voicer]
@="URL: voicer Protocol"
"URL Protocol"=""

[HKEY_CLASSES_ROOT\voicer\shell]

[HKEY_CLASSES_ROOT\voicer\shell\open]

[HKEY_CLASSES_ROOT\voicer\shell\open\command]
@="\"C:\\j2sdk1.4.2_03\\jre\\bin\\JAVAW.EXE\"    \"-cp\"
\"C:\\voicer\\voicer.jar;C:\\voicer\\lib\\commons-
beanutils-1.6.jar;C:\\voicer\\lib\\commons-collections-
2.1.1.jar;C:\\voicer\\lib\\commons-collections-
3.1.jar;C:\\voicer\\lib\\commons-jxpath-
1.2.jar;C:\\voicer\\lib\\commons-logging-
1.0.3.jar;C:\\voicer\\lib\\genesis-client-0.2-
dev.jar;C:\\voicer\\lib\\reusable-components-
20040406.jar;C:\\voicer\\lib\\sounds.zip;C:\\voicer\\lib
\\thinlet.jar\" \"com.coop4.voicer.ui.Console\" \"%1\""
```

Figura A.5 – Exemplo de Conteúdo Adicionado ao Registro do Windows

Após a primeira execução do aplicativo, o computador do usuário possuirá em sua bibliotecas de protocolos URL, mais um protocolo URL chamado voicer. Assim como as referências que iniciam com *http://* ou *https://* apontam para páginas disponíveis na internet, as referências iniciadas com *voicer://* apontarão para conteúdos que devem ser vocalizados.

Isso traz muitos benefícios ao usuário final, e podemos citar um em especial: Aprendizado de crianças. Com a tecnologia presente no aplicativo *Voicer* integrada com a Internet, pode-se fazer sites infantis onde as crianças podem navegar e

aprender a ler e falar. Os sites infantis terão várias fotos e palavras escritas mostrando, através de associação, o significado de cada figura e como se escreve. Disponibilizando nesses sites links entendidos pelo aplicativo *Voicer*, as crianças poderão não somente ver as figuras e palavras no site, mas poderão aprender a falar cada uma dessas palavras, apenas clicando nos links.

Uma evolução e extensão esperada para essa integração WEB do aplicativo *Voicer* seria criar um módulo que convertesse o padrão XML usado pelo RSS para o padrão criado pela nossa equipe. Dessa forma, as pessoas que utilizam programas agregadores de notícia (RSS Aggregators) poderiam instalar apenas um plugin referente ao aplicativo *Voicer***.** Isso possibilitaria que fosse provido conteúdo além de visual, conteúdo falado, trazendo maiores opções e comodidade para o usuário final.

A.3 Considerações Finais

A equipe conseguiu realizar a implementação do módulo multimídia com simplificações. Infelizmente, uma solução *on demand* para a sintetização texto-voz conforme foi inicialmente cogitado, não era possível para o prazo e atividades realizadas em paralelo no meio acadêmico pelos membros da equipe.

Por outro lado, a integração desta atividade com o Projeto de Formatura, pertencente a outra disciplina, representou um fator decisivo para a concretização da solução adotada pela equipe e apresentada neste documento.

A equipe durante o processo de desenvolvimento, em conjunto com os docentes responsáveis pelas avaliações intermediárias, levantou possíveis melhorias que poderiam ser implementadas na apresentação e padrões utilizados pelo sistema.

Algumas destas melhorias correspondem a aspectos que abrangem desde a interface com o usuário, possibilitando facilidades de configuração do módulo, até mecanismos semelhantes à tecnologia PUSH, onde o usuário pode ser informado sobre novidades que seu provedor de conteúdo possa estar disponibilizando via Internet, no momento em que estas notícias são geradas virtualmente.

O projeto tem grande potencial de viabilidade comercial se comparado com a idéia de sistemas que provém conteúdo dinâmico que podem ser disponibilizados em

servidores de terceiros, como o RSS que já é adotado por grandes portais de informação, alguns dos quais conhecidos no mercado brasileiro: http://www.estadao.com.br (portal do Jornal O Estado de S. Paulo), http://www.folha.uol.com.br (portal do Jornal Folha de S. Paulo) que representam os dois maiores jornais de circulação nacional.

O conteúdo é disponibilizado em XML por um aplicativo RSS (Publisher) e o RSS (client) se incumbe de manter o cliente com informações atualizadas dos fornecedores de informação.

ANEXO B – ESPECIFICAÇÃO DE CLASSES

O presente anexo apresenta a especificação de classes do Sintetizador Texto-Voz desenvolvido. É mostrado um diagrama de classes de cada pacote e os métodos e atributos de cada classe existente.

B.1 Package com.coop4.voicer.sound:

Pacote referente às classes responsáveis pelo tratamento sonoro da aplicação. É composto por duas classes somente: um *player* (tocador) de arquivos *waves* e outra classe que implementa a *interface Semantics,* responsável por atribuir semântica aos fonemas passados a ela.

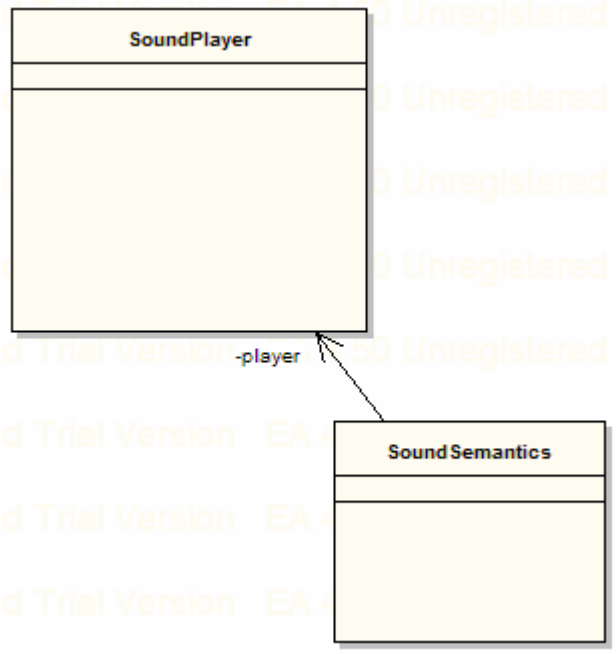

Figura B.1 : com.coop4.voicer.sound

B.1.1 SoundPlayer

public Class: Classe responsável por tocar arquivos de som em formato wave.

Atributo	**Tipo**	**Descrição**
log	private const static : *Log*	Instância de *logging* para realizar diversos níveis de *logging* Valor Inicial: LogFactory.getLog;
extension	private const static : *String*	Extensão dos arquivos de som utilizados Valor Inicial: ".wav";
errorWav	private const static : *String*	Arquivo de som a ser tocado caso não seja encontrado o arquivo de som correto a ser tocado Valor Inicial: "ding";
myClips	private const : *Map*	Mapa contendo os clipes de som a serem tocados Valor Inicial: new HashMap();
Método	**Tipo**	**Descrição**
play (*String*)	public: *void*	Toca o arquivo de som cujo nome é fornecido como parâmetro, aguardando que o arquivo seja completamente tocado antes finalmente retornar da função.
getClip (*String*)	private: *Clip*	Obtém e retorna um clipe de som, carregando o arquivo se não foi utilizado anteriormente ou utilizando o arquivo previamente carregado e presente no mapa de arquivos de som carregados.
stopClip (*Clip*)	private: *void*	Finaliza a execução do clipe de áudio fornecido como parâmetro
getAudio (*String*)	private: *InputStream*	Obtém o *stream* de áudio através do nome normalizado do arquivo de som. Os arquivos de som devem estar presentes no *classpath* da aplicação.
normalize (*String*)	private: *String*	Normaliza o nome do arquivo de som para que seu arquivo de áudio possa ser localizado pelo *SoundPlayer* através do

		classpath da aplicação.

B.1.2 SoundSemantics

public Class Implements Semantics: Dá sentido aos fonemas reconhecidos (transforma texto em voz), fornecendo métodos para tocar os arquivos de fonemas.

Atributo	**Tipo**	**Descrição**
player	private const : *SoundPlayer*	Instância de nosso *player* Java
Método	**Tipo**	**Descrição**
SoundSemantics ()	public:	Construtor Padrão da Classe. Apenas instancia um novo *player*
execute (*String*)	public: *Token*	Método para acesso à execução de um som referente ao fonema cujo nome é passado como parâmetro
say (*String*)	private: *Token*	Método que representa a fala do Sintetizador Texto-Voz, tocando um fonema e aguardando o final de sua execução.

B.2 Package com.coop4.voicer.ui

Pacote referente às classes de interface gráfica com o usuário. É utilizado como *API* gráfica o componente *Thinlet* onde podemos definir os componentes da interface gráfica e sua disposição na tela através de um arquivo *XML* muito parecido com o padrão *HTML* para páginas na *WEB*.

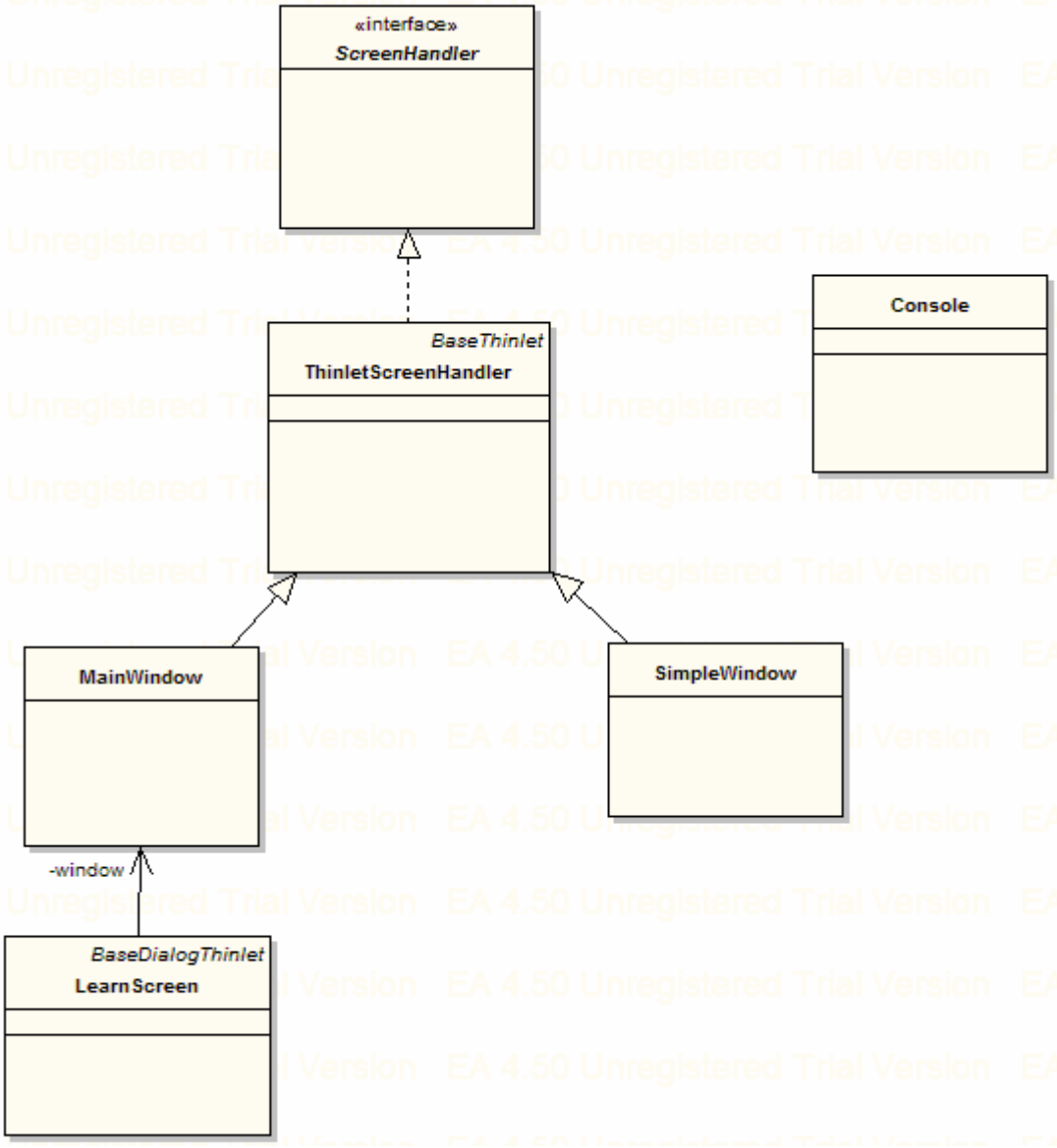

Figura B.2: com.coop4.voicer.ui

B.2.1 Console

public Class: Classe para execução do *Voicer* em modo console. Não possui nenhuma interface gráfica, e recebe como parâmetro uma *String* que representa um *voicer link* (exemplo: *voicer://http://www.noticias.com.br/esportes?news*).

Atributo	Tipo	Descrição

vmds	private const : *Vmds*	Instância da *virtual machine data structure*
Método	**Tipo**	**Descrição**
Console ()	public:	Construtor padrão do console que instancia uma *virtual machine data structure* e a inicializa.
say (*String*)	private: *void*	Método que pronuncia o texto passado como parâmetro
main (*String[]*)	public static: *void*	Método *main* para execução dessa classe

B.2.2 LearnScreen

Public Class Implements BaseDialogThinlet: Define a tela de aprendizado do aplicativo. Essa tela possui apenas dois campos de texto e dois botões, onde o usuário pode cadastrar uma regra de exceção, testá-la e armazená-la.

Atributo	**Tipo**	**Descrição**
window	private const : *MainWindow*	Instância da janela principal do aplicativo que é pai dessa janela.
oldValue	private const : *Object*	Campo com a pronúncia antiga.
newValue	private const : *Object*	Campo para a entrada da nova pronúncia.
Método	**Tipo**	**Descrição**
LearnScreen (*MainWindow*)	public:	Construtor que faz o *parse* do arquivo que define a interface gráfica.
learn ()	public: *void*	Método de aprendizado e representa o botão *learn* da tela.
valid ()	private: *boolean*	Verifica se o usuário digitou pelo menos algum texto em ambos os campos da tela.
test ()	public: *void*	Toca o som para teste antes do

		aprendizado

B.2.3 MainWindow

Public Class Extends ThinletScreenHandler: A classe que representa a janela principal e acadêmica da aplicação

Método	**Tipo**	**Descrição**
MainWindow ()	public:	Bloco estático para fazer o registro da aplicação no windows
go ()	public: *void*	Método que executa a *virtual machine data structure* o que provoca a pronuncia do texto presente no campo de entrada na interface gráfica
learn ()	public: *void*	Método que mostra a tela de aprendizado
main (*String[]*)	public static: *void*	Método *main* da classe

B.2.4 SimpleWindow

Public Class Extends ThinletScreenHandler: Classe simplificada da interface gráfica com o usuário. A tela que essa classe representa possui apenas o campo de texto de entrada e o botão de execução do aplicativo. Pode-se dizer que essa classe é um sub conjunto do que é apresentado na classe MainWindow.

Método	**Tipo**	**Descrição**
SimpleWindow ()	public:	Construtor padrão que faz o *parse* do arquivo XML da tela
go ()	public: *void*	Método de execução que faz a pronuncia da palavra de entrada do usuário
main (*String[]*)	public static: *void*	Método *main* da classe

B.2.5 ThinletScreenHandler

Public Class Implements ScreenHandler, BaseThinlet: *ScreenHandler* implementado utilizando como *API* gráfica *AWT* através do projeto *Thinlet*.

Atributo	**Tipo**	**Descrição**
rules	private const : *Object*	Objeto da interface gráfica que representa a tabela de regras
finals	private const : *Object*	Objeto da interface gráfica que representa a lista de estados finais
state	private const : *Object*	Objeto da interface gráfica que representa o estado atual.
stackTable	private const : *Object*	Objeto da interface gráfica que representa a pilha
watchTable	private const : *Object*	Objeto da interface gráfica que representa a tabela de variáveis.
speedSlider	private const : *Object*	Objeto da interface gráfica que representa a barra de rolagem referente a velocidade de execução
inputText	private const : *Object*	Objeto da interface gráfica que representa o texto de entrada do usuário
outputText	private const : *Object*	Objeto da interface gráfica que representa o texto de saída
transducerText	private const : *Object*	Objeto da interface gráfica que representa o transdutor
runButton	private const : *Object*	Objeto da interface gráfica que representa o botão de execução
rulesProperties	private const : *List*	Objeto da interface gráfica que representa os nomes das colunas da tabela de regras
rulesRows	private const : *List*	Objeto da interface gráfica que representa as linhas da tabela de regras.

vmds	private const : *Vmds*	Instância da *virtual machine data structure*
Método	**Tipo**	**Descrição**
ThinletScreenHandler (*String*)	public:	Construtor que recebe como parâmetro o nome do arquivo *XML* da interface gráfica e já armazena nas instâncias de classe todos os objetos que serão utilizados posteriormente, não necessitando buscá-los no momento de execução. Também há código para centralizar a janela.
getVmds ()	protected: *Vmds*	Retorna a instância da *virtual machine data structure* associada à execução do *Voicer*
initVmds ()	private: *void*	Método usado para inicializar a *virtual machine data structure*
populateFromMap (*Object, Map*)	private: *void*	Método auxiliar que dado um mapa qualquer, preenche a tabela representada pelo objeto de interface gráfica passado como parâmetro
syncTable ()	public: *void*	Método que sincroniza a tabela de regras, removendo as regras da tabela e limpando todas as linhas da tabela.
selectRule (*Rule, boolean*)	public: *void*	Método que adiciona a regra passada como parâmetro à tabela de regras. Se estiver no modo passo a passo (*debug on*) a linha referente a regra é adicionada no mesmo momento. Caso contrário é apenas armazenada em cache para adicionar todas juntas posteriormente.
showRules ()	public: *void*	Mostra todas as regras que foram armazenadas em *cache* na tabela de regras
setFinals (*Collection*)	public: *void*	Armazena na tela a coleção de estados finais passados como parâmetro.
appendFinal (*String*)	public: *void*	Adiciona o estado final passado como parâmetro à lista de estados finais já

		existentes.
appendOutputText (*String*)	public: *void*	Concatena o texto passado como parâmetro ao resto do texto de saída (*output*)
appendTransducerText (*String*)	public: *void*	Concatena o texto passado como parâmetro ao resto do texto do *transdutor* (*transducer output*)
setInputText (*String*)	public: *void*	Armazena o texto passado como parâmetro como o texto de entrada (*input*)
setOutputText (*String*)	public: *void*	Armazena o texto passado como parâmetro como o texto de saída (*output*)
setSpeed (*int*)	public: *void*	Armazena uma nova velocidade
setState (*int*)	public: *void*	Armazena um novo estado
setTransducerText (*String*)	public: *void*	Armazena o texto passado como parâmetro para o transdutor (*transducer output*)
setWatch (*Map*)	public: *void*	Popula a tabela referente as variáveis com o novo mapa de variáveis passado como parâmetro
stackPop ()	public: *void*	Desempilha o último elemento da pilha
stackPush (*int*)	public: *void*	Empilha o estado na pilha
getInputText ()	public: *String*	Retorna o texto de entrada do usuário (*input*)
getSpeed ()	public: *int*	Retorna a velocidade atual de execução do *Voicer*
setRunEnabled (*boolean*)	public: *void*	Habilita ou desabilita o botão *run*.
onClose ()	private: *void*	Método chamado quando a aplicação é encerrada. Ele deve chamar a classe *Knowledge* e provocar a escrita de todas as exceções presentes em memória em um arquivo.

B.2.6 ScreenHandler

public abstract «interface» Interface: Interface que auxilia a sincronia de dados das telas com a *virtual machine data structure*. Assim, sempre que for alterada alguma entrada na tela, a *virtual machine data structure* será automaticamente alterada também.

Método	**Tipo**	**Descrição**
showRules ()	public: *void*	Método que deve mostrar todo o conjunto de regras armazenado em *cache* na tabela para o usuário visualizar
selectRule (*Rule, boolean*)	public: *void*	Método que deve chamar a atenção do usuário para determinada regra adicionando a regra ao final da lista de regras ou, alternativamente, colorindo a regra.
setFinals (*Collection*)	public: *void*	Método que dado uma lista de estados finais, deve mostrá-los na tela
appendFinal (*String*)	public: *void*	Método que deve adicionar um estado final à lista de estados finais presente da tela
setState (*int*)	public: *void*	Método que deve mostrar na tela o estado atual do autômato
setInputText (*String*)	public: *void*	Método que deve mostrar no campo de entrada do usuário o texto passado como parâmetro
setOutputText (*String*)	public: *void*	Método que deve mostrar no campo de saída o texto passado como parâmetro
setTransducerText (*String*)	public: *void*	Método que deve mostrar no campo transdutor o texto passado como parâmetro
appendOutputText (*String*)	public: *void*	Método que deve adicionar ao texto já existente do campo de saída, o texto passado como parâmetro.

appendTransducerText (*String*)	public: *void*	Método que deve adicionar ao texto já existente do campo transdutor, o texto passado como parâmetro.
setSpeed (*int*)	public: *void*	Método que deve mover a barra de velocidade para a velocidade passada como parâmetro
stackPop ()	public: *void*	Método que deve remover o topo da tabela que representa a pilha de estados
stackPush (*int*)	public: *void*	Método que deve adicionar o estado passado como parâmetro ao topo da pilha de estados representada graficamente por uma tabela de uma coluna
setWatch (*Map*)	public: *void*	Método que deve mostrar na tabela de variáveis o mapa passado como parâmetro
getInputText ()	public: *String*	Método que deve retorna o texto de entrada da tela.
getSpeed ()	public: *int*	Método que deve retornar a velocidade de execução do *Voicer*
syncTable ()	public: *void*	Método que deve sincronizar os dados da tela
setRunEnabled (*boolean*)	public: *void*	Método que deve habilitar ou desabilitar o botão *run* da tela

B.3 Package com.coop4.voicer.util

Pacote referente às classe utilitárias, que possuem funcionalidades úteis em diversos pontos do sistema.

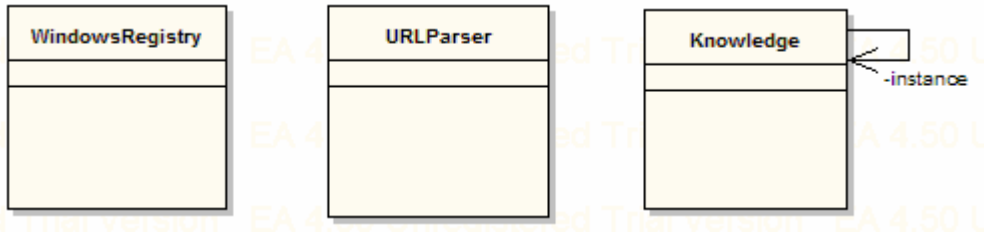

Figura B.3: com.coop4.voicer.util

B.3.1 Knowledge

public Class: Classe *singleton Knowledge* responsável pelo registro de todas as exceções da língua portuguesa. As exceções podem ser registradas pelo usuário do *Voicer* e são armazenadas em um arquivo de nome '*.knowledge*'. Essa classe é responsável por substituir palavras ou frases de entrada por suas exceções. Dessa forma as exceções já aprendidas serão faladas corretamente.

Atributo	Tipo	Descrição
log	private const static : *Log*	Instância de *logging* dessa classe Valor Inicial: LogFactory.getLog;
instance	private const static : *Knowledge*	Instância única dessa classe Valor Inicial: new Knowledge();
file	private const : *File*	Instância do arquivo.*knowledge*
knowledge	private const : *Map*	Mapa contendo todas as exceções registradas. A chave é a palavra em letras minúsculas e a chave é o modo que essa palavra é pronunciada, guardada também em letras minúsculas.

		Valor Inicial: new HashMap();
Método	**Tipo**	**Descrição**
Knowledge ()	private:	Construtor padrão e privado
getInstance ()	public static: *Knowledge*	Retorna a instância *singleton* dessa classe. É a única forma de se obter uma instância dessa classe nessa aplicação
init ()	private: *void*	Método que inicializa a leitura do arquivo *.knowlegde* para armazenar em memória todas as exceções previamente cadastradas. Se o arquivo não existir nada é feito.
destroy ()	public: *void*	Método que deve ser chamado antes da aplicação ser fechada. Ao ser chamado ele registra todas as exceções presentes em memória para o arquivo *.knowledge*, dessa forma, tudo o que foi registrado persistirá e poderá ser utilizado em outras execuções do programa.
getWord (*String*)	private: *String*	Dada uma palavra ele retorna outra palavra corrigida com as exceções. Por exemplo, se a palavra for *cebola* e nas regras de exceções estiver cadastrado *cebola=cebôla* a *String* a ser retornada será *cebôla*.
doYouKnow (*String*)	public: *String*	Dada uma palavra ou frase ele retorna outra palavra ou frase corrigida de acordo com as exceções. Por exemplo, se a frase for "*Uma cebola.*" e nas regras de exceções estiver cadastrado *cebola=cebôla* a *String* a ser retornada será "*Uma cebôla*."
learn (*String*, *String*)	public: *void*	Método de aprendizado onde são passados como parâmetros a palavra a ser aprendida e a forma como essa palavra deve ser pronunciada.

B.3.2 URLParser

public Class: Classe utilizada para fazer o *parser* dos *voicer links*. Os *voicer links* são registrados no registro do windows como protocolos *URL*. Ao clicarem em seus *links* o *Voicer* será executado em modo console e o som referenciado pelo link será pronunciado.

Atributo	**Tipo**	**Descrição**
id	private const : *String*	*String* id que identifica qual o texto a ser falado dentro de um arquivo *XML*.
url	private const : *URL*	*URL* que referencia o endereço do arquivo *XML* de conteúdo do *Voicer*
root	private : *Container*	Container utilizado pelo *JXPath* Valor Inicial: null;
text	private : *String*	Texto a ser falado
Método	**Tipo**	**Descrição**
URLParser (*String*)	public:	O parâmetro passado no construtor é a string que representa o *voicer link*. Exemplo de *voicer link*: *voicer://file:///F:/teste.xml?one voicer://http://www.somehost.com/some data.xml?news*
getRoot ()	public: *Container*	Root utilizado pelo *JXPath*
getText ()	public: *String*	Retorna o texto a ser lido

B.3.3 WindowsRegistry

public Class: Classe responsável por guardar no registro do windows o conteúdo necessário para reconhecer os *voicer links*.

Atributo	**Tipo**	**Descrição**

CMD1	public const static : *String*	Primeiro comando a ser registrado Valor Inicial: "REG ADD HKEY_CLASSES_ROOT\\voicer "+"/ve /d \"URL: voicer Protocol\" /f";
CMD2	public const static : *String*	Segundo comando a ser registrado Valor Inicial: "REG ADD HKEY_CLASSES_ROOT\\voicer "+"/v \"URL Protocol\" /f";
CMD3	public const static : *String*	Terceiro comando a ser registrado. Indica como executar o voicer. Valor Inicial: "REG ADD HKEY_CLASSES_ROOT\\voicer\\shell\\open\\command "+"/ve /d \"\\\"{0}\\bin\\JAVAW.EXE\\\" \\\"-cp\\\" \\\"{1}\\\" \\\"{2}\\\" \\\"%1\\\"\" /f";
Método	**Tipo**	**Descrição**
install ()	public static: *void*	Instala os *voicer links* através de adição de conteúdo ao registro do windows. Se o sistema operacional não for windows nada é feito
normalizeClassPath ()	private static: *String*	Normaliza o *classpath* para executar o *Voicer*

B.4 Package com.coop4.voicer.vm

Pacote referente às classes da *virtual machine*. As classes foram originalmente baseadas em classes do aplicativo *Adaptools.*

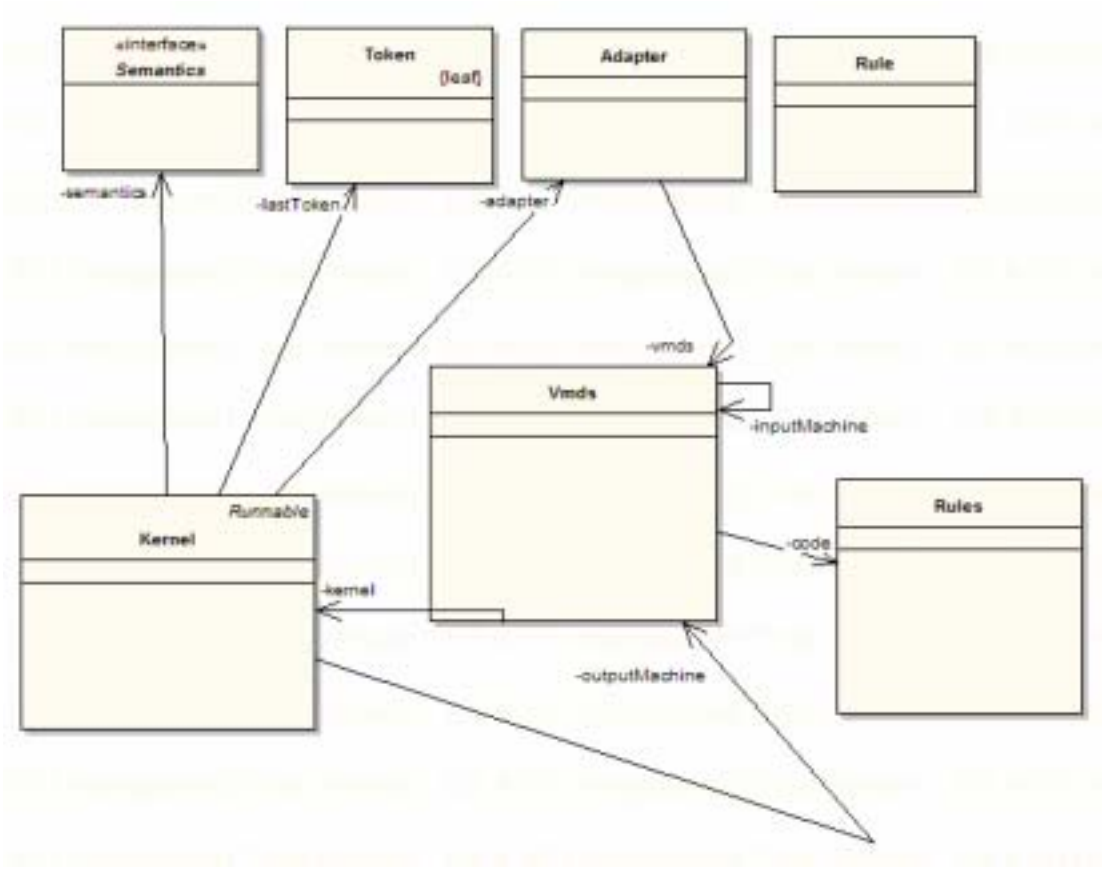

Figura B.4: com.coop4.voicer.vm

B.4.1 Adapter

public Class: Classe responsável pela lógica de Adaptatividade do reconhecedor de fonemas.

Atributo	Tipo	Descrição
vmds	private const : *Vmds*	Instância da estrutura de dados da Máquina Virtual do Sintetizador
insActions	private : *List*	Lista com os números das linhas das ações de inserção primitivas

remActions	private : *List*	Lista com os números das linhas das ações de remoção primitivas
queActions	private : *List*	Lista com os números das linhas das ações de consulta/busca primitivas
bindingsList	private : *List*	Lista com as variáveis criadas pelas ações de consulta/busca primitivas
nextGenState	private : *int*	Representa o próximo estado do autômato Valor Inicial: 9000;
time	private : *int*	Timing (número seqüencial) de controle de execução de máquinas e suas sub-máquinas - incrementado quando houver chamadas de sub-máquinas Valor Inicial: 0;
subMachineName	private : *String*	Contém a regra que iniciou determinada função adaptativa
actionName	private : *String*	Contém o campo com nome da ação
actionParameters	private : *Collection*	Contém os Parâmetros relacionados à ação corrente
Método	**Tipo**	**Descrição**
Adapter (*Vmds*)	public:	Construtor que deve receber a estrutura de dados da Máquina Virtual do Sintetizador para inicialização.
execute (*String, String*)	public: *void*	Efetua a execução de máquina/sub-máquina do autômato
extractActions ()	private: *void*	Realiza a análise dá acão a ser realizada e realiza a correspondente inserção, remoção ou consulta à regras
addSystemInformation (*Map*)	private: *void*	Adiciona ao mapa de bindings o estado corrente da Máquina Virtual e a entrada lida
addParametersInformat	private: *void*	Adiciona parâmetros ao mapa de bindings

ion (*Map*)		
executeQueryActions ()	private: *void*	Executa ação de Consulta/Busca
executeQueryAction (*int, Map, List*)	private: *void*	Executa ação de Consulta/Busca
executeRemoveActions ()	private: *void*	Realiza a remoção de regras
executeInsertActions ()	private: *void*	Realiza a chamada da inserção de Regras a um Mapa de *bindings*
insertRule (*Rule, Map*)	private: *void*	Insere uma regra efetivamente, atualizando a máquina virtual em execução. Verifica se a regra action passada como parâmetro tem variáveis internas ou, caso contrário, seta a ação adaptativa a executar.
substActionField (*Rule, Rule, Map*)	private: *void*	Efetua substituições de variáveis dentro de uma ação adaptativa
existNullBinding (*Rule, Map*)	private: *boolean*	Verifica se na lista de *bindings* apresenta algum valor nulo para determinado *actionField*
removeRule (*Rule*)	private: *void*	Remove uma regra
getTime ()	private: *String*	Retorna o valor da variável de "*timing*" de máquinas e sub-máquinas
unify (*Rule, Rule, Map*)	private: *boolean*	Realiza a unificação, tratando o caso especial de existir variáveis internas em uma ação adaptativa.
unifyActionField (*String, String, Map*)	private: *void*	Efetua união no caso especial do campo da ação adaptativa em que pode haver variáveis internas
isBinded (*String, Map*)	private: *boolean*	Verifica se determinado campo esta presente no Mapa de *bindings*
bindedValue (*String, Map*)	private: *String*	Passando-se o nome de um campo (chave) de interesse, encontra-se o valor dessa chave

isUnifiable (*Rule, Rule*)	private: *boolean*	Verifica se é possível a unificação dados *action* e *rule*
isUnifiable (*Rule, Rule, Map*)	private: *boolean*	Verifica se é possível a unificação a partir do *action*, *rule* e mapa de *bindings*
isUnifiableActionField (*String, String, Map*)	private: *boolean*	Testa se é possível a unificação do campo da ação adaptativa (que deve ser tratado como um caso especial devido à possibilidade de ocorrência de variáveis internas)

B.4.2 Kernel

Public Class_Implement Runnable: Classe que contém as chamadas para as principais funções de controle dos autômatos utilizados no reconhecimento de textos, realiza também o controle do que será exibido pela máquina virtual. Implementa a interface *Runnable*

Atributo	**Tipo**	**Descrição**
log	private const static : *Log*	Log do aplicativo para controle de erros Valor Inicial: LogFactory.getLog;
vmds	private const : *Vmds*	Instância da máquina virtual em execução (aplicativo)
adapter	private const : *Adapter*	Instância da classe *Adapter* com os métodos de tratamento adaptativo para reconhecimento de fonemas
semantics	private const : *Semantics*	Instância da interface semântica (mapea os fonemas de texto para arquivos de voz gravados - Interface *Semantics*)
lastToken	private : *Token*	último token lido Valor Inicial: null;
hasToken	private : *boolean*	sinaliza se existe um token Valor Inicial: false;

toBeExecuted	private const : *List*	Lista com as regras a serem executadas
contaLoop	private : *int*	contador
running	private : *boolean*	*flag* que indica reconhecimento de texto em andamento
Método	**Tipo**	**Descrição**
Kernel (*Vmds*)	public:	Construtor que efetua as inicializações do Aplicativo
run ()	public: *void*	Inicia o processo de reconhecimento do texto
isRunning ()	public: *boolean*	Indica se o processo de reconhecimento de fonemas está em andamento
stepByStep ()	private: *void*	Método que efetua as principais chamadas de máquina e sub-máquinas de reconhecimento, utilizando intensivamente para isso as classes *Rule* e *Adapter*
getNextRule ()	private: *Rule*	Obtém a próxima regra
aheadIsOK (*Rule*)	private: *boolean*	Mecanismo de *Lookahead*. Retorna *true* se há como prosseguir no reconhecimento, ou *false*, caso contrário (se não há outro *token* a ser lido ou se não há um retorno de alguma sub-máquina a ser tratada)
getDestValue (*Rule*)	private: *int*	Obtém o valor do campo de Destino de uma regra
tryToExecute (*Rule*)	private: *void*	Simula a execução de uma regra, não realizando as operações de mudança de estado e empilhamento e desempilhamento efetivamente.
executeList ()	private: *void*	Executa a lista de regras
setLastToken (*Token*)	private: *void*	Se não há nenhum token sendo tratado define-se o novo token
getLastToken ()	public: *Token*	Obtém o último token lido

waitUser ()	private: *void*	Faz com que o sistema aguarde uma ação do usuário.
userClicked ()	public: *void*	Método executado quando o usuário clica para continuar a execução.
tokenRead ()	public: *void*	Método chamado externamente para indicar que o último *token* lido foi consumido.
accepted ()	private: *boolean*	Indica se o texto inserido foi completamente reconhecido e aceito (validação léxica). Retorna *true* se a cadeia de caracteres foi aceita (texto válido), ou *false*, caso o texto não seja aceito.

B.4.3 Rule

public Class: Classe que representa uma Regra que faz parte do processo de reconhecimento de fonemas do Sintetizador Texto-Voz.

Atributo	**Tipo**	**Descrição**
index	private const : *int*	Índice associado a uma instância da regra
time	private : *String*	*Timing* das regras (profundidade da regra)
header	private : *String*	Representa o *header* típico de uma regra
origin	private : *String*	Representa o campo de estado de origem de uma regra
input	private : *String*	Representa o valor da entrada à qual a regra é sensível
destination	private : *String*	Representa o estado de destino de uma regra
push	private : *String*	Conteúdo a ser empilhado

output	private : *String*	Conteúdo de saída
adaptiveAction	private : *String*	Ação Adaptativa
unifiable	private : *boolean*	*flag* que indica a possibilidade de unificação entre regras
Método	**Tipo**	**Descrição**
Rule (*int*)	public:	Construtor que associa um índice a uma nova instância de uma regra
getTime ()	public: *String*	Retorna o timing da regra (em formato de String)
setTime (*String*)	public: *void*	Armazena o timing de uma regra (valores inteiros e seqüenciais)
getHeader ()	public: *String*	Obtém a variável que representa o *header* da regra
setHeader (*String*)	public: *void*	Define o cabeçalho desta regra
getOrigin ()	public: *String*	Obtém o estado de Origem
setOrigin (*String*)	public: *void*	Define o estado de Origem da regra
getInput ()	public: *String*	Obtém a entrada à qual a regra é sensível
setInput (*String*)	public: *void*	Define a entrada à qual a regra é sensível
getDestination ()	public: *String*	Obtém o estado de destino da regra
setDestination (*String*)	public: *void*	Define qual o estado de destino da regra
getPush ()	public: *String*	Obtém o Estado a ser empilhado
setPush (*String*)	public: *void*	Define o estado a ser empilhado por esta regra instanciada
getOutput ()	public: *String*	Obtém o valor de saída desta regra
setOutput (*String*)	public: *void*	Define o Valor de saída da regra atual
getAdaptiveAction ()	public: *String*	Obtém o valor da ação adaptativa a ser

		executada
setAdaptiveAction (*String*)	public: *void*	Atribui o valor da Ação Adaptativa
isUnifiable ()	public: *boolean*	Retorna o valor da *flag unifiable*
setUnifiable (*String*)	public: *void*	Atribui o valor *boolean* da *flag unifiable*
getIndex ()	public: *int*	Obtém o índice associado à regra instanciada
finalState ()	public: *String*	Obtém a *String* representando o estado final de acordo com o retorno da Sub-máquina
hasFinal ()	public: *boolean*	Retorna *true* se não for chamada de função adaptativa e estiver empilhando o estado "*fin*"
remove ()	public: *void*	Insere o caractere # no início da regra para que a regra seja ignorada (removida)
undoRemove ()	public: *void*	Elimina o símbolo # do início da regra para que ela deixe de ser ignorada
reads (*Token*)	public: *boolean*	Retorna *true* se o *token* passado como parâmetro for lido
inputIsRange ()	private: *boolean*	Caso a entrada seja um intervalo de valores retorna *true*, ou, caso contrário, retorna *false*
inRange (*String, String*)	private: *boolean*	Verifica se um caractere está em um certo intervalo de caracteres.
space (*String*)	private: *boolean*	Verifica se a string passada é um espaço em branco ou qualquer caractere que represente um espaço em branco
inputIsLetter ()	private: *boolean*	Verifica se a entrada é uma letra
inputIsDigit ()	private: *boolean*	Verifica se o input é a palavra reservada *digit*
inputIsSpecial ()	private:	Verifica se o input é a palavra reservada

	boolean	*special*
inputIsSpace ()	private: *boolean*	Verifica se o input é a palavra reservada *spc*
other ()	public: *boolean*	Verifica se o input é a palavra reservada *other*
special (*String*)	private: *boolean*	Verifica se a string não é um caractere especial.
hasOutput ()	public: *boolean*	Verifica se a regra possui *output*, ou seja, não é a palavra reservada *nop*
subMachineCall ()	public: *boolean*	Verifica se não é uma chamada a uma sub máquina
removed ()	public: *boolean*	Verifica se a regra foi removida
valid ()	public: *boolean*	Verifica se a regra é uma regra válida
emptyJump ()	public: *boolean*	Verifica se a regra é uma transição vazia
errorJump ()	public: *boolean*	Verifica se a ação adaptativa é um erro
subMachineReturn ()	public: *boolean*	Verifica se é um retorno de chamada de sub máquina
function ()	public: *boolean*	Verifica se é uma função
memberOf (*String*)	public: *boolean*	Verifica se é uma ação adaptativa
hasAfterAction ()	public: *boolean*	Verifica se possui uma ação adaptativa a ser executada posteriormente
hasBeforeAction ()	public: *boolean*	Verifica se possui uma ação adaptativa a ser executada anteriormente
afterAction ()	public: *String*	Retorna o nome da ação adaptativa
beforeAction ()	public: *String*	Retorna o nome da ação adaptativa
query ()	public: *boolean*	Verifica se a regra se trata de uma ação adaptativa de consulta @return
insertion ()	public: *boolean*	Verifica se a regra se trata de uma ação adaptativa de inserção

remotion ()	public: *boolean*	Verifica se a regra se trata de uma ação adaptativa de remoção
actionName (*String*)	public static: *String*	Retorna o nome da ação retirando os parênteses
parseActionField (*String*)	public static: *List*	Faz o *parse* do campo de ação
actionParameters (*String*)	public static: *List*	Retorna uma lista com os parâmetros da ação
parseList (*String*)	private static: *List*	Faz o *parse* da lista de parâmetros
hasInternalVar (*String*)	public static: *boolean*	Verifica se possui uma variável interna a ação
isQueryVar (*String*)	public static: *boolean*	Verifica se a ação é uma consulta.
isGenVar (*String*)	public static: *boolean*	Verifica se a ação é uma variável gerada
isPrimitive (*String*)	private static: *boolean*	Verifica se é uma primitiva
isQuery (*String*)	private static: *boolean*	Verifica se é a ação adaptativa de consulta
isInsertion (*String*)	private static: *boolean*	Verifica se é a ação adaptativa de inserção
isRemotion (*String*)	private static: *boolean*	Verifica se é a ação adaptativa de remoção
normalizeMap (*Map*)	private: *Map*	Normaliza o mapa de propriedades retirando propriedades que não equivalem às colunas utilizadas para reconhecer uma regra.
obtainBeanProperties ()	public: *List*	Retorna as propriedades da regra em uma lista.

B.4.4 Rules

public Class: Classe que representa uma coleção de regras. A estrutura para guardar as regras deve ser bem definida para que a busca pelas regras seja eficaz.

Atributo	**Tipo**	**Descrição**
originalSet	private : *MultiMap*	*MultiMap* contendo as regras identificadas pelo *origin* Valor Inicial: new MultiHashMap();
unifiableSet	private : *MultiMap*	*MultiMap* contém as regras unificáveis tendo *origin* como chave Valor Inicial: new MultiHashMap();
adaptiveSet	private : *MultiMap*	*MultiMap* contém as regras adaptativas tendo o *origin* como chave Valor Inicial: new MultiHashMap();
adaptiveFunctions	private : *MultiMap*	*MultiMap* contém as funções adaptativas tendo como chave o nome da função adaptativa Valor Inicial: new MultiHashMap();
Método	**Tipo**	**Descrição**
add (*Rule*)	public: *void*	Adiciona a regra ao conjunto de regras
adaptiveAdd (*Rule*)	public: *void*	Adiciona uma regra adaptativa
set (*Rules*)	public: *void*	Copia uma instância de *Rules* para esta instância
size ()	public: *int*	Retorna o tamanho da coleção de regras
clear ()	public: *void*	Limpa todo o conjunto de regras
reset ()	public: *void*	Limpa todas as regras adaptativas
getRulesWithOrigin (*int*)	public: *List*	Procura um conjunto de regras dado um *origin*
getRulesWithOrigin (*String*)	public: *List*	Procura um conjunto de regras dado um *origin*
getUnifiableRules ()	public: *List*	Retorna todas as regras unificáveis

normalizeIntegerString (*String*)	private: *String*	Normaliza a string que representa um inteiro, tirando os zeros à esquerda.
getAdaptiveFunction (*String*)	public: *List*	Dado um *header* retorna todas as listas de regras adaptativas

B.4.5 Token

public Class: Classe Token que representa um token. Nada mais é que uma string.

Atributo	**Tipo**	**Descrição**
token	private const : *char*	String referente ao *token*
Método	**Tipo**	**Descrição**
Token (*char*)	public:	Construtor que cria um *Token* dada uma *String* que o representa
toString ()	public: *String*	Retorna a *String* que representa esta instância do *Token*

B.4.6 Vmds

public Class: Classe que representa toda a estrutura de dados de uma máquina virtual. Chamada de *virtual machine data structure*.

Atributo	**Tipo**	**Descrição**
log	private const static : *Log*	Instância estática de *logging* Valor Inicial: LogFactory.getLog;
RULES_FILE	private const : *String*	Nome do arquivo que contém o código objeto das regras da língua portuguesa Valor Inicial: "voicer.spa";
FIELD_DELIMITER	public const static : *String*	Delimitador utilizado para separar as colunas no código objeto

		Valor Inicial: ";";
STACK_UNDERFLOW	public const static : *int*	Código usado para identificar *stack underflow* Valor Inicial: -1;
MAX_SPEED	public const static : *int*	Velocidade máxima Valor Inicial: 150;
loadedObjects	private static : *Map*	Mapa que contém os códigos objetos já carregados Valor Inicial: new HashMap();
handler	private const : *ScreenHandler*	*Handler* da tela
code	private const : *Rules*	Conjunto de regras carregadas
finals	private const : *List*	Lista de estados finais
state	private : *int*	Estado atual da máquina Valor Inicial: 0;
stack	private const : *Stack*	Pilha da máquina
watch	private : *Map*	Mapa de variáveis
speed	private : *int*	Velocidade atual
inputText	private : *String*	Texto de entrada
transducerText	private : *StringBuffer*	Texto do transdutor
outputText	private : *StringBuffer*	Texto de saída
tokenPosition	private : *int*	Posição da leitura do *token*
inputMachine	private :	Sub máquina de entrada

	Vmds	Valor Inicial: null;
outputMachine	private : *Vmds*	Sub máquina de saída Valor Inicial: null;
kernel	private : *Kernel*	Instância do *kernel* Valor Inicial: null;
objectFile	private : *String*	Nome do arquivo do código de objeto
inputFile	private : *String*	Nome do arquivo de entrada
invert	private : *boolean*	*flag* que indica se a entrada deve ser lida de trás pra frente ou não
Método	**Tipo**	**Descrição**
Vmds ()	public:	Construtor padrão que não possui *ScreenHandler*
Vmds (*ScreenHandler*)	public:	Construtor que recebe um *ScreenHandler*
setInvert (*boolean*)	public: *void*	Armazena se a leitura dever ser feita invertida ou não
getHandler ()	public: *ScreenHandler*	Retorna o *handler* associado a essa máquina virtual
getCode ()	public: *Rules*	Retorna a lista de regras
getState ()	public: *int*	Retorna o estado atual
setState (*int*)	public: *void*	Altera o estado atual
getStack ()	public: *Stack*	Retorna a pilha de estados da máquina virtual
setWatch (*Map*)	public: *void*	Altera o mapa de variáveis
getSpeed ()	public: *int*	Retorna a velocidade atual
getInputText ()	public: *String*	Retorna o texto de entrada
setInputText (*String*)	public: *void*	Armazena o texto de entrada
setTransducerText	public: *void*	Altera o texto do transdutor

(*StringBuffer*)		
appendTransOutputTex t (*String*)	public: *void*	Concatena o valor passado como parâmetro com o valor atual do transdutor
setOutputText (*StringBuffer*)	public: *void*	Altera o texto de saída
appendOutputText (*String*)	public: *void*	Concatena o valor passado como parâmetro com o valor atual do texto de saída
setTokenPosition (*int*)	private: *void*	Altera a posição de leitura do *token*
setInputMachine (*Vmds*)	private: *void*	Altera a máquina virtual de entrada
getOutputMachine ()	public: *Vmds*	Retorna a máquina virtual de saída
setOutputMachine (*Vmds*)	private: *void*	Altera a máquina virtual de saída
getKernel ()	private: *Kernel*	Retorna o *kernel* da aplicação
setInputFile (*String*)	private: *void*	Seleciona o arquivo de entrada do usuário
setObjectFile (*String*)	private: *void*	Seleciona o arquivo de regras a ser utilizado
reset ()	public: *void*	Reseta a máquina virtual
end ()	public: *void*	Finaliza a *virtual machine data structure*. Geralmente essa é a hora de mostrar ao usuário a lista ordenada de todas as regras utilizadas na execução da máquina virtual.
curToken ()	public: *Token*	Retorna o *token* corrente
getLastToken ()	private: *Token*	Retorna o último *token* lido
readToken ()	public: *void*	Lê o *token*
pop ()	public: *int*	Remove o estado do topo da pilha do autômato
push (*int*)	public: *void*	Adiciona um estado ao topo da pilha do

		autômato
allInputRead ()	public: *boolean*	Verifica se toda a entrada foi lida
appendFinalState (*Rule*)	public: *void*	Se a regra possuir um estado final, então adiciona à lista de estados finais.
appendFinal (*String*)	private: *void*	Adiciona um estado final à lista de estados finais.
currentInFinals ()	public: *boolean*	Verifica se o autômato atualmente está em um estado final.
selectRule (*Rule*)	public: *void*	Seleciona a regra passada como parâmetro.
loadObjectCode ()	private: *void*	Carrega o arquivo de regras que está descrito em formato texto para a estrutura de dados da máquina virtual.
initConnection (*String, Vmds, String*)	private: *void*	Inicia uma conexão.
connectInput ()	public: *void*	Método que cria uma sub-máquina dessa máquina e inicia uma conexão com ela. Por padrão todas as execuções sempre têm no mínimo duas máquinas, sendo uma na raiz e outra com toda a lógica a ser executada.
run ()	public: *void*	Método de execução da máquina virtual.
isRunning ()	private: *boolean*	Método que diz se a máquina virtual está em execução ou não.
syncWithScreen ()	private: *void*	Sincroniza todos os dados dessa máquina virtual com a tela.
readFile (*String*)	private: *String*	Retorna uma string representando todo o conteúdo do arquivo lido.

B.4.7 Semantics

public abstract «interface» Interface: Interface *Semantics*. Para dar semântica aos *tokens*, deve-se implementar esta interface. No caso do *Voicer* a única classe que

implementa esta interface é a classe *SoundSemantics* que dá aos *tokens* significado sonoro.

Método	**Tipo**	**Descrição**
execute (*String*)	public: *Token*	Único método que deve ser implementado. Recebe como parâmetro o nome do *token* e sua execução deve dar semântica ao *token* e pode ou não retornar um *token* como saída da ação semântica.

REFERÊNCIAS BIBLIOGRÁFICAS

BASSETO, B. A.; NETO, J. J. A stochastic musical composer based on adaptative algorithms. In: **Anais do XIX Congresso Nacional da Sociedade Brasileira de Computação. SBC-99**. PUC-RIO, Rio de Janeiro, Brazil: [s.n.], 1999. v. 3, p.105-130.

CAMOLESI, A.R.; NETO, J.J. **Modelagem Adaptativa de Aplicações Complexas**. XXX Conferencia Latinoamericana de Informática - CLEI'04. Arequipa - Peru, Setiembre 27 - Octubre 1, 2004.

CAMPOS,G.L. **Síntese de Voz para o Idioma Português**. São Paulo, 1980. Tese(Doutorado) - Escola Politécnica, Universidade de São Paulo.

CASEIRO, D.; TRANCOSO, I.; OLIVEIRA, L.; VIANA, C. Grapheme-to-Phone Using Finite State Machine Transducers. In: **IEEE Workshop on Speech Synthesis.** Lisboa, Portugal, 2002.

CHBANE, D. T. **Desenvolvimento de Sistema para Conversão de Textos em Fonemas no Idioma Português**. 1993, 130p. Disseratação (Mestre) - Escola Politécnica, Universidade de São Paulo, São Paulo, Brasil, 1994.

ESQUIVEL,A.S. Um Sistema de Síntese de Voz. In: Congresso Nacional de Informática, 18., São Paulo, 1985. **Anais**. São Paulo, Sucesu, 1985. p. 776-82.

JUNIOR, J. R. A.; NETO, J. J.; HIRAKAWA, A. R. Adaptive automata for independent autonomous nagivation in unknown enviroment. In: **Proceedings of IASTED International Conference on Applied Simulation and Modelling - ASM 2000**. Ban_, Alberta: [s.n.], 2000.

LEMMETTY, S. **Review of Speech Synthesis Technology**. 1999, 104 p. Master's Thesis – Electrical and Communication Engineering Department, Helsink University of Tecnology, Helsinque, Finlândia, 1999.

MENEZES, C. E. D. de; NETO, J. J. Um método híbrido para a construção de etiquetadores morfológicos, aplicado à língua portuguesa, baseado em autômatos

adaptativos. In: **Anais da Conferencia Iberoamericana en Sistemas, Cibernética e Informática**. Orlando, Florida: [s.n.], 2002.

NETO, J. J. **Introdução a Compilação**. Rio de Janeiro: LTC, 1987. 222p.

NETO, J. J. **Contribuição à metodologia de construção de compiladores**. 1993, 130p. Tese (Livre Docência) - Escola Politécnica, Universidade de São Paulo, São Paulo, Brasil, 1993.

NETO, J. J.; JUNIOR, J. R. A.; SANTOS, J. M. N. dos. Synchronized statecharts for reactive systems. In: **Proceedings of the IASTED International Conference on Applied Modelling and Simulation**. Honolulu, Hawaii: [s.n.], 1998. p.246-251.

NETO, J. J. Solving complex problems e_ciently with adaptative automata. In: **Conference on the Implementation and Application of Automata - CIAA 2000**. Ontario, Canada: [s.n.], 2000.

NETO, J. J.; MORAES, M. de. Formalismo adaptativo aplicado ao reconhecimentode linguagem natural. **Anais da Conferencia Iberoamericana en Sistemas, Cibernética e Informática**, Orlando, Florida, July 2002.

PIMENTEL, M. da G. C.; INÁCIO, V. dos R.. **Integrando Reconhecimento e Síntese de Voz à Computação Ubíqua**, 25 p. Monografia (Projeto de Graduação) - ICMC, Universidade de São Paulo, São Carlos, Brasil, 2003.

PISTORI, H. **Tecnologia Adaptativa em Engenharia de Computação: Estado da Arte e Aplicações**. Edição Revisada. 2003, 174 p. Tese (Doutorado) - Escola Politécnica, Universidade de São Paulo, São Paulo, Brasil, 2003.

ROCHA, R. L. A.; NETO, J. J. Autômato adaptativo, limites e complexidade em comparacão com máquina de turing. In: **Proceedings of the first Congress of Logic Applied to Technology - LAPTEC'2000**. São Paulo: Faculdade SENAC de Ciências Exatas e Tecnologia: [s.n.], 2000. p.33-48.

SILVA, T.C. **Fonética e fonologia do português: roteiro de estudos e guia de exercícios**, 7.ed – São Paulo: Editora Contexto, 2003.

TRANCOSO, I.; CASEIRO, D. Spoken Language Processing Using Wighted Finite State Transducers. In: SWIM'2004 - Special Workshop in Maui: Lectures by Masters

in Speech Processing, 2004. **Anais**. Laboratório de Sistemas de Linguagens Faladas, Lisboa, Portugal, 2004.

WEISSER, M. The computer for the 21st century. **Scientific American**, 265(3):94–104, 1991.

ZUFFO, F.A.; PISTORI, H. Tecnologia Adaptativa e Síntese de Voz: Primeiros Experimentos. **Anais do V Workshop de Software Livre - WSL**. Porto Alegre, 2-5 de Junho, 2004.