UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
ESCOLA POLITÉCNICA
DEPARTAMENTO DE ELETRÔNICA E DE COMPUTAÇÃO

MusiC – Um Framework para Classificação de Música

Autor:

Marcos José Sant'Anna Magalhães

Banca Examinadora:

Orientador:

Prof. Luiz Wagner Pereira Biscainho, D.Sc.

Examinador:

Eng. Dirceu Gonzaga da Silva, M.C.

Examinador:

Prof. Marcelo Luiz Drumond Lanza, M.Sc.

DEL
15 de outubro de 2009

UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
Escola Politécnica - Departamento de Eletrônica e de Computação
Centro de Tecnologia, bloco H, sala H-217, Cidade Universitária
Rio de Janeiro - RJ CEP 21949-900

DEDICATÓRIA

A meu avô Ayrton (im memoriam) por ter sido meu primeiro professor de eletrônica, pelas aulas nas incontáveis luzes de natal que enfeitavam a casa dele.

AGRADECIMENTO

A Deus, por ter me dado forças para realizar minhas tarefas.

Aos que vieram antes de mim, por terem me dado o conhecimento necessário para ir além.

Aos meus pais, por terem me dado conforto, sem eles as derrotas seriam ainda maiores.

Ao meu padrinho, Luiz Antonio, pelo apoio incondicional.

Aos meus amigos, por terem me dado momentos inesquecíveis todos esses anos. Em especial Diego Trally, Paulo Cezar, Paulo Vinicius e Rafael Amaro; Aninha e Suzana. Também ao Allan Dieguez e ao Ivan alvos preferidos das minhas ironias e gozações.

Ao meu orientador, por ser o melhor referencial que poderia ter na minha vida profissional.

RESUMO

A cada dia se tornam mais raras as coleções de músicas armazenadas em vinil, fitas ou até mesmo CDs. Com o crescimento da capacidade de armazenamento dos discos rígidos e a queda no preço desses produtos, as coleções musicais migram para armazenamento nesses discos. Esta mudança envolve desde usuários individuais carregando grandes quantidades de faixas de áudio nos *ipods*® e celulares até lojas virtuais como *iTunes*® e *Deutsche Grammophon*®. Entretanto, entre as desvantagens de se armazenar uma grande quantidade de dados no mesmo meio está a dificuldade de organização e busca. Para ajudar nessa tarefa, muitas pesquisas foram feitas em MIR (Music Information Retrieval), campo que estuda entre outras coisas como extrair de faixas de áudio informações como autores, instrumentos ou gênero. Neste projeto de final de curso, criamos um *framework* para dar mais agilidade e flexibilidade aos pesquisadores desta área. Escolhemos um algoritmo de classificação de gênero musical para estudo de caso e os resultados podem ser encontrados no último capítulo.

Palavras-Chave:
sistema, automático, classificação, áudio, música.

"Assim como o diamante não é puro se não for lapidado, o homem não pode considerar-se homem se não souber superar obstáculos."

Earl Derr Biggers, Charlie Chan e o Camelo Preto.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Trechos de Código

Capítulo 1

Introdução

Desde os primórdios da espécie, o homem teve que aprender a se relacionar com o ambiente onde vive. À medida que a nossa relação com o meio externo foi evoluindo, as nossas necessidades foram mudando e exigindo cada vez mais desse relacionamento. Mas embora as necessidades tenham mudado, as formas básicas com as quais percebemos o ambiente permanecem as mesmas: somente através dos sentidos conseguimos nos relacionar com o que está a nossa volta.

Inicialmente associados à sobrevivência, os sentidos foram aos poucos ampliando sua importância até o ponto de se tornarem uma potencial fonte de prazer. Em particular, de guia importante nas tarefas cotidianas, a audição também se tornou viabilizadora de uma fascinante forma de entretenimento, a música.

A música passou por inúmeras transformações que afetaram os instrumentos, os ritmos, as letras e até a forma de ouvir, numa vagarosa evolução. Mais tarde, com o aprimoramento do conhecimento científico, a música ganhou novas possibilidades. Foi apenas uma questão de tempo até chegarmos a potentes amplificadores, alto-falantes cada vez mais fiéis, codificadores sem perda e tocadores portáteis, cada vez mais populares.

Estes tocadores contam com grande capacidade de armazenamento, mas os recursos para organização das músicas ainda são pobres. Recentemente, um campo do processamento de áudio, o MIR (do inglês *Music Information Retrieval*), se dedica a obter informações sobre determinada faixa de música exclusivamente através das informações acústicas gravadas. Existem trabalhos que tentam descobrir o título da faixa, o intérprete, o compositor e até a sensação que a música passa ao ouvinte.

Mais do que um grande desafio, o MIR encontra diversos tipos de aplicações. Desde tocadores inteligentes capazes de sugerir faixas e alterar a equalização para cada uma delas até aplicações criadas únicamente para permitir a organização de coleções, empresariais (por exemplo, de lojas virtuais) ou pessoais. Entretanto, ainda existe um longo caminho a se percorrer para viabilizar todas essas aplicações.

Este projeto foi criado para ajudar os pesquisadores desta área a percorrer

este caminho. Nossa meta é dar mais agilidade à pesquisa de algoritmos de classificação de áudio, criando um conjunto de "pedaços" de *software* e permitindo que outras pessoas criem seus algoritmos através do encadeamento desses pedaços com, possivelmente, outros que elas tenham desenvolvido. Uma descrição mais detalhada do projeto e dos seus objetivos será encontrada no Capítulo 4. Já o Capítulo 5 mostra os resultados obtidos usando o trabalho de Barbedo e Lopes [13], que está descrito no Capítulo 3, enquanto que o Capítulo 2 descreve as tecnologias usadas para implementar este projeto.

Capítulo 2

Descrição da Tecnologia

A primeira parte deste capítulo pretende enfatizar algumas diferenças entre as linguagens de programação gerenciadas (ex. C♯, Java, Eiffel) e as não-gerenciadas (ex. Fortran, C, C++). Em seguida, daremos algumas características do C♯, a linguagem usada predominantemente neste projeto, buscando mostrar toda a estrutura por trás da linguagem.

Para não sobrecarregar a nomenclatura, por vezes nos referiremos às linguagens não-gerenciadas como linguagens tradicionais.

2.1 Linguagens de Programação

Uma definição de linguagem de programação é, segundo John C. Mitchel [3], um canal de expressão da arte de programar computadores. Uma definição mais pragmática diria que é o conjunto de palavras-chave e regras que permitem aos programadores dar instruções a máquinas.

Considera-se que a primeira máquina programável foi desenvolvida em 1206 (!!) pelo inventor árabe Al-Jazari [27]. No Livro dos Conhecimentos de Engenhosos Dispositivos Mecânicos aparece, entre muitos outros autômatos, uma banda de música (Figura 2.1) composta por quatro integrantes que poderiam tocar vários ritmos musicais programáveis através de alavancas e ainda realizavam, segundo Charles B. Fowler [6], até 50 movimentos corporais e faciais durante cada uma dessas seleções.

Entretanto, o primeiro computador programável[1] só foi desenvolvido em 1941. Entre 1939 e 1941 Konrad Zuse desenvolveu e montou o Z3 com 2400 relés (aproximadamente 600 para a unidade aritmética e outros 1800 para memória e controle [2]) que eram usados no sistema telefônico da época. O Z3 foi desenvolvido para realizar análises estatísticas de dados das asas dos aviões durante o projeto das aero-

[1] O primeiro computador mecânico foi desenvolvido ainda em 1623 por Wilhelm Schickard. Era uma calculadora automática que usava o sistema decimal. Mais tarde seria aprimorada por Leibniz para usar a base binária.

Figura 2.1: Ilustração de uma das máquinas de Al-Jazari: Um artefato musical em forma de barco. Extraído de [17]. *Copyright* © Smithsonian Institute.

naves. Entre 1942 e 1946, Zuse desenvolveu a primeira linguagem de programação, a *plainkalkül* (do alemão, plano de cálculo) embora só a tenha publicado em 1972. Apenas em 2000, é que engenheiros da *Freie Universität Berlin* implementaram o primeiro compilador de *plainkalkül* [11].

A página *99 bottles of beer* [31] mantém um registro informal das linguagens de programação e suas variações (também chamadas de dialetos). Este site pede que os visitantes contribuam com programas que, quando executados, escrevam a letra da música *99 bottles of beer* [32]. No último acesso (22 de fevereiro de 2009) o site contabilizava 1253 linguagens e variações. Uma outra lista mantida por Bill Kinnersley tem o (ousado) objetivo de ser a mais completa listagem de linguagens de programação. Esta lista conta com cerca de 2500 (!) linguagens e dialetos.

Não existe uma forma precisa de classificar esta verdadeira babel de linguagens de programação. Uma forma comum de fazer isso é se referir aos paradigmas de programação que a linguagem favorece (no caso de linguagens que suportam diversos desses tipos) ou adota, como na Figura 2.2. Ainda uma outra forma de classificar uma linguagem é "posicionando-a" entre a linguagem natural[2] e a linguagem de máquina. As linguagens também podem ser classificadas quanto à forma de

[2]De forma geral, as linguagens de programação se baseiam no inglês, mas existem linguagens que são baseadas em outras linguas. A *Fjölnir* é baseada no islandês.

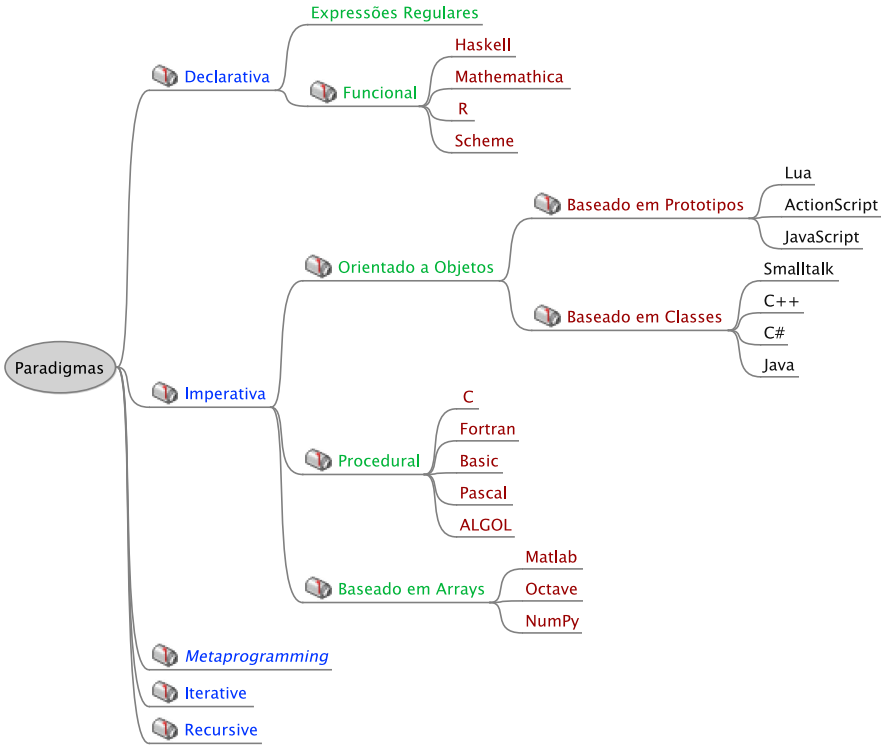

Figura 2.2: Paradigmas de linguagens de programação.

execução dos seus programas (ex. compiladas ou interpretadas) ou ainda quanto ao gerenciamento de memória (ex. gerenciada ou não-gerenciada)[3].

Consideramos uma linguagem gerenciada[4] aquela que executa, automaticamente e de maneira autônoma, o gerenciamento das áreas de memória alocadas pelos seus programas. Gerenciar memória significa fazer operações sobre essas áreas, incluindo (mas não se limitando a) alocá-las e removê-las.

Embora as linguagens gerenciadas tenham uma forte conexão com as linguagens interpretadas, como veremos a seguir, existem linguagens compiladas que são gerenciadas. Essas linguagens se utilizam de toda uma estrutura que se interpõe entre o binário gerado na fase de compilação e o processador para que possa agir como um gerente. Essa estrutura se torna um processador fictício, uma máquina virtual, para essas linguagens, que no decorrer do texto chamaremos de linguagens dependentes de máquina virtual.

A próxima seção apresenta uma discussão sobre as linguagens interpretadas mostrando quão estreito é o seu relacionamento com as linguagens gerenciadas.

[3]Separar linguagens apenas em gerenciadas ou não-gerenciadas é uma forma bastante primitiva de classificação que esconde as diferenças que existem em como as linguagens lidam com os recursos, especificamente, a memória. Entretanto, como discutiremos apenas aspectos gerais das linguagens gerenciadas, esta classificação se mostra adequada.

[4]Não foi possível encontrar na literatura uma descrição formal.

2.1.1 Linguagens Interpretadas

Nesta seção faremos uma apresentação das linguagens interpretadas (ou *scriptadas*[5]) mostrando a sua relação com os outros tipos de linguagens.

A palavra *script* significa, em português, roteiro: um texto que descreve os passos que os atores devem seguir para realizar uma determinada montagem. Do ponto de vista da informática, um *script* é exatamente a mesma coisa: um texto seguindo uma estrutura padronizada (tal qual um roteiro no sentido da palavra), que descreve a forma com que as variáveis e/ou objetos (os atores, comparativamente com as montagens artísticas) exercem e sofrem ações.

Por ser um texto e estar, portanto, muito mais próximo do homem que da máquina, o processo de "tradução" para a linguagem de máquina é caro[6]. Este processo é ainda mais custoso se considerarmos que ele ocorre concomitantemente com a execução (ou que ele **é** a execução). Assim, algumas medidas que modificam o custo do intérprete foram adotadas para reduzir o seu impacto no desempenho sem remover a flexibilidade e a facilidade permitidas por esta família de linguagens.

Algumas linguagens interpretadas, como Python e Lua, suportam a geração de um tipo de código intermediário, frequentemente referido como *bytecode*[7], para diminuir o custo da tradução. Este código transforma o texto do *script* em um conjunto de instruções binárias, portanto mais palatável às máquinas, de forma que quando este código intermediário for executado a tradução seja muito mais simples.

Aqui aparece uma primeira diferença em relação às linguagens compiladas. As linguagens compiladas têm em seus executáveis instruções para o **processador**, enquanto os "executáveis" das linguagens interpretadas têm instrucões para o **interpretador**, através de texto ou de *bytecode*. (Esta diferenciação é mais sutil nos casos de linguagens compiladas que fazem uso de uma arquitetura semelhante à das linguagens interpretadas, já que existe um processo de compilação e a tentativa de tornar esta arquitetura transparente aos olhos do usuário.)

É esse aspecto que permite que uma linguagem possa ser verdadeiramente gerenciada: sem o interpretador uma linguagem não pode garantir o gerenciamento de memória sem a colaboração do usuário. Para ser independente, esse gerente precisa estar ciente de todo o contexto de execução de um programa, ou seja, precisa saber de "tudo que está acontecendo".

Nas linguagens compiladas isso não é possível, já que o processador, que é o

[5]Este é um jargão da área originado da palavra inglesa *script* que denomina este tipo de linguagem neste idioma.

[6]Isto significa que procedimentos que demandam **muito processamento**, em geral, não devem ser implementados numa linguagem interpretada. Entretanto, os processadores estão cada vez mais capazes e muda, a cada dia, a noção de **muito processamento**.

[7]Nome originalmente atribuído aos binários criados com Java.

responsável pela execução, só sabe o que está acontecendo no momento presente. Já nas linguagens gerenciadas, o "gerente", qualquer que seja o nome que ele receba, sabe de tudo o que acontece agora, já aconteceu e irá acontecer. É esta "não-causalidade" que lhe permite fazer o gerenciamento de memória.

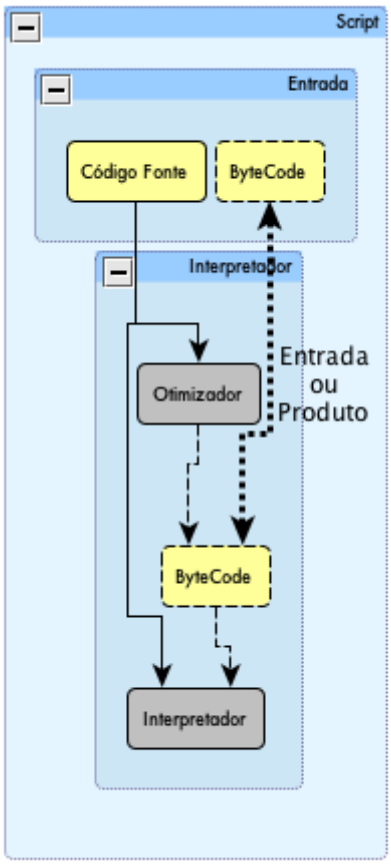

Figura 2.3: Ciclo de vida de um "*script*".

A Figura 2.3 mostra como um *script* pode ser executado. Isto pode acontecer diretamente a partir do código-fonte, a partir do *bytecode* ou, no caso híbrido, quando o binário for criado. Dois fatos devem ser salientados nesta figura: não há separação entre os ambientes de desenvolvimento e execução; e o *bytecode* é tanto entrada quanto produto.

Já na Figura 2.4, pode ser vista a etapa de compilação necessária para execução de linguagens que dependem de máquina virtual, bem como a estrutura da qual elas necessitam. Note-se que nesta arquitetura há uma clara separação entre os ambientes de desenvolvimento e de execução.

Não haver esta separação significa que o desenvolvedor e o usuário têm os mesmos "direitos", o que implica o acesso aos códigos-fonte e a eventual capacidade de modificar a aplicação. Esta característica pode ser tanto muito bem vinda, por exemplo, para customizações, quanto inaceitável, se o autor original estiver tentando

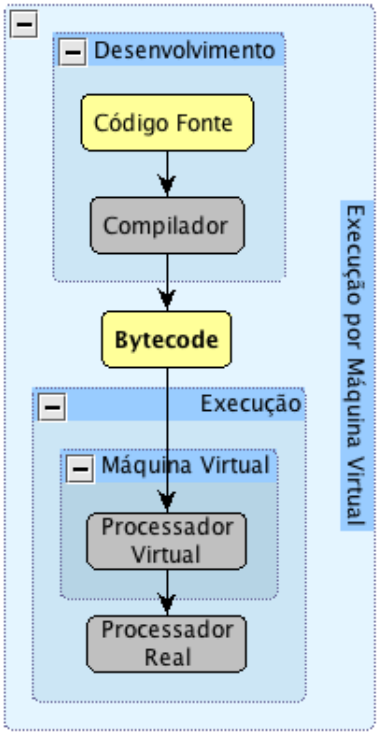

Figura 2.4: Ciclo de vida de um executável produzido por uma linguagem gerenciada dependente de máquina virtual.

proteger seus direitos sobre as possíveis cópias. É através da fase de compilação que os ambientes de desenvolvimento e execução serão separados nas linguagens dependentes de máquina virtual, como mostrado na Figura 2.4, tornando o *bytecode* um intermediário entre esses ambientes.

As motivações para a existência desta fase de compilação também incluem a melhora do desempenho (como nas linguagens interpretadas) e limitação do acesso ao código-fonte (embora ainda com sucesso bastante limitado). Outras razões serão discutidas na Seção 2.1.3, quando falarmos sobre gerenciamento de memória.

O Trecho 2.1 é o código de uma função do projeto usada para imprimir o código-fonte através do LaTeX®. O Trecho 2.2 foi obtido usando o **Red Gate's .NET Reflector®** [35] para gerar o código-fonte da função **exportToLatex** a partir do binário. Apesar seu do tamanho reduzido, pode-se constatar que o código obtido com a reversão é impecável[8]. Com a riqueza de informações existente nos *bytecodes*, mesmo em aplicações maiores, a acurácia se mantém, diferentemente do que ocorre com as linguagens compiladas.

```
1 public static void exportToLatex(string dir, string ext, TextWriter f)
  {
```

[8]Em C♯ a inserção de um caracter "@" antecedendo uma *string* significa que nenhum caractere de escape será aceito; portanto, "\n" não é uma quebra de linha, e sim a própria sequência de caracteres.

```
3   string[] dirList = Directory.GetDirectories(dir);

5   if (dirList.GetLength(0) > 0)
    {
7     foreach (string d in dirList)
      {
9       exportToLatex(d, ext, f);
      }
11  }

13  foreach (string file in Directory.GetFiles(dir))
    {
15    if (Path.GetExtension(file) == ext && Path.GetFileName(file) != "AssemblyInfo.
          cs")
      {
17      string relPath = RelativePathTo(Directory.GetCurrentDirectory(), file);
        relPath = relPath.Replace('\\', '/');
19      f.WriteLine("\\subsection{" + relPath + "}");
        f.WriteLine("\\lstinputlisting{" + relPath + "}");
21    }
    }
23 }
```

Trecho 2.1: Código original da função *reflected*.

```
1 public static void exportToLatex(string dir, string ext, TextWriter f)
  {
3   string[] dirList = Directory.GetDirectories(dir);

5   if (dirList.GetLength(0) > 0)
    {
7     foreach (string str in dirList)
      {
9       exportToLatex(str, ext, f);
      }
11  }

13  foreach (string str2 in Directory.GetFiles(dir))
    {
15    if (Path.GetExtension(str2) == ext && Path.GetFileName(str2) != "AssemblyInfo.
          cs")
      {
17      string str3 = RelativePathTo(Directory.GetCurrentDirectory(), str2);
        relPath = str3.Replace('\\', '/');
19      f.WriteLine(@"\subsection{" + str3 + "}");
        f.WriteLine(@"\lstinputlisting{" + str3 + "}");
21    }
    }
23 }
```

Trecho 2.2: Código gerado pelo *reflector*.

Muitas linguagens interpretadas também são gerenciadas. No entanto, há linguagens gerenciadas que não são interpretadas nem puramente compiladas como as tradicionais C, C++ e Fortran. As linguagens que se encaixam neste grupo, que no decorrer do texto chamaremos de linguagens dependentes de máquina virtual, herdam características tanto das linguagens compiladas quanto das interpretadas.

A próxima seção discutirá as linguagens gerenciadas.

2.1.2 Linguagens Gerenciadas

A Tabela 2.1 mostra as dez linguagens mais populares[9] da internet segundo o *site* `www.tiobe.com` [36] que divulga mensalmente uma listagem dessas linguagens.

Maio 2009	Maio 2008	Linguagem	Pontuação	Variação 08/09	Gerenciada
1	1	Java	19,537%	-1,35%	Sim
2	2	C	16,128%	+0,62%	Não
3	3	C++	11,068%	+0.26%	Não
4	4	PHP	9,921%	-0,28%	Sim
5	5	Visual Basic	8,631%	-1,16%	Sim
6	7	Python	5,548%	+0,65%	Sim
7	8	C♯	4,266%	+0,21%	Sim
8	9	JavaScript	3,548%	+0,62%	Sim
9	6	Perl	3,525%	-2,02%	Sim
10	10	Ruby	2,692%	+0,05%	Sim

Tabela 2.1: Listagem das linguagens mais populares da internet. Adaptado de [36].

Como pode ser visto na tabela, C ainda é uma das linguagens mais populares. Essa linguagem, que oferece ao programador eficiência e controle sobre o que está acontecendo, além de ser, em muitos casos, a primeira que os programadores aprendem. Na verdade, essas mesmas características que atraem tantos programadores também podem trazer consequências indesejáveis. Os Trechos 2.3 e 2.4 são implementações de um erro comum durante o desenvolvimento de um programa de computador.

```
1 #innclude <stdio.h>

3 int main()
  {
5   char myString[] = "1234";
    int i;
7
    for(i = 0; i < 6; i++)
9     printf("%i:"t, *(myString+i));
  }
```

Trecho 2.3: *Buffer overflow* versão C.

```
  class OutOfBound
2 {
    static int Main( string[] str )
4   {
      string myString = "1234";
```

[9] A metodologia usada pode ser encontrada em `http://www.tiobe.com/index.php/content/paperinfo/tpci/tpci_definition.htm`

```
6       for( int i = 0; i < 6; i++ )
          System.Console.Write( (int) myString[i] + ":" );
8
        return 0;
10    }
  }
```

Trecho 2.4: *Buffer overflow* versão C♯.

Esses trechos são dois programas, o primeiro em C e o outro em C♯, que tentam ler uma *string* além da área de memória alocada para ela. Na Tabela 2.4 está a saída gerada pelos dois programas quando executados. Com este exemplo pretendemos destacar a diferença entre as propostas das linguagens gerenciadas e tradicionais. Por trás das linguagens gerenciadas existe uma filosofia de minimizar os erros, mesmo que isso resulte em um desempenho pior ou em não finalizar uma instrução, enquanto que nas tradicionais o objetivo é concluir a tarefa com eficiência.

49:50:51:52:0:5

Tabela 2.2: *
Saída produzida pelo código em C[ab]

[a]Versão do Compilador: i686-apple-darwin9-gcc-4.0.1 (GCC) 4.0.1 (Apple Inc. build 5465)

[b]Comando: gcc -Wall -ansi -pedantic

49:50:51:52 Unhandled Exception: System.IndexOutOfRangeException: Array index is out of range. at OutOfBound.Main (System.String[] str) [0x00000]

Tabela 2.3: *
Saída produzida pelo código em C♯[ab]

[a]Versão do Compilador: Mono C♯ compiler version 2.2.0.0

[b]Comando: mcs -checked+

Tabela 2.4: Resultados obtidos após a execução dos Trechos 2.3 e 2.4, respectivamente.

Um dos recursos mais poderosos e utilizados das linguagens tradicionais é o ponteiro. Por ser tão utilizado, por vezes indevidamente, ele está relacionado com vários erros comuns durante a implementação de um algoritmo (como o mostrado anteriormente), além de ser um desafio para os novatos. Por ser tão notório cúmplice, uma proposta que tente prevenir erros de implementação não pode deixar de envolvê-lo.

A forma adotada nessas linguagens foi tentar retirar do programador o acesso direto a regiões de memória através dos ponteiros, encarregando um "gerente virtual" de realizar a alocação, liberação e referenciamento da memória. Assim, o programador só teria acesso à memória través de referências, um tipo de variável parecida com o ponteiro, mas com limitações e garantias diferentes.

As referências são **objetos** que armazenam informações (entre elas o endereço) sobre certas regiões da memória, diferentemente do ponteiro, que é apenas uma variável que armazena um endereço. Embora a diferença possa parecer sutil, ela é suficiente para ter grandes consequências. Por ser um objeto, uma referência pode armazenar outras informações, além de estar apta a receber alguma capacidade de processamento. As referências usadas em Java e C♯ têm a forma parecida com a dos ponteiros; entretanto, em condições normais[10], escondem do usuário o endereço real do recurso referenciado, além de contar o número de referências ativas que apontam para o mesmo recurso[11] e atestar a compatibilidade dos tipos da referência e dos dados.

Essas referências são parte fundamental da forma de gerenciamento de memória, que será discutida na próxima seção.

2.1.3 Gerenciamento de Memória

Como visto na seção anterior, para tentar diminuir erros de execução relacionados aos ponteiros, foram usadas as referências e um "gerente virtual"; assim, cada aplicação teria o seu gerente, buscando ser eficaz e evitar erros. Mas esta não é a forma como o gerente é implementado nas principais linguagens que usam esta tecnologia.

Normalmente o gerente é implementado de forma centralizada, já que a alternativa de cada aplicação e/ou biblioteca ter o seu próprio gerente não é eficiente, além de diminuir as possibilidades de otimização. Entretanto, sincronizar potencialmente inúmeros acessos de vários processos que tentam requisitar recursos da máquina virtual não é tarefa simples.

Uma forma de simplificar o gerente é trazer esta disputa para dentro dele. Com isso, evita-se o uso de complexas rotinas de comunicação entre processos que degradariam o desempenho das aplicações e do próprio gerente. Assim, os executáveis e bibliotecas desenvolvidos com as linguagens gerenciadas são como um *script*, e são executados como se fizessem parte da estrutura responsável pelo gerenciamento. Neste tipo de arquitetura é como se o gerente representasse uma máquina (processador + sistema operacional) em que arquivos binários estão executados. Daí surge a noção da máquina virtual (que não precisa, necessariamente[12], fazer o gerenciamento

[10] Em C♯, por exemplo, é possível obter ponteiros para os dados através do uso das palavras-chave *unsafe* e *fixed*.

[11] Este tipo de contagem é realizado por inúmeras linguagens para se certificar de que um recurso não será mais utilizado e, portanto, pode ser removido. O *Python* é um exemplo de linguagem que usa esta técnica.

[12] Este é o caso quando uma aplicação nativa é executada dentro de um *software* de virtualização. A máquina virtual faz apenas o "repasse" das instruções ao processador.

dos recursos).

O conjunto de instruções que são usadas para formar o *bytecode*, como são chamados os binários produzidos por linguagens que usam máquina virtual, forma um *pseudo-assembly*. Dentro desta estrutura, ele funciona como o conjunto básico de instruções de um determinado tipo de processador (o da máquina virtual). Este *bytecode* será convertido em instrução para determinada arquitetura (processador real) pelo compilador em tempo de execução (tradução livre do inglês *Just-In-Time Compiler*, JIT) apenas no momento[13] de ser enviada ao processador. O processo pode ser visualizado na Figura 2.4.

Um aspecto importante de ser ressaltado na diferença entre os executáveis nativos (das linguagens compiladas) e os da máquina virtual é a separação entre dados e instruções. Como vimos anteriormente, esses executáveis são diferentes do ponto de vista de quem o está executando (o processador real ou o virtual). Esta diferença também se revela na forma de ver o que é dado dentro de um executável. Enquanto no modelo de execução nativo existe a separação entre o que é dado e o que é instrução, no modelo da máquina virtual e do interpretador tudo é dado: as constantes, os tipos, os algoritmos.

Uma grande vantagem de ver o executável em si como dado é dar à máquina virtual o conhecimento de tudo que o programa vai fazer. Em um *script*, o interpretador pode se adiantar à execução para obter informações adicionais que possam determinar o comportamento atual. Esta capacidade (não-causal) cria a possibilidade de otimizações que não eram possíveis nas linguagens tradicionais[14]. Nestas últimas, as instruções vão imediatamente para o processador, que só sabe o que já aconteceu de forma muito limitada (através do estado dos registradores e da memória) e o que está acontecendo agora (a instrução atual).

Uma outra característica desta abordagem se refere à invocação de funções e métodos. Para invocar esses procedimentos, algumas informações devem estar disponíveis, uma vez que a máquina virtual não conhece os tipos que, possivelmente, foram definidos pelo programador. Para executar um método, a máquina virtual deve primeiramente achar este método entre os demais que compõem a classe[15]

[13]Cada implementador é livre para limitar em quantas instruções o gerente pode estar adiante da execução. A situação é similar à gravação de um CD ou o clássico problema dos produtores e dos consumidores (ou do *buffer* associado), descrito por Tanenbaum em [8], onde o leitor (produtor – gerente + JIT) deve estar sempre um pouco adiante do gravador (consumidor – processador).

[14]Em algumas das linguagens gerenciadas, porções de memória podem ser movidas durante o processo de otimização da execução, o que causaria sérios erros se fosse feito em uma linguagem tradicional.

[15]Stroustrup [4] define classe como um tipo definido pelo usuário. Mais do que isso, classes são gabaritos (*blueprint*) para se criar objetos, e podem conter atributos (que descrevem propriedades que definem os objetos), ou métodos (ações que podem ser efetuadas pelos objetos ou sobre ele). Normalmente, as classes representam unidades lógicas da solução implementada.

e as suas superclasses, checar a visibilidade do método, obter informações sobre o protótipo[16], checar se os objetos que estão sendo passados podem ser "interpretados" como os estabelecidos pelo protótipo, e finalmente executar os métodos. Assim, para conseguir realizar todas essas etapas com sucesso é preciso ter uma descrição das classes contendo seus métodos, a visibilidade deles, e informações sobre sua posição na hierarquia das classes, além da descrição dos próprios métodos e possivelmente outras informações específicas das linguagens ou implementações. A todos esses dados que descrevem as classes dá-se o nome de metadados.

Esse metadados criam a possibilidade de um programa olhar para um executável ou biblioteca e saber quais e como são os tipos que estão ali e, se for necessário, instanciá-los e executar os métodos neles contidos. Dada a importância do metadado, sua construção se torna uma tarefa crítica, não devendo ficar a cargo do usuário. É o compilador que se encarrega da construção desses dados, implicando a necessidade de uma fase de compilação.

Em 2005, Detlef Vollmann [1] fez uma avaliação das técnicas existentes de construção de metadados para C++. Esta avaliação é parte dos trabalhos na elaboração de uma atualização desta linguagem, que está sendo chamada de C++0x. Nesta revisão se pretendia incluir o suporte à reflexão, uma técnica que depende dos metadados para sua execução.

A reflexão é a capacidade de um programa se automodificar, escolhendo determinado rumo que só será conhecido durante a execução. Esta técnica foi largamente utilizada neste projeto, e por isso apresentaremos uma discussão sobre este tema.

2.1.3.1 Gerenciamento de Memória em Linguagens Não-Gerenciadas

De fato, existem algumas técnicas que, através do uso de bibliotecas, acrescentam às linguagens tradicionais a capacidade de algum gerenciamento de memória. É importante notar que isto não as torna gerenciadas, já que é um recurso não inerente à linguagem e de uso facultativo. Essas bibliotecas ainda dependem do usuário para garantir o funcionamento correto, o que certamente não é desejável. Entretanto, elas são utilizadas porque reduzem o esforço necessário para o programador realizar as operações com a memória de forma adequada.

Em C, a APR (*Apache Portable Runtime*) usa *pools* (reservatórios) de alocação e garante a desalocação se for usado unicamente o reservatório padrão. Em C++, bibliotecas como Qt e wxWidgets implementam um coletor de lixo nos seus objetos, usando as facilidades de ter uma hierarquia de classes que derivam de uma

[16]Os protótipos correspondem à assinatura de uma função ou método. Fazem parte do protótipo os argumentos, o identificador do procedimento e o tipo retornado, embora este último não seja usado para fins de polimorfismo, quando dois procedimentos se diferenciam unicamente pelos argumentos.

única classe. Entretanto, se o usuário desalocar um desses objetos causará um erro na rotina de encerramento. Ao encerrar as tarefas, a biblioteca tentará desalocar todos os objetos que "pertencem" a ela, inclusive o objeto que já foi desalocado pelo usuário, pois não há como verificar se o objeto já foi destruído.

2.1.4 Reflexão

De acordo com J. Malenfant, M. Jacques e F. N. Demers [9] reflexão é a habilidade de um programa observar e modificar aspectos da linguagem de programação[17] (sintaxe, gramática e implementação) e até mesmo o próprio código. Para nós a habilidade de modificar aqueles aspectos não é interessante, entretanto a capacidade de modificar o próprio código é muito bem vinda.

Como o código pode ser modificado? Na verdade, o código que foi escrito inicialmente não será alterado. Cada vez que este código for executado, o que variará serão as instruções transmitidas ao processador. A máquina virtual decide, como vimos anteriormente, somente na hora de executar, como serão as instruções que devem ser passadas ao processador; podemos aproveitar esta característica para mudar os rumos de cada execução, dependendo do que está acontecendo.

Neste projeto, esta técnica foi usada largamente com, principalmente, duas finalidades: achar e importar tipos definidos pelo usuário; e invocar o construtor (dos tipos importados) mais adequado aos parâmetros passados pelo usuário. A reflexão é, de modo geral, muito útil para implementar aplicações que ofereçam a possibilidade de extensões, principalmente se essas extensões podem ser desenvolvidas por todos os usuários, por se tratar de técnica mais robusta que a padrão, como será mostrado no decorrer desta seção.

2.1.4.1 Reflexão e Extensões

Uma aplicação não-gerenciada pode ser estendida de duas formas: através de bibliotecas ou de extensões. Quando usamos uma biblioteca, seja estática ou dinâmica, usamos os arquivos-fonte e (em C, C++, Fortran e outras) os arquivos de cabeçalho como metadados; portanto, podemos exportar as classes que só existem na biblioteca para a aplicação. Entretanto, quando usamos as extensões não existem metadados, e as classes definidas internamente à extensão ficam confinadas dentro dela.

Para a aplicação ter acesso a essas classes, a extensão deve fazer uma adaptação entre as classes que existem dentro dela e a aplicação, fornecendo um construtor

[17] Reflexão comportamental, do inglês *behavioural reflection*

e um destrutor e garantindo que estas classes sejam derivadas de uma classe conhecida pela aplicação.

Outro problema enfrentado pelos desenvolvedores que utilizam linguagens não-gerenciadas é que essas linguagens, em geral, não suportam o carregamento de extensões. Para realizar esta tarefa devem recorrer às bibliotecas do sistema operacional, o que torna o código específico para este sistema. Algumas bibliotecas criam uma camada de abstração entre a aplicação e as bibliotecas do sistema operacional, melhorando (porque a forma como o sistema operacional trata a extensão ainda pode variar) a portabilidade do código.

O Trecho 2.5 é a implementação de uma aplicação que carrega uma extensão chamada my_plugin.dll definida pelo Trecho de código 2.6.

```
#include <dlfcn.h>
#include <stdlib.h>
#include <stdio.h>

// gcc reflection_host.c -o host_c */
// To compile with -pedantic change comments to /* */  */

int main()
{
   void * plugin = dlopen("my_plugin.so", RTLD_LAZY);

   char (*myfunc)() = (char (*)()) dlsym(plugin, "exported_func");
   void (*say_version)() = (void (*)()) dlsym(plugin, "say_version");

   printf("Exported value: %c\n", myfunc());
   say_version(1);

   dlclose(plugin);
   return 0;
}
```

Trecho 2.5: Versão C (POSIX) de aplicação com incompatibilidades na interface com a extensão.

```
#include <stdio.h>

// Mac: gcc -dynamiclib reflection_plugin.c -o libplugin.dylib */
// Linux: gcc -dynamic reflection_plugin.c -o libplugin.so */
// To compile with -pedantic change comments to /* */  */

int exported_func()
{
   return 48; /* Zero ascii code*/
}

void say_version()
{
   printf("I am version 2 !\n");
}
```

Trecho 2.6: Versão C de extensão com incompatibilidades na interface com a

aplicação.

Note que os protótipos das funções **exported_func** e **say_version** estão também definidos do lado da aplicação, de forma estática, e não correspondem aos protótipos das funções originais. Nesse exemplo, a não correspondência dos protótipos não provoca erros graves, mas nem sempre é esse o caso. Usando as técnicas de reflexão, não só os protótipos dessas funções não precisariam ser enunciados como poderiam variar a cada execução e ainda contar com uma verificação para garantir sua compatibilidade.

O Trecho 2.7 é um exemplo de como uma aplicação pode usar a reflexão em C♯. Ele carrega uma biblioteca (my_plugin.dll, Trecho 2.8) e procura pela classe **Global**, já que em C♯ não existe o *namespace* global[18], embora membros estáticos sejam permitidos. Uma vez obtido o objeto que descreve a classe, **tGlobal**, procuram-se os métodos **exported_func** e **say_version**. Deve-se notar que C♯ suporta polimorfismo, e portanto os nomes dos métodos não necessariamente identificam o método desejado. Somente com o chamado da função **Invoke**, que define os argumentos que serão passados ao método e o objeto "dono" do método, é que a identificação será completa, já que dois métodos não podem ser diferentes apenas pelo tipo que retornam.

```
1 using System;
  using System.Reflection;
3
  // mcs -target:exe -out:host.exe reflection_host.cs
5
  class Host
7 {
    public static int Main(string[] args)
9   {
      Assembly plugin = Assembly.LoadFile("plugin_cs.dll");
11    Type tGlobal = plugin.GetType("Global");

13    MethodInfo myfunc = tGlobal.GetMethod("exported_func");
      MethodInfo say_version = tGlobal.GetMethod("say_version");
15
      Console.WriteLine("Exported Value: "+ myfunc.Invoke(null, null));
17
      // argumentos do Invoke:
19    // o objeto do qual o método será executado. null significa que o método é
          estático.
      // lista de argumentos do método. null siginifica que  método não tem
          argumentos.
21    // Retorna: O objeto retornado pelo método ou null.

23    say_version.Invoke(null, null);

25    return 0;
    }
```

[18]Isto significa que não há funções globais. Todas as funções devem ser definidas dentro de classes.

```
27 }
```

Trecho 2.7: Versão C♯ de aplicação usando reflexão para executar um método em uma biblioteca dinâmica.

```
1 using System;

3 //mcs -target:library -out:plugin.dll reflection_plugin.cs

5 // Em C♯ não existem funções globais. Todas as funções pertencem a uma classe.
  public class Global
7 {
    public static int exported_func()
9   {
      return 48;
11  }

13  public static void say_version()
    {
15    Console.WriteLine("I am version 2 !\n");
    }
17 }
```

Trecho 2.8: Versão C♯ do Trecho 2.6.

É interessante notar que no Trecho 2.7, diferentemente do Trecho 2.5, os protótipos dos métodos importados não aparecem claramente definidos e que, a cada execução, o protótipo do método que seria importado pode variar, bastando que os tipos dos objetos passados como parâmetros variem.

Esta característica de evitar o contato direto do programador com o protótipo é interessante para as aplicações que suportam extensões, principalmente se o suporte se estende às extensões desenvolvidas por usuários, como neste projeto.

2.1.5 Comentários Sobre o Desempenho

Comparar de forma clara e inequívoca o desempenho das linguagens gerenciadas com as tradicionais é uma tarefa que ainda está por ser feita. Entretanto, podemos perceber da Seção 2.1.3 que existe uma "disputa" entre os custos adicionais gerados pelo gerenciamento e as otimizações que podem ocorrer devido à máquina virtual.

Existem defensores fervorosos das duas correntes, mas parece haver uma forte tendência para a implementação de tarefas cujo desempenho é crítico em linguagens tradicionais, e a implementação de tarefas cujo tempo de desenvolvimento é crítico em linguagens gerenciadas, uma vez que as facilidades oferecidas reduzem o tempo de desenvolvimento.

2.2 C♯ e CLI

Nesta seção será feita uma breve apresentação do C♯ e da Infraestrutura Compartilhada entre Linguagens (CLI – *Common Language Infraestructure*). Faremos inicialmente uma breve revisão geral sobre a história e a estrutura dessas duas tecnologias. Em seguida, discutiremos aspectos mais específicos, tanto do C♯ quanto da CLI, que serão necessários futuramente.

De maneira alguma esta seção pretende ser um tutorial sobre o assunto, mas apenas permitir uma breve familiarização do leitor para melhor compreensão do projeto.

2.2.1 Visão Geral

O C♯[19] é uma linguagem de programação de alto nível, imperativa e orientada a objetos, que suporta diversos tipos de paradigmas de programação, entre eles o funcional, e definida pelos padrões ISO/IEC 23270:2006 e ECMA[20] 334:2006 [37]. Esta é uma das linguagens que integram a CLI, que é um esforço, também iniciado pela Microsoft®, de interoperabilidade entre linguagens, e mais tarde definido pelos padrões ISO/IEC 23271:2006 e ECMA 335:2006.

A primeira versão começou a ser desenvolvida no início de 1999 e foi anunciada publicamente em julho de 2000 em um dos eventos promovidos[21] pela Microsoft®. Ainda em 2000, Microsoft®, HP® e Intel® propuseram a adoção do C♯ como um padrão ECMA, o que aconteceu no ano seguinte. Em 2003, a ISO, depois de alterações que também seriam adotadas pelo padrão ECMA, também criou padrões para o C♯ e a CLI. Atualmente o C♯ está na versão 3.5, estando a versão 4.0 em desenvolvimento.

As regras que pautaram o desenvolvimento do C♯ foram:

- Ser simples, moderna, de propósito geral e orientada a objetos.
- Suportar os princípios da engenharia de *software*.
- Oferecer a possibilidade de ser executada em sistemas distribuídos.
- Prever portabilidade do código e do programador[22].

[19] O nome curioso que foi dado à linguagem vem da teoria musical. Assim como C++ representa um incremento sobre o C, C♯ representa um semitom acima de C (dó).

[20] A ECMA (European Computer Manufacturers Association) é uma associação que reune importante empresas para criação de padrões para equipamentos eletrônicos e de informática. Entre os membros estão gigantes do setor, como Adobe®, Google®, IBM®, Intel®e Microsoft®.

[21] PDC - Professional Developer Conference.

[22] Será discutida na Seção 2.2.2.1.

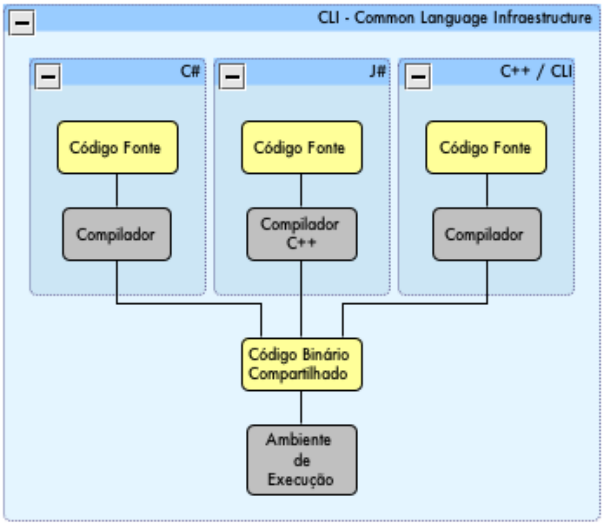

Figura 2.5: Visão geral da CLI.

- Prever suporte a internacionalização.
- Ser capaz de rodar tanto em sistemas complexos quanto em embarcados.
- Preocupar-se com o consumo de recursos (memória e processador), mas não a ponto de tentar concorrer com C e Assembly.

Nos cinco primeiros tópicos pode-se perceber certa semelhança com Java, já que essas características estão fortemente presentes naquela linguagem e não poderiam ser negligenciados por uma nova linguagem que estivesse disputando "espaço" com ela. É interessante notar que o sexto tópico (sobre sistemas complexos e embarcados) seria de vital importância para o futuro, pois estaria aliando esta nova linguagem às plataformas móveis da Microsoft®. Já o sétimo tópico é apenas a formalização de que não se pretendia realizar uma tarefa quase impossível. Como vimos anteriormente, as máquinas virtuais se encarregam de muitas operações extras, o que, mesmo com as otimizações, torna muito difícil a obtenção de desempenho, em velocidade ou tamanho, comparável com a de C ou de Assembly. O resultado produzido por essas linhas de projeto foi uma linguagem parecida com C e C++ que é muito produtiva e capaz.

Uma idéia muito interessante que veio junto com o C♯ foi a CLI. Ela permite que as linguagens que implementem certas regras compartilhem seus binários. A Figura 2.5 mostra uma visão geral da arquitetura da CLI, que será discutida posteriormente.

Com a CLI a Microsoft® pretendia oferecer aos seus clientes a possibilidade de migrar para a nova linguagem (C♯) sem ter que rejeitar o legado pré-existente que usava outras tecnologias (Visual Basic ou C++), criando uma base inicial de

consumidores. Hoje, mais de 20 linguagens compõem a CLI, entre elas C++[23], C♯, F♯, Boo e reimplementações de Lisp (L♯), Prolog (P♯), Python (IronPython), Ruby (IronRuby) e Java (IKVM[24]).

Embora C♯ seja uma das linguagens que suportam a CLI, ela tem extensões que podem impossibilitar o compartilhamento de parte dos binários. A CLI não prevê o uso de ponteiros, e portanto o uso deles nas interfaces torna o método impossível de se compartilhar. É interessante notar que um método que não esteja conforme a CLI não torna todo o binário não conformante, mas impede a classe onde ele foi definido de ser compartilhada com as demais linguagens.

Uma consequência interessante da CLI é a necessidade de uma bibloteca padrão ampla, para garantir a uniformidade entre as linguagens que a compõem. Ainda como efeitos desta característica temos a facilidade de escrever códigos portáveis (pois as diferenças ficam concentradas na biblioteca padrão), o desenvolvimento mais ágil (não há necessidade de aprender novas bibliotecas para cada plataforma e/ou atividade) e ainda a facilidade de manutenção do código.

O padrão da CLI define dois perfis de bibliotecas que podem ser adotados pelas implementações: um básico (*Kernel Profile* – semelhante, na filosofia, ao J2ME, que é um subconjunto das funcionalidades oferecidas pelo padrão Java para viabilizar o uso em dispositivos portáteis), que tem apenas as classes básicas; e um mais completo (*Compact Profile*) que conta com bibliotecas para rede e XML (do inglês, *Extensible Markup Language*.). Esses perfis aparecem em negrito na Figura 2.6, onde também se vêem as bibliotecas que fazem parte dos perfis. Deve-se notar que o *Kernel Profile* é apenas um subconjunto (bastante restrito) do *Compact Profile*.

Ainda na Figura 2.6 pode se perceber que algumas bibliotecas, as que estão em itálico, não fazem parte de nenhum perfil. Muitas outras bibliotecas podem ser oferecidas pelos implementadores, mas o padrão define um conjunto mínimo de classes e funcionalidades que devem aparecer nessas bibliotecas.

Na Figura 2.7 podemos ver alguns exemplos de classes que compõem as bibliotecas. Em negrito estão as bibliotecas acompanhadas pelo número de classes definidas no padrão (ex. **Reflection (28)**); em itálico estão categorias que **não** aparecem no padrão e foram inseridas apenas para facilitar a visualização.

Embora o número de classes especificadas seja grande, apenas uma fração do *framework* oferecido pela Microsoft® está incluído na especificação; isso poderia impedir outros implementadores de oferecer alternativas semelhantes, já que estariam violando direitos de propriedade intelectual. Microsoft®, HP® e Intel® detêm a patente do C♯ e da CLI, mas tanto a ECMA quanto a ISO requerem que as partes

[23]Na verdade uma variante de C++ disponibilizada pela Microsoft® no Visual Studio.

[24]Esta é uma reimplementação da máquina virtual Java que roda sobre o projeto Mono ou o .Net Framework.

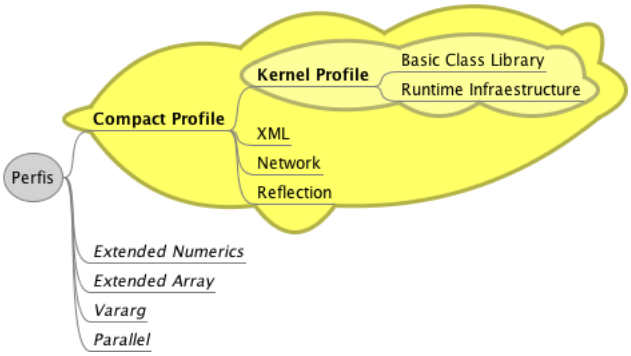

Figura 2.6: Organização das bibliotecas que compõem a CLI. Baseado em [38].

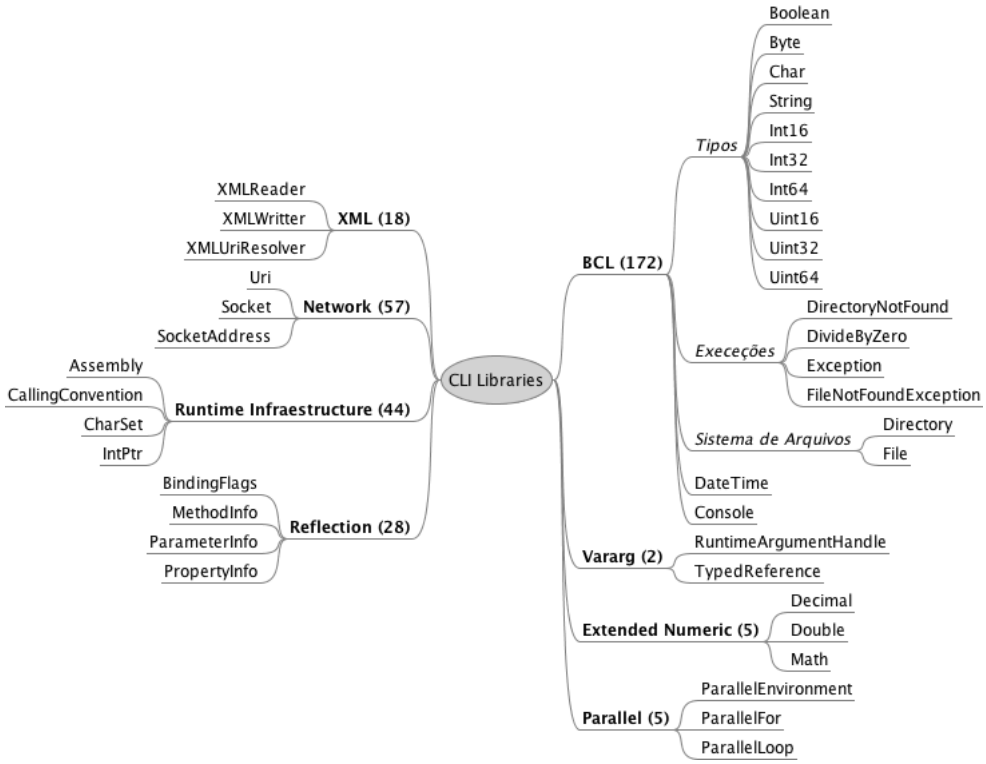

Figura 2.7: Exemplos de classes definidas no padrão ECMA 335. Baseado em [38].

padronizadas estejam sob licença RAND (Reasonable And Non Discriminatory).

Nesta seção fizemos uma apresentação tanto do C♯ quanto da CLI. O leitor pode ficar com a impressão de que tratamos a CLI com maior profundidade que o C♯, mas na verdade, por ser o C♯ uma das linguagens compatíveis com a CLI, tudo que falamos sobre esta é válido para aquela. Para mais detalhes sobre o C♯, a oitava cláusula do padrão da linguagem (ECMA 334 [37]) oferece um extenso tutorial.

2.2.2 CLI

Como visto anteriormente, a CLI é um esforço para viabilizar a interoperabilidade entre linguagens de programação[25], ou seja, a capacidade de linguagens compartilharem procedimentos, áreas de memória e até mesmo objetos. Esta não é uma ideia nova nem tampouco uma nova possibilidade.

Linguagens compiladas, como C e Fortran, podem compartilhar seus recursos somente adaptando os nomes dos símbolos se tiverem os arquivos intermediários (*.o, *.obj) mesclados. Linguagens interpretadas têm alto grau de "permeabilidade" com as linguagens de seus compiladores e, usualmente, compartilham todo tipo de informação com elas. Mais recentemente, CORBA (*Common Object Request Broker Architecture*) trouxe uma nova noção de compartilhamento, ao viabilizar a sincronização de objetos em diferentes linguagens sendo executados em diferentes computadores.

O compartilhamento usando os códigos intermediários só funciona em situações especiais em que não existe ou não é usado o conceito de objeto, e, ainda assim, depende do programador[26] para adaptá-la à nomenclatura usada pelos compiladores[27] ou utilizar tipos correspondentes nas duas linguagens. Para esta forma de compartilhamento se aplicar às linguagens orientadas a objetos, é necessária a compatibilidade dos modelos de memória das linguagens, ou seja, as linguagens precisam entender os objetos das outras, o que dificilmente ocorre sem ser um requisito de projeto.

O compartilhamento de váriaveis e funções de linguagens interpretadas com as linguagens que implementam o seu compilador, como entre Lua e C, também ocorre largamente. Tanto novas funções em C podem estender o elenco de funções disponíveis em Lua quanto *scripts* Lua podem ser chamados em um programa C. O mesmo ocorre com os pares Python e C; Iron Python e C♯; Jython e Java; e muitas outras possibilidades.

O compartilhamento usando CORBA exige que se monte uma grande estrutura em torno de cada código envolvido nesse relacionamento, o que torna o processo lento e demorado. Entretanto, esta tecnologia mostrou que seria necessário, e possível, ter uma estrutura intermediária que funcionasse como elemento de ligação entre as linguagens. A CLI se baseia nesta idéia para atingir seus objetivos.

---

[25]Durante a pesquisa bibliográfica não foi posssível encontrar uma definição formal. Entretanto, [16] propõe uma definição para o caso de linguagens orientadas a objeto que, em suma, exige compatibilidade das classes e dos objetos, dispensa o uso de códigos ou compiladores intermediários e permite interoperabilidade segura entre os tipos.

[26]O programador pode fazer as adaptações necessárias manualmente ou utilizar ferramentas como a Chasm[24] e a Babel[20].

[27]Os compiladores C, usualmente, adicionam um "_" na frente dos símbolos.

Para alcançar o nível de integração proposto pela CLI, deve haver um acordo prévio entre os desenvolvedores de bibliotecas, compiladores e aplicações, de forma a criarem um núcleo comum às linguagens. Segundo o padrão que normatiza a CLI [38], a CLS (do inglês, *Common Language Specification*) é um acordo estabelecido entre os projetistas das linguagens e das bibliotecas que estabelece uma série de convenções, garantindo que as bibliotecas possam ser usadas por várias linguagens.

A CLS é um conjunto de regras (sobre a nomenclatura, os tipos e os métodos, entre outras que estão listadas em [33]) comuns às linguagens que suportam a CLI, que garante o "entendimento" entre elas. Uma linguagem pode ser fiel à CLS e, ainda assim, oferecer recursos adicionais que não estão disponíveis neste núcleo e, possivelmente, em outras linguagens que fazem parte da infra-estrutura. Esta é uma situação interessante para aumentar o número de linguagens que podem aderir à CLI, como no caso de C♯, que suporta o uso de ponteiros, ao contrário da CLI.

Como foi dito anteriormente, um dos impedimentos na criação deste ambiente compartilhado é a necessidade de compatibilidade entre os objetos das linguagens. Na estrutura da CLI é o Sistema Compartilhado de Tipos (CTS – *Common Type System*) a entidade que garante que os tipos são compatíveis, abstraindo das diferenças entre as linguagens.

O CTS define de que forma os tipos são declarados, usados e manipulados, além de definir alguns tipos básicos, como **float32**, **float64** (IEC 60559:1989) e **native int** (size_t), e ainda algumas operações-padrão sobre os tipos. Este sistema oferece a base sobre a qual os compiladores das linguagens que oferecem suporte à CLI serão implementados. Note que existe uma área em comum entre o CTS e a CLS; com efeito, a CLS define um subconjunto do CTS.

O CTS é implementado e tem o seu perfeito funcionamento assegurado pelo Sistema de Execução Virtual (VES – Virtual Execution System). Este sistema também é responsável por carregar e executar os programas feitos para a CLI e os metadados que serão usados pelo VES para "conectar" em tempo de execução módulos que foram gerados separadamente (por exemplo, um executável e uma biblioteca).

Os metadados, que, como vimos anteriormente, são informações extras sobre o conteúdo do binário, descrevem de forma neutra às linguagens as classes e seus membros (métodos, campos[28] e outros tipos), e atributos[29]. Durante a execução, esses dados serão usados pelo sistema de execução para achar informações vitais para o processo, entre outras as interfaces implementadas e as heranças. Deve-se notar que os metadados são focados na descrição das classes. A CLI não prevê a possi-

[28]Estes são referidos em C++como atributos.

[29]Estes são modificadores de tipos e membros.

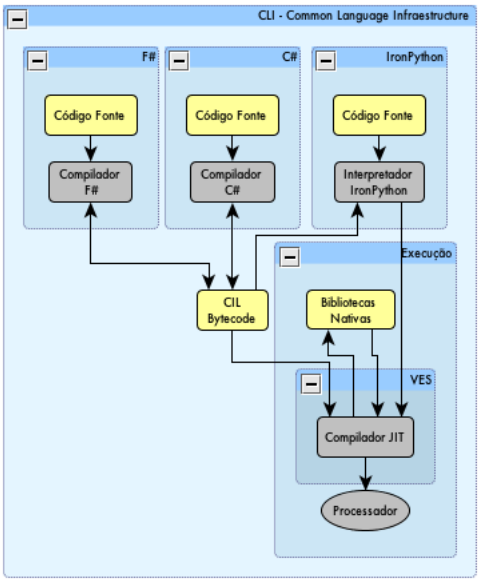

Figura 2.8: Visão detalhada da CLI.

bilidade de artefatos não pertecentes a classes, o que justifica esta "imparcialidade" dos metadados.

Vale enfatizar que uma linguagem de programação, entidade puramente abstrata, se realiza unicamente através dos compiladores ou do interpretador. Sendo assim, essas ferramentas desempenham um papel fundamental no funcionamento da CLI. Toda essa estrutura foi construída sempre com o principio de reúso do código-fonte em mente, mas as mudanças propostas nas linguagens são extensas. É nesse momento que os compiladores se mostram fundamentais, assumindo a responsabilidade de mudar toda a estrutura interna das linguagens, mantendo a mesma interface (gramática, sintaxe) com o usuário.

Os compiladores das linguagens da CLI transformam código-fonte, como vimos nas seções anteriores, em *bytecode*, ao qual, no caso da CLI, dá-se o nome de Linguagem Intermedária Compartilhada (CIL – *Common Intermediate Language*). A Figura 2.8 mostra o funcionamento da CLI do ponto de vista do usuário.

Como ilustrado na Figura 2.8, na estrutura da CLI os compiladores de todas as linguagens geram binários CIL; e portanto, uma vez o que binário foi criado, todas as linguagens podem compartilhá-lo, inclusive as interpretadas. O VES (Virtual Execution System) é o responsável por, durante a fase de execução, converter as instruções CIL em instruções binárias para o processador. O VES também se encarrega da interface com o código nativo, o que torna muito simples a invocação desses códigos, como veremos adiante.

É importante ressaltar que a CIL assegura a portabilidade dos códigos gerados pela CLI. A especificidade de cada plataforma ou arquitetura fica encapsulada na

máquina virtual, deixando a CIL – e portanto os *assemblies* (nome dado aos binários CIL) – invariantes com respeito às plataformas e arquiteturas.

Na próxima seção serão discutidas algumas questões relacionadas à portabilidade dos binários gerados para a CLI.

2.2.2.1 Portabilidade

Como vimos na seção anterior, os binários gerados para a CLI são independentes de uma plataforma ou arquitetura, o que significa que não precisam de alterações para serem "executados" em qualquer sistema operacional ou arquitetura de processador, graças ao uso da CIL e das máquinas virtuais. Entretanto, é necessário que haja uma implementação do sistema de execução disponível para a configuração em que se pretende usar esses binários.

O Projeto Mono é uma das implementações disponíveis da CLI. Este é um esforço da comunidade de *software* livre para criar todo o ambiente necessário para desenvolver e executar um aplicativo que utiliza a CLI. O projeto, que é gerido pela Novell®, obtém grande sucesso na implementação de uma IDE (*Integrated Development Environment* – o Monodevelop), compilador e máquina virtual. Este projeto disponibiliza seus produtos para Windows®, Linux (OpenSUSE, SLE, Debian/Ubuntu) e Mac OS X.

O projeto DotGNU, que é parte do projeto GNU da *Free Software Foundation*, portanto *software* livre, também implementa toda a infra-estrutura da CLI necessária para o desenvolvimento e execução dos binários, inclusive em arquiteturas (ppc, arm, parisc, s390, ia64, alpha, mips, sparc) e sistemas (GNU/Linux em – PlayStation 2, Xbox –, BSD, Cygwin/Mingw32, Solaris, AIX) mais exóticos.

Por ter sido desenvolvido originalmente pela Microsoft®, existe a crença de que o C♯ só seja capaz de ser desenvolvido ou ter seus programas executados em ambiente Windows®. Como mostramos, a CLI como um todo, o que inclui linguagens como C♯ e VB, pode ser executada em diversas configurações.

Quando falamos das idéias por trás do projeto do C♯, na Seção 2.2.1, uma das diretivas adotadas pelos projetistas dessa linguagem era a portabilidade do programador. Incluímos um breve exemplo para ilustrar este conceito.

Este exemplo consiste de três trechos de código: um trecho simples de código XML (Trecho 2.9) e dois outros trechos escritos em Boo (Trecho 2.10) e C♯ (Trecho 2.11), respectivamente, que processarão o trecho inicial.

```
<MusiC>
  <MusiC-Classify >
    <File path="file1" />
    <Dir path="dir1" />
    <Playlist path="playlist1" />
  </MusiC-Classify >
</MusiC>
```

Trecho 2.9: Arquivo XML que será processado pelos Trechos 2.10 e 2.11.

Os códigos 2.10 e 2.11 devem ser comparados, para além das diferenças sintáticas das linguagens. Por exemplo, deve perceber-se que as funções e classes que estão sendo usadas são as mesmas e que são compostas pelos mesmos métodos. Repare no uso da classe **XmlDocument** e dos métodos **XmlDocument.Load** e **Console.WriteLine**.

```
import System.Xml

#booi xml_parser.boo

class MusiCXmlHandler:
  def Evaluate( node as XmlNode ):
    if node.Name == "MusiC-Classify":
      MusiC_Classify(node);
      return;

  def MusiC_Classify(music_class as XmlNode) :
    for node as XmlNode in music_class.ChildNodes:
      System.Console.WriteLine(node.Name + ": " + node.Attributes["path"].Value);


doc = XmlDocument();

doc.Load("xml_source.xml");

root = doc.ChildNodes[0];

if root.Name != "MusiC":
  System.Console.WriteLine("Wrong Format !"); # ou print "Wrong Format !"

hnd = MusiCXmlHandler();

for child as XmlNode in root.ChildNodes:
  hnd.Evaluate(child);
```

Trecho 2.10: Programa em Boo para ilustrar a portabilidade do programador.

```
using System.Xml;

class MusiCXmlHandler
{
  public void Evaluate( XmlNode node )
  {
    if (node.Name == "MusiC-Classify")
      MusiC_Classify(node);
  }

  public void MusiC_Classify( XmlNode music_class )
  {
    foreach (XmlNode node in music_class.ChildNodes)
      System.Console.WriteLine(node.Name + ": " + node.Attributes["path"].Value);
  }
}
```

```
18 class app
   {
20   public static void Main(string[] args)
     {
22     XmlDocument doc = new XmlDocument();
       doc.Load("xml_source.xml");
24
       XmlNode root = doc.ChildNodes[0];
26
       if ( root.Name != "MusiC" )
28       return;

30     MusiCXmlHandler  hnd = new MusiCXmlHandler();

32     foreach (XmlNode child in root.ChildNodes)
         hnd.Evaluate(child);
34   }
   }
```

Trecho 2.11: Programa em C♯ para ilustrar a portabilidade do programador.

Esses exemplos mostram que, com o uso da CLI, mesmo códigos escritos em linguagens bem diferentes podem ser suficientemente parecidos, o que facilita a manutenção e a migração (portabilidade) do programador de uma linguagem para outra.

2.2.2.2 Interfaceamento com Código Nativo

Quando surge uma nova tecnologia, a compatibilidade é um fator importante no processo de aceitação. Esta questão é ainda mais relevante se a tecnologia atual já foi depurada ao longo dos anos, e especialmente em aplicações sensíveis a erros. Este é o caso de aplicações na área médica, na aviação e no sistema financeiro.

Para resolver os problemas de compatibilidade com os códigos legados (frequentemente C, C++, Fortran, Algol), as aplicações em Java fazem uso da Interface Nativa Java (JNI – *Java Native Interface*). Os Trechos 2.12 e 2.13 usam a JNI para importar a função **exported_func** usada nas seções anteriores (Trecho 2.6).

```
1 // javac -encoding UTF-8 NativeInterface.java

3 public class NativeInterface
  {
5   static
    {
7     System.loadLibrary("native2java");
    }
9
    public static void main(String [] args)
11  {
      System.out.println(exported_func(8));
13  }

15  public static native int exported_func(int i);
```

```
}
```

Trecho 2.12: Programa Java que chama função nativa.

No Trecho 2.12, a classe **NativeInterface** implementa um construtor estático, o método **main**, e declara um método **exported_func**. O construtor, que só precisa ser executado uma única vez, foi marcado como estático. Ele é o responsável por carregar a biblioteca **native2java**, da qual falaremos adiante. O método **NativeInterface.main** é o ponto de entrada da aplicação, e portanto completamente equivalente ao tradicional **int main(int argc, char ** argv)**. Por último, o método **exported_func** funciona como um ponteiro para uma casca em volta da função que queremos importar. Este método foi marcado pela palavra-chave **native**, indicando que a implementação dele será fornecida através de um método nativo. O Trecho 2.13 mostra a implementação desta casca.

```
#include <jni.h>

// g++ -dynamiclib native_wrapper.cpp -o libnative2java.dylib -I/Library/Java/Home/
    include -L. -lplugin

extern "C" int exported_func();

extern "C" jint Java_NativeInterface_exported_1func(JNIEnv *e, jobject _this, jint
    arg1)
{
  // arg1 + 48
  return arg1 + exported_func();
}
```

Trecho 2.13: native2java. Biblioteca criada para formar uma "casca" em volta da biblioteca que queremos importar.

No caso específico que estamos implementando, a biblioteca **native2java** não seria realmente necessária, pois temos acesso ao código-fonte. Entretanto, queremos simular a situação em que não possuíssemos o código-fonte, ou não fosse conveniente mudá-lo.

É de causar estranheza o nome da função que está sendo implementada pela biblioteca. Infelizmente, este é o único nome que esta função pode ter. Ele é composto pelo prefixo "Java_" acrescido do nome da classe (NativeInterface) modificado, seguido por um "_" e pelo nome da função-alvo também modificado e, em casos onde estas informações não determinam somente uma função, seguido por "__" e pelos nomes dos tipos que compõem a lista de argumentos após terem sido processados. Tanto o nome da classe quanto das eventuais funções devem ser modificados segundo [39]. Cada caractere "_" será transformado para "_1", ocorrendo transformações semelhantes com caracteres como ";", "[" e todos os não-ASCII. Esta regra pode ser encontrada na documentação do pacote de desenvolvimento Java versão 1.6, mantido pela Sun® [39]

Felizmente, o C♯ usa um método bem mais amistoso para incorporar código legado em uma aplicação. O Trecho 2.14 usa a **PInvoke** (*Platform Invocation*), uma técnica que permite que o C♯ tenha acesso aos códigos legados de maneira bastante simples.

```
1 using System;

3 class Native
  {
5   public static void Main(string[] s)
    {
7     System.Console.WriteLine( exported_func() + 8 );
    }
9
    [DllImport("MusiC.Native.Core.dll", EntryPoint="GetFeature")]
11  public static extern int exported_func();
  }
```

Trecho 2.14: Programa C♯ que chama função nativa.

A classe **Native** implementa um método **Main** que, assim como na versão Java, é completamente equivalente ao **int main(int argc, char ** argv)** de C. O método **exported_func** é também um ponteiro, mas que aponta diretamente para a implementação nativa. Este método está marcado com a palavra-chave **extern**, que desempenha papel semelhante ao da palavra-chave **native** que aparece na implementação em Java. O atributo **DllImport** passa informações sobre como achar e se comunicar com o método que será importado.

O primeiro "argumento" do atributo é o nome do binário, que não precisa seguir nenhuma convenção específica. Através do **EntryPoint** pode-se passar o nome da função que será importada. Este argumento é opcional, e caso não seja especificado será usado o nome do método. Esta é a forma "curta" de usar o **PInvoke**, que, embora prática, mantém o nome da biblioteca fixo. O uso do atributo **DllImport** é apenas uma forma mais palatável de chamar os métodos da API responsáveis por carregar a biblioteca, que podem ser chamados diretamente pelo programador para maior flexibilidade.

Assim encerramos a discussão sobre a CLI, dando-se por encerrada a discussão sobre as tecnologias que foram usadas no desenvolvimento desse projeto. Na próxima seção será feito um breve resumo do que tratou este capítulo.

2.3 Resumo

Começamos este capítulo com as origens dos sistemas programáveis. Lembramos momentos importantes nessa longa história, como o primeiro dispositivo programável, uma banda de música do século XIII, e a primeira linguagem de programação, a *plainkalkül*. Na sequência, chamamos a atenção para algumas característcas das

linguagens interpretadas, principalmente, a "não-causalidade" do interpretador e as semelhanças dessas linguagens com outros tipos de linguagem. Ainda na primeira seção mostramos os conceitos básicos das linguagens gerenciadas e do gerenciamento de memória. Finaliza esta seção um breve tutorial sobre Reflexão, uma técnica muito conveniente para a criação de extensões e usada largamente neste processo.

Na segunda seção fizemos uma breve introdução histórica do C♯ e da CLI, ambos padrões ECMA e ISO orignalmente desenvolvidos pela Microsoft®. Em seguida foi feita uma extensa descrição da CLI, mostrando uma implementação real do que fora discutido na seção anterior, incluindo comentários sobre a portabilidade e como ocorre a interface com códigos nativos neste sistema.

No próximo capítulo discutiremos alguns detalhes do algoritmo de classificação de gêneros musicais que foi implementado usando este projeto.

Capítulo 3

O Método de Barbedo e Lopes

Neste capítulo faremos algumas considerações sobre um trabalho de Jayme Barbedo e Amauri Lopes, *Automatic Genre Classification of Music Signals* [13]. Este artigo foi escolhido como o estudo de caso a ser implementado usando as facilidades desenvolvidas neste projeto.

3.1 Visão Geral

O processo que Barbedo e Lopes descrevem em seu artigo é, do ponto de vista matemático, bastante simples, e não exige sofisticação matemática do leitor. Entretanto, o algoritmo descrito é complexo e exige muito dos recursos computacionais.

A execução do MBL[1], assim como a da maioria dos métodos de recuperação de informações de músicas (*Music Information Retrieval* – MIR), é formado por duas etapas [14]. A primeira etapa, a extração, se baseia no cálculo de características (não necessariamente espectrais) de seções dos dados acústicos (tipicamente a forma de onda, mas existem trabalhos que usam MIDI, por exemplo [14, 7]). A outra etapa é a aplicação de algoritmos de reconhecimento de padrões aos dados gerados na etapa anterior para estimar, no caso do MBL, o gênero. Posteriormente, serão dados os detalhes de como o algoritmo executa cada uma dessas fases.

Vale dizer, existe uma terceira fase "escondida" nesse processo. Normalmente, um algoritmo de reconhecimento de padrões precisa de um referencial (um padrão) que possa ser comparado com os dados submetidos a ele para então determinar qual o padrão mais próximo. Assim, é usual encontrar nesses algoritmos uma fase de treinamento que criará os padrões que serão usados no reconhecimento e que, com efeito, raramente (apenas nos casos em que o sistema evolui) faz parte da execução.

[1]Com o objetivo de tornar o texto mais simples para o leitor, sempre que for preciso fazer referência ao Método de Barbedo e Lopes será usada a sigla MBL.

Na Seção 3.5 discutiremos o método de treinamento adotado pelo MBL.

Existem ainda duas outras etapas que não são, de fato, pertencentes ao processo de classificação, mas das quais este processo depende: a criação da base de dados e a seleção das características que serão usadas para descrever as músicas e os gêneros.

Durante o desenvolvimento deste trabalho, encontramos a necessidade de criar uma base de dados para a implementação deste trabalho, mas deixamos tanto a discussão sobre as circunstâncias que nos levaram a isto quanto um detalhamento desta base para o Capítulo 5, no qual discutiremos os testes realizados.

Este artigo foi escolhido para implementação por ser um trabalho recente (2005) e para se prestigiar a produção científica nacional, já que este trabalho é uma das poucas contribuições de autores brasileiros nesta área.

3.2 Considerações sobre a Base de Dados

Os autores testaram o método proposto com duas coleções de músicas: uma que aparenta ter sido montada por eles e a *Magnatune Database*[2], que está disponível na página do *5th International Conference on Music Information Retrieval* (Ismir'04) [18]. Cada uma dessas coleções apresenta particularidades que serão discutidas a seguir.

A coleção montada pelos autores[3], e usada para o desenvolvimento do método, é composta por 2266 arquivos organizados em 29 gêneros. Cada estilo é representado por, no mínimo, 40 arquivos, dos quais os 20 considerados mais característicos do gênero, segundo critérios subjetivos, serão usados para treinamento. Esses arquivos foram amostrados a 48 kHz com 16 bits e se originam de fontes diversas, tais como *compact discs*, transmissões de rádio pela internet e arquivos codificados usando MP3, WMA, OGG ou AAC. Os arquivos desta coleção são **trechos** de músicas, entretanto não há informação sobre a duração (possivelmente 32 segundos[4]) ou sobre como este trecho foi escolhido (do inicio? do fim?). Os autores tomaram alguns cuidados ao construir a base: evitaram repetições de intérpretes e de músicas, tentando descorrelacionar ao máximo os arquivos entre si; e também não incluíram faixas que foram submetidas a codificadores psicoacústicos no conjunto de treinamento.

[2]Para não sobrecarregar o texto com uma nomenclatura extensa, usaremos MD para fazer referência a esta coleção.

[3]Aparentemente os autores consideram esta base parte da metodologia proposta. Entendemos a estreita conexão que existe entre esses dois elementos (base e metodologia), mas não compartilhamos da mesma opnião.

[4]Tamanho do trecho de análise usado no artigo.

Já a MD é composta por 1458 arquivos codificados em MP3 a 128 kbps[5], organizados em 6 grupos de gêneros[6] (Clássico, Eletrônica, *Jazz_Blues*, *Metal_Punk*, *Rock_Pop* e *World*). Ela é formada por arquivos de músicas **completas**, e não trechos como a coleção anterior, e inclui diversas contribuições de um mesmo intérprete, o que pode distorcer os resultados.

Esta base já está dividida em dois conjuntos de faixas: o de treinamento e o de teste. O conjunto de treinamento é composto pelas faixas que servem de modelo para classificar as demais. Idealmente, este conjunto deveria conter apenas faixas cuja classificação subjetiva fosse inequívoca, o que raramente ocorre. Já o conjunto de testes é composto pelas faixas que serão classificadas pelo algoritmo.

A Tabela 3.1 caracteriza a distribuição dos gêneros do conjunto de treinamento da MD pelo número de faixas e pela duração total. Em negrito estão os totais dos grupos que contam com mais de 1 gênero. Como os arquivos da MD têm duração variável, o número de vetores possíveis por arquivo também é variável; a caracterização pela duração total é, em geral, mais adequada, já que é uma boa estimativa da distribuição do número de vetores por gênero. Entretanto, este algoritmo trabalha com um número variável de arquivos, mas considera apenas 32 segundos de cada um deles, o que torna a descrição pelo número de faixas mais relevante.

Como podemos ver na Tabela 3.1, a distribuição está longe de ser equilibrada. Alguns gêneros, como o *jazz*, não estão bem representados na MD (este, em particular, está mal representado pelos dois critérios), enquanto que outros gêneros, como o clássico, ocupam grande parte da base.

A Tabela 3.2 ajuda a entender a composição dos grupos de gênero. Esses grupos estão dispostos nas colunas, ao passo que nas linhas aparecem os gêneros que as faixas apresentavam no respectivo campo do formato MP3. Pode-se perceber que alguns gêneros, notadamente o grupo *Rock_Pop*, são formados por músicas de uma grande variedade de estilos. Ainda na Tabela 3.2 é preocupante o "compartilhamento" de estilos que ocorre entre, por exemplo, *Metal_Punk* e *Rock_Pop*; e Clássico e *World*.

Com esta discussão, tentamos descrever algumas das sutilezas que estão presentes nas bases de dados usadas no artigo. Uma vez concluída a descrição, discutiremos na seção que segue o método usado por Barbedo e Lopes para selecionar os descritores que fazem parte do MBL.

---

[5]Nota de implementação: Esses arquivos foram convertidos para formato *wave*, 44.1kHz para serem processados.

[6]Embora a base agrupe esses gêneros no gabarito, eles estão disponíveis na estrutura de diretórios separadamante.

Conjunto	Gênero	Faixas	% de Faixas	Duração	% da Duração
Conjunto de Treinamento	Clássico	320	43,90 %	17:49:16	34,74 %
	Eletrônica	115	15,78 %	10:30:06	20,47 %
	Jazz	12	1,65 %	1:03:32	2,06 %
	Blues	14	1,92 %	35:52	1,17 %
	subtotal	**26**	**3,57 %**	**1:39:24**	**3,23 %**
	Metal	29	3,98 %	2:24:15	4,69 %
	Punk	16	2,20 %	43:54	1,43 %
	subtotal	**45**	**6,18 %**	**3:08:09**	**6,12 %**
	Rock	95	13,03 %	5:49:43	11,36 %
	Pop	6	0,82 %	27:29	0,89 %
	subtotal	**101**	**13,85 %**	**6:17:12**	**12,25 %**
	World	122	16,74 %	11:54:01	23,20 %
	Total	**729**	**100,02**%	**51:18:08**	**100,01**%

Tabela 3.1: Distribuição dos gêneros da *Magnatune Database* no conjunto de treinamento por número de faixas e por duração total.

	Grupos de Gêneros									
Gêneros	Cl	El	Jz	Bl	Mt	Pk	Rk	Po	Wl	Legenda
Cl	**315**									Clássico
Et	5						5		95	Étnico
El		**105**								Eletrônica
Ac		1								*Acid*
Am		2								Ambiente
Tc		2								*Techno*
Tr		2								*Trance*
Jz			**12**							*Jazz*
Bl				**14**			2			*Blues*
Mt					**9**					Metal
Hr					11		2			*HardRock*
Pk						**11**	9			*Punk*
Pr						5	7			*Punk Rock*
Rk					9		**63**			*Rock*
Na							2		8	*New Age*
Rt							4			Retro
Th							1			*Trip-Hop*
Po								**6**		*Pop*
Fl									11	Folclórico
Ot										Outros
NAv		3							8	Não Definidos
total	320	115	26		45		101		122	

Tabela 3.2: Composição dos gêneros no conjunto de treinamento da *Magnatune Database*.

3.3 Seleção das Características

Nesta seção vamos descrever o processo descrito por Barbedo e Lopes para determinar quais as **características** que melhor se adaptam ao algoritmo. Este método otimiza o conjunto de descritores, a partir de um conjunto inicial arbitrário, sempre buscando a melhor taxa de acerto.

Mas o que são características? Características (do inglês *feature* ou ainda *fingerprint*) ou descritores são operações matemáticas feitas sobre um trecho de áudio, frequentemente mas não exclusivamente no domínio da frequência, com o objetivo de extrair propriedades particulares do trecho. Essas propriedades podem ser usadas para diversas atividades, por exemplo, classificação de gênero, intérprete ou compositor, ou identificação de humor.

O processo descrito pelos autores parte de um conjunto inicial de 13 descritores: audibilidade (*loudness*), largura de banda (*bandwidth*), fluxo espectral (*spectral flux*), *spectral rollof*, centróide espectral, taxa de cruzamento por zero, freqüência fundamental, razão de quadros de baixa energia e os 5 primeiros coeficientes cepstrais; e chega ao conjunto ótimo de características para a trinca algoritmo – base de dados – conjunto inicial.

Primeiramente, foi criado um *ranking* baseado no resultado obtido por cada uma das características ao classificar os gêneros musicais individualmente. A seguir, o MBL foi executado incluindo todos os descritores do conjunto inicial, o que deu ínicio a um processo recursivo que retirava a pior característica, segundo o *ranking* criado inicialmente, e realizava um novo teste. Este procedimento foi reiterado até que restassem apenas dois descritores.

Os autores relatam que se o número de características for menor ou maior que quatro ocorre redução na taxa de acerto. Este curioso comportamento acontece porque: quando o número de dimensões aumenta, a informação que está sendo adicionada não é relevante para a classificação, o que confunde o classificador; quando o número de características decresce, detalhes importantes podem ser jogados fora, o que também é prejudicial.

A Figura 3.1 ilustra uma situação (ideal) onde aumentar o número de dimensões seria prejudicial. A figura mostra um espaço E_1 onde as classes C_1 e C_2 são separáveis. Suponha que ao adicionar uma nova dimensão, D_n, a projeção das duas classes sobre esta dimensão, sejam coincidentes, por exemplo, R_1. Como seria impossível diferenciar as ocorrências de cada classe nesta dimensão, o resultado final desse acréscimo de informação seria a piora do sistema de classificação.

É importante ressaltar que o resultado obtido pelo autor é, a princípio, válido

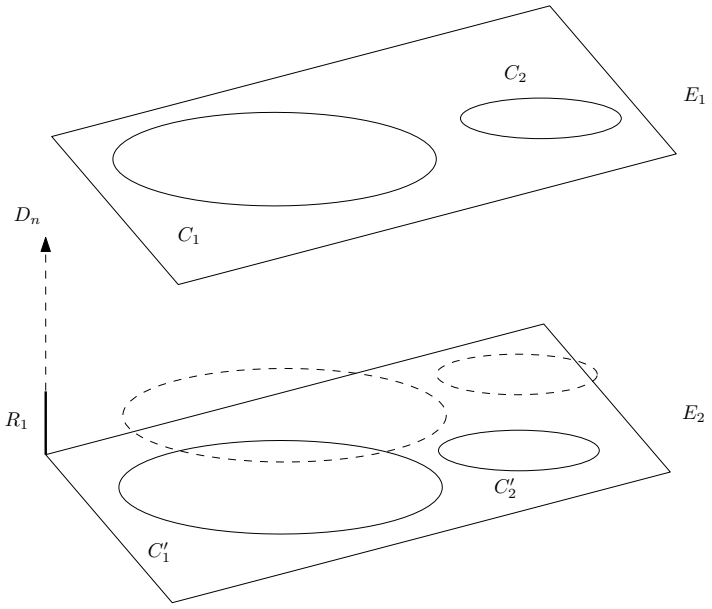

Figura 3.1: Exemplo de degradação na separabilidade de 2 classes por acréscimo de dimensão.

unicamente para o conjunto de características com que ele trabalha e para o algoritmo proposto. Mas isto de forma nenhuma invalida a estratégia que foi descrita. Este é um dos poucos esforços encontrados na literatura para se criar uma avaliação objetiva de um dado conjunto de características aplicadas à classificação de gênero.

Com isso, encerramos a discussão sobre o algoritmo para seleção de descritores. Na próxima seção, que trata da extração de características, veremos como serão usadas e quais foram as *features* selecionadas.

3.4 Extração

Nesta seção discutiremos a primeira etapa do MBL, a fase de extração de características. Esta fase é responsável por descrever os arquivos de áudio, e assim, prepará-los para as fases de treinamento e de classificação.

Como vimos na seção anterior, os autores utilizam apenas 4 características: *Spectral Rolloff*, *Loudness*, *Bandwidth* e *Spectral Flux*. Algumas dessas características, notadamente o *Spectral Rolloff*, possuem várias definições possíveis. Em [13] elas são definidas da seguinte forma:

Spectral Rolloff – SRO

$$\sum_{k=0}^{\mathrm{SRO}} X(k) = 0{,}95 \cdot \sum_{k=0}^{K} X(k) \tag{3.1}$$

Loudness – LD

$$\mathrm{LD}_i = \sum_{k=1}^{K} X(k)^2 \cdot 10^{W(k)/20} \tag{3.2}$$

$$W(k) = -0{,}6 \cdot 3{,}64 \cdot f(k)^{-0{,}8} - 6{,}5 \cdot e^{-0{,}6\cdot(f(k)-3{,}3)^2} + 10^{-3} \cdot f(k)^{3{,}6} \tag{3.3}$$

$$\tag{3.4}$$

Bandwidth – BW

$$\mathrm{BW}_i = \sqrt{\frac{\sum_{k=1}^{K}[(\mathrm{CE}_i - k^2) \cdot |X(k)|^2]}{\sum_{k=1}^{K} |X(k)|^2}} \tag{3.5}$$

$$\mathrm{CE}_i = \frac{\sum_{k=1}^{K} k \cdot X(k)^2}{\sum_{k=1}^{K} |X(k)|^2} \tag{3.6}$$

Spectral Flux – SF

$$\mathrm{SF}_i = \sum_{k=1}^{K} \{\log_{10}[X_i(k)] - \log_{10}[X_{i-1}(k)]\}^2 \tag{3.7}$$

Nas expressões, k é o índice da raia espectral, $|X_i(k)|$ é a magnitude da raia k de um quadro da STFT, i é o quadro da STFT e $f(k)$ é a freqüência em kHz associada a raia k.

A extração do MBL consiste no janelamento de arquivos de 32 segundos[7] usando a janela de Hamming com 1024 amostras de comprimento e sobreposição de 512. Para cada uma dessas janelas é calculada a DFT, que, por sua vez, será usada para o cálculo das *features*. Assim, teremos para cada quadro da STFT um vetor com dimensão 4.

No próximo passo, esses vetores serão agrupados de modo que cada grupo represente cerca de um segundo de música. Por exemplo, usando a taxa de 48 kHz, temos em um segundo $\frac{48000 \text{ pontos}}{512} - 1 = 92$ vetores. Sobre cada um desses grupos serão calculados média e variância, além da prevalência do pico principal (PPP), medida proposta pelo autor para medir a diferença entre os picos e os vales, e definida na equação (3.8). Assim, como cada arquivo tem 32 segundos, ao final do processo cada um deles será representado por 32 vetores, agora com dimensão 12 (média, variância e PPP de cada característica).

[7]Nota de Implementação: Nas bases compostas por arquivos completos de música, e não por trechos de 32 segundos, optou-se, no presente projeto, por processar apenas os 32 segundos centrais das faixas de áudio.

$$\mathrm{PPP}_{\mathrm{feat}}(j) = \frac{\max[ft(i,j)]}{(1/I)\cdot\sum_{i=1}^{I} ft(i,j)}, \tag{3.8}$$

onde i é o quadro da STFT, j é o grupo de 1 segundo e I é o numero de vetores em um grupo e feat é o descritor para o qual a PPP está sendo calculada.

Cabe dizer ainda que, como a representação de cada faixa de áudio é constante (32 segundos gerando 32 vetores), o número de faixas é, como argumentamos anteriormente, mais importante para descrever o volume de informação disponível sobre cada classe do que a duração total dele, pois nesse caso, o número de arquivos é diretamente proporcional ao número de vetores.

Desta forma, ao final da etapa de extração, os arquivos foram descritos e estão prontos para serem submetidos tanto ao processo de treinamento quanto ao de classificação. A próxima seção mostra o processo de treinamento proposto por Barbedo e Lopes.

3.5 Treinamento

Esta seção discute o método de treinamento proposto para o MBL. O objetivo desta fase é estabelecer, com base na representação das faixas de músicas, um modelo que represente cada um dos gêneros analisados. O MBL adota um esquema de modelagem dos gêneros que, ao invés de diferenciar os gêneros entre si, modela as diferenças par a par.

O primeiro passo é filtrar os vetores que descrevem o gênero, removendo aqueles que estão **longe** da média. Infelizmente o autor não revela detalhes do processo usado, dizendo apenas que um **grande** conjunto era formado após o descarte de vetores considerando fatores como a média e a faixa de valores esperada para cada dimensão.

Em seguida, é construído o modelo que será usado pelo algoritmo de reconhecimento. A idéia é construir um modelo para cada **par** de gêneros contendo um conjunto de seis vetores, três de cada gênero. Para um dado par de gêneros, todos os conjuntos possíveis são avaliados e o mais bem a avaliado será escolhido para modelar o par de gêneros. A avaliação consiste em calcular a distância euclidiana de cada vetor do conjunto a cada vetor de ambos os gêneros. Se o vetor que pertence ao conjunto tiver originado do mesmo gênero que o vetor da base, o conjunto tem a sua pontuação incrementada; em caso contrário, ela é mantida. Este método garante que o modelo escolhido é formado pelos 6 vetores que fornecem a maior separação para o par de gêneros em questão.

Barbedo e Lopes argumentam que o uso de mais componentes na composição do conjunto aumenta a probabilidade de que um vetor próximo de outra classe seja

incluído na representação do gênero, o que diminuiria a taxa de acerto. Afirmam também, que este fato é ainda mais grave em classes similares, e sugerem que o uso de um número variável de vetores por conjunto pode ser benéfico ao processo.

Como se pode perceber, o número de operações envolvidas no processo é gigantesco. Se apenas 20 vetores de cada gênero saírem da filtragem incial, o número de conjuntos possíveis pode ser calculado através de

$$C_3^{k_1} * C_3^{k_2} = \tag{3.9}$$

$$C_3^{20} * C_3^{20} = \tag{3.10}$$

$$1140 * 1140 \approx 1.300.000, \tag{3.11}$$

onde k_p é o número de vetores do p-ésimo gênero. Entretanto, se aumentarmos o número para 30 vetores (o que ainda significa descrever o gênero com representação menor que a de uma faixa) esse número cresce para cerca de **16 milhões** de iterações. Vale lembrar que a cada iteração são calculadas 6 distâncias a **todos** os outros vetores vetores que fazem parte dos dois gêneros (inclusive os que foram filtrados). Por questões práticas, e por omissão do autor, a implementação neste projeto limita o número máximo de vetores por gênero a 20.

Uma vez que são conhecidas as representações de cada par de gêneros, e das músicas, podemos dar início ao processo de classificação das faixas. A próxima seção trata do processo de classificação proposto por Barbedo e Lopes.

3.6 Classificação

Esta seção discute o processo de classificação proposto para o MBL. O objetivo desta fase é, com base na descrição dos gêneros, criada pelo treinamento, e das músicas, criada pela extração, avaliar qual seria o gênero musical desta faixa.

Este algoritmo se baseia em um processo de votação em que o mais votado é considerado o vencedor. Esta votação acontece em dois níveis: no nível da representação de 1 segundo de áudio, em que todas as combinações dos gêneros votam para decidir o "gênero" do trecho; e no nível da música, onde cada segundo vota para determinar qual gênero será atribuido à faixa.

Cada vetor que representa a faixa de áudio a ser classificada será marcado com o gênero vencedor da eleição, sendo cada par de gêneros votante. Havendo empate entre os gêneros, o vetor atual fica marcado como inconclusivo e o processo é continuado. Com todos os vetores avaliados, é feita nova eleição, onde, desta vez, os votantes são os vetores que representam a faixa de áudio que votarão nos gêneros com os quais foram marcados na eleição anterior. Caso haja empate nesta última

	Clássico	Eletrônico	Jazz	Metal	Rock	World
Clássico	**0,93**	0	0,01	0	0	0,06
Eletrônica	0	**0,73**	0	0,09	0,16	0,02
Jazz	0,04	0	**0,80**	0	0,08	0,8
Metal	0	0,06	0	**0,79**	0,15	0
Rock	0	0,02	0,02	0,13	**0,76**	0,07
World	0,02	0,06	0,08	0	0,11	**0,73**

Tabela 3.3: Matriz de confusão apresentada por Barbedo e Lopes para a *Magnatune*. Extraído de [13]

avaliação, é considerado vencedor o gênero mais votado na primeira eleição, aquela que decide o voto dos vetores. Se ainda assim permanecer o empate, o vencedor é escolhido através da realização de novo processo de classificação onde apenas os pares que contêm ao menos um dos gêneros empatados são usados. Caso ocorra novo empate, o vencedor é escolhido aleatoriamente.

Na Tabela 3.3, extraída de [13], estão os resultados obtidos por Barbedo e Lopes ao usar o MBL para classificar a MD. Chama a atenção o excelente desempenho obtido pelo algoritmo quando classificando músicas clássicas (93%). Entretanto, não temos como afirmar que esse resultado foi obtido treinando o MBL com a MD. Na verdade, o texto dá a entender que duas versões diferentes (para compatibilizar a taxonomia) da biblioteca criada pelos autores foram usadas nos dois testes apresentados no artigo.

A discussão sobre o processo de classificação adotado por Barbedo e Lopes encerra a discussão sobre este artigo. Na próxima seção será apresentado um pequeno resumo do capítulo antes de iniciarmos a apresentação do sistema.

3.7 Resumo

Neste capítulo discutimos o MBL [13], método este que foi escolhido para demonstração da biblioteca em funcionamento. Foram feitas, quando cabíveis, notas referentes à implementação, além de um estudo sobre a base de dados utilizada pelos autores e detalhamento do processo como um todo.

Começamos a apresentação do trabalho de Barbedo e Lopes fazendo criteriosa análise da base de dados utilizada em [13], mostrando algumas características que poderiam afetar os resultados encontrados. Em seguida, tratamos dos métodos usados para selecionar as características: descrevemos o processo e fizemos uma breve análise sobre a quantidade de *features* que foram usadas.

A Seção 3.4 deu início à discussão do artigo, tratando do processo de extração. Nesta seção foi discutido o processo de extração e definimos as características que são usadas no MBL. Na sequência foi apresentado o processo de treinamento e foram

feitas algumas críticas com relação à omissão de certos detalhes que impediram implementação mais fiel do método e as soluções que arbitramos. E, finalizando o capítulo, apresentamos o algoritmo de classificação além de uma breve análise sobre os resultados encontrados por Barbedo e Lopes.

Na sequência será apresentada a biblioteca desenvolvida por este projeto. Serão discutidos detalhes relacionados a uso, projeto e implementação.

Capítulo 4

Descrição do Projeto

Neste capítulo faremos a apresentação do *framework* desenvolvido durante este projeto. Descreveremos os objetivos, o funcionamento e a configuração do *software* desenvolvido, sempre enfatizando a participação do usuário no processo.

4.1 Visão Geral

Batizado de MusiC — a contração da expressão *Music Classification* — este projeto inicialmente faria a comparação de dois artigos de classificação de gêneros musicais (um deles descrito no capítulo anterior e usado para estudo de caso). Durante o desenrolar do projeto, julgamos que seria muito interessante se houvesse uma aplicação que permitisse ao usuário se preocupar-se apenas com o algoritmo e sua implementação, e não com a construção de todo um sistema.

Este *framework* foi criado para permitir que a pesquisa de novos algoritmos de classificação, principalmente de áudio[1], seja mais ágil. A idéia é fazer com que os pesquisadores desta área não precisem se preocupar com o desenvolvimento de partes de *software* que não estão diretamente relacionadas com os algoritmos (por exemplo, ler os arquivos de áudio e salvar os resultados da extração para minimizar o tempo de execução) e que seja fácil alterá-lo.

Nossa solução simplifica, através de arquivos XML, a configuração e descrição dos algoritmos, o que, em alguns casos, elimina a necessidade de qualquer conhecimento de programação por parte do usuário. Entretanto, na maior parte das vezes, as características, os classificadores, as janelas e as codificações implementadas pela biblioteca não serão suficientes, e por isso os usuários podem desenvolver suas próprias extensões e usá-las nos seus algoritmos; até mesmo outros configuradores que usem formatos diferentes do XML podem ser criados.

[1] A versão atual considera os dados unidimensionais e armazenados usando WAVE (representação de áudio). Algumas modificações podem ser feitas na biblioteca para aceitar dados de entrada mais genéricos.

Como o MusiC foi desenvolvido predominantemente em C♯, todas as extensões devem ser implementadas na mesma linguagem ou em linguagem que também faça uso da CLI. Se for necessário, o usuário pode, como foi feito com o classificador MBL, implementar em C♯ apenas uma casca sobre outra implementação (no exemplo citado em C++). O classificador foi implementado nesta linguagem para fazer uso das facilidades oferecidas pela GSL (*Gnu Scientific Library*), bem como das otimizações possibilitadas pelo uso de ponteiros em C++.

O C♯ foi a linguagem escolhida para o desenvolvimento do MusiC por permitir desenvolvimento bastante ágil, diminuindo o tempo necessário para completar a implementação do *software*; por ser mais compacta, o que leva a códigos menores; e por ser considerada relativamente fácil de aprender. Sendo este projeto um *framework* voltado para a comunidade científica, a facilidade de distribuição e a capacidade de operação em vários sistemas operacionais (Windows®, Macintosh® e Linux®) é desejável, o que também justifica a escolha do C♯ como a linguagem predominante neste projeto.

O *framework* desenvolvido é formado por duas bibliotecas (uma em C♯ e outra em C++), uma aplicação e sete extensões. A biblioteca em C++ oferece suporte à criação e manipulação das estruturas que armazenam os dados de entrada (dados de áudio e da extração) enquanto a biblioteca em C♯ é o núcleo deste projeto e implementa grande parte do seu funcionamento. A aplicação usa a biblioteca em C♯ para ler um arquivo XML e "executá-lo". As extensões oferecem um conjunto mínimo necessário para implementar o artigo descrito no capítulo anterior: leitor de arquivo (Wave), janela (Hamming), os 4 descritores (largura de banda, fluxo espectral, *loudness*, *spectral rolloff*) e o classificador, além da extensão para possibilitar a configuração.

Na revisão 212, este projeto contava com 9117 linhas de código, em C♯ e C++, onde 4258 linhas são efetivas e 3332 são comentários (basicamente documentação). Segundo o *site* `www.ohloh.net`, que faz análise de código de projetos com código aberto, este número nos coloca entre os melhores 10% de todos os projetos em C♯ listados neste *site*, que vasculha repositórios como *Google Code*, *SourceForge* e *FreshMeat* por novos projetos. Resumo mais atualizado pode ser encontrado em `https://www.ohloh.net/p/music-cs`.

O código-fonte deste projeto pode ser encontrado em `http://code.google.com/p/music-cs/`, e todos os arquivos estão sob licença MIT/X11, o que permite seu uso em qualquer aplicação, inclusive aplicações comerciais.

4.2 Objetivo

O objetivo deste processo é criar uma infraestrutura que facilite e torne mais ágeis futuros estudos de classificação de áudio. Esta preocupação se manifesta de várias formas:

- Facilidade de uso.
- Adoção de interface-padrão com o usuário por meio de arquivo XML.
- Possibilidade de uso de outras interfaces, possivelmente mais adequadas.
- Possibilidade de extensão dos formatos de arquivo reconhecido.
- Extensibilidade de partes vitais para a implementação de novos algoritmos.
- Suporte a operações básicas.
- Escolha de uma linguagem fácil de aprender[2].
- Preocupação com a portabilidade.

Este projeto foi desenvolvido sempre tendo em mente aplicações futuras desta biblioteca, por exemplo um reprodutor de áudio capaz de classificar ou sugerir músicas; ou um laboratório de classificação, a exemplo do MATLAB®.

Na seção seguinte veremos mais detalhes sobre a operação desta biblioteca.

4.3 Funcionamento

Esta seção pretende descrever as atividades que a biblioteca deve realizar para completar as tarefas a que se propõe. Começando da leitura do arquivo com a descrição dos algoritmos, passando pela execução, até chegar aos resultados, mostraremos o que está acontecendo longe dos olhos do usuário.

Ao ser inicializada, a biblioteca carregará todas as extensões encontradas no diretório indicado, por padrão o diretório do executável. Em seguida, o arquivo de configuração será lido. Este arquivo é uma das interfaces entre o usuário e a biblioteca, é através dele que devem ser informados os arquivos de áudio que serão usados para treinamento e classificação e os algoritmos; podem ser passadas algumas opções como, por exemplo, obrigar a realização da extração ou impedir que os dados dessa extração sejam salvos. A Seção 4.4 descreve o formato que deve ser usado para que a leitura seja bem sucedida.

[2]Embora a escolha natural para uma aplicação voltada para área acadêmica fosse o C++, julgamos que o C♯ traria mais benefícios para o desenvolvedor sem prejuízos para o usuário.

Os algoritmos, que são compostos por janela, característica e classificador, serão montados a partir das informações disponíveis naquele arquivo e adicionados a uma fila, o que possibilita a execução de vários algoritmos compartilhando o mesmo conjunto de treinamento e teste. Uma vez concluída a leitura do arquivo de configuração, cada algoritmo da lista será executado sequencialmente.

Quando um algoritmo recebe a instrução para executar, ele precisa verificar se estão disponíveis todos os elementos necessários: janela e descritores. Os classificadores são opcionais, uma vez que alguém pode estar interessado somente no cálculo das *features*. Após esta confirmação, os arquivos de treinamento serão encaminhados à extração.

Para iniciar a extração, o sistema deverá encontrar um manipulador (do inglês, *handler*) capaz de ler o arquivo. Os manipuladores são extensões capazes de ler e expor os dados dos arquivos de áudio para a biblioteca. Os usuários podem criar esses manipuladores para oferecer suporte a outros tipos de arquivo além do WAVE, para qual já existe um manipulador disponível. Uma vez encontrado um *handler* capaz de ler o arquivo, o próximo passo será submeter o arquivo ao extrator.

O extrator verificará se o arquivo que está sendo processado já teve alguma das características previamente extraída. Para considerar duas características iguais, tanto o nome da característica, formado por classes e parâmetros, quanto o da janela, formado por classe, tamanho, sobreposição e parâmetros, devem coincidir com os nomes armazenados. Caso tenha ocorrido uma extração prévia de alguma característica, o extrator recuperará os seus valores, tornando o processo mais ágil; em caso contrário, a extração será realizada e os dados obtidos para o novo par janela-característica serão salvos.

Se a biblioteca estiver detectado um classificador no algoritmo que está sendo executado, os dados obtidos na extração serão submetidos ao classificador; em caso contrário, o algoritmo atual é considerado finalizado.

O próximo passo é submeter as representações obtidas na extração ao classificador para que possa pré-processar os dados antes de iniciar o treinamento. Depois os dados são novamente submetidos ao classificador para que realize o treinamento. Por último, cada arquivo-alvo (que deve ser classificado) será extraído, pré-processado e submetido ao classificador para classificação.

Nesta seção buscamos descrever, com riqueza mas sem detalhes técnicos, qual o algoritmo que realiza as atividades a que a biblioteca se propõe. Os detalhes técnicos que "faltaram" nesta seção estão descritos na seção que segue.

4.3.1 Descrição Técnica

Esta seção pretende trazer especificidades técnicas do projeto que foram omitidas na seção anterior. Primeiramente é dada uma visão geral sobre a organização interna e o projeto da biblioteca; em seguida, faz-se um passo-a-passo da execução, semelhante ao mostrado na seção anterior, revelando os detalhes da implementação.

Abaixo estão relacionados os *namespaces* usados por esta biblioteca, contendo uma breve descrição.

- MusiC – *Core*
- MusiC.Apps – Aplicações
- MusiC.Data – Manipulação dos dados (vetores de áudio já janelados)
- MusiC.Extensions – Funcionalidades comuns a todas as extensões
- MusiC.Extensions.Classifiers – Classificadores
- MusiC.Extensions.Configs – Configuradores
- MusiC.Extensions.Features – Características
- MusiC.Extensions.Handlers – Manipuladores
- MusiC.Extensions.Windows – Janelas
- MusiC.Native – Facilidades implementadas em C++.

Também listamos algumas classes que serão citadas no passo-a-passo que segue com uma breve descrição para facilitar o entendimento.

- Algorithm – Representa um algoritmo. É uma "casca" em torno da Pipeline.
- Pipeline – Coordena a execução e montagem dos *pipelines*. Decide se a execução será gerenciada ou não-gerenciada invocando o *pipeline* correto.
- mPipeline e uPipeline – Respectivamente, representam o *pipeline* gerenciado (*managed*) e o não-gerenciado (*unmanaged*)
- Extension – Classe base para todas as demais extensões. Entretanto, as extensões não podem derivar diretamente dela e devem ser derivadas das classes correspondentes ao seu tipo, como Window, Classifier, Configurator e Handler.
- ExtensionCache – Localiza, carrega e armazena as extensões.

- ExtensionInfo – Cada objeto desta classe representa uma extensão. Também identifica o tipo (por exemplo, janela ou classificador) e instancia objetos da classe que ela representa.
- MusiC – Dá ao usuário acesso às funções da biblioteca.
- MusiCObject – Controla o acesso a um único arquivo de registros. Fornece facilidades para reportar erros.

Dentre todas as classes que compõem o projeto, duas têm papel importante na estrutura montada: a **MusiC** e a **MusiCObject**, ambas contidas no *namespace* **MusiC**. A primeira é a interface do usuário com todas as operações realizadas pela biblioteca, como inicialização, configuração e execução. A outra é a classe da qual as demais, inclusive a MusiC, derivam, herdando as capacidades de *logging* presentes nela. Entretanto, algumas classes não derivam de **MusiCObject** por alguma inconveniência técnica, por exemplo, o C♯ não suportar herança múltipla. Este é o caso das exceções, que precisam derivar de uma classe de exceção da biblioteca padrão (no caso, **System.ApplicationException**); e das estruturas (em C♯, classes e estruturas são entidades com diferenças mais significativas que em C++) que fazem a comunicação com o código nativo.

A inicialização da biblioteca consiste em descobrir e carregar as extensões. A criação dos algoritmos é tarefa da execução por dois motivos: a leitura do arquivo de configuração depende de uma extensão (um Configurator); a "montagem" de um algoritmo também depende das extensões (as janelas, as características e os classificadores estão disponíveis?); e para facilitar a re-execução de algoritmos contendo alterações.

Para ser considerado uma extensão, um arquivo deve ser uma biblioteca dinâmica, ter a extensão .dll, ser gerenciado; e os tipos, para serem aceitos, devem ser públicos e únicos e derivar de uma das classes básicas. A unicidade é uma questão complexa. Até a presente data, não fomos capazes de distinguir tipos "iguais" (mesmos membros, direitos de acesso, código-fonte) mas de diferentes bibliotecas.

Após passar pelas verificações descritas acima, as extensões serão registradas em estruturas similares a bancos de dados na memória. Usamos as LINQ (*Language Integrated Query*) para manipular esta estrutura. Isto significa que ao invés de usar construções tradicionais das linguagens de programação, como *for* e *while*, usamos uma sintaxe mais semelhante ao SQL (Structured Query Language), o que facilita a implementação.

O Trecho 4.1 mostra o código usado para verificar se a classe já foi adicionada à base de extensões (_infoList). Toda a parte de gerenciamento das extensões, como adição, busca de extensões e busca de manipuladores, foi implementada usando as *queries* integradas, o que exige **.Net Framework 3.5** ou equivalente.

```
// Procura por extensões(_info) que tenha o mesmo FullName que o tipo que está
    sendo adicionado(extensionType)
IEnumerable<ExtensionInfo> result =
from ExtensionInfo _info in _infoList where _info.Class.FullName == extensionType.
    FullName select _info;

if (result.Count() != 0)
{
    Message(extensionType.ToString() + " ... [REJECTED - DUPLICATED]");
    return;
}
```

Trecho 4.1: Exemplo de código usando LINQ TO XML.

Uma vez montado o *cache*, está finalizada a inicialização. A execução se inicia buscando no cache um configurador capaz de ler o arquivo de configuração que foi informado. A classe **Configurator** define o método abstrato **CanHandle** que é utilizado para esta verificação. O primeiro configurador presente no *cache* que for capaz de lidar com o arquivo será selecionado.

Em seguida, o configurador será usado para ler o arquivo e gerar um objeto de configuração (**Config**). Este objeto armazena todas a informações sobre a execução[3], por exemplo, quais os arquivos de treinamento e de classificação. É durante a construção do objeto de configuração que os algoritmos serão criados, as extensões serão instanciadas e os parâmetros para ela passados.

O configurador criará os algoritmos adicionando extensões pelos nomes completos da classe, o que inclui o *namespace*. Junto com os nomes também devem ser passados objetos **ParamList**. Esses objetos representam a lista de parâmetros que devem ser usados para instanciar o objeto da classe que foi informada. Para cada deve ser informada a classe e, opcionalmente, o valor. Recomenda-se que seja acrescentado um nome para auto-documentação da configuração, mas sem nenhum efeito interno.

O algoritmo, usando o nome da classe que foi passado, buscará no *cache* o objeto que contém as informações sobre essa extensão. Usando a lista de parâmetros passada para o algoritmo, este objeto pode instanciar um objeto da classe da extensão. Para isso, os próprios parâmetros devem ser instanciados, o que é feito usando os métodos estáticos **<CLASS> Parse(string)** presentes nas classes numéricas. Qualquer classe que implemente o método **Parse** e um construtor padrão (sem parâmetros) pode ser passada como parâmetro. Dois *arrays* contendo os tipos dos parâmetros e os valores já instanciados deles serão criados e serão passados ao construtor da extensão usando a reflexão. Por limitação da API disponível, os parâmetros devem ser passados NA ORDEM CORRETA. Caso contrário, a construção do objeto será equivocada ou o construtor não será encontrado.

[3]A seção seguinte trará mais detalhes sobre quais informações podem ser passadas a biblioteca

Uma vez a extensão instanciada, o algoritmo irá adicioná-la ao *pipeline*. Este sistema de *pipelines* (ou linhas de execução) foi criado para permitir que duas interfaces, uma gerenciada e outra não (o que significa protótipos diferentes), possam conviver. A implementação não-gerenciada (e não compatível com a CLS) está completa e foi usada nos testes que serão mostrados no próximo capítulo. Já a implementação gerenciada só tem os protótipos e ainda não foi implementada, o que está nos nossos planos futuros. Esta divisão foi feita por acreditarmos que a implementação "*unsafe*" teria melhor desempenho e facilitaria a comunicação com o código nativo, mas gostaríamos de manter a possibilidade de comparar os desempenhos em momento mais oportuno.

O próximo passo é executar todos os algoritmos presentes na lista de execução. Estes transmitirão a ordem de execução aos *pipelines*, que determinam se alguma das linhas de execução disponíveis pode ser executada e, em caso de existência, ela será executada[4].

Considerando que a linha de execução não-gerenciada, representada pela classe **uPipeline**, foi acionada, então a lista de CLASSES (no sentido de classe de dados, e doravante referida como rótulo) será obtida do objeto de configuração. Caso haja um classificador, este será perguntado sobre a necessidade de retreinar através do método virtual **bool Classifier.NeedTraining(Config)**.

Se novo treinamento for necessário ou não houver classificador, cada um dos rótulos terá todos os seus arquivos extraídos. Para isso os arquivos são listados, procura-se no *cache* um manipulador adequado, associa-se o arquivo ao manipulador selecionado, associa-se o manipulador à janela e submetem-se a janela e a lista de características ao extrator.

O extrator é apenas um coordenador do processo de extração. Ele verifica se o par janela-característica já foi extraído anteriormente e leva em consideração as opções pertinentes, mas quem faz os cálculos relacionados à extração são as janelas, os *frames* e as características. As janelas usam o método virtual **Calculate** para fazer a combinação entre os dados da janela e o áudio. A implementação padrão MULTIPLICA a janela pelo trecho de áudio, mas basta sobrecarregar o método (ao implementar uma nova janela) para que o usuário redefina este comportamento da janela. Os *frames* armazenam os dados calculados pelas janelas e as transformadas para evitar o cálculo desnecessário das transformadas. Na presente data, a classe **Frame** só é capaz de calcular a DFT (FFT) de potências de 2 e não conta, por enquanto, com capacidades de extensão. As características usam os *frames* para, através do método abstrato **float Extract(Frame)**, finalmente realizar as extrações. Todos os dados e contas são realizados em **float**, com raras exceções. Não

[4]A implementação atual considera um erro a possibilidade de executar dois *pipelines*.

Tamanho	Nome do Campo	Descrição
4 bytes	sectionLabelSz	Tamanho do nome da janela.
4 bytes	dataLabelSz	Tamanho do nome do descritor.
4 bytes	dataSectionSz	Tamanho, em bytes, dos dados.
sectionLabelSz	sectionLabel	Nome da Janela
dataLabelSz	dataLabel	Nome do descritor
dataSectionSz	dataSection	Dados (float).

Tabela 4.1: Descrição do formato usado para armazenar as características pré-extraídas.

existem planos para converter a biblioteca para **double** ou ainda tornar o modo de funcionamento opcional.

Durante a extração será usado um manipulador da base de características para verificar se alguma característica já foi extraída para o arquivo em questão e reutilizá-la. Uma base de características é um arquivo <nome-completo-do-original>.db, criado no mesmo diretório do arquivo original com o formato apresentado na Tabela 4.1. Este manipulador está implementado na biblioteca em C♯ como um envoltório sobre uma implementação em C++ e não pode ser estendido. Não existem planos para torná-lo extensível; entretanto, o uso de outros formatos, possívelmente padronizados, não está descartado.

Outro ponto importante é quanto às estruturas de dados usadas para armazenar os resultados da extração. Elas foram implementadas em C♯ (através de **struct**s) e espelhadas no ambiente C++. A forma como foram implementadas nos dois ambientes permite a troca de dados entre o C♯ e o C++, possibilitando a implementação do classificador nesta linguagem.

O *namespace* **MusiC.Native** guarda algumas facilidades para manipular essas estruturas, bem como as define. Fazem parte deste arranjo: DataCollection, ClassData, FileData e FrameData, além da DataHandler, que apenas facilita o manuseio das demais. Elas representam, respectivamente, a coleção como um todo, cada classe, cada arquivo e cada segmento de áudio. A Figura 4.1 mostra como elas se relacionam.

É importante notar que todos os segmentos estão encadeados e que, particularmente, o último segmento de uma classe está ligado ao primeiro segmento da próxima classe. A estrutura foi montada para facilitar a varredura de toda a base de dados, bem como a de todos os segmentos de determinada classe ou arquivo. Acreditamos que esta estrutura é suficientemente flexível para permitir que o usuário implemente classificadores eficientemente, mesmo que complexos.

Embora a próxima etapa, a classificação, seja o objetivo de todo este projeto, não há muitos detalhes a serem esclarecidos. Os dados criados no extrator são submetidos ao classificador para uma filtragem inicial (TrainingFilter) e por fim

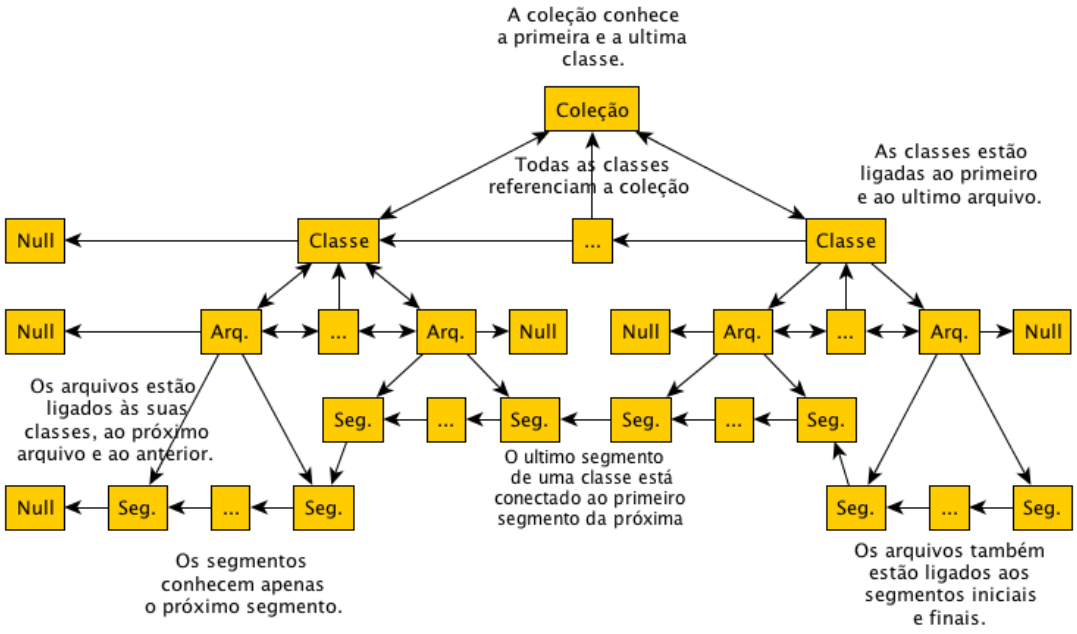

Figura 4.1: Esquema ilustrando a estrutura de dados que recebe os dados extraídos dos arquivos de áudio.

submetidos ao classificador para treinamento. Após o fim do treinamento, será dada a ordem de armazenamento ao classificador, deixando a cargo do desenvolvedor a implementação da função de gravação. Por enquanto, não existe uma forma mais amigável de salvar esses dados, mas achamos que seria interessante que esses dados fossem guardados pela biblioteca, o que implicaria algumas restrições quanto às estruturas de dados que poderiam ser usadas na implementação do classificador.

Enfim, cada arquivo relacionado para classificação será submetido ao processo de extração (idêntico ao processo de extração para o treinamento), a uma filtragem inicial (ClassificationFilter) e por fim ao classificador. O resultado esperado é que o classificador retorne um número inteiro não-negativo para informar qual a classe do arquivo, com base na ordem em que as classes foram passadas ao classificador.

4.4 Configuração

Esta seção explica quais as opções de configuração que a biblioteca possui. Por isso, todas as informações contidas nesta seção dizem respeito apenas ao configurador padrão.

Como foi dito anteriormente, este configurador lê um arquivo XML, do qual consegue extrair as configurações que serão passadas à biblioteca. Ao buscar essas informações, o configurador ignorará silenciosamente todas as marcações que não

foram reconhecidas, realizando as tarefas previstas para as demais. Todas as marcações criadas para este projeto estão prefixadas por "MusiC-", o que possibilita que o usuário tenha outras informações no mesmo arquivo sem interferir no funcionamento da biblioteca. Entretanto, recomenda-se que as marcações que iniciem por "MusiC-" sejam consideradas como reservadas para o futuro.

A próxima seção descreve as ações realizadas por cada marcação e as opções disponíveis.

4.4.1 Marcações

Nesta seção listamos todas as marcações reconhecidas pelo configurador padrão disponibilizado pelo projeto. Para cada marcação existe uma tabela que resume as características mais importantes, além de exemplos explicando qual o seu uso correto.

4.4.1.1 MusiC-Algorithm

A marcação MusiC-Algorithm define um algoritmo que deve ser executado pela biblioteca. Nela serão declarados os demais elementos que fazem parte do algoritmo.

Esses elementos podem ser declarados através da marcação Extension ou usando apelidos (*alias*). No decorrer desta seção aprensentaremos as marcações Extension e MusiC-Alias, responsável pela criação de apelidos. A Tabela 4.2 mostra um resumo desta marcação e a Tabela 4.3 traz detalhes sobre a marcação Extension, que só pode ser usada no escopo da marcação MusiC-Algorithm.

MusiC-Algorithm	
Atividades:	Declara um novo algoritmo.
Marcações Suportadas:	Extension e nomes criados usando a marcação MusiC-Alias.
Atributos Requeridos:	–
Atributos Opcionais:	name.

Tabela 4.2: Informações sobre a marcação MusiC-Algorithm.

O atributo *name* estabelece o nome que as mensagens geradas pelo sistema usará para se referir aos algoritmos. Caso um nome não seja estabelecido, será criado um nome padrão (Algorithm_N, onde N é relativo a posição que o algoritmo aparece no arquivo de configuração. Para o primeiro N é 1, para o segundo N é 2 ...).

Já a marcação Extension define quais as extensões que serão usadas no algoritmo. Só serão reconhecidas as extensões que foram carregadas quando a biblioteca

foi inicializada. A Tabela 4.4 traz detalhes sobre a única marcação que pode ser definida em seu escopo, a marcação Param.

Extension	
Atividades:	Informa que uma extensão deve ser carregada.
Marcações Suportadas:	Param
Atributos Requeridos:	class
Atributos Opcionais:	–
Observação:	Esta marcação só está disponível no interior da marcação MusiC-Algorithm.

Tabela 4.3: Informações sobre a marcação Extension.

O atributo *class* indica qual a classe, obrigatoriamente incluindo o *namespace*, da extensão que deve ser adicionada no algoritmo. Só classes conhecidas pelo *cache* serão adicionadas com sucesso.

A marcação Param faz a declaração de um parâmetro que será passado ao construtor da extensão. Esses parâmetros também são usados para construir a identificação das janelas e dos descritores quando as características extraídas são gravadas ou recuperadas. Dois atributos são suportados por essa extensão: *class* e *value*. O atributo *class* informa qual a classe do parâmetro e o atributo *value* diz o valor do parâmetro. Caso este último não seja especificado o valor padrão da classe do atributo será usado.

Param	
Atividades:	Declara um parâmetro de uma extensão.
Marcações Suportadas:	–
Atributos Requeridos:	class
Atributos Opcionais:	value e name[5]
Observação:	Esta marcação só está disponível no interior da marcação Extension ou de marcações criadas com a *tag* MusiC-Alias.

Tabela 4.4: Informações sobre a marcação Param.

O Trecho 4.2 mostra um exemplo de como utilizar as marcações MusiC-Algorithm, Extension e Param.

```
1 <MusiC-Alias name=''Hamming'' class=''MusiC.Extensions.Windows.Hamming'' />

3 <MusiC-Algorithm name=''Quasi-Barbedo''>
    <!-- Extension + Param -->
5   <Extension class=''MusiC.Extensions.Windows.Hamming'' >
      <!-- A ordem que os parâmetros são passados é a mesma que será passado para o
          construtor -->
7     <Param name=''so-para-documentacao'' class=''System.Int32'' value=''1024'' />
    </Ex  tension>
9
    <!-- Param trabalhando com um \textit{alias} -->
```

```
11  <Hamming>
      <!-- A ordem que os parâmetros são passados é a mesma que será passado para o
           construtor -->
13    <Param name=''size'' class=''System.Int32'' value=''1024'' />
    </Hamming>
15
    <!-- Extension sem parâmetros -->
17  <Extension class=''MusiC.Extensions.Features.SpectralRollOff'' />
  </MusiC-Algorithm>
```

Trecho 4.2: Exemplo de como usar a marcação MusiC-Algorithm.

4.4.1.2 MusiC-Alias

Esta marcação define um apelido para uma extensão. Através dela o usuário não precisa redefinir as extensões da forma completa (através da *tag* Extension) em todos os algoritmos. A Tabela 4.5 contém detalhes sobre esta marcação.

Para usar um apelido basta criar uma marcação com o apelido (ao invés de Extension), conforme mostra o Trecho 4.3. O objetivo desta marcação é evitar a redefinição de extensões em vários algoritmos. A biblioteca não suporta o uso de parâmetros na declaração de um apelido, entretanto é uma das características previstas em versões futuras.

Alias	
Atividades:	Cria um nome para uma extensão.
Marcações Suportadas:	–
Atributos Requeridos:	name e class.
Atributos Opcionais:	–

Tabela 4.5: Informações sobre a marcação MusiC-Alias.

O atributo *name* especifica o apelido que será dado à extensão especificada, através do nome completo de sua classe, no atributo *class*.

```
  <!-- Definição -->
2 <MusiC-Alias name=''Hamming'' class=''MusiC.Extensions.Windows.Hamming'' />

4 <!-- Em um algoritmo -->
  <Hamming>
6   <!-- PARÂMETROS: Definir os parâmetros que devem ser usados. -->
    <Param name=''so-para-documentação'' class=''System.Int32'' value=''1024'' />
8 </Hamming>

10 <!-- Extension Equivalente (também em um algoritmo) -->
  <Extension class=''MusiC.Extensions.Windows.Hamming'' >
12  <Param name=''so-para-documentação'' class=''System.Int32'' value=''1024'' />
  </Extension>
```

Trecho 4.3: Exemplo de como usar a marcação MusiC-Alias.

4.4.1.3 MusiC-Classify

A marcação MusiC-Classify declara quais os arquivos que serão classificados. Usando as marcações suportadas internamente dela podem ser adicionados tanto arquivos quanto diretórios. A Tabela 4.6 mostra os detalhes para esta marcação e as Tabelas 4.7 e 4.8 mostram os detalhes das marcações que aparecem no interior desta.

MusiC-Classify	
Atividades:	Informa os arquivos que devem ser classificados.
Marcações Suportadas:	TrainDir e TrainFile.
Atributos Requeridos:	–
Atributos Opcionais:	–

Tabela 4.6: Informações sobre a marcação MusiC-Classify.

A marcação ClassDir, que só é válida no interior da MusiC-Classify, permite que todos os arquivos contidos em um diretório, especificados pelo atributo *path*, sejam marcados para classificação. Se o atributo opcional *recursive* for verdadeiro, os sub-diretórios também serão lidos e seus arquivos serão marcados para classificação; em caso contrário, se o atributo for falso ou não especificado, apenas os arquivos que estão no diretório padrão serão marcados.

Esta marcação também permite que um filtro seja especificado através do atributo opcional *filter*. Este filtro é uma expressão regular que será usada para conferir se um arquivo deve ser acrescentado à fila de classificação.

ClassDir	
Atividades:	Informa que todos os arquivos de um diretório e, opcionalmente, seus sub-diretórios devem ser classificados.
Marcações Suportadas:	–
Atributos Requeridos:	path
Atributos Opcionais:	filter
Observação:	Esta marcação só está disponível no interior da marcação MusiC-Classify.

Tabela 4.7: Informações sobre a marcação ClassDir.

Já a marcação ClassFile apenas acresecenta um arquivo, especificado pelo atributo *path*, à lista de arquivos a serem classificados.

O Trecho 4.4 mostra como as marcações MusiC-Classify, ClassDir e ClassFile devem usadas.

```
1 <MusiC-Classify>
    <Dir path=''..\data'' recursive=''true'' filter=''(.wav$)''/>
3   <File path=''.\exemplo\teste1.wav'' ''/>
  </MusiC-Classify>
```

Trecho 4.4: Exemplo de uso da marcação MusiC-Classify.

ClassFile	
Atividades:	Informa que um arquivo deve ser incluído para classificação.
Marcações Suportadas:	–
Atributos Requeridos:	path
Atributos Opcionais:	–
Observação:	Esta marcação só está disponível no interior da marcação MusiC-Classify.

Tabela 4.8: Informações sobre a marcação ClassFile.

MusiC-Option	
Atividades:	Define os valores de opções.
Marcações Suportadas:	–
Atributos Requeridos:	name e value
Atributos Opcionais:	–

Tabela 4.9: Informações sobre a marcação MusiC-Option.

4.4.1.4 MusiC-Option

A marcação MusiC-Option define valores que alteram o comportamento do algoritmo. Essas marcações devem aparecer múltiplas vezes para alterar o valor de mais de uma opção, como mostrado no Trecho 4.5, enquanto que a Tabela 4.9 resume a sintaxe desta marcação.

O atributo *name* especifica qual opção será modificada enquanto que o *value* indica qual será o novo valor da opção.

Listamos as opções, suas funções e os respectivos valores padrão na Tabela 4.10. Os nomes das opções não são sensíveis a variações de maiúsculas e minúsculas.

Opções Disponíveis		
ALLOW_SAVE:	Altera a gravação de dados pré-extraídos	TRUE
FORCE_EXTRACT:	Força a extração mesmo quando dados pré-extraídos estão disponíveis	FALSE

Tabela 4.10: Informações sobre as opções disponíveis e seus valores padrão.

```
<!-- Os nomes das opções não são sensíveis a maiusculas e minusculas -->
<MusiC-Option name=''allow_save'' value=''false''/>
<MusiC-Option name=''force_extract'' value=''false''/>
```

Trecho 4.5: Exemplo de definição de opções.

4.4.1.5 MusiC-Train

A marcação MusiC-Train cria rótulos nos quais os arquivos especificados pela marcação MusiC-Classify serão classificados. Esta marcação também declara quais

os arquivos devem ser incluídos no treinamento dos rótulos criados. A Tabela 4.11 resume a sintaxe esperada para esta marcação.

MusiC-Train	
Atividades:	Informa os arquivos que serão usados para o treinamento.
Marcações Suportadas:	Label
Atributos Requeridos:	–
Atributos Opcionais:	dir.

Tabela 4.11: Informações sobre a marcação MusiC-Train.

O atributo opcional *dir* funciona como um diretório base para os arquivos de treinamento. Ele pode ser usado para adicionar automaticamente um diretório à lista de classificação, desde que este tenha o mesmo nome que o *label* e esteja neste diretório.

A única marcação disponível é a *Label* (Tabela 4.12). Este nó define um rótulo (uma classe de dados) e os arquivos que serão usados no treinamento. As Tabelas 4.13 e 4.14 resumem a sintaxe das marcação *Label*, que só pode ser usadas no escopo da marcação MusiC-Train.

Label	
Atividades:	Informa os arquivos que serão usados para o treinamento.
Marcações Suportadas:	TrainDir e TrainFile
Atributos Requeridos:	name
Atributos Opcionais:	–
Observação:	Esta marcação só está disponível no interior da marcação MusiC-Train.

Tabela 4.12: Informações sobre a marcação Label.

Ele se comporta de maneira muito similar à MusiC-Classify, suportando as mesmas marcações; entretanto, requer o atributo *name*, que especifica o nome do rótulo.

A marcação TrainDir, que só é válida no interior da *Label*, permite que todos os arquivos contidos em um diretório, especificados pelo atributo *path*, sejam marcados para classificação.

Uma outra diferença é que o atributo *path* da marcação TrainDir não é obrigatório. Caso ele não esteja especificado, será considerado o diretório com o nome da rótulo dentro do diretório de treinamento; e se este último também não estiver definido, então será considerado o diretório do executável.

Se o atributo opcional *recursive* for verdadeiro, os sub-diretórios também serão lidos e seus arquivos serão incluídos no treinamento do rótulo definido pela

TrainDir	
Atividades:	Informa que todos os arquivos de um diretório e, opcionalmente, seus sub-diretórios devem ser classificados.
Marcações Suportadas:	–
Atributos Requeridos:	–
Atributos Opcionais:	path, recursive e filter
Observação:	Esta marcação só está disponível no interior da marcação Label.

Tabela 4.13: Informações sobre a marcação TrainDir.

Label; em caso contrário, se o atributo for falso ou não especificado, apenas os arquivos que estão no diretório padrão serão incluídos.

Esta marcação também permite que um filtro seja especificado através do atributo opcional *filter*. Este filtro é uma expressão regular que será usada para conferir se um arquivo deve ser acrescentado à fila de treinamento do rótulo em questão.

Já a marcação TrainFile apenas acrescenta um arquivo, especificado pelo atributo *path*, à lista de arquivos que serão usados no treinamento.

TrainFile	
Atividades:	Informa que um arquivo deve ser incluído para classificação.
Marcações Suportadas:	–
Atributos Requeridos:	path
Atributos Opcionais:	–
Observação:	Esta marcação só está disponível no interior da marcação Label.

Tabela 4.14: Informações sobre a marcação TrainFile.

O Trecho 4.6 mostra como as marcações MusiC-Train, Label, TrainDir e TrainFile devem usadas.

```
<MusiC-Train dir=''..\data''>
  <Label name=''classical''>
    <Dir path=''..\data2'' recursive=''true'' filter=''(.wav$)''/>
    <File path=''..\meu_arquivo.wav''/>
  </Label>
  <Label name=''electronic''>
    <!-- Adiciona o diretório -- \data\electronic -- Se o atributo dir (MusiC-Train
         ) não for declarado então é usado o diretório do executável que chamou a
         biblioteca. -->
    <Dir />
  </Label>
</MusiC-Train>
```

Trecho 4.6: Exemplo de como usar as marcações MusiC-Train.

Vale notar que as *tags* sempre começam com letra maiúscula e os atributos

sempre começam com letra minúscula. Todos esses nomes são sensíveis a essas variações. Além disso, *tags* e atributos que não estão descritos aqui podem ser usados à vontade pelos usuários, pois serão ignorados, bem como comentários.

Capítulo 5

Testes

Este capítulo traz os resultados obtidos usando o *framework* desenvolvido para implementar o classificador de Barbedo e Lopes [13], do qual tratamos no Capítulo 3. Começamos descrevendo os parâmetros usados nos testes e em seguida descrevemos a base de dados que usamos. A última seção analisa os resultados encontrados.

5.1 Descrição do Teste

Como discutido anteriormente, o trabalho de Barbedo e Lopes [13] deixou algumas lacunas que dificultariam a reprodução dos experimento descrito por eles. Na tentativa de reproduzi-lo, fomos obrigados a tomar algumas decisões que podem influenciar no desempenho do algoritmo. Esta seção pretende documentar estas decisões.

Quando implementamos o MBL, tivemos que definir o critério de seleção dos vetores mais relevantes. No artigo, Barbedo e Lopes dizem que selecionam os vetores usando fatores como a média dos vetores de treinamento e a faixa esperada para cada característica. Assim, arbitramos que selecionaríamos os vetores que estivessem na vizinhança da média e que a distância máxima à média seria dada por uma fração do desvio-padrão.

Esta fração deve ser determinada pelo valor que seleciona, aproximadamente, o número de candidatos em que estamos interessados. Nesta implementação, o número de vetores candidatos que serão selecionados não é fixo, o que significa que se a fração for muito pequena, poucos vetores representarão o gênero; em caso contrário, como os vetores são selecionados na ordem em que aparecem, vetores possivelmente melhores (mais perto da média) podem ser desconsiderados (se a quota-alvo de vetores já tiver sido atingida).

Para os testes que foram realizados a seguir, trabalhamos com 20 vetores representando o gênero, e a distância máxima permitida da média é 70% do desvio-

padrão. Esses valores nos dão, em média, cerca de 18 vetores representando cada gênero para a base de dados que usamos.

Escolhemos uma representação com 20 vetores para reduzir o tempo de processamento. Entretanto, permitir um número maior de vetores significa também aumentar a distância máxima a que eles estão sendo coletados (ou somente os mesmos 18 serão selecionados), o que pode piorar o desempenho do algoritmo.

Esses valores podem, e provavelmente devem, ser alterados dependendo da base de dados, principalmente do conjunto de treinamento. Por isso, a seção seguinte tenta definir a base de dados que usamos nos testes.

5.2 Bases de Dados

Esta seção pretende descrever a base de dados utilizada no teste. Por ser uma base que foi criada por nós para este trabalho, optamos por descrever mais os processos percorridos que propriamente a base de dados.

A base que criamos para os testes descritos neste capítulo é formada por quatro gêneros: Barroco, Choro, MPB e Piano. Formada a partir de CDs, as faixas foram selecionadas subjetivamente de forma que definissem bem cada um gênero (quanto à abrangência), mas que permitissem que os gêneros fossem suficientemente diferentes entre si, para evitar uma seleção pobre ou tendenciosa. Cada classe conta com, no mínimo, 30 arquivos em formato WAVE codificados com 16 bits a 44.1 kHz.

Não houve seleção especial para o conjunto de treinamento (50 % da base) ou teste; entretanto, onde foi possível, tentamos deixar apenas uma faixa de cada de autor/intérprete no conjunto de treinamento. Houve um esforço no sentido de que os autores que apareciam no conjunto de treinamento também aparecessem no conjunto de teste, mas isso não foi possível para todos os gêneros. A Tabela 5.1 mostra a composição quanto ao número de arquivos dos conjuntos de treinamento e teste.

	MPB	Piano	Barroco	Choro
Treinamento	16	15	15	15
Teste	16	15	14	15

Tabela 5.1: Número de arquivos de cada gênero que compõem os conjuntos de treinamento e teste.

Criar uma base tem suas vantagens, como maior controle sobre o que está acontencedo, mas também tem desvantagens (por exemplo, em alguns casos fica mais difícil determinar se um problema é causado pelo algoritmo ou pela base). Na sequência, discutimos os fatores que nos levaram a criar a uma base e não utilizar uma pré-existente, por exemplo, a *Magnatune*.

5.2.0.6 Justificativa

Iniciamos o desenvolvimento do *framework* (e das extensões que implementam o MBL) realizando testes com a *Magnatune Database* (MD). Entretanto, esta base possui características muito diferentes da base que parece ter sido usada para desenvolver o MBL, como mostramos na Seção 3.2. Relembrando, as bases usam codificações e comprimentos diferentes, além de políticas diferentes quanto à repetição de autores ou intérpretes. Além disso, o fato de toda a MD, inclusive os arquivos de treinamento, ter sido submetida a um codificador perceptivo parece interferir com os resultados do algoritmo, que usa informações descartadas pelo codificador através da medição da audibilidade (*loudness*). Isto parece ser ainda mais relevante quando os autores evitaram usar arquivos que usam codificação perceptiva no conjunto de treinamento.

Outro fator determinante para a escolha de uma nova base de dados (culminando com sua criação) foi a composição do conjunto de treinamento da MD. Como também foi mostrado na Seção 3.2 através das Tabelas 3.1 e 3.2, este conjunto é superficial em alguns gêneros, e existem interseções indesejadas entre os rótulos que estão sendo treinados.

Embora não tenha sido um fator considerado, mas que foi certamente notado, a *Magnatune* é colossal. São mais de 4 GB de arquivos MP3 (compactados) que formam uma base de quase 14 GB de arquivos WAVE para ser processada. A duração do treinamento[1] fica reduzida de cerca de 10 horas de processamento para algo em torno de 10 minutos (ambos os tempos em um Intel® Core 2 Quad Q8200 4 GB RAM DDR2 rodando Windows® XP SP3)[2] com o uso da base nova.

5.3 Resultados

Esta seção mostra os resultados obtidos nos testes que realizamos. O primeiro teste usou apenas duas classes (MPB e Piano) para tornar mais fácil a análise do que estava acontecendo. Em seguida, acrescentamos o Barroco e para último o teste acrescentamos apenas o Choro ao conjunto inicial.

O Trecho 5.1 mostra o arquivo de configuração usado no testes. Para cada teste alteramos apenas os diretórios onde os arquivos estavam listados.

```
<?xml version="1.0"?>
<MusiC version="1.0">
```

[1] Os tempos de extração e classificação não são significativos se os dados estiverem pré-extraídos; em caso contrário, são certamente muito menor que o tempo de treinamento.

[2] Não foi feito um *benchmark* específico e nem os tempos citados são exatos. Nosso objetivo foi apenas mostrar a redução significativa ocorrida no tempo de processamento, bem como o sucesso do uso de dados pré-extraídos.

```
<MusiC-Alias name="Hamming" class="MusiC.Extensions.Windows.HammingU"/>
<MusiC-Alias name="RollOff" class="MusiC.Extensions.Features.SpecRollOffU"/>
<MusiC-Alias name="BW" class="MusiC.Extensions.Features.Bandwidth"/>
<MusiC-Alias name="Barbedo" class="MusiC.Extensions.Classifiers.Barbedo"/>

<!--<MusiC-Option name="force_extract" value="true" />-->
<!--<MusiC-Option name="force_training" value="true" />-->

<MusiC-Train dir="d:\db">
  <Label name="MPB">
    <TrainDir path="MPB_cantada" recursive="true" filter="(.wav$)"/>
  </Label>

  <Label name="Piano" >
    <TrainDir path="Piano_Romantismo" recursive="true" filter="(.wav$)"/>
  </Label>
</MusiC-Train>

<MusiC-Classify>
  <ClassDir path="d:\db\classify"/>
</MusiC-Classify>

<MusiC-Algorithm>

  <Hamming>
    <Param name="Size" value="1024" class="System.Int32"/>
    <Param name="Overlap" value="512" class="System.Int32"/>
  </Hamming>

  <RollOff/>
  <BW/>
  <Extension class="MusiC.Extensions.Features.Loudness" />
  <Extension class="MusiC.Extensions.Features.SpectralFlux" />

  <Barbedo />

</MusiC-Algorithm>
</MusiC>
```

Trecho 5.1: Arquivo de configuração usado nos testes.

5.3.1 MPB-Piano

O primeiro teste realizado só incluia as faixas de MPB e Piano. Escolhemos estas classes por que acreditamos que elas estariam bem separadas; e por conveniência, já que seria mais fácil montá-las. A Tabela 5.2 traz a matriz de confusão para este teste.

	MPB	Piano	Total
MPB	0.81 (13)	0.23 (3)	16
Piano	0.06 (1)	0.93 (14)	15

Tabela 5.2: Matriz de confusão para o teste MPB-Piano.

A princípio não existem comentários relevantes para serem feitos sobre este teste. Entretanto, esses dados serão usados na análise dos demais, onde algumas situações curiosas ocorreram.

5.3.2 MPB-Piano-Barroco

Para realizar este teste deixamos apenas as faixas subjetivamente classificadas como MPB, Piano ou Barroco em ambos os conjuntos. A Tabela 5.3 mostra a matriz de confusão para este teste.

	MPB	Piano	Barroco	Total de Arquivos
MPB	0.81 (13)	0 (0)	0.23 (3)	16
Piano	0.06 (1)	0.73 (11)	0.2 (3)	15
Barroco	0 (0)	0.29 (4)	0.71 (10)	14

Tabela 5.3: Matriz de confusão para o teste MPB-Piano-Barroco.

Abaixo citamos alguns fatos que a tabela não revela:

- O sistema acerta as mesmas 13 faixas de MPB que o teste anterior.
- As 3 faixas de MPB que foram classificadas como Piano no teste anterior agora são classificadas como Barroco.
- A faixa de piano classificada como MPB no primeiro teste manteve a mesma classificação.

A repetição da classificação do mesmo conjunto das 13 faixas como MPB indica robustez do algoritmo, bem como uma boa representação do gênero na base.

Foi curiosa a "migração" das 3 faixas de MPB que haviam sido classificadas como Piano para o Barroco. De certa forma reflete que essas faixas não estão bem representadas nem pelos vetores de MPB nem pelos de Piano e, provavelmente, nem pelos de Barroco. Esta mudança também sugere uma certa superposição das classes Barroco e Piano, o que pode afetar a discriminação entre as duas classes; de fato isso ocorre levemente, como pode ser visto na Tabela 5.3.

A única faixa de Piano que havia sido classificada como MPB manteve a sua classificação. Isto sugere que esta faixa está mais deslocada das demais, representando um ponto fora do contexto.

A taxa de acerto final fica em 75,55%, o que se aproxima da taxa de média de acerto da classificação por pessoas reportada por Lippens, Martens, Mulder e Tzanetakis em [10] de 76%.

5.3.3 MPB-Piano-Choro

Para este teste acrescentamos o Choro ao teste inicial. Os resultados podem ser vistos na Tabela 5.4.

	MPB	Piano	Choro	Total de Arquivos
MPB	0.69 (11)	0.25 (4)	0.06 (1)	16
Piano	0.07 (1)	0.93 (14)	0 (0)	15
Choro	0.6 (9)	0.2 (3)	0.2 (3)	15

Tabela 5.4: Matriz de confusão para o teste MPB-Piano-Choro

Algumas observações que não aparecem na tabela:

- Das 13 faixas que haviam sido classificadas corretamente como MPB nos testes anteriores, 11 mantiveram a classificação.
- As 3 faixas de MPB que foram classificadas como Barroco no teste anterior (e haviam sido classificadas como Piano anteriormente) agora são novamente classificadas como Piano.
- A faixa de piano classificada como MPB nos testes anteriores manteve a mesma classificação.
- Uma faixa que fora corretamente classificada como MPB nos dois testes anteriores foi classificada como Piano.
- 9 faixas de Choro (69%) são classificadas como MPB, mas apenas uma faixa deste gênero é classificada como Choro.

A faixa de Piano que fora classificada como MPB nos testes anteriores teve sua classificação confirmada mais uma vez. Isto reforça a suspeita de que esta faixa seja um ponto destoante dos demais do conjunto.

Curiosamente, uma das faixas que havia sido classificada corretamente nos dois testes anteriores teve sua classificação alterada para Piano. Isto ocorreu porque a pontuação que se acumulava em MPB agora se dividiu entre MPB e Choro, tornando o Piano o campeão.

O percentual tão elevado de faixas de Choro classificadas como MPB sugere que as faixas escolhidas para o conjunto de treinamento de Choro não formam uma representação satisfatória do gênero (no espaço dos descritores usados), que está melhor representado pelas faixas de MPB do que por seu próprio gênero.

Com este teste encerramos o ciclo de desenvolvimento do *framework* apresentado. Em conjunto com os demais testes, pudemos ver que existe uma base sólida, mas há espaço para melhorias.

Capítulo 6

Conclusão

Após um longo ciclo de desenvolvimento, conseguimos desenvolver um *framework* que oferecesse facilidades a pesquisadores de classificação de áudio oferecendo um conjunto reduzido de limitações. Esperamos ter demonstrado de forma satisfatória as capacidades deste *framework* ao longo deste trabalho.

Realizamos um estudo de caso do método que Barbedo e Lopes apresentam em [13]. As tarefas foram concluídas com sucesso, e as técnicas que usamos para aumentar a velocidade do processo foram extremamente eficientes. Quanto ao desempenho, não houve nenhuma situação em que o algoritmo fosse muito rápido (exceto quando apenas classificando as faixas) ou muito lento, o que nos indica que estamos fazendo as escolhas corretas.

Apesar de ser uma base montada apenas para este trabalho, portanto ainda com as suas particularidades desconhecidas, os testes indicam que ela, em geral, se comportou bem. Talvez uma seleção um pouco diferente do conjunto de treinamento de Choro (ou o uso de um conjunto diferente de descritores) melhorasse o desempenho nesta classe.

De qualquer forma, embora bem sucedido nas suas ambições, ainda existe muito trabalho que pode ser feito para melhorar este projeto.

6.1 Trabalhos Futuros

Embora este trabalho cumpra a proposta feita inicialmente, diversas melhorias podem ser realizadas. Destacamos duas áreas que poderiam receber esforços adicionais: o desenvolvimento de outras extensões e o uso da biblioteca em aplicações mais abrangentes.

Atualmente, o conjunto de extensões oferecidas é muito restrito e está limitado apenas ao escopo do trabalho de Barbedo e Lopes. É desejável criar outras extensões como, por exemplo, as janelas de Hanning e Kaiser; permitir o uso de

coeficientes cepstrais como descritores; e incluir classificadores mais genéricos, como máquinas de vetor suporte ou modelos de misturas de gaussianas.

Uma primeira aplicação que poderia ser acrescentada ao *framework* seria uma interface gráfica que permitisse ao usuário testar as configurações e gerar os arquivos de configuração automaticamente. Isto tornaria o uso das capacidades oferecidas por este *framework* o mais simples possível.

Acreditamos que também seria interessante, principalmente para usuários mais avançados, a criação de um laboratório virtual de classificação. Este laboratório ofereceria, com base neste *framework*, diversos algoritmos recorrentes, maior flexibilidade para montar os algoritmos e, idealmente, capacidade de vizualizar os dados.

Ainda nesta mesma linha, a implementação de um *player* de áudio que pudesse classificar e sugerir faixas de áudio de diversas maneiras diferentes seria uma ótima forma de continuar e divulgar o projeto.

Outra linha possível seria a reimplementação do *framework* em C++. Um grande atrativo dessa idéia linha é tornar a biblioteca disponível em outras linguagens (inclusive C♯, usando o Mono). Também seria possível o uso de extensões em diversas outras linguagens (não só em C♯) através do uso de interpretadores embarcados; seria possível usar extensões em Python, Lua e Matlab (embarcando o Octave), para citar algumas possibilidades. Entretanto, pela falta de suporte a reflexão no C++ o processo de criação de extensões nessa linguagem seria bem mais complicado, bem como o desenvolvimento da biblioteca.

De qualquer forma, acreditamos que este projeto abre uma porta para grandes possibilidades e que a sua continuação seria bastante interessante.

Referências Bibliográficas

[1] VOLLMANN, D., "Aspects of Reflection in C++", webpage, `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1751.html`, Created: 2005-01-14; Last Accessed: 2009-05-22. 14

[2] ZUSE, H., "The Life and Work of Konrad Zuse", webpage, `http://www.epemag.com/zuse/part4a.htm`. 3

[3] MITCHEL, J. C., *Concepts In Programming Languages*. Cambridge University Press, 2003. 3

[4] STROUSTRUP, B., *A Linguagem de Programação C++*. Bookman, 2000. 13

[5] DOLGE, A., *Pianos and Their Makers: A Comprehensive History of the Development of the Piano from the Monochord to the Concert Grand Player Piano By Alfred Dolge*. Courier Dover Publications, 1972.

[6] FOWLER, C. B., "The Museum of Music: A History of Mechanical Instruments", *Music Educators Journal*, v. 54, n. 2, pp. 45–49, October 1967. 3

[7] MCKAY, C., FUJINAGA, I., "automatic genre classification using large high-level musical feature sets", . 32

[8] TANENBAUN, A. S., WOODHUL, A. S., *Sistemas Operacionais*. Bookman, 2000. 13

[9] MALENFANT, J., JACQUES, M., DEMERS, F.-N., "A Tutorial on Behavioural Reflection and its Implementation". In: *Proceedings of Reflection*, 1996. 15

[10] LIPPENS, S., MARTENS, J. P., MULDER, T. D., *et al.*, "A Comparison of Human and Automatic Musical Genre Classification", *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP'04)*, v. 4, pp. 233–236, 2004. 65

[11] ROJAS, R., GöKTEKIN, C., FRIEDLAND, G., *et al.*, *Plankalkül: The First High-Level Programming Language and its Implementation*, Report, Freie Universität Berlin, February 2000. 4

[12] AUCOUTURIER, J.-J., PACHET, F., "Representing musical genre: a state of the art", *Journal of New Music Research*, v. 32, pp. 83–93, 2003.

[13] BARBEDO, J. A. G., LOPES, A., "Automatic genre classification of musical signals", *EURASIP Journal on Advances in Signal Processing*, v. Volume 2007, pp. 12 pages, 2007. Article ID 64960. x, 2, 32, 37, 41, 61, 67

[14] CATALTEPE, Z., YASLAN, Y., SONMEZ, A., "Music genre classification using MIDI and audio features", *EURASIP Journal in advances on advances in signal processing*, v. Volume 2007, pp. 8 pages, 2007. Article ID 36409. 32

[15] ELLIS, D., "mp3read and mp3write for Matlab", webpage, `http://www.ee.columbia.edu/~{}dpwe/resources/matlab/mp3read.html`, 07/07/2007.

[16] ENEVOLDSEN, M. B., *Object Oriented Language Interoperability*. Tese de M.Sc., University of Aarhus, 2004. 23

[17] INSTITUTE, S., "A Musical Toy in the Form of a Boat", webpage, `http://www.asia.si.edu/collections/zoomObject.cfm?ObjectId=9919`. ix, 4

[18] ISMIR2004, "5th international conference on music information retrieval, Barcelona, Spain. 10-14 outubro 2004", `http://ismir2004.ismir.net/genre{_}contest/index.htm`, Last Accessed: 30/07/2007. 33

[19] JIANG, D. N., ZHANG, H.-J., TAO, J.-H., *et al.*, "Music type classification by spectral contrast feature", *Proceedings. ICME 2002 - IEEE International Conference on Multimedia and Expo, Lausanne, Suíça. 26-29 de agosto de 2002*, v. Volume 1, Issue , 2002, pp. 113 – 116, 2002.

[20] LLNL, "Babel", webpage, `https://computation.llnl.gov/casc/components/babel.html`, LastAccessed: 2009.06.03 Last Updated: 2004.01.30. 23

[21] MITRI, G., UITDENBOGERD, A. L., CIESIELSKI, V., "Automatic music classification problems", .

[22] PACHET, F., CAZALY, D., "A taxonomy of musical genres", *Content-Based Multimedia Information Access Conference (RIAO), Paris, April, 2000.*, , 2000.

[23] PEETERS, G., "A large set of audio features for sound description", , 2004.

[24] RASMUSSEN, C., SOTTILE, M., RICKETT, C., "Chasm – Language Interoperability Tools", webpage, http://chasm-interop.sourceforge.net/, Last Accessed: 2009.06.03 Last Updated: 2005.07.13. 23

[25] RWC, "RWC music database: music genre", http://staff.aist.go.jp/m.goto/RWC-MDB/.

[26] School of Information Management and Systems, Berkeley, "How Much Info 2003", 2003, http://www2.sims.berkeley.edu/research/projects/how-much-info-2003/internet.htm#p2p, 19.Ago.2007.

[27] SHEFFIELD, T. U. O., "13th century programmable robot", webpage, http://www.shef.ac.uk/marcoms/eview/articles58/robot.html, LastAccessed:2009-06-30. 3

[28] SPTK Working Group, "Speech signal processing toolkit", http://www.sp.nitech.ac.jp/~tokuda/SPTK/index-j.html,30/07/2007. Developed by Department of Computer Science, Nagoya Institute of Technology e Interdisciplinary Graduate School of Science and Engineering, Tokyo Institute of Technology.

[29] TZANETAKIS, G., COOK, P., "Musical Genre Classification of Audio Signals", *IEEE Transactions on speech and audio processing*, v. 10, N 5, pp. 83–93, 2002.

[30] WEST, K., COX, S., "Features and classifiers for the automatic classification of musical audio signals", *Proceedings. ISMIR2004 - 5th International conference on music information retrieval, Barcelona, Espanha. 10-14 outubro 2004*, , 2004.

[31] "99 Bottles Of Beer", webpage, http://99-bottles-of-beer.net/, Last Accessed: 2009-02-22. 4

[32] "99 Bottles Of Beer", webpage, http://99-bottles-of-beer.net/lyrics.html, Last Accessed: 2009-02-22. 4

[33] "Common Language Specification", webpage, http://msdn.microsoft.com/en-us/library/12a7a7h3.aspx, Last Accessed: 2009.06.04. 24

[34] http://www.music-ir.org/mirex/2007/index.php/Audio_Artist_Identification, 18.Jan.2008.

[35] "Red Gate .Net Reflector", webpage, http://www.red-gate.com/products/reflector/index.htm. 8

[36] "Tiobe Index", webpage, www.tiobe.com, Last Accessed: 2009-05-21. x, 10

[37] “ECMA 334 - C♯ Specification”, June 2006. 19, 22

[38] “ECMA 335 - Common Language Interface”, June 2006. ix, 22, 24

[39] “Java Native Interface Specification”, webpage, 2003, `http://java.sun.com/javase/6/docs/technotes/guides/jni/spec/design.html#wp16696`, Last Accessed: 2009.06.04. 29