Sumário

[1 Introdução 5](#_Toc260771409)

[2 Referencial Teórico 6](#_Toc260771410)

[3 Ferramentas 7](#_Toc260771411)

[3.1 .NET Framework 7](#_Toc260771412)

[3.2 Visual studio 2008 7](#_Toc260771413)

[3.3 DirectX 7](#_Toc260771414)

[3.4 SlimDX 8](#_Toc260771415)

[3.5 NUnit 8](#_Toc260771416)

[3.6 TestMatrix 8](#_Toc260771417)

[3.7 DB4Objects 8](#_Toc260771418)

[3.8 Subversion 9](#_Toc260771419)

[3.9 UML 9](#_Toc260771420)

[4 Metodologia 10](#_Toc260771421)

[4.1 Pesquisa Inicial 10](#_Toc260771422)

[4.2 Especificação das Classes 11](#_Toc260771423)

[4.3 Especificação dos Casos de Teste 11](#_Toc260771424)

[4.4 Implementação das Classes 13](#_Toc260771425)

[4.5 Validação dos Casos de Teste 13](#_Toc260771426)

[4.6 Experimentos 14](#_Toc260771427)

[5 Implementação 15](#_Toc260771428)

[5.1 Songer.UnitTests 16](#_Toc260771429)

[5.1.1 Classe MusicalAnalyzerTests 17](#_Toc260771430)

[5.1.2 Classe NotesAndChordsTests 17](#_Toc260771431)

[5.2 Songer.SoundInput 18](#_Toc260771432)

[5.2.1 Classe SoundSource 19](#_Toc260771433)

[5.2.2 Classe LineInCapture 20](#_Toc260771434)

[5.2.3 Classe WaveFileCapture 21](#_Toc260771435)

[5.2.4 Classe SoundDetectedEventArgs 22](#_Toc260771436)

[5.3 Songer.SoundAnalysis 22](#_Toc260771437)

[5.3.1 Estrutura ComplexNumber 23](#_Toc260771438)

[5.3.2 Classe CooleyTukeyFFT 23](#_Toc260771439)

[5.4 Songer.MusicalInterpreter 24](#_Toc260771440)

[5.4.1 Classe MusicalNote 25](#_Toc260771441)

[5.4.2 Classe MusicalNoteDictionary 25](#_Toc260771442)

[5.4.3 Classe Chord 26](#_Toc260771443)

[5.4.4 Classe ChordDictionary 27](#_Toc260771444)

[5.4.5 Classe MusicalAnalyzer 27](#_Toc260771445)

[5.5 Songer.Database 27](#_Toc260771446)

[5.5.1 Classe Song 28](#_Toc260771447)

[5.5.2 Classe SongStore 28](#_Toc260771448)

[5.6 Songer.Presentation 30](#_Toc260771449)

[5.6.1 Aba Inserir 30](#_Toc260771450)

[5.6.2 Aba Buscar 31](#_Toc260771451)

[6 Experimentos 32](#_Toc260771452)

[7 Conclusão 33](#_Toc260771453)

[8 Referência Bibliográfica 34](#_Toc260771454)

Lista de Figuras

[Figura 4‑1: Fluxo da metodologia utilizada 10](#_Toc260862897)

[Figura 4‑2: Exemplo do menu Generate Method Stub 12](#_Toc260862898)

[Figura 5‑1: Diagrama dos módulos do projeto. 15](#_Toc260862899)

[Figura 5‑2: Diagrama de classes do módulo Songer.UnitTests 16](#_Toc260862900)

[Figura 5‑3: Exemplo de falha de teste. 18](#_Toc260862901)

[Figura 5‑4: Diagrama de classes do módulo Songer.SoundInput 19](#_Toc260862902)

[Figura 5‑5: Diagrama de classes do módulo Songer.SoundAnalysis 23](#_Toc260862903)

[Figura 5‑6: Diagrama de classes do módulo Songer.MusicalInterpreter 24](#_Toc260862904)

[Figura 5‑7: Diagrama de classes do módulo Songer.Database 33](#_Toc260862905)

[Figura 5‑8: Tela de inserção de nova música 35](#_Toc260862906)

[Figura 5‑9: Tela de busca de música 36](#_Toc260862907)

Lista de Quadros

[Quadro 4‑1: Exemplo do método gerado pelo Visual Studio 12](#_Toc260862991)

[Quadro 5‑1: Exemplo de teste de unidade 17](#_Toc260862992)

[Quadro 5‑2: Código-fonte do método Chord.Matches 27](#_Toc260862993)

[Quadro 5‑3: Código-fonte do método MusicalAnalyzer.OnSoundDetected 28](#_Toc260862994)

[Quadro 5‑4: Código-fonte do método MusicalAnalyzer.GetNotesBeingPlayed 30](#_Toc260862995)

[Quadro 5‑5: Código-fonte do método MusicalAnalyzer.GetChordBeingPlayed 31](#_Toc260862996)

[Quadro 5‑6: Código-fonte do método MusicalAnalyzer.FindBestChord 31](#_Toc260862997)

[Quadro 5‑7: Código-fonte do método MusicalAnalyzer.ProcessChordsSequence 32](#_Toc260862998)

[Quadro 5‑8: Código-fonte do método Song.Matches 33](#_Toc260862999)

[Quadro 5‑9: Código-fonte dos métodos SongStore.Add e SongStore.Delete 34](#_Toc260863000)

[Quadro 5‑10: Código-fonte do método SongStore.Search 34](#_Toc260863001)

# Introdução

# Referencial Teórico

Este capítulo trata dos conceitos necessários para o desenvolvimento deste trabalho. Mesmo sendo um trabalho que trata do desenvolvimento de software, e voltado para a área de informática, este utiliza diversos conceitos de som e música, que não são comuns a essas pessoas. Aqui serão apresentados a mecânica do som, e como música pode ser criada a partir de diferentes sons.

## Som

## Música

Segundo a definição do dicionário Gama Kury (2002, p. 728), música é a “arte de combinar sons para que produzam um efeito agradável”, sendo estes sons conhecidos como notas musicais.

Uma nota musical é o menor elemento de som perceptível pelo ouvido humano, caracterizado pela sua freqüência fundamental (medida em Hertz). No caso de um violão, essa é a medida do número de vezes que a corda irá vibrar a cada segundo quando uma nota qualquer for tocada.

Para Holst (2004, p. 4), uma nota musical é um nível de som claro e sustentado, sendo diferenciados de outros sons não-musicais pela regularidade no vai e vem das ondas sendo geradas pela mesma.

Um conjunto de notas sendo tocadas juntas formam um acorde, podendo estes serem tríades, quando forem um conjunto de três sons, ou tétrades, quando forem um conjunto de quatro sons simultâneos.

De acordo com Straus (2008, p. 291), uma tríade em notação musical pode ser “escrita em três linhas ou espaços consecutivos [...] a nota mais baixa é chamada de raiz, a do meio é chamada de terça e a mais alta é chamada de quinta”[[1]](#footnote-2). A tétrade é formada pela adição da sétima à tríade.

Para Adolfo (1989, p. 32), uma música é formada por quatro elementos básicos, sendo eles:

* Melodia – representada pela linha melódica principal, normalmente preenchida pela voz, e onde se encaixa a letra da música;
* Harmonia – representada pelo conjunto de acordes;
* Pulsação – representada pelo ritmo do acompanhamento;
* Baixo – representada pela nota fundamental (ou raiz) do acorde.

Em um sistema para detecção automática de acordes, as partes mais importantes da música a se levar em consideração são a harmonia, pois é onde estão descritos os mesmos; e a pulsação, que permitirá extrair onde, em uma música, são realizadas as mudanças de acorde, permitindo assim gerar uma representação da mesma.

# Ferramentas

Este capítulo apresenta as ferramentas que serão utilizadas na implementação do sistema proposto neste trabalho científico.

## .NET Framework

O .NET Framework é uma plataforma para desenvolvimento de aplicativos, que provê uma grande biblioteca para construção rápida de aplicativos (RAD), abstraindo diversos aspectos do sistema operacional, e uma máquina virtual para execução gerenciada dos mesmos. Atualmente está na versão 4.0 (lançada em Abril de 2010), mas para este trabalho foi escolhida a versão 3.5 pois era a mais recente quando o mesmo foi iniciado (em Dezembro de 2009).

O .NET permite o desenvolvimento em qualquer linguagem que possuam um compilador específico para a plataforma. Para este trabalho foi escolhida a linguagem C#, que além de ser a mais utilizada na plataforma, é aquela em que o autor possui mais de sete anos de experiência.

## Visual studio 2008

O Visual Studio é um ambiente integrado de desenvolvimento, que permite a criação de aplicativos para desktop (Windows) e web, tanto em código nativo (com linguagem C++) e para o .NET Framework. Novamente, a escolha foi realizada devido à experiência do autor, de mais de sete anos com o ambiente, tendo trabalhado com o mesmo desde a versão 2002 (conhecido popularmente como apenas como Visual Studio.NET).

## DirectX

O DirectX é uma coleção de API[[2]](#footnote-3)s para tratar de tarefas relacionadas à multimídia no sistema operacional Windows, provendo uma abstração do hardware para chamadas de software, utilizado inicialmente para o desenvolvimento de jogos. Neste trabalho, foi utilizada para ter acesso a recursos de captura de áudio, através das APIs do DirectSound.

## SlimDX

A SlimDX é uma biblioteca de código-aberto que disponibiliza acesso às APIs do DirectX através do .NET Framework. Foi criada como um substituto ao Managed DirectX da Microsoft (descontinuado em Outubro de 2006, antes do lançamento da segunda versão), e dá suporte a estruturas de código do .NET Framework 2.0 em diante, além de ter compatibilidade total com o Visual Studio 2008.

## NUnit

O NUnit é uma plataforma de testes de unidade, composta de uma biblioteca de classes para desenvolvimento dos testes e aplicativos para execução dos mesmos. É utilizada para a validação de pequenas unidades de código-fonte.

## TestMatrix

É uma ferramenta que se integra ao Visual Studio e permite a execução dos testes de unidade dentro do mesmo, indicando quais testes executaram, se a execução ocorreu com sucesso ou não, e informando todas as linhas de código que a execução percorreu, facilitando a depuração e análise, não apenas dos testes, mas também do código do sistema em si.

## DB4Objects

Este é um banco de dado orientado a objetos, de código aberto, e totalmente desenvolvido em C#. Ele pode ser integrado ao sistema, eliminando assim a necessidade de um gerenciador de banco de dados executando em paralelo, e facilitando a distribuição do mesmo. No versão final, é apenas uma DLL que é distribuída junta ao aplicativo, não sendo necessária a instalação de um aplicativo adicional.

## Subversion

O Subversion é um sistema de controle de versão, utilizado para manter as mudanças nas versões atuais e anteriores de arquivos como código-fonte e documentos, permitindo assim que sejam analisadas e/ou desfeitas as mudanças no mesmo. O site XP-Dev foi utilizado para fornecer o servidor subversion para este projeto, além de algumas ferramentas para gerenciamento de projeto.

## UML

A UML é uma linguagem de modelagem que permite que o desenvolvedor visualize o desenvolvimento do código de seu trabalho em diagramas padronizados. Para a criação dos diagramas de classe foi utilizado o próprio Visual Studio (ferramenta Class Diagram) e para os demais diagramas foi utilizado o Microsoft Visio.

# Metodologia

Devido à natureza experimental deste trabalho, foi detectada a necessidade de uma sistemática para validação do código implementado. Este capítulo descreve a metodologia utilizada e suas etapas, conforme a Figura 4‑1.

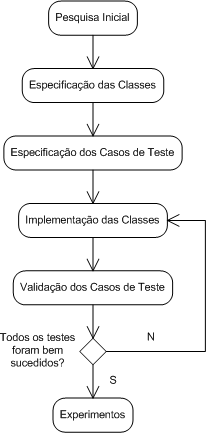


Figura ‑: Fluxo da metodologia utilizada

## Pesquisa Inicial

Nesta estapa foram realizados estudos sobre as implementações das tarefas básicas do sistema, como a obtenção de dados de áudio, através de arquivos do tipo Wave e captura de som pela entrada de linha, além da obtenção do espectrograma do sinal de áudio, para sua posterior análise. Este trabalho não irá se deter a melhorar estes algoritmos já que os mesmos são utilizados em outros sistemas há bastante tempo e são considerados triviais.

## Especificação das Classes

Todas as principais classes devem ser especificadas, mas somente o seu nome e sua utilização são importantes neste momento, suas propriedades e métodos serão idealizados no passo a seguir. Esta é também a primeira etapa do TDD.

## Especificação dos Casos de Teste

Já com as classes definidas, deverão ser criados casos de teste que validem a implementação do analisador musical. Estes casos serão criados utilizando-se o NUnit para implementação dos testes de unidade, que referenciam as classes criadas anteriormente. Durante a implementação destes testes serão definidos os métodos das classes do sistema, através de ferramentas disponíveis no Visual Studio. Um exemplo pode ser visto na Figura 4‑2, que mostra o menu Generate Method Stub, o qual permite adicionar métodos ainda não implementados a uma classe de forma automática, e seu resultado no Quadro 4‑1.

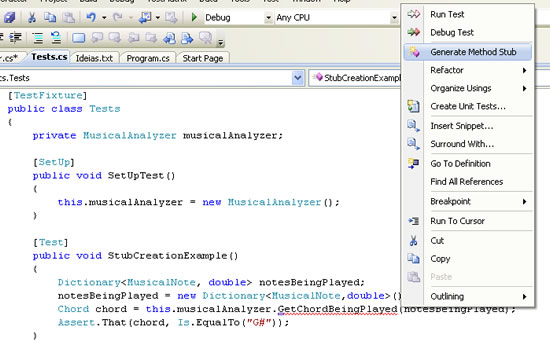


Figura ‑: Exemplo do menu Generate Method Stub

|  |
| --- |
| public class MusicalAnalyzer  {  public Chord GetChordBeingPlayed(Dictionary<MusicalNote, double>  notesBeingPlayed)  {  throw new NotImplementedException();  }  } |

Quadro ‑: Exemplo do método gerado pelo Visual Studio

Analisando o código gerado automaticamente, nota-se que ele dispara uma exceção do tipo NotImplementedException. Essa exceção é importante, pois nos informa que o método ainda não foi implementado e faz com que o teste que utiliza este método falhe. O desenvolvimento utilizando TDD preza que se escreva o máximo de testes e que todos eles falhem nesta fase.

## Implementação das Classes

Após especificados os testes, o código dos métodos gerados deverá ser implementado, bem como métodos auxiliares (privados) que estes necessitem. A idéia aqui é escrever o código que faça com que os testes de unidade deixem de falhar, validando assim a implementação das classes.

A fim de facilitar o desenvolvimento, serão definidos como base o acorde G (Sol Maior) para verificar o reconhecimento de um acorde único, e a seqüência G D Em C (Sol Maior, Ré Maior, Mi Menor e Dó Maior) para verificar a extração de uma seqüência de acordes em uma passagem de som.

## Validação dos Casos de Teste

Neste ponto os testes deverão ser executados. O que se espera é que todos eles não falhem para que possa se prossiga com os experimentos, caso contrário, deverá voltar à fase anterior e tentar encontrar as falhas de implementação das classes.

Como visto na figura 4.1, esta etapa deverá ser repetida diversas vezes durante a implementação do projeto, até que se tenha certeza de que nenhuma das classes possua alguma falha que possam interferir nos experimentos e invalidar este trabalho.

Para a execução dos testes, serão utilizados arquivos Wave gravados a partir de acordes tocados no instrumento musical selecionado, a fim de automatizá-los. Aqui serão testados os reconhecimentos de vários acordes e seqüências, diferentes dos citados anteriormente.

## Experimentos

Por fim, após os testes serem executados exaustivamente, deverão ser realizados experimentos que comprovem a eficácia dos algoritmos implementados em relação à detecção de acordes. Para esta etapa os experimentos deverão ser realizados tanto com arquivos Wave pré-gravados, como com o instrumento musical real, e seus resultados analisados.

# Implementação

Este capítulo apresenta os conceitos e algoritmos utilizados na implementação do sistema proposto neste trabalho, bem como as classes pelas quais é composto o sistema. Os algoritmos foram desenvolvimentos, tanto o código como os comentários, utilizando somente palavras da língua inglesa, visando à internacionalização do código-fonte desenvolvido. No corpo desde capítulo será explicado o que cada um deles realiza.

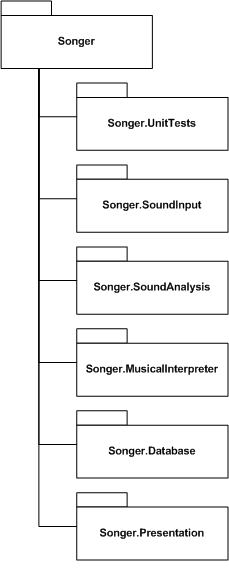


Figura ‑: Diagrama dos módulos do projeto.

O sistema foi divido em módulos para facilitar a implementação e futuro entendimento do mesmo. Como o Visual Studio possui o conceito de uma área de trabalho (conhecida como solução) composta de um ou mais projetos, cada modulo foi desenvolvido como um projeto separado dentro de uma única solução chamada Songer. A Figura 5‑1 apresenta a representação gráfica da solução, mostrando os módulos e seus namespaces[[3]](#footnote-4).

## Songer.UnitTests

Este módulo contém os testes de unidade que serão escritos antes da implementação dos outros módulos, para fazer a posterior validação dos mesmos, de acordos com as regras de TDD.

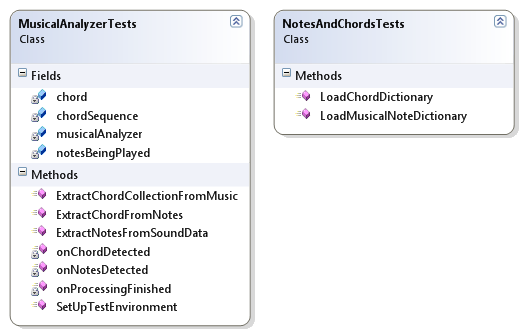


Figura ‑: Diagrama de classes do módulo Songer.UnitTests

De acordo com a Figura 5‑2, há apenas duas classes de testes, e ambas são referentes ao módulo Songer.MusicalInterpreter. A opção de não escrever testes para os outros módulos é porque os mesmos servem de apoio ao Songer.MusicalInterpreter, que é a parte central deste trabalho, e portanto, serão indiretamente testados quando este o for.

### Classe MusicalAnalyzerTests

Esta classe possui os principais testes do projeto. Eles verificarão se o sistema consegue extrair um conjunto de notas musicais a partir de uma leitura de entrada de áudio, em seguida se ele consegue informar qual acorde é formado por este conjunto de notas, e por último se ele é capaz de mostrar toda a seqüência de acordes contida em uma passagem de uma música. O Quadro 5‑1 mostra o último teste, que se certifica de que o módulo de análise musical consegue devolver uma sequencia de acordes correta, a partir de uma passagem de som.

|  |
| --- |
| [Test]  public void ExtractChordCollectionFromMusic()  {  this.musicalAnalyzer.AnalyzeAudio(@"..\Sounds\Seq1.wav");  while (this.chordSequence == string.Empty)  {  Thread.Sleep(100);  }  Assert.That(this.chordSequence, Is.EqualTo("G D Em C"));  }  private void onProcessingFinished(object sender,  AudioProcessingFinishedEventArgs e)  {  this.chordSequence = e.Chords;  } |

Quadro ‑: Exemplo de teste de unidade

### Classe NotesAndChordsTests

Esta classe é um pouco mais simples que a anterior e irá averiguar apenas se os dicionários de notas musicas e de acordes (contidos em arquivos de textos, nomeados Notes.txt e Chords.txt, respectivamente) são lidos de forma correta, já que eles serão utilizados pelo MusicalAnalyzer.

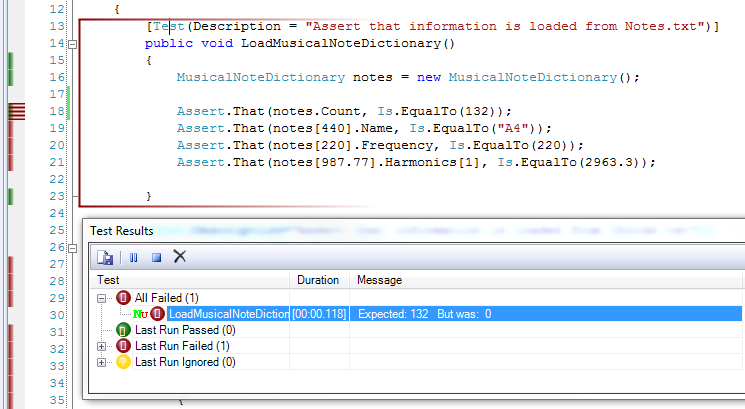


Figura ‑: Exemplo de falha de teste.

Como mencionado no item 4.3 (Especificação dos casos de teste), o código destas classes deverá ser escrito antes da implementação dos outros módulos do sistema, fazendo com que os testes falhem. A Figura 5‑3 mostra o resultado da execução do teste nesta fase.

## Songer.SoundInput

Este módulo é responsável pela obtenção de dados referentes a sinais de áudio, para que estes possam ser posteriormente analisados pelo módulo Songer.SoundAnalysis. A Figura 5‑4 apresenta as classes que o compõe.

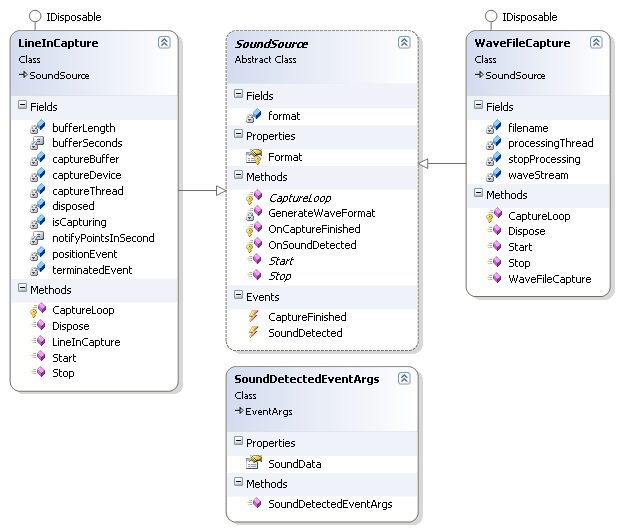


Figura ‑: Diagrama de classes do módulo Songer.SoundInput

Este módulo é composto por quatro classes, sendo uma abstrata que serve de modelo para implementação das classes de obtenção de dados, duas implementações de formas de obtenção de dados e uma classe para passar objetos em eventos gerados por estas classes.

### Classe SoundSource

Esta é uma classe abstrata, não sendo utilizada diretamente, mas servindo de base às classes WaveFileCapture e LineInCapture. Ela é responsável por criar um modelo de funções que as classes que herdam dela devem implementar, além de implementar algumas funções essenciais, como por exemplo, fornecer o formato padrão de arquivo wave utilizado.

A função GenerateWaveFormat cria um instancia da classe WaveFormat, que contém as informações de formato do arquivo wave com o qual se está trabalhando. Para este trabalho, o formato é o mesmo utilizado em CD-Áudio, com 44100 amostragens de 16 bits por segundo (44100Hz/16-bit), mas com apenas um canal (mono), já que o instrumento utilizado nos testes (guitarra) fornece apenas um único canal de áudio.

A classe SoundSource fornece também dois eventos: SoundDetected e CaptureFinished, que são disparados, respectivamente, sempre que um conjunto de dados de som está disponível para ser processado e quando não há mais dados a serem processados.

Por fim, a classe fornece três métodos abstratos que todas as classes que herdam desta devem implementar: Start e Stop, responsáveis por iniciar e parar a captura de som; e CaptureLoop, executado como uma *thread[[4]](#footnote-5)* secundária, onde ocorre a leitura do buffer de som. Vale ressaltar que esta *thread* adicional é importante para que o sistema continue realizando a captura (principalmente quando de um instrumento musical) continuamente, mesmo quando os outros módulos estão processando as informações que já lhe foram passadas.

### Classe LineInCapture

A primeira classe que herda de SoundSource é a LineInCapture, responsável pela captura de som a partir de um dispositivo de entrada (por padrão, o line-in da placa de som). Para isso, foram utilizadas as APIs do DirectSound, em especial as classe DirectSoundCapture, que realiza a captura do som, e a classe CaptureBuffer, que armazena temporariamente a informação que foi capturada.

No método Start são criadas instancias destas duas classes, para se realizar a captura. Além disso, são criadas também posições de notificação (duas a cada segundo), que fará o papel de semáforo, permitindo que o buffer do som que já foi capturado seja lido rapidamente e repassado para o processamento nas classes de análise de sinal.

Em seguida, a classe inicializa a thread secundária, tendo o método CaptureLoop como código de execução. Nele o buffer de captura é inicializado e fica aguardando uma das mensagens de notificação para realizar a leitura. Quando isto ocorre, ele dispara o evento OnSoundDetected, passando os dados lidos do buffer para as classes inscritas neste evento, e em seguida volta a esperar a próxima notificação.

Por fim, o método Stop elimina a thread, fechando todos os objetos de captura. Vale salientar que esta classe funciona de forma automática. Quando o usuário informar que quer gravar um som, a mesma deve ler o buffer continuamente até que seja detectado um sinal de entrada de som, e quando o mesmo acabar (o que implica que o usuário parou de tocar), a própria classe deve chamar o método Stop automaticamente.

### Classe WaveFileCapture

A outra classe que herda de SoundSource é a WaveFileCapture, que é responsável por ler a informação de áudio diretamente de um arquivo wave previamente gravado. A principal utilidade da mesma é para fazer a inserção de informações no banco de dados de forma mais rápida e automatizada, podendo o operador fazer a leitura de diversos arquivos ao invés de precisar “tocar” cada uma das músicas em um instrumento.

O seu construtor permite informar o nome do arquivo que será lido, sendo necessário que seja criado um novo objeto sempre que se quiser ler importar uma nova música ao banco, tendo em vista que após o processamento do arquivo a thread de captura é finalizada automaticamente. Para a leitura, foi utilizada a classe WaveStream (disponível no SlimDX), que facilita a leitura e interpretação de arquivos wave.

Nesta classe o método Start realiza apenas a criação da thread de captura. O buffer de captura e as notificações não são necessárias pois, diferente de uma entrada de linha, todos os dados do áudio já são conhecidos e finitos.

No loop de captura, é realizada a leitura do arquivo wave, enviando para processamento um segundo de áudio por vez, e saindo do loop (encerrando assim a thread) assim que for processado o último segundo de áudio do arquivo.

O método Stop desta classe aborta a thread imediamente, sendo útil quando se quiser cancelar a leitura de um arquivo.

### Classe SoundDetectedEventArgs

A utilidade desta classe é fornecer um conjunto de dados relevantes ao som detectado e que serão disponibilizados à classe que se inscreveu ao evento SoundDetected da classe SoundSource quando o mesmo for disparado.

## Songer.SoundAnalysis

Este módulo é responsável pelo processamento do áudio capturado pelo Songer.SoundInput e geração do espectrograma do mesmo, que será analisado posteriormente pelo módulo Songer.MusicalInterpreter. A Figura 5‑5 apresenta as classes pertencentes a este módulo.

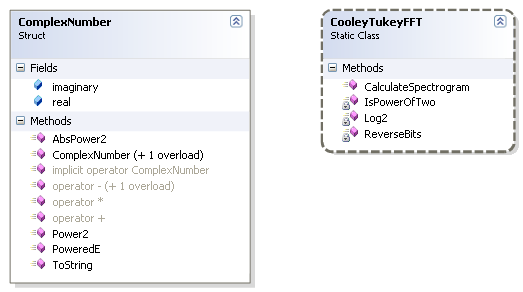


Figura ‑: Diagrama de classes do módulo Songer.SoundAnalysis

Este módulo é composto de apenas uma classe e uma estrutura de dados. A estrutura é uma implementação de números complexos e a classe é uma implementação do algoritmo Cooley-Tukey. Ambos são baseados nas implementações do artigo *FFT Guitar Tuner*. **(AQUI FALTA A REFERENCIA)**

### Estrutura ComplexNumber

Esta estrutura armazena um número complexo, tendo suporte a quase todas as operações básicas (soma, subtração e multiplicação), além de algumas adicionais, como potência de 2 (normal e absoluta) e exponenciação. Esta estrutura é necessário para o calculo da FFT.

### Classe CooleyTukeyFFT

Realiza a extração do espectrograma do som através do algoritmo FFT de Cooley-Tukey. Primeiramente se certifica de que o tamanho do vetor é uma potencia de dois (senão, adiciona itens zerados até que seja uma potencia de dois), em seguida gera um vetor inverso dos dados de som, calculando então a FFT. Com o resultado da FFT, cria-se um vetor to tipo Double[], onde cada item é uma representação de uma freqüência, e o seu valor é a amplitude do som referente àquela freqüência. Por fim, retorna este vetor para a função de origem.

## Songer.MusicalInterpreter

Este é o módulo principal do projeto, responsável pela interpretação do espectrograma (freqüência e amplitude dos sinais) e sua conversão em notas musicais (de acordo com a freqüência fundamental de cada nota). Além disso, este módulo também realiza a conversão do conjunto de notas em um determinado espaço de tempo para o acorde que as possui. A Figura 5‑6 apresenta as classes que compõem este módulo.

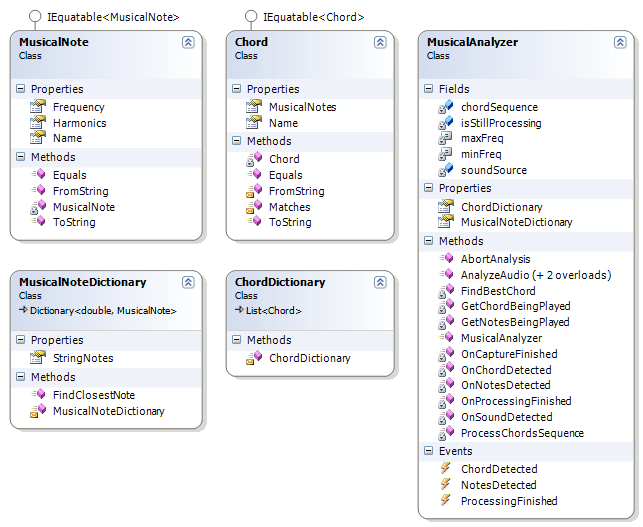


Figura ‑: Diagrama de classes do módulo Songer.MusicalInterpreter

O módulo é composto de cinco classes, sendo duas delas utilizadas para representação de notas e acordes (MusicalNote e Chord), outras duas para representar dicionários de notas e acordes (MusicalNoteDictionary e ChordDictionary), e a última responsável pela análise de espectrograma (MusicalAnalyzer).

### Classe MusicalNote

Esta classe representa uma nota musical, tendo o seu nome (no padrão <nota><oitava><sustenido?>), sua freqüência fundamental (que representa a nota), e os harmônicos que esta nota também gera. Para se criar um instancia dessa classe, todas estas informações devem ser passadas em forma de string, no padrão <nome nota>,<freqüência fundamental>,<harmônico 1>,...,<harmônico n>.

Esta classe implementa a interface IEquatable, que será útil para fazer a comparação de duas instancias apenas pelo seu nome, ignorando as freqüências, e tornando-a mais rápida.

### Classe MusicalNoteDictionary

Esta classe representa o conjunto de todas as notas musicais audíveis, da oitava 0 (C0 ou Dó 0) até a oitava 10 (B10 ou Si 10). Ela o faz através da leitura do arquivo Notes.txt, que possui todas as informações de nota no padrão informado no parágrafo anterior, sendo uma nota por linha lida. Ao ler o arquivo, a classe armazena a posição da coleção onde estão armazenadas as informações das notas correspondentes às cordas soltas da guitarra, disponibilizando esta informação através da propriedade StringNotes.

O único método disponível nesta classe, FindClosestNote, é responsável por encontrar a nota mais próxima à freqüência informada. Ela é necessária pois nem todas as freqüências que espectrograma disponibilizou são absolutas em relação às notas musicas, além de a guitarra poder estar um pouco desafinada ou até mesmo a posição do dedo pode causar uma variação na casa de décimos de hertz, imperceptível ao ouvido humano, mas não para o computador, que é mais sensível a estas mudanças de freqüência.

### Classe Chord

Semelhante à classe MusicalNote, esta classe armazena as informações de um acorde, sendo elas o nome do mesmo (por exemplo Gm, para Sol Menor) e as notas que compõem o mesmo (de 3 a 6 notas).

Assim como a classe MusicalNote, esta classe é instanciada a partir de uma string com as informações do acorde, no padrão <nome> <6 5 4 3 2 1>, onde os números representam a posição do dedo no braço da guitarra para a formação de um acorde. A partir da nota da corda solta (como mencionado anteriormente, disponibilizada pela classe MusicalNoteDictionary), é possível somar o número de casas até e saber qual exatamente é a nota musical naquela casa específica.

A interface IEquatable é implementa também na classe Chord devido à necessidade de comparar acordes. Aqui são comparados apenas os nomes dos mesmos, já que, independente das notas que o formem, se eles possuem o mesmo nome é porque, para o ouvido humano, o som de ambos os acordes, em uma música, são complementares, então um pode facilmente substituir o outro. Além disso, esse tipo de comparação será importante mais à frente, na classe MusicalAnalyzer.

Esta classe também implementa o método Matches, que recebe como parâmetro um conjunto de notas musicais e verifica se o acorde da mesma pode ser formado com estas notas, sendo este método um dos pontos chave do sistema. O Quadro 5‑2 mostra a sua implementação.

|  |
| --- |
| internal bool Matches(IEnumerable<MusicalNote> notesBeingPlayed)  {  foreach(MusicalNote note in this.MusicalNotes)  {  if (!notesBeingPlayed.Contains(note))  return false;  }  return true;  } |

Quadro ‑: Código-fonte do método Chord.Matches

### Classe ChordDictionary

Esta classe representa um dicionário de acordes, lido a partir do arquivo Chors.txt, contendo uma lista de strings no padrão de acorde informado no parágrafo anterior. Nenhuma propriedade ou método adicional é fornecido, somente a lista de acordes, que é a própria classe.

### Classe MusicalAnalyzer

Esta é a principal classe do sistema, que irá realizar o processamento dos dados obtidos (espectrograma) para extração da informação procurada (notas e acordes). Devido ao grande número de métodos exclusivos deste projeto implementados nesta classe, será dada uma ênfase maior à explicação de cada um deles.

Além dos métodos, esta classe possui também três eventos que podem ser inscritos por classes que a utilizem: NotesDetected, que retorna um conjunto de notas musicais a cada notificação de processamento; ChordDetected, que retorna o possível acorde formado pelo conjunto de notas repassado anteriormente; e por fim, o evento ProcessingFinished, que além de informar que a análise foi finalizada, retorna também o conjunto de acordes encontrado no áudio.

O primeiro método é o AnalyzeAudio, que pode ser chamado de duas formas: passando como parâmetro uma música (informando o caminho do arquivo Wave), ou sem parâmetro, o que automaticamente inicia a recepção de áudio pela entrada de linha do computador. A partir disto, ele se inscreve nos eventos do objeto de entrada (que é do tipo SoundSource, explicado anteriormente), que fornecerão a ele os dados para processamento (espectrograma e informação de término da captura).

O método AbortAnalysis permite que a análise seja interrompida prematuramente, mas ainda assim, as informações já processadas são retornadas através da chamada do evento ProcessingFinished.

Estes dois métodos supracitados são os únicos públicos (isto é, que podem ser chamados por outras classes) que esta classe possui. Os demais são todos privados e se comunicam com as outras classes através de eventos.

O primeiro deles é o OnSoundDetected, que é chamado quando ocorre o evento SoundDeteced no objeto SoundSource criado na chamada do método AnalyzeAudio. Sua função é receber o espectrograma, e chamar os métodos para extração de notas musicais e acordes, disparando então os eventos NotesDetected e ChordDetected, além de armazenar o acorde em uma lista para posterior formação da seqüência de acordes. O Quadro 5‑3 mostra esta sua implementação.

|  |
| --- |
| private void OnSoundDetected(object sender, SoundDetectedEventArgs e)  {  Dictionary<MusicalNote, double> notesBeingPlayed =  this.GetNotesBeingPlayed(e.SoundData);  if (notesBeingPlayed.Count > 0)  {  this.OnNotesDetected(notesBeingPlayed);  }    Chord chordBeingPlayed = this.GetChordsBeingPlayed(notesBeingPlayed);  if (chordBeingPlayed != null)  {  this.OnChordDetected(chordBeingPlayed);  this.chordSequence.Add(chordBeingPlayed);  }  } |

Quadro ‑: Código-fonte do método MusicalAnalyzer.OnSoundDetected

O método GetNotesBeingPlayed é o responsável pela extração das notas musicais. Isso é realizado primeiramente extraindo-se o espectrograma do som, e em seguida fazendo a tradução do mesmo para notas musicais, armazenando apenas as que têm uma amplitude aceitável. O mostra a implementação do mesmo. Devido ao tamanho do código, foram incluídos alguns comentários em português para facilitar o entendimento. Sua implementação pode ser vista no Quadro 5‑4.

|  |
| --- |
| private Dictionary<MusicalNote, double> GetNotesBeingPlayed(  short[] soundData)  {  Dictionary<MusicalNote, double> notesBeingPlayed = new  Dictionary<MusicalNote, double>();  // Chama o módulo SoundAnalysis para retornar  // o espectrograma dos dados de som recebidos  double[] spectrogram = CooleyTukeyFFT.CalculateSpectrogram(soundData);    // Descobre qual a frequência que possui a amplitude  // mais alta de som  int index = 0;  double max = spectrogram[0];  int usefullMaxSpectr = Math.Min(spectrogram.Length,  (int)(maxFreq \* spectrogram.Length / 44100) + 1);  for (int i = 1; i < usefullMaxSpectr; i++)  {  if (max < spectrogram[i])  {  max = spectrogram[i];  index = i;  }  }  // Processa apenas se a frequência encontrada for maior  // que a frequência mínima que um violão/guitarra pode emitir  if (((double)44100 \* index / spectrogram.Length) > minFreq)  {  // Percorre todo o espectrograma e guarda na lista apenas  // as notas com frequência e amplitude aceitáveis  for (int i = 1; i < usefullMaxSpectr; i++)  {  // Encontra a frequência da nota  double freq = (double)44100 \* i / spectrogram.Length;  double amplitude = spectrogram[i];  if (freq < minFreq || amplitude < 80000000000000)  continue;  amplitude /= 1000000000000;  MusicalNote musicalNote =  MusicalAnalyzer.MusicalNoteDictionary.FindClosestNote(freq);  // Como uma mesma nota pode aparecer mais de uma vez,  // salva a maior amplitude em que ela aparecer  if (notesBeingPlayed.ContainsKey(musicalNote))  {  if (notesBeingPlayed[musicalNote] < amplitude)  notesBeingPlayed[musicalNote] = amplitude;  }  else  {  notesBeingPlayed.Add(musicalNote, amplitude);  }  }  }  return notesBeingPlayed;  } |

Quadro ‑: Código-fonte do método MusicalAnalyzer.GetNotesBeingPlayed

Quanto à extração de acordes, o método GetChordBeingPlayed faz uso das notas musicais encontradas, filtrando-as para que sejam utilizadas apenas as 12 de maior amplitude, já que o resto é apenas ruído. Para isso foi utilizado o LINQ[[5]](#footnote-6).

Em seguida, é chamado o método Matches da classe Chord (mostrado no item 5.4.3) para cada objeto de acorde no dicionário ChordDictionary, salvando em uma lista de possíveis acordes. Terminada a busca, deverá ser retornado o acorde correto, mas podem ocorrer três casos diferentes:

* Nenhuma acorde foi encontrado: retorna nulo;
* Apenas um acorde foi encontrado: retorna o mesmo;
* Dois ou mais acordes encontrados: chama o método FindBestChord para encontrar o de maior probabilidade.

O apresenta a implementação deste método, novamente com comentários em português para melhor explicação de suas partes.

|  |
| --- |
| private Chord GetChordBeingPlayed(Dictionary<MusicalNote, double>  notesBeingPlayed)  {  List<Chord> possibleChords = new List<Chord>();  // Certifica-se que pelo menos 3 notas foram tocadas,  // já que um acorde é formado de pelo menos 3 notas (tríade)  if (notesBeingPlayed.Count > 2)  {  // Filtra apenas as 12 notas com maior amplitude utilizando LINQ  IEnumerable<MusicalNote> filteredNotes =  (from note in notesBeingPlayed  orderby note.Value select note.Key).Take(12);    // Verifica quais são os possíveis acordes, de acordo  // com as notas musicais  foreach (Chord chord in MusicalAnalyzer.ChordDictionary)  {  if (chord.Matches(filteredNotes) &&  !possibleChords.Contains(chord))  {  possibleChords.Add(chord);  }  }  }  switch (possibleChords.Count)  {  case 0: //Nenhum acorde encontrado  return null;  case 1: //Apenas um acorde encontrado  return possibleChords[0];  default: //Vários acordes encontrados  //Tentar encontrar o de maior probabilidade  return this.FindBestChord(notesBeingPlayed, possibleChords);  }  } |

Quadro ‑: Código-fonte do método MusicalAnalyzer.GetChordBeingPlayed

O método FindBestChord realiza um cálculo de probabilidade de acordo com a amplitude das notas, trazendo aquela que possuir um conjunto de notas com maiores amplitudes, como pode ser visto no Quadro 5‑6.

|  |
| --- |
| private Chord FindBestChord(Dictionary<MusicalNote, double>  notesBeingPlayed, List<Chord> chords)  {  return chords[0]; //TODO!!!!!  } |

Quadro ‑: Código-fonte do método MusicalAnalyzer.FindBestChord

O último método desta classe a ser mencionado é o ProcessChordsSequence, que faz a organização da cifra antes de devolvê-la através do evento ProcessingFinished.

A sequência de acordes capturada deverá ter vários acordes repetidos de forma continua, já que ele pode ter sido tocado, sem parar, por mais de meio segundo, ou seja, uma sequencia “G D Em C” poderá ter sido armazenada até aqui como, por exemplo, “G G D D D Em Em Em C”.

Como neste trabalho o ritmo e tempo da música foram ignorados, pois os mesmos não são representados por cifras, basta apenas saber os acordes a cada nova troca. O Quadro 5‑7 apresenta esta implementação.

|  |
| --- |
| private string ProcessChordsSequence()  {  List<Chord> finalChordSequence = new List<Chord>();  StringBuilder s = new StringBuilder();  //Elimina os acordes repetidos continuamente  //Adiciona somente se for diferente do último acorde inserido  foreach (Chord chord in this.chordSequence)  {  if (finalChordSequence.Last() != chord)  {  finalChordSequence.Add(chord);  }  }  //Adiciona todos os acordes a uma única string  foreach (Chord chord in finalChordSequence)  {  s.AppendFormat("{0} ", chord.Name);  }  //Remove o espaço a mais no final da string  return s.Remove(s.Length - 1, 1).ToString();  } |

Quadro ‑: Código-fonte do método MusicalAnalyzer.ProcessChordsSequence

## Songer.Database

Este é o módulo responsável pela persistência e busca dos dados processados pelo sistema. Com a utilização do Db4Objects, o código pode ser bastante resumido. A Figura 5‑7 mostra as classes do mesmo.

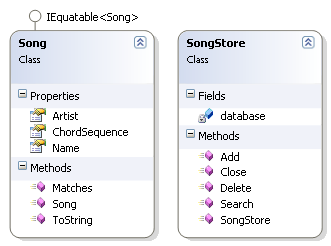


Figura ‑: Diagrama de classes do módulo Songer.Database

### Classe Song

Esta classe armazena as informações de uma música processada, como o nome do artista e o nome da mesma, além da seqüência de acordes que foi encontrada após o seu processamento.

Um ponto importante para se ressaltar nesta classe é o método Matches, que deve receber uma pequena seqüência de acordes e verificar se ele faz parte da música em questão. Com a escolha da plataforma .NET, este trabalho ficou resumido a uma única linha de código, como se pode visualizar no Quadro 5‑8.

|  |
| --- |
| public bool Matches(string chordSequence)  {  return this.ChordSequence.Contains(chordSequence);  } |

Quadro ‑: Código-fonte do método Song.Matches

### Classe SongStore

Esta classe é responsável pelo armazenamento e busca dos dados. Como foi utilizado um banco de dados orientado a objetos, instancias da classe Song podem ser armazenadas diretamente, através dos métodos Add e Delete, que recebe uma delas como parâmetro e realizam a operação requisitada, como se pode ver no Quadro 5‑9.

|  |
| --- |
| private IObjectContainer database;  public SongStore()  {  this.database = Db4oEmbedded.OpenFile("SongStore.yap");  }  public void Add(Song song)  {  this.database.Store(song);  this.database.Commit();  }  public void Delete(Song song)  {  this.database.Delete(song);  this.database.Commit();  } |

Quadro ‑: Código-fonte dos métodos SongStore.Add e SongStore.Delete

Ainda em relação ao Quadro 5‑4, nota-se que o construtor apenas chama uma função para abrir um arquivo de extensão yap. Este arquivo é onde o Db4Objects armazena todos os dados após a execução do método Commit.

O método Search faz uso de predicados (disponível somente a partir da versão 3.0 do .NET Framework), que permite definir critérios e avaliar quais objetos atendem a este critério. Sua implementação é apresentada no Quadro 5‑10.

|  |
| --- |
| public IList<Song> Search(string chordSequence)  {  return this.database.Query<Song>(s => s.Matches(chordSequence));  } |

Quadro ‑: Código-fonte do método SongStore.Search

O método genérico Query necessita que seja passado um tipo, que neste caso foi a classe Song, que será utilizada no predicado e também como tipo de retorno (IList<Song>). O parametro do método é um predicado que informa que, para todo objeto s (que é do tipo Song) encontrado no banco de dados, chame o método Matches (que retorna um tipo booleano) do mesmo, passando chordSequence como parâmetro. Caso o retorno seja verdadeiro, o objeto será adicionado à lista de retorno.

## Songer.Presentation

Este é módulo que possui a tela de apresentação ao usuário. Diferente dos anteriores, este módulo não se preocupa em realizar qualquer tipo de processamento, mas utiliza-se (direta ou indiretamente) de todos os outros módulos para receber e apresentar informações para o usuário. A tela é dividida em duas abas, uma para a busca de informações, e a outra para inserção das mesmas.

### Tela Inserir

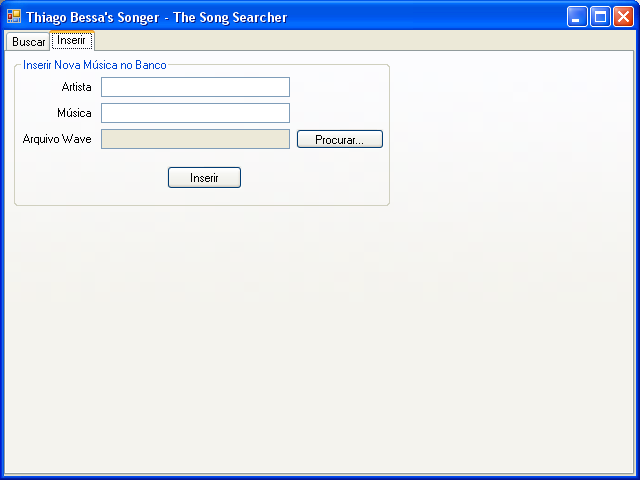


Figura ‑: Tela de inserção de nova música

Esta tela permite ao usuário inserir novas músicas ao banco de dados. Aqui o mesmo deverá informar o nome do artista, o nome da música e o arquivo Wave da mesma. O sistema irá então chamar o método AnalyzeAudio da classe MusicalAnalyzer e, quando terminado o processamento, juntar às outras informações a sequencia de acordes que foi retornada, criar uma instancia da classe Song com estes dados e salvá-lo no banco de dados. Essa tela pode ser vista na Figura 5‑8.

### Tela Buscar

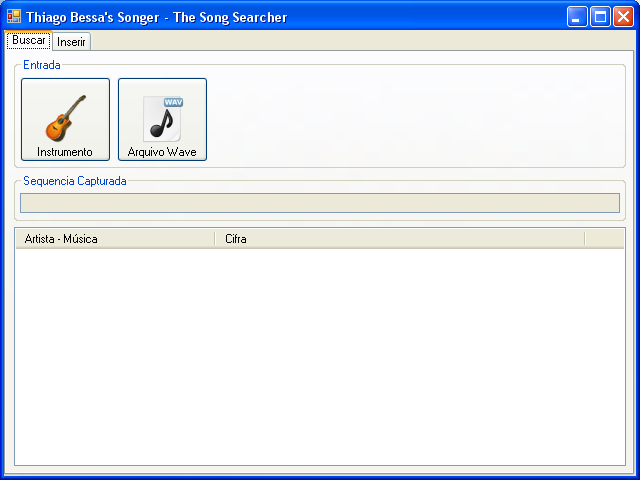


Figura ‑: Tela de busca de música

Como se pode notar na Figura 5‑9, a tela de busca permite ao usuário escolher entre duas formas de entrada: uma de instrumento (através da entrada de linha, ou microfone) e outra por arquivo Wave. Após a entrada do som, o mesmo será processado, e com a seqüência de acordes retornada, será realizada uma busca no banco de dados pelas músicas que possuam pelo menos parte da seqüência igual, sendo as músicas semelhantes apresentadas na lista inferior. Além disso, a seqüência de acordes capturada também é apresentada ao usuário.

# Experimentos

# Conclusão

# Referência Bibliográfica

1. Tradução livre do autor [↑](#footnote-ref-2)
2. Do inglês Application Programming Interface, ou Interface de Programação de Aplicativo, é o conjunto de métodos e regras que um software fornece para que outros softwares possam fazer uso de suas funcionalidades. [↑](#footnote-ref-3)
3. Em .NET Framework, Namespace é uma representação de pacote que pode conter outros namespaces, classes e estruturas. [↑](#footnote-ref-4)
4. Conhecida também como *linha de execução,* é o resultado da divisão de um processo em duas ou mais tarefas que executam concorrentemente. [↑](#footnote-ref-5)
5. Do ingles Language-Integrated Query, ou Consulta integrada à linguagem. Permite que sejam realizadas consultas em coleções de objetos, XML ou até mesmo em banco de dados (LINQ to SQL) utilizando a própria linguagem de programação, mas com uma sintaxe semelhante à SQL. [↑](#footnote-ref-6)