**Sumário**

[1 Introdução 3](#_Toc260251229)

[2 Referencial Teórico 4](#_Toc260251230)

[3 Ferramentas 5](#_Toc260251231)

[3.1 .NET Framework 5](#_Toc260251232)

[3.2 Visual studio 2008 5](#_Toc260251233)

[3.3 DirectX 5](#_Toc260251234)

[3.4 SlimDX 5](#_Toc260251235)

[3.5 NUnit 6](#_Toc260251236)

[3.6 TestMatrix 6](#_Toc260251237)

[3.7 Subversion 6](#_Toc260251238)

[3.8 UML 6](#_Toc260251239)

[4 Metodologia 7](#_Toc260251240)

[4.1 Pesquisa Inicial 7](#_Toc260251241)

[4.2 Especificação das Classes 8](#_Toc260251242)

[4.3 Especificação dos Casos de Teste 8](#_Toc260251243)

[4.4 Implementação das Classes 9](#_Toc260251244)

[4.5 Validação dos Casos de Teste 9](#_Toc260251245)

[4.6 Experimentos 10](#_Toc260251246)

[5 Implementação 11](#_Toc260251247)

[5.1 Songer.UnitTests 12](#_Toc260251248)

[5.1.1 Classe MusicalAnalyzerTests 12](#_Toc260251249)

[5.1.2 Classe NotesAndChordsTests 13](#_Toc260251250)

[5.2 Songer.SoundInput 13](#_Toc260251251)

[5.2.1 Classe SoundSource 14](#_Toc260251252)

[5.2.2 Classe LineInCapture 15](#_Toc260251253)

[5.2.3 Classe WaveFileCapture 16](#_Toc260251254)

[5.2.4 Classe SoundDetectedEventArgs 16](#_Toc260251255)

[5.3 Songer.SoundAnalysis 16](#_Toc260251256)

[5.3.1 Estrutura ComplexNumber 17](#_Toc260251257)

[5.3.2 Classe CooleyTukeyFFT 17](#_Toc260251258)

[5.4 Songer.MusicalInterpreter 18](#_Toc260251259)

[5.4.1 Classe MusicalNote 18](#_Toc260251260)

[5.4.2 Classe MusicalNoteDictionary 19](#_Toc260251261)

[5.4.3 Classe Chord 19](#_Toc260251262)

[5.4.4 Classe ChordDictionary 20](#_Toc260251263)

[5.4.5 Classe MusicalAnalyzer 20](#_Toc260251264)

[5.5 Songer.Database 20](#_Toc260251265)

[5.6 Songer.Presentation 20](#_Toc260251266)

[6 Experimentos 21](#_Toc260251267)

[7 Conclusão 22](#_Toc260251268)

**Lista de Figuras**

[Figura 4‑1: Fluxo da metodologia utilizada 7](#_Toc260403038)

[Figura 4‑2: Exemplo do menu Generate Method Stub que permite adicionar métodos ainda não implementados a uma classe. 8](#_Toc260403039)

[Figura 5‑1: Diagrama dos módulos do projeto. 11](#_Toc260403040)

[Figura 5‑2: Diagrama de classes do módulo Songer.UnitTests 12](#_Toc260403041)

[Figura 5‑3: Exemplo de falha de teste. Na linha de código número 18, ele espera por um retorno de valor 132, mas o valor retornado foi de 0, como informa a mensagem de erro na janela Test Results. 13](#_Toc260403042)

[Figura 5‑4: Diagrama de classes do módulo Songer.SoundInput 14](#_Toc260403043)

[Figura 5‑5: Diagrama de classes do módulo Songer.SoundAnalysis 17](#_Toc260403044)

[Figura 5‑6: Diagrama de classes do módulo Songer.MusicalInterpreter 18](#_Toc260403045)

**Lista de Quadros**

[Quadro 4‑1: Exemplo do método gerado pelo Visual Studio 9](#_Toc260403046)

[Quadro 5‑1: Código-fonte do método Chord.Matches 20](#_Toc260403047)

# Introdução

# Referencial Teórico

# 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. Permite que se desenvolva em quaisquer linguagens que possuam um compilador específico para a plataforma. Para este trabalho foi escolhida a linguagem C#, que é a mais utilizada na plataforma.

## 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.

## DirectX

O DirectX é uma coleção de APIs 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 após a descontinuação do Managed DirectX da Microsoft.

## 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 dos testes.

## Subversion

O Subversion é um sistema de controle de versão, utilizado para manter versões atuais e anteriores de arquivos como código-fonte e documentos. O site [www.xp-dev.com](http://www.xp-dev.com) foi utilizado para fornecer o servidor subversion para este 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.

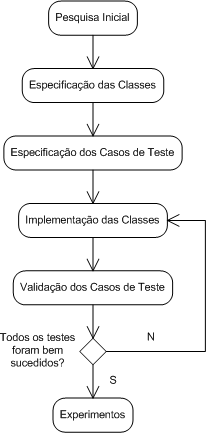


Figura ‑: Fluxo da metodologia utilizada

## Pesquisa Inicial

Estudar 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 e seu resultado no quadro 4.1.

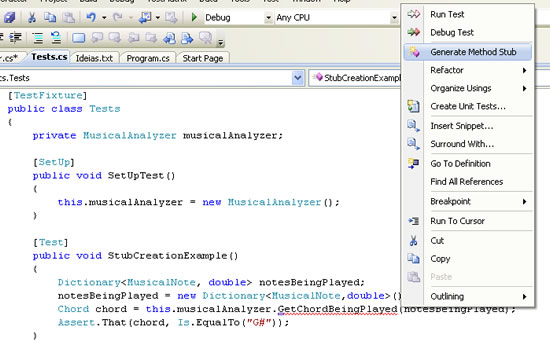


Figura ‑: Exemplo do menu Generate Method Stub que permite adicionar métodos ainda não implementados a uma classe.

|  |
| --- |
| 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 um 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.

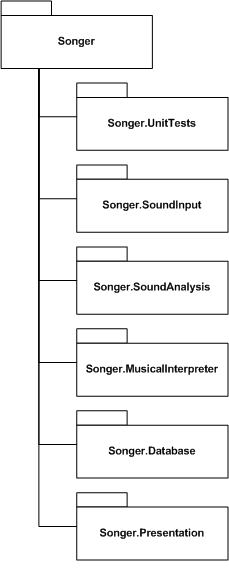


Figura ‑: Diagrama dos módulos do projeto.

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[[1]](#footnote-2).

## 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.

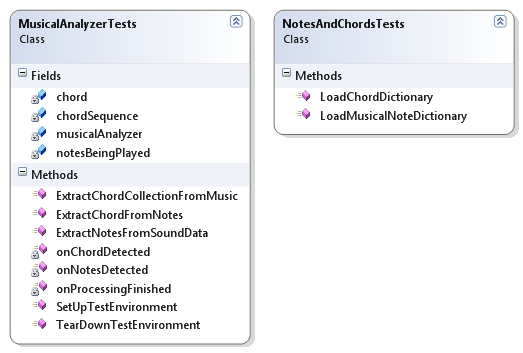


Figura ‑: Diagrama de classes do módulo Songer.UnitTests

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 do conjunto dos três 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.

### 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.

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.

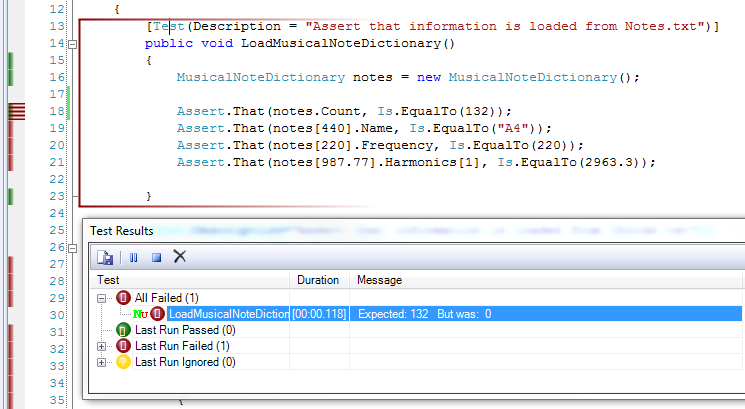


Figura ‑: Exemplo de falha de teste. Na linha de código número 18, ele espera por um retorno de valor 132, mas o valor retornado foi de 0, como informa a mensagem de erro na janela Test Results.

## 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.

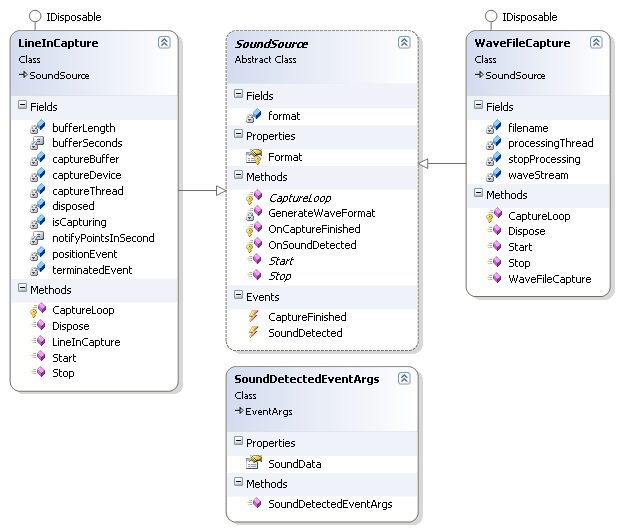


Figura ‑: Diagrama de classes do módulo Songer.SoundInput

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 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 secundário, 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.

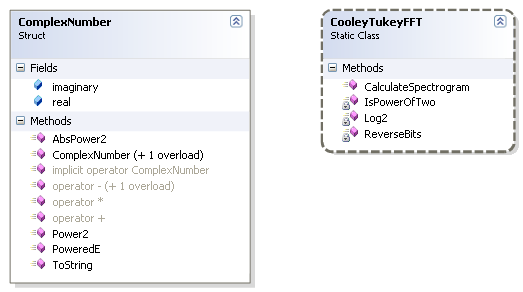


Figura ‑: Diagrama de classes do módulo Songer.SoundAnalysis

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*.

### 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 5.4 apresenta as classes que compõem este módulo.

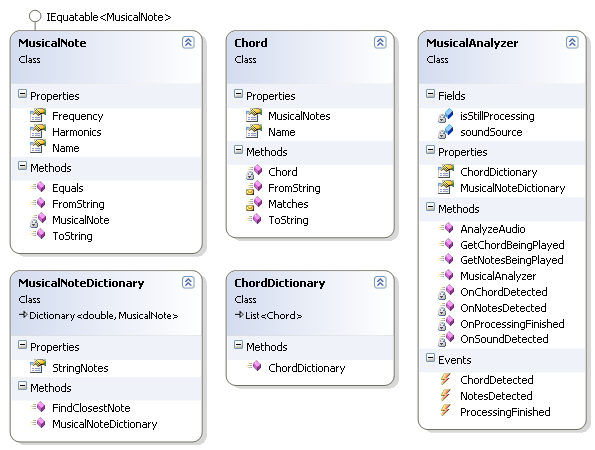


Figura ‑: Diagrama de classes do módulo Songer.MusicalInterpreter

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 necessário 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.

Esta classe também implementa o método Matches, que recebe como parâmetro um conjunto de notas musicas e verifica se o acorde da mesma pode ser formado com estas notas, sendo este método um dos pontos chave do sistema. O quadro 5.1 mostra a sua implementação. TODO – Colocar a imp. Final.

|  |
| --- |
| internal bool Matches(List<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 terá acesso direto a todos os outros módulos do mesmo.

## Songer.Database

Módulo de banco de dados

## Songer.Presentation

Módulo de apresentação

# Experimentos

# Conclusão

1. Em .NET Framework, Namespace é uma representação de pacote que pode conter outros namespaces, classes e estruturas. [↑](#footnote-ref-2)