GREGORY SHOLL E SANTOS

GUILHERME CARLOS POLITTA

TRABALHO SEM TITULO

Trabalho apresentado como requisito parcial à Obtenção de grau de Bacharel em Ciências da Computação no curso de graduação em Bacharelado em Ciências da Computação, Departamento de Informática da Universidade Federal do Paraná.

Orientador: Prof. Bruno Muller Junior

CURITIBA

2014

**SUMÁRIO**

**1 Introdução** 3

**2 Revisão Bibliográfica** 4

2.1 Linguagem SQL 4

2.1.2 Linguagem de definição de dados 4

2.1.3 Linguagem de manipulação de dados 5

2.2 SGBD 5

2.3 *Framework* *Ruby on Rails* 7

2.3.1 Comandos *Rails* 9

2.4 Analisadores sintáticos 12

2.4.1 YACC 12

2.5 Analisadores Léxicos 13

2.5.1 Lex 14

**3 Proposta do Trabalho** 16

**4 Implementação** 18

**5 Conclusão** 25

**1 Introdução**

Empresas com mais tempo no mercado possuem sistemas criados há muitos anos, chamados “sistemas legados”. Entre os diversos problemas com esses sistemas estão à contratação de pessoal qualificado, obsolescência de hardware, a dificuldade de integrar com outros sistemas e escalar para problemas maiores, custo alto de manutenção e documentação incompleta (referencias).

Uma solução para esses problemas é a migração do sistema para uma tecnologia mais nova. Entretanto, esse processo de migração normalmente não é simples (referencia), uma vez que requer fazer o sistema do zero, recriando a lógica, as telas, os fluxos, o banco de dados etc.

Essa migração está vinculada a diversos custos, como o aprendizado de ambas as tecnologias, a original e a que de deseja assumir, o aprimoramento do consumo de espaço e do desempenho e o desafio de manter o banco de dados novo consistente com o atual.

Existem *frameworks* capazes de criar uma aplicação a partir do modelo de banco de dados, como o ASP.NET, o JavaEE, o OpenACS e o Ruby on Rails. Um processo automatizado de migração, de um modelo de dados de uma aplicação legado para um *framework* desse estilo, pode reduzir os custos envolvidos, uma vez que esse processo seria responsável por manter a consistência dos dados entre as aplicações e o *framework* criaria a estrutura básica de telas de controladores.

Este trabalho propõe um *software* que gere a estrutura básica de uma aplicação em *Ruby on Rails*, dado o modelo de dados de uma aplicação já existente. É importante reforçar que a proposta é da geração da estrutura básica da aplicação, ou seja, toda e qualquer lógica envolvida na manipulação e utilização ainda terá de ser implementada em uma segunda fase.

O trabalho está organizado da seguinte maneira: o capítulo 2 descreve os conceitos e ferramentas utilizados, o capítulo 3 descreve o software proposto, o capítulo 4 demonstra uma implementação da proposta e o capítulo 5 apresenta a conclusão.

**2 Revisão Bibliográfica**

Esse capítulo apresenta as ferramentas e conceitos utilizados no trabalho. A Seção 2.1apresenta a linguagem declarativa SQL, à Seção 2.2os Sistemas de Gerenciamento de Banco de Dados (SGBD), a Seção 2.3o *framework* *Ruby on Rails*, a Seção 2.4os analisadores sintáticos e a Seção 2.5os analisadores léxicos.

**2.1 Linguagem SQL**

O SQL (do inglês, *Structured Query Language*) é uma linguagem de programação feita para gerenciar sistema de banco de dados relacionais que foi desenvolvida no começo dos anos 70 por Donald D. Chamberlin e Raymond F. Boyce. Atualmente, SQL é muito utilizado em gerenciamento e desenvolvimento de banco de dados *Web*.

A linguagem é originalmente baseada em álgebra relacional e em cálculo relacional de tuplas. Ela é dividida nas linguagens de definição e de manipulação de dados, definidas a seguir.

**2.1.2 Linguagem de definição de dados**

A linguagem de definição de dados é a que altera as estruturas e restrições dos dados, no caso do SQL essas estruturas são as tabelas do banco de dados. Esse subconjunto na linguagem SQL é denominado de SQL *Schema*. A figura 1 mostra um exemplo de criação de uma tabela Pessoa.

Existem comandos para criar, destruir, permitir e revogar acesso às tabelas (todas as funções consideram que já foram criadas as tabelas previamente). As chamadas para essas funções são CREATE, DROP, GRANT e REVOKE, respectivamente. Neste trabalho, utilizaremos apenas a função CREATE, pois é a única que interessa no processo de criação do sistema novo.

Figura 1 – Exemplo SQL *Schema*

1. **CREATE** **TABLE** Pessoa **(**
2. id **INTEGER** PRIMARY\_KEY**,**
3. nome **VARCHAR(**50**)** **NOT** **NULL,**
4. sobrenome **VARCHAR(**100**)** **NOT** **NULL,**
5. telefone **INTEGER** **NULL**
6. **);**

A função DROP pode, por exemplo, tirar somente a coluna de data de nascimento da tabela, assim como também pode remover a tabela inteira. Já a função GRANT permite acesso aos dados do objeto alterado a um grupo de usuários. A função REVOKE é a que revoga acesso a dados do objeto a certos usuários.

**2.1.3 Linguagem de manipulação de dados**

As linguagens de manipulação de dados são aquelas que lidam com os dados propriamente ditos. Essas linguagens são as que inserem, apagam, atualizam e selecionam os dados desejados.

A manipulação de dados incluem funções como INSERT (inserção), DELETE (remoção) e SELECT (seleção) dos dados dos banco de dados. Essa parte da linguagem não é relevante para o domínio deste trabalho e, portanto, não será detalhada.

**2.2 SGBD**

SGBD vem da sigla Sistema de Gerenciamento de Banco de Dados (em inglês, *Database Management System*) e é um conjunto de programas que gerenciam o acesso e manutenção de um banco de dados. Os SGBDs de banco de dados relacionais são geralmente acompanhados de uma interface simples e intuitiva para o usuário, retirando a necessidado do usuário intervir na funcionalidade do gerenciamento.

Um banco de dados geralmente não é portável para outro SGBD que não seja aquele onde foi criado. Entretanto, SGBDs distintos podem se comunicar através de algum padrão, estabelecido previamente pelos usuários de ambos os sistemas, para fornecerem dados entre si.

Existem três características fundamentais que todos SGBDs devem ter: linguagem de definição de dados, linguagem de manipulação de dados e processamento eficaz de consultas. As relevantes a esse trabalho, linguagem de definição e linguagem de manipulação de dados, já foram definidas na seção 2.1.

Um SGBD precisa ter integridade semântica, ou seja, manter os seus dados sempre correto em relação ao domínio da aplicação. Caso contrário, perde-se o principal propósito de se utilizar um banco de dados. Há outras duas integridades fundamentais para um SGBD, a integridade referencial e a integridade de relação. **(referência)**

A integridade referencial garante que todo valor de uma tabela que referência a um atributo de outra (ou as vezes da mesma) tabela, exista um valor corresponde na tabela referenciada. Essa propriedade é usada muito como chave estrangeira em banco de dados. **(referência)**

A integridade de relação é a propriedade que garante que toda tabela de um banco de dados possua um identificador, sendo que uma ou mais colunas da tabela podem ser usadas para defini-lo. Essas colunas em conjunto devem ser únicas e não nulas, para que possa identificar cada tupla da tabela unicamente.

Os SGBDs também possuem outras características importantes como a segurança no armazenamento dos dados e sua concorrência. Entretanto, estas características não fazem parte do escopo deste trabalho e não serão aprofundadas.

**2.3 *Framework* Ruby on Rails**

*Ruby on Rails*, ou somente *Rails*, é um *framework* gratuito e de código aberto, baseado em Ruby e destinado a aumentar a velocidade de construção de aplicativos *web*, em especial aqueles orientados a banco de dados.

Esse *framework* utiliza diversos padrões e paradigmas de engenharia de *software*, como a convenção sobre configuração (do inglês, *Convention over Configuration*), *Don’t Repeat Yourself* (traduz para Não Se Repita), o padrão *Active Record* e o modelo MVC. Para esse trabalho os mais importantes são o *Active Record* e o MVC.

O padrão *model-view-controller* (também conhecido como MVC) serve para organizar a programação do projeto. Esse modelo é dividido em três componentes, como o nome sugere: modelo, visão e controlador.

O componente modelo é responsável por gerenciar diretamente os dados, lógicas e regras da aplicação. Ele também notifica os outros dois componentes que mudanças foram feitas no seu estado, permitindo mudança na saída da visão e no conjunto de comandos disponível no controlador.

A visão é responsável por gerar alguma forma de representação dos dados armazenados, seja por texto, diagrama, tabelas etc.

O controlador pode enviar comandos para o modelo para que mudanças sejam feitas nos dados, assim como também pode mandar comandos para a visão associadas para mudar a forma de exibição dos dados (por exemplo, rolar por um documento mais extenso).

A figura 2 mostra um diagrama da interação desse componentes entre si e com o usuário. Note que o usuário não sabe de que forma os dados estão armazenados, pois para ele isso não importa, geralmente. Ele visualiza apenas as respostas dos comandos gerados por ele. Os comandos são interpretados pelo controlador que os traduz para um comando que o modelo possa executar. Ao realizar o comando, o modelo adquire um retorno e o passa para a visão atualizar os seus elementos.

****

Figura 2 – Diagrama de relacionamento modelo MVC

O *Active Record* é um padrão de projeto de *softwares* que aborda o acesso ao banco de dados relacionais. Uma tabela desse banco de dados é embrulhada (do inglês, *wrapped*) em uma classe de programação orientada a objetos. Assim, um objeto dessa classe fica ligado a somente uma tupla dessa tabela.

Ao criar um novo objeto, uma nova tupla também é criada na tabela, portanto toda vez que o objeto é acessado ou modificado, o resultado é refletido no banco de dados, mas sem a necessidade de um acesso explícito pelo programador.

A figura 3 mostra um exemplo de um pseudocódigo utilizando o padrão *Active Record* e sua versão equivalente em SQL.

Figura 3 – Exemplo de funcionamento do *Active Record*

1. p **=** Pessoa**()**
2. p**.**nome **=** "João"
3. p**.**sobrenome **=** "Mendonça"
4. p**.**save**()**
5. **INSERT** **INTO** Pessoa **(**nome**,** sobrenome**)** **VALUES** **(**"João"**,** "Mendonça"**);**

A biblioteca *Active Record* do rails implementa ORM (do inglês, mapeamento objeto-relacional), uma técnica de desenvolvimento utilizada para diminuir os conflitos entre a programação orientada a objetos e o banco de dados relacionais. Ela cria um modelo de domínio persistente, onde a lógica e os dados são apresentados como um pacote unificado, fazendo com que a obtenção e escrita de dados possam ser feitas sem a utilização direta de expressões SQL.

Outra característica importante do *Active Record* é associação entre objetos. Eles não se relacionam diretamente no banco de dado, através de chaves estrangeiras, mas sim por associações no modelo que mapeia esse banco de dados. Isso se deve a ideologia do *Active Record* de que toda e qualquer lógica deve ficar no modelo, não no banco de dados, esta servindo apenas como uma base de armazenamento “burra”.

**2.3.1 Comandos *Rails***

O *Rails* possui diversas linhas de comandos para se utilizar no *shell*. Por eles são criadas novas aplicações, modificações de banco de dados e diversos outras operações (referencia). Para esse trabalho, os comandos mais importantes são o *new* e o *generate.*

O *rails new* cria a base de uma nova aplicação. Esse comando gera a estrutura de pastas do projeto, um pacote básico de gemas do *ruby* e alguns arquivos de configurações. A figura 4 mostra a estrutura de arquivos gerada pelo comando.

O *rails generate* utiliza *templates* para facilitar a criação de diversos arquivos ou estruturas, como controladores e visões (referencia). Ele utiliza a estrutura de pastas e as gemas criadas pelo *rails new*, portanto é necessário que o comando já tenha sido executado anteriormente.

****

Figura 4 – Estrutura de arquivos do *rails new*

A maior parte sites orientados a banco de dados precisam de quatro operações básicas (referencia):

* Criar novos registros no banco de dados
* Ler e exibir os registros criados
* Atualizar registros já existentes
* Excluir os registros

Isso é chamado de CRUD, acrônimo em inglês que significa Criar, Ler, Atualizar e Destruir. Como é tão comum a necessidade dessas operações, o *rails generate* possui um *template* específico para isso: o scaffold.

Para utilizar o *scaffold*, devem ser especificados os atributos de um objeto que se deseja criar. É possível também, passar as relações que esse objeto vai ter com outros.O resultado do comando é a criação do *model* para esses atributos, o banco de dados associado ao modelo, o *controller* para manipula-lo, as *views* para o CRUD e um conjunto de teste para cada um dos itens anteriores.

A figura 5 mostra um exemplo da execução de um comando *rails generate scaffold.* Podemos observar que foram criados todos os arquivos referentes aos três componentes do modelo MVC, como foi visto no capítulo 2.3.



Figura 5 – Resultado *rails scaffold*

**2.4 Analisadores sintáticos**

Análise sintática é uma técnica emprega no estudo da estrutura sintática de uma linguagem (citação), conforme as regras de uma gramática formal. Essa análise resulta em uma árvore ou outra estrutura hierárquica que mostra o relacionamento entre cada símbolo reconhecido.

Um analisador sintático tem a função de determinar como a entrada pode ser derivada a partir do símbolo inicial da gramática. Existe duas maneiras de realizar essa tarefa:

1. *Top-down* ou descendente. O analisador começa do símbolo mais alto da hierarquia de análise e tenta transforma-lo na entrada
2. *Bottom-up* ou Ascendente. O analisador começa com a entrada de dados e tenta reescreve-la até chegar ao símbolo inicial mais alto da hierarquia.

O analisador LL (Left-to-right, Leftmost derivation) faz a análise *Top-down* indo da direita pra esquerda, preferindo sempre derivar o símbolo mais à esquerda. Já o analisador LR (Left-to-right, Rightmost derivation) faz a análise *Botton-up*, indo da direita pra esquerda, preferindo derivar o símbolo mais à direita. Ambos são normalmente acompanhados de um número k para indicar que são autorizados a olhar k entradas a frente para evitar *backtracking* ou adivinhação.

**2.4.1 YACC**

YACC (do inglês, *Yet Another Compiler Compiler*) é um gerador de analisadores sintáticos LALR (*Look-Ahead Left-to-Right parser*), desenvolvido no começo dos anos 1970 por Stephen C. Johnson.

Normalmente utilizado para a construção de compiladores, o YACC proporciona uma ferramenta na qual o usuário específica uma estrutura de entrada junto com um código para ser invocado cada vez que uma estrutura é reconhecida.

A figura 6 mostra o YACC sendo utilizado. No painel A coluna temos um trecho de código em Pascal que será utilizado como entrada para o código YACC, no painel B temos a estrutura definida por um usuário com o YACC e no painel C a saída gerada em Java. Como pode ser visto, o código YACC detecta quando uma linha começa com a palavra-chave *var* seguida de um identificador qualquer, dois pontos, um segundo identificador e um ponto e vírgula, gerando a saída em Java “*String x;*”. Em blocos entre colchetes é possível escrever qualquer código em C, portanto poderíamos, por exemplo, fazer a verificação de que se *String* é um tipo suportado para a estrutura.

Figura 6 – Exemplo de funcionamento do YACC

**Painel B**

1. var **:** VAR IDENT **{**
2. **printf(**"%s"**,** token**);**
3. **}** DOIS\_PONTOS IDENT **{**
4. **printf(**"%s"**,** token**);**
5. **}** PONTO\_E\_VIRGULA **{**
6. **printf(**";"**);**
7. **}**

**Painel A**

1. var x**:** String**;**

**Painel C**

1. String x**;**

**2.5 Analisadores Léxicos**

Análise léxica é o processo de converter uma sequência de caracteres em símbolos ou *tokens*, permitindo que seja feita a verificação de que esses caracteres pertencem ao alfabeto de análise. Portanto, um analisador léxico implementa um autômato finito que reconhece símbolos como sendo válidos ou não a uma certa linguagem.

A implementação desses analisadores requer a descrição do autômato que reconhece a gramática ou expressão regular desejada. A sequência de caracteres de entrada é estruturada como uma lista de símbolos, que o analisador vai utilizar como entradas para o autômato.

Se o analisador terminar de consumir os símbolos em um estado final, a entrada é dada como válida, se acabar em um estado não final ou não houver estado para o qual possa ir, a entrada é considerada inválida para a gramática.

**2.5.1 Lex**

Lex é um gerador de analisadores léxicos, escrito por Mike Lesk e Eric Schmidt em 1975. Ajuda a escrever programas cujo fluxo de controle é dirigido por instâncias de expressões regulares, ou seja, as entradas podem ser interpretadas por essas expressões.

Um código Lex pode ser dividido em duas partes: a declaração da expressão regular e a sequência de ações de devem ser executadas quando essa expressão é reconhecida.

A figura 7 mostra um trecho de código que utiliza Lex. Na linha 5 temos uma expressão que reconhece sequências de caracteres que começam com uma letra (maiúscula ou minúscula) seguida de zero ou mais letras e números, em qualquer ordem. Já a linha 9 reconhece quando a expressão previamente declarada é reconhecida e execute um código qualquer, nesse caso só imprime um texto no terminal.

Figura 7 – Exemplo de funcionamento do Lex

1. **%{**
2. #include <stdio.h>
3. **%}**
4. ident **[**a**-**zA**-**Z**][**a**-**zA**-**Z1**-**9\_**]\***
5. **%%**
6. **{**ident**}** **{** printf**(**"encontrei um identificador válido!"**)** **}**

Como o Lex só consegue trabalhar com máquinas de estado finito e o YACC não consegue ler simples entradas de dados, trabalhando apenas com uma série de *tokens,* as duas ferramentas são utilizadas em conjunto, de forma que o Lex serve como um pré-processador para o YACC, gerando os *tokens* que ele necessita. As figura 8 e 9 explicam essa relação. O Lex recebe a entrada de dados e as expressões regulares, gerando a rotina chamada *yylex,* a saída gerada serve como entrada para a rotina *yyparse* criada pelo YACC usando as regras gramaticais. Portanto, cada vez que o YACC precisa de um novo *token* ele invoca o Lex, que processa os dados de entrada e retorna a primeira expressão identificada.

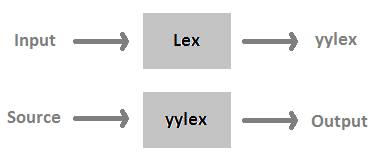


Figura 8 – Entras de saídas do Lex e do YACC

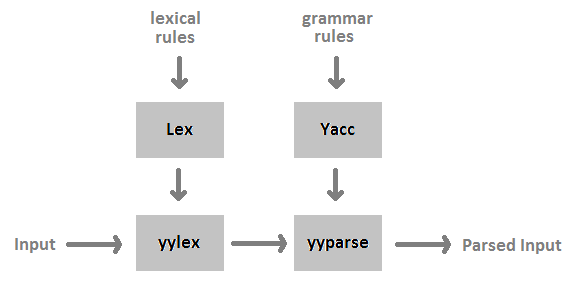


Figura 9 – Relacionamento entre o Lex e o YACC

Esse trabalho tem como objetivo propor uma aplicação que recebe um SQL *schema* e utiliza o YACC e o Lex para traduzi-lo para um conjunto de comandos *Rails,* que juntos constroem o banco de dados e um esqueleto de telas para a entrada.

**3 Proposta do Trabalho**

A implementação proposta nesse trabalho é da construção de uma ferramenta que, a partir de um SQL *Schema* de entrada, gere comandos *rails* que façam a migração de uma aplicação web baseada em banco de dados, para *Ruby on Rails.*

Para isso, utilizamos as ferramentas YACC e Lex, como interpretadores do SQL *Schema*, e a linguagem de programação C, para unir as duas ferramentas e gerar os comandos de saída.

O SQL *Schema* segue uma gramática bem definida (referencia), que é reconhecida pelo YACC. Como visto na seção 2.4.1, ele permite gerar comandos à medida que vai fazendo a análise sintática. Isto normalmente é utilizado em compiladores, para fazer a análise sintática e geração de código.

No presente trabalho, o YACC foi utilizado para gerar comandos rails à medida que faz a análise sintática do SQL *Schema*. Especificamente, ele reconhece os vários tipos de tabelas que podem estar incluídos no SQL *Schema* e gera os comandos que criam os modelos *rails*, assim como os *views* e *controllers* correspondentes, da nova aplicação.

Já o Lex, como foi visto na seção 2.5.1, faz a análise semântica da entrada. No nosso trabalho ele certifica que todos os símbolos lidos são válidos para o SQL *Schema* e gera os *tokens* com os quais o YACC trabalha.

A figura X mostra um exemplo da entrada da nossa ferramenta. Essa é composta de uma ou mais tabelas descritas em SQL *Schema* e separadas por uma linha em branco. A saída esperada para dita entrada, exemplificada na figura X, são comandos *rails* que descrevem o mesmo modelo passado como entrada, mantendo as mesmas relações entre as tabelas.

1. **CREATE** **TABLE** customers **(**
2. customer\_id **INT** AUTO\_INCREMENT **PRIMARY** **KEY,**
3. customer\_name **VARCHAR(**100**)**
4. **);**
5. **CREATE** **TABLE** orders **(**
6. order\_id **INT** AUTO\_INCREMENT **PRIMARY** **KEY,**
7. customer\_id **INT,**
8. amount **DOUBLE,**
9. **FOREIGN** **KEY** **(**customer\_id**)** **REFERENCES** customers**(**customer\_id**)**
10. **);**
11. rails generate scaffold customers customer\_name**:**string --force
12. rails generate scaffold orders customers**:references** amount**:decimal{**42.6**}** –force

No exemplo, os campos com que possuem atributos *INT AUTO\_INCREMENT PRIMARY KEY,* ou seja, inteiros de incremento automático e que são chaves primárias, não são explicitamente mapeados para o rails, uma vez que este gera um campo com essas características automaticamente para cada tabela.

Já o campo *costumer­­­\_name*, que é originalmente do tipo *VARCHAR,* foi mapeado para rails como *string*. O campo *amount*, primeiramente do tipo *DOUBLE*, foi mapeado para *decimal{42.6},* ou seja, um decimal com tamanho de 46 caracteres e 6 caracteres de precisão. A lista com todos os atributos suportados e seus mapeamentos para o rails está no apêndice X.

**4 Implementação**

A implementação da ferramenta proposta foi dividida em três partes: a leitura do arquivo do SQL *Schema*, a intepretação desse arquivo e a geração dos comandos *Rails.*

A primeira foi feita com o Lex, onde criamos uma séries de expressões regulares que definem todos os *tokens* possíveis para o SQL *Schema*, portanto ele não só lê a entrada de dados, mas também verifica se os símbolos lidos são válidos. A lista de expressões regulares, assim como os símbolos que elas definem, estão disponíveis no apêndice 1.

CRIAR UMA TABELA COM AS COLUNAS: EXPRESSÃO REGULAR NO LEX E SÍMBOLOS RECONHECIDOS PELO COMANDO COMO APENDICE

Já a interpretação dos símbolos lidos foi feita com o YACC. Com ele criamos uma estrutura que reconhece as sequências possíveis de símbolos e, com isso, verificando se a entrada está de acordo com o padrões do SQL *Schema*, reconhecendo os identificadores das tabelas, como nome da tabela e de atributos, e também a relação de atributos da tabela com outras tabelas.

Com essas informações, o terceiro passo foi criar registros em memória que guardam as relações entre tabelas, seus nomes, os atributos que compõem cada tabela, os tipos de cada um desses atributos e quais outras características eles têm, por exemplo, se é chave primária ou estrangeira. Ao final é feita uma avaliação de todas as tabelas e, enfim, é gerado os comandos *Rails* adequados.

Para testar a ferramenta desenvolvida, utilizamos o projeto Paje (referencia) como caso de teste. O Pajé é um sistema de gestão hospitalar, feito em JSenna (referência jsenna), com licença de software livre.

Extraímos algumas tabelas desse sistema para teste. O SQL *Schema* da criação dessas tabelas podem ser visto na figura X. Temos três tabelas: paciente, profissional e agenda. As tabelas paciente e profissional são referenciadas pela tabela agenda, em uma relação 1-N, ou seja, cada paciente e profissional podem ter várias agendas, mas cada agenda só possui um paciente e um profissional.

1. **CREATE** **TABLE** paciente **(**
2. id\_paciente **INT** AUTO\_INCREMENT **PRIMARY** **KEY,**
3. nome\_paciente **VARCHAR(**255**),**
4. descricao\_cama **VARCHAR(**255**)**
5. **);**
6. **CREATE** **TABLE** profissional **(**
7. id\_profissional **INT** AUTO\_INCREMENT **PRIMARY** **KEY,**
8. nome\_usuaraio **VARCHAR(**255**),**
9. cronograma\_horas **VARCHAR(**255**)**
10. **);**
11. **CREATE** **TABLE** agenda **(**
12. id **INT** AUTO\_INCREMENT **PRIMARY** **KEY,**
13. id\_paciente **INT,**
14. id\_profissional **INT,**
15. protocolo\_registro\_hora\_entrada **VARCHAR(**255**),**
16. **FOREIGN** **KEY** **(**id\_paciente**)** **REFERENCES** paciente**(**id\_paciente**),**
17. **FOREIGN** **KEY** **(**id\_profissional**)** **REFERENCES** profissional**(**id\_profissional**)**
18. **);**

Após a extração, chamamos a nossa ferramenta por linha de comando passando essas tabelas como parâmetro, como pode ser visto na figura X.



O resultado obtido é exibido na figura X. Podemos observar quatro pontos importantes:

* Os campos de *ID* não são explicitamente declarados no comando *rails* de saída. A ferramenta detecta que o campo é chave primária, inteiro e com incremento automático e omite no comando final gerado, já que o *rails* cria automaticamente um campo com essas categorias para cada tabela.
* Os campos que referenciam outras tabelas não possuem um tipo de dados declarado. Isso é consequência do primeiro ponto, como uma chave primaria do tipo inteiro e de incremento automática é sempre criado para cada tabela, não é necessário identificar qual vai ser tipo do campo que referencia essa tabela, vai ser sempre inteiro.
* O campo que referencia outras tabelas tem o seu nome alterado. O campo vai ter sempre nome da tabela a qual referência.
* O comando da linha 4 serve para criar efetivamente o banco de dados, já que cada comando *rails generate* cria uma migração que pode ainda pode ser modificada. Com o *rake db:migrate* todos as migrações criadas são aplicadas simultaneamente à nova aplicação.

1. rails generate scaffold paciente nome\_paciente:string descricao\_cama:string --force
2. rails generate scaffold profissional nome\_usuaraio:string cronograma\_horas:string --force
3. rails generate scaffold agenda paciente:references profissional:references protocolo\_registro\_hora\_entrada:string --force
4. rake db:migrate

Os comandos agora foram criados, mas antes de podermos executa-los, precisamos criar um novo ambiente de aplicação rails. Para isso, é necessário criar um pasta e *rails new <nome da aplicação>*. Esse comando, como visto na seção 2.3.1, gera a estrutura básica de pastas do projeto, os arquivos inicias de configurações e as gemas padrões.

Com o ambiente criado, podemos agora executar os comandos gerados pela ferramenta. A figura X mostra o arquivo de migração criado pelos comandos para a tabela agenda, onde cada parâmetro foi mapeado para o *Active Record* de acordo com o especificado no *rails generate*. O campo *timestamp,* na linha 8, é criado automaticamente para todas as tabelas.

É importante ressaltar que não são criadas chaves estrangeiras no banco de dados, ao invés são criadas associações entre os objetos no modelo, como visto na seção 2.3. A figura X mostra as associações no modelo da agenda.

1. **class** **CreateAgendas** **<** ActiveRecord**::**Migration
2. **def** **change**
3. create\_table :agendas **do** **|**t**|**
4. t**.**references :paciente**,** index: **true**
5. t**.**references :profissional**,** index: **true**
6. t**.**string :protocolo\_registro\_hora\_entrada
7. t**.**timestamps
8. **end**
9. **end**
10. **end**
11. **class** **Agenda** **<** ActiveRecord**::**Base
12. belongs\_to :paciente
13. belongs\_to :profissional
14. **end**

Como visto na seção 2.3.1, o *rails* *generate scaffold* cria uma série de outros arquivos para as operações básicas sobre os dados, chamadas CRUD. Esses arquivos podem ser encontrados na pasta *app/views*. A lista de arquivos gerados para o nosso teste podem ser vistos na figura X. 

Para visualizarmos as telas criadas pelos comandos, executamos o comando *rails server* para iniciar a aplicação web, depois acessamos por um navegador. Na figura X1, podemos ver a tela de criação de um paciente. A figura X2 mostra a lista de agendas e a X3 a exclusão de um profissional. 





**5 Conclusão**