UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

GABRIEL MAIER FERNANDES VIDUEIRO
PEREIRA

RailsOnDia: Integrando aplicações *Rails* com ferramentas de modelagem existentes

Trabalho de Graduação.

Prof. Dr. Marcelo Pimenta
Orientador

Porto Alegre, julho de 2011

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos Alexandre Netto
Vice-Reitor: Prof. Rui Vicente Oppermann
Pró-Reitora de Graduação: Profa. Valquiria Link Bassani
Diretor do Instituto de Informática: Prof. Flávio Rech Wagner
Coordenador do CIC: Prof. João César Netto
Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

AGRADECIMENTOS

Agradeço a dedicação do Prof. Dr. Marcelo Pimenta, pelos conhecimentos passados durante e ao final do meu período junto à faculdade.

Agradeço todo o suporte dado pelos meus colegas e amigos de curso Leonardo Borba e Vanius Zapalowski. Sem a ajuda e apoio de vocês a jornada da faculdade se tornaria penosa.

Agradeço a compreensão de minha companheira Juliana, pela correção e leitura do trabalho que tanto me tornou ausente nessas últimas semanas. Tenha a certeza, de que você me deu forças para alcançar meus objetivos e construir uma história ao seu lado.

Agradeço especialmente a minha mãe, Sandra Fernandes, que tornou o sonho da faculdade de ciência da computação possível mesmo frente a tantas dificuldades. Se cheguei até aqui, foi por teu esforço. Obrigado.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

RoR	Ruby on Rails
MVC	Model-View-Controller
ER	Entidade relacionamento
XML	Extensible Markup Language
XMI	XML Interchange
CRUD	Create, read, update and delete
PNG	Portable Network Graphics

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

As aplicações escritas em Rails estão se tornando mais populares em função da simplicidade da linguagem Ruby e das funcionalidades *easy-to-use* do framework Ruby on Rails. No entanto, o hábito de modelar as aplicações antes de criá-las ainda é pouco comum na comunidade de usuários. Com o propósito de mudar essa situação, esse trabalho propõe uma solução de integração entre diagramas e aplicações escritas em Rails, chamada de RailsOnDia. A integração consiste na criação de uma biblioteca capaz de modificar a estrutura de uma aplicação. Essa modificação é realizada em duas etapas. Primeiramente, é necessária a existência de um arquivo capaz de entender e extrair informações sobre a estrutura do diagrama modelado. Com esse arquivo em mãos, caberá ao programador utilizar as informações e modificar a estrutura da aplicação utilizando a funcionalidade do RailsOnDia. RailsOnDia não restringe o modelo de diagrama a ser usado, apenas necessita de uma versão em XML do mesmo para que sejam retiradas as informações necessárias. Portanto, este trabalho propõe uma estrutura XML que possibilita o teste e a extração de resultados da utilização do RailsOnDia.

Palavras-chave: Modelagem, Ruby on Rails, engenharia WEB.

ABSTRACT

Applications written in Rails are becoming more popular due to the simplicity of the Ruby language and the easy-to-use features of the Ruby on Rails framework. However, the habit of modeling applications before creating them remains unusual in the users' community. In order to change this situation, this paper proposes an integration solution between models and applications written in Rails, called RailsOnDia. The integration consists in the creation of a library that is able to handle the structure of an application. This modification is carried out in two steps. First, it is necessary to have a file that is capable of understanding and extracting information about the structure of the modeled diagram. With this file, the programmer might use this information and modify the structure of the application by using the RailsOnDia features. RailsOnDia does not narrow the diagram model to be used, it only needs an XML version of it to retrieve the necessary information. Therefore, this paper proposes an XML structure that allows for testing and extraction of results from the proposed library.

Keywords: Modelling, Ruby on Rails, WEB engineering.

1 INTRODUÇÃO

Com o constante crescimento da Internet, vem aumentando muito o número de aplicações para a Web, e, com isso, começam a surgir diferentes abordagens para a criação e manutenção dessas aplicações. Uma das soluções que está alcançando respeito nesse ambiente é o framework Ruby on Rails, que visa melhorar o processo de desenvolvimento de software, criando um ambiente ágil com qualidade certificada por baterias de testes automatizados (RUBY; THOMAS; HANSSON, 2011).

O Rails trabalha sobre uma arquitetura MVC (Model-View-Controller) de software. Este trabalho irá propor uma diferente maneira de manipulação do *Model*, que é responsável por controlar o comportamento e os dados do sistema (BURBECK, 1992). Toda a modificação de estrutura dos modelos de uma aplicação é realizada através de uma convenção interna do Rails, chamada Migration. Essa convenção permite o controle de alterações estruturais da aplicação, auxiliando no desenvolvimento de aplicações em ambientes colaborativos. No entanto, atualmente os Migrations são gerenciados através da criação de arquivos com código Ruby que são posteriormente interpretados, o que muitas vezes torna-se pouco prático.

Este trabalho propõe a criação de uma biblioteca que possibilite a integração entre aplicações Rails e ferramentas de modelagem existentes, tornando possível a modificação estrutural de uma aplicação *Rails* através de uma interface de diagramas. Para tal integração, será utilizado o padrão XML como meio de comunicação, por ser amplamente difundido e bem aceito pela maioria dos softwares (BRAY; PAOLI; SPERBERG, 1998). Não está no âmbito desse trabalho a criação de uma ferramenta de modelagem que interaja diretamente com a aplicação Rails, e sim a criação de uma biblioteca que seja capaz de se comunicar, através de XML, com diversas ferramentas de modelagem já existentes, permitindo assim a utilização de diferentes notações gráficas para a modelagem do sistema.

1.1 Contexto Histórico

Ainda não é comum a modelagem de sistemas quando eles possuem como alvo a Web. Esse se torna um grande problema, visto que a engenharia Web possui diversas metodologias que facilitam a compreensão e manutenção de sistemas. Dentre as metodologias existentes, uma das mais comuns é a escolha de uma arquitetura em camadas para a criação de sistemas (KAPPEL, 2006). Uma famosa estrutura em camadas é o MVC, no qual existe uma divisão entre os *Models*, que modelam os dados e a lógica do sistema, a View que representa a camada de apresentação do sistema e o Controller, que é o responsável pela comunicação das outras duas camadas (BURBECK, 1992).

O framework Ruby on Rails utiliza como base uma estrutura MVC, mas não apresenta

intrinsicamente métodos de modelagem de seus *Models* (RUBY; THOMAS; HANSSON, 2011). Sendo assim, cabe ao responsável pelo sistema a elaboração de diagramas que representem a lógica do sistema e seu comportamento frente a interação do usuário final.

A filosofia ágil proposta pelos usuários e criadores do framework não descarta a utilização de modelos, mas sim a utilização da modelagem quando ela clarificar pontos críticos no fluxo de desenvolvimento de software. Os modelos devem ser simples o suficiente para a compreensão do problema por parte dos envolvidos, visando garantir a qualidade e agilidade do software (PRESSMAN; LOWE, 2008).

Isto posto, fica evidente a importância da modelagem para sistemas Web e a atual deficiência do framework Ruby on Rails frente à necessidade de uma ferramenta de modelagem ágil e integrada ao resto do framework.

1.2 Objetivo

A meta deste trabalho é a criação de um ambiente de desenvolvimento ainda mais ágil para o framework, que permita ao desenvolvedor criar e manter aplicações Rails utilizando modelagem diagramática, o que é uma postura recomendada pela engenharia WEB para a manutenção de sistemas de médio e grande porte (KAPPEL, 2006). Em vista de não limitar os tipos de modelos utilizados, será criada uma biblioteca com operações genéricas que manipulam a estrutura da aplicação.

Essa biblioteca permitirá que as aplicações Rails sejam manipuladas por diferentes modelos através da criação de arquivos que interpretem esses modelos e modifiquem a estrutura da aplicação conforme necessário. Desta forma, torna-se possível a utilização da biblioteca junto à modelos bastante conhecidos, como diagramas de entidade-relacionamento e diagramas de classes.

1.3 Estrutura do Documento

O capítulo 2 introduz o problema de modelagem em aplicações desenvolvidas no framework Rails. No capítulo 3 são abordadas as soluções existentes semelhantes ao proposto pelo trabalho. No capítulo 4 é apresentada a biblioteca RailsOnDia. Na seção 4.1 é apresentada uma explicação da estrutura de uma aplicação desenvolvida em Rails. A seção 4.2 apresenta a linguagem de marcação utilizada como meio de comunicação entre aplicações e modelos. Na seção 4.3 estão listadas e explicadas duas abordagens utilizadas na criação do trabalho. A seção 4.4 explica o funcionamento do RailsOnDia. O capítulo 5 encerra o trabalho com as conclusões e perspectivas futuras.

2 INTEGRANDO MODELAGEM A APLICAÇÕES RAILS: CONCEITOS E FUNDAMENTOS

O framework Ruby on Rails não provê uma ferramenta de modelagem nativa ao desenvolvedor. No entanto, provê um recurso que possibilita o controle das modificações nos modelos da aplicação. Esse recurso é chamado de Migration, que nada mais é do que um arquivo de código em Ruby com uma sintaxe específica que proporciona informações suficientes para o Rails modificar a estrutura da aplicação. O arquivo é apenas informativo, pois quem de fato realiza a alteração estrutural é uma ferramenta chamada Rake. Essa ferramenta é nativa do framework e possui funcionamento semelhante ao comando Make do Linux. Seu nome, inclusive, deriva de Ruby Make.

Por exemplo, para a criação de um modelo de produtos que possua os atributos nome, descrição e datas de criação e edição dos produtos, é criado um arquivo como o da figura 2.1. Neste arquivo são definidas duas funções: *self.up* e *self.down*. A primeira é executada quando o interesse do usuário é em efetivar a alteração proposta pelo Migration na estrutura da aplicação. A segunda é executada quando o usuário decide desfazer a modificação, ou seja, todo arquivo Migration possui um método que realiza a modificação e um que a desfaz. Esse arquivo é composto por uma classe em Ruby, rotulada com o nome da migração, que estende a classe *ActiveRecord::Migration*. O *ActiveRecord* é a classe do framework que controla os modelos do sistema, provendo mecanismos de independência de banco de dados, funcionalidades básicas de CRUD (Create, Read e Update), capacidades avançadas de pesquisa e a habilidade de relacionar modelos entre si. Essa classe deve ser estendida de modo a possibilitar a utilização dos métodos que ela dispõe para a manipulação de modelos e seus atributos (MARSHALL; PYTEL; YUREK, 2007).

```
class CreateProducts < ActiveRecord::Migration
  def self.up
    create_table :products do |t|
      t.string :name
      t.text :description

      t.timestamps
    end
  end

  def self.down
    drop_table :products
  end
end
```

Figura 2.1: Estrutura de um arquivo Migration

De modo a facilitar as pequenas e corriqueiras alterações que ocorrem durante o ciclo de desenvolvimento de software, como adicionar ou remover atributos de um modelo, o Rails possui um gerador de Migrations que é utilizado através de linha de comando. Esse gerador permite a criação de arquivos de migração padrões, nos quais o nome da migração informa a operação que se deseja realizar. Por exemplo, se o nome da migração for *AddNameToPerson*, o arquivo criado irá possuir as informações necessárias para adicionar o atributo *Name* no modelo *Person*. O tipo do atributo é passado ao gerador como parâmetro na linha de comando. Esse gerador, apesar de muito útil, não abrange os casos mais complexos, nos quais várias operações necessitam ser realizadas, cabendo ao usuário criar arquivos de Migrations manualmente (MARSHALL; PYTEL; YUREK, 2007).

Os relacionamentos entre os modelos de uma aplicação em Rails são elencados nos arquivos de classes de cada *Model*. Existem quatro métodos para implementar as associações entre os modelos da aplicação. São eles: *has_one*, *has_many*, *belongs_to* e *has_and_belongs_to_many*. Com esses métodos é possível criar associações do tipo *um pra muitos*, *muitos pra um* e *muitos pra muitos*. Esses relacionamentos são criados apenas logicamente ao ser carregada a estrutura do framework, ou seja, não são criadas estruturas de persistência para o armazenamento dessas informações de relacionamento. Por isso, para obter a persistência da informação de relacionamento torna-se necessária a criação de um atributo que realize a gravação desta informação.

O framework Rails disponibiliza uma estrutura de *Models* junto a um conjunto de métodos que implementam as associações entre eles, porém, não apresenta uma maneira fácil e prática de manipulá-los. Inclusive os relacionamentos são tratados separadamente da estrutura dos modelos, visto que não é possível a definição de um relacionamento em um arquivo de Migrations. Desta forma, o trabalho do desenvolvedor que está criando um novo conjunto de *models* para a aplicação é dividido em duas partes: a criação dos *modelos* e, posteriormente, a criação das associações entre eles (AKITA, 2006).

O framework Rails disponibiliza um conjunto de ferramentas bastante robusto para o controle de modificações na estrutura da aplicação, como os arquivos *Migrations* e a ferramenta Rake. No entanto, não possui um modo gráfico de tratar essa estrutura, cabendo ao desenvolvedor a realização de diversos passos para a obtenção de modificações estruturais na aplicação, que poderiam ser resolvidos mais intuitivamente através da manipulação visual de um diagrama.

A modelagem diagramática - ainda mais hoje em dia com a efervescência em torno da linguagem UML, de abordagens guiadas por modelos (MDD, MDE, MDA, etc) - é um tópico com muita bibliografia e material. O leitor interessado pode buscar mais detalhes em (SOMMERVILLE, 2007; PRESSMAN, 2010; FOWLER, 2004).

3 TRABALHOS RELACIONADOS

Em 2004 o criador do Rails tornou-o open-source. Com isso, pessoas começaram a utilizar e criar ferramentas para melhorar ou complementar o framework. Esse conjunto de soluções criadas e a comunidade formada pelos usuários é chamado de ecossistema Rails. Dentro desse ecossistema podem ser encontradas algumas soluções para a visualização da aplicação de uma forma mais gráfica do que a textual padrão. No entanto, não existe ainda uma boa solução para a edição dos modelos de uma aplicação, a não ser por meio de manipulação de arquivos com sintaxe Ruby.

Nesta seção serão listadas algumas das soluções que mais aproximam-se das funcionalidades propostas pelo RailsOnDia, e, ao final, serão analisadas e comparadas de acordo com as suas funcionalidades.

3.1 SQL Editor

A alternativa que mais se aproxima do propósito desse trabalho se encontra em um software proprietário chamado SQL Editor, comercializado pela MalcolmHardie. Nele é possível a importação e exportação das estruturas de uma aplicação Rails. No processo de importação o software realiza uma leitura nos códigos fonte da aplicação a procura de informações para desenhar o diagrama, exemplificado na figura 3.1. Já no processo de exportação, o software exporta as modificações realizadas no formato de arquivos Migrations.

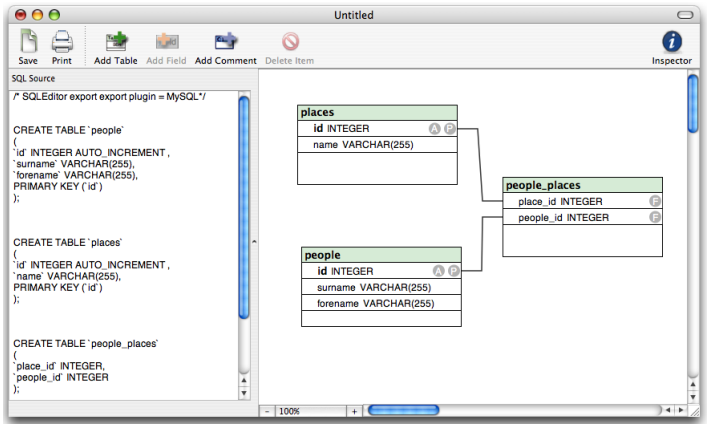


Figura 3.1: Interface do software SQL Editor

Este software trabalha sobre uma estrutura relacional de bancos de dados, ou seja,

com diagramas ER. Em função dos bancos de dados relacionais possuírem um paradigma diferente do orientado a objetos, essa abordagem acaba resultando em um conjunto de problemas comumente conhecidos como incompatibilidade de impedância objeto relacional (AMBLER, 2003), nos quais o banco de dados relacional não consegue expressar fielmente o que a aplicação orientada a objetos deseja. Logo, essa é uma solução que possibilita tratar apenas com um tipo de modelagem, tornando-se um limitante para aqueles que preferem uma abordagem diferente da existente.

3.2 Rubymine

Rubymine é uma plataforma de desenvolvimento totalmente projetada para trabalhar com aplicações desenvolvidas em Rails. Uma de suas funcionalidades é a visualização dos modelos e relacionamentos da aplicação em forma de diagramas, chamada de *Model Dependency Diagram*. Esse diagrama sinaliza ao programador possíveis erros na estrutura, como a relação que não possui cardinalidade na figura 3.2, que é exibida com um sinal de interrogação avermelhado. No entanto, essa ferramenta não permite a edição de dados e relacionamentos através do modo gráfico, funcionando apenas como uma ferramenta de leitura e compreensão da aplicação.

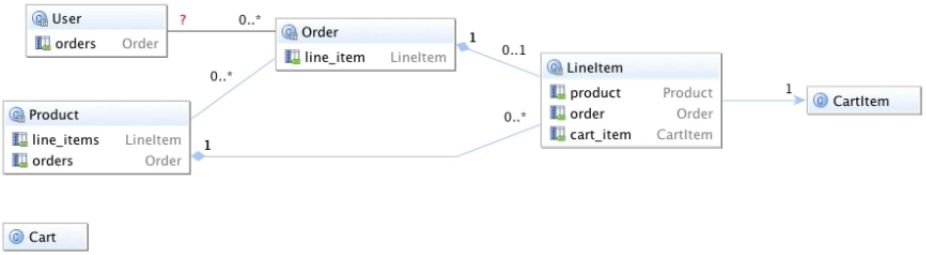


Figura 3.2: Exemplo de *Model Dependency Diagram*

3.3 Railroad

O RailRoad é uma solução open source, criada por Javier Smaldone, para a criação de diagramas a partir de aplicações Rails. Essa ferramenta carrega os dados dos modelos da aplicação e monta um diagrama na linguagem DOT, que é uma linguagem textual utilizada para representar diagramas (KOUTSOFIOS; NORTH, 2002). A leitura de um arquivo DOT, apesar de compreensível para um humano, é pouco prática, devido ao grande número de palavras e expressões utilizadas para representar um simples relacionamento entre dois *models* do sistema. Por isso o autor aconselha a utilização de algum software que traduza de DOT para um arquivo de imagem, como o PNG.

Por apresentar etapas de conversões de formato, o RailRoad não é uma solução prática para a visualização do diagrama de modelos de uma aplicação. Esta solução também não possibilita a edição de atributos e dados dos modelos, se tornando apenas uma ferramenta de visualização da estrutura, como pode ser visto na figura 3.3. Por criar diversos estilos diferentes de diagramas e possibilitar uma série de configurações para os diagramas, esta é uma ferramenta bastante utilizada para documentação de sistemas.

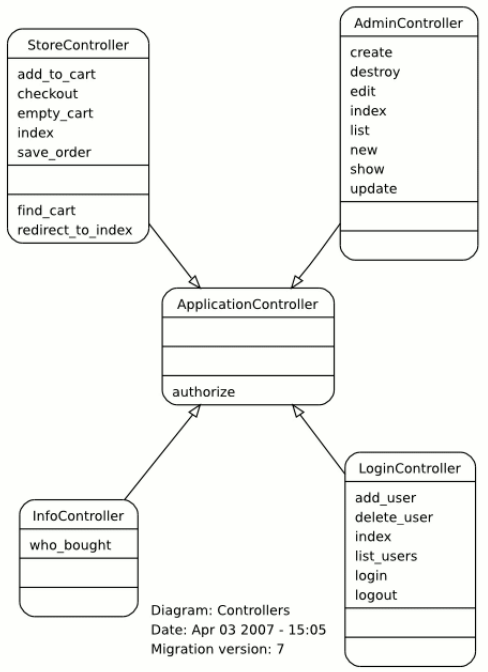


Figura 3.3: Diagrama gerado pelo *RailRoad*

3.4 Quadro comparativo

Como visto, as soluções existentes para visualização diagramática da estrutura de uma aplicação são pouco funcionais para a finalidade de modelagem. Apesar de algumas proverem ferramentas interessantes para a visualização da estrutura, o propósito de cada um dos aplicativos é diferente do proposto por este trabalho, visto que são ferramentas muito específicas que não se aplicam para a modelagem de sistemas. A tabela 3.1 elucida as diferenças entre os trabalhos existentes citados.

Tabela 3.1: Comparativo entre os trabalhos existentes.

Ferramenta	**Modifica a estrutura**	**Permite diferentes modelos conceituais**
SQL Editor	Sim	Não
Rubymine	Não	Não
RailRoad	Não	Não

4 RAILSONDIA: UMA BIBLIOTECA PARA INTEGRACAO DE FERRAMENTAS DE MODELAGEM A APLICACOES RAILS

Este trabalho tem como objetivo a criação de uma biblioteca que possibilite a criação e manutenção da estrutura de *models* de uma aplicação através de uma ferramenta gráfica genérica, desde que tal ferramenta possua a funcionalidade de exportar para XML seus diagramas. A biblioteca irá manipular aplicações desenvolvidas em Rails, utilizando principalmente as funcionalidades da classe de persistência de dados do framework e a capacidade do framework de manipular sua estrutura através de arquivos com código Ruby, chamados de Migrations.

4.1 Contexto Rails

O framework Ruby on Rails é escrito sobre a linguagem interpretada Ruby, utilizando o paradigma de orientação a objetos. Ele é desenvolvido sobre a arquitetura MVC, que é amplamente utilizada em desenvolvimento Web por manter separada a camada de visualização das camadas de lógica e regras de negócios da aplicação. Essa separação se torna atraente para a estrutura de uma aplicação baseada na Web, pois essas aplicações costumam ser desenvolvidas para diferentes navegadores de Internet que possuem diferentes padrões de exibição e estilização de suas páginas. Com isso, a separação da camada de visão permite que o problema de incompatibilidade dos navegadores seja tratado em uma camada apenas, e não na aplicação como um todo (BURBECK, 1992).

O Rails possui dois princípios básicos de existência. O DRY (Don't repeat yourself) e o "Convenção sobre configuração". O DRY estimula a diminuição do retrabalho visando a melhora de produtividade, ou seja, se uma característica da aplicação já foi declarada uma vez, ela não deve ser redeclarada em outro local. O framework deve ser inteligente o suficiente para buscar o local da primeira declaração. O princípio da 'Convenção sobre configuração' reflete que o framework não deve ser configurável para cada tipo de aplicação que ele irá manter, mas sim que cada aplicação deve tomar certas convenções para que ela funcione corretamente sobre o ambiente Rails. Essas convenções são basicamente os locais onde cada tipo de arquivo será armazenado dentro da estrutura do framework e a nomenclatura desses arquivos e classes. Com essas simples convenções o Rails consegue carregar e manipular as estruturas das quais necessita ao executar uma aplicação (AKITA, 2006).

Para este trabalho, dois locais na estrutura do framework merecem especial atenção. A primeira é a pasta dos modelos. Ela é armazenada dentro da pasta de códigos da aplicação e é mantida em separado das camadas de visão e controle. Cada modelo da aplicação é identificado por um arquivo com o nome do modelo no singular e a extensão padrão

para arquivos Ruby, como por exemplo task.rb. Nesse arquivo estará definida uma classe estendida da biblioteca padrão para o controle do modelos em Rails, a classe itActiveRecord::Base. Nesse arquivo serão definidas os relacionamentos que o modelo terá com outros modelos, bem como as funções que ele deverá executar em determinados momentos da execução.

Outra pasta de suma importância para este trabalho é a pasta que armazena os arquivos do tipo Migration. É nessa pasta que serão armazenados os Migrations criados e posteriormente executados por comandos internos do Rails. Cabe citar que o armazenamento desses arquivos nessas pastas é de vital importância para o correto funcionamento do framework, pois ele opera sobre convenção. Portanto, o framework infere que os arquivos estarão nessa pasta.

O RailsOnDia modifica o conteúdo e a semântica dos arquivos contidos nas pastas acima citadas. Atualmente, o trabalho de modificação é realizado manualmente e isso se torna um fator de erros, visto que uma modificação de estrutura às vezes carece da modificação nas duas pastas de forma sequencial. Logo, este trabalho tende a diminuir os erros gerados por modificações na estrutura da aplicação.

4.2 Extensible Markup Language (XML)

O XML é um padrão criado pela W3C, originalmente desenvolvido para lidar com os desafios da publicação em larga escala de informações em meio eletrônico. É considerado um padrão muito importante por lidar com a troca de informações dos mais diversificados tipos na Web. A representação XML permite a criação de arquivos que informam o conteúdo e a estrutura do dado representado, separando o conteúdo da informação através de uma organização hierárquica de marcação de texto. Um documento XML é composto por unidades chamadas de entidades, que possuem atributos que as caracterizam. Cada entidade pode referenciar outras de modo a incluí-las no documento. Será considerado um documento XML válido aquele que for composto por entidades e que possua necessariamente apenas um elemento raiz (BRAY; PAOLI; SPERBERG, 1998).

Para a realização desse trabalho foi criada uma estrutura XML simples, capaz de representar os dados e relacionamentos dos *models* presentes em uma aplicação Rails. Estruturas mais completas, que implementam em diagramas todos conceitos da orientação a objetos exigem um grau maior de representação, e já existe atualmente uma extensão do XML que possui uma série de regras e recursos com essa finalidade. Essa extensão é conhecida por XMI (XML Interchange) (OMG, 2005). No entanto, não está no âmbito deste trabalho a criação de um algoritmo de interpretação para cada tipo de representação existente, e, por esse motivo, foi desenvolvida uma representação mais simples, de modo a facilitar a compreensão e desenvolvimento do trabalho.

A estrutura criada possui uma raiz nomeada como *diagram*. Cada *model* da aplicação é representado como uma entidade filha da raiz e possui um atributo *name* para representar seu nome. Cada *model* possui duas entidades filhas: uma para representar seus atributos e outra para representar seus relacionamentos. Os atributos, representados pela entidade *attribute*, possuem um atributo name para representar seu nome. O tipo de um atributo é representado por uma entidade XML *datatype* hieraquicamente filha de uma entidade *attribute*. Cada *model* possui entidades, rotuladas *relation*, para representar os seus relacionamentos os demais *models* da aplicação. Elas possuem um atributo *type* que indica a cardinalidade do relacionamento no modelo. A figura 4.1 mostra em diagrama como é a organização hierárquica da estrutura criada.

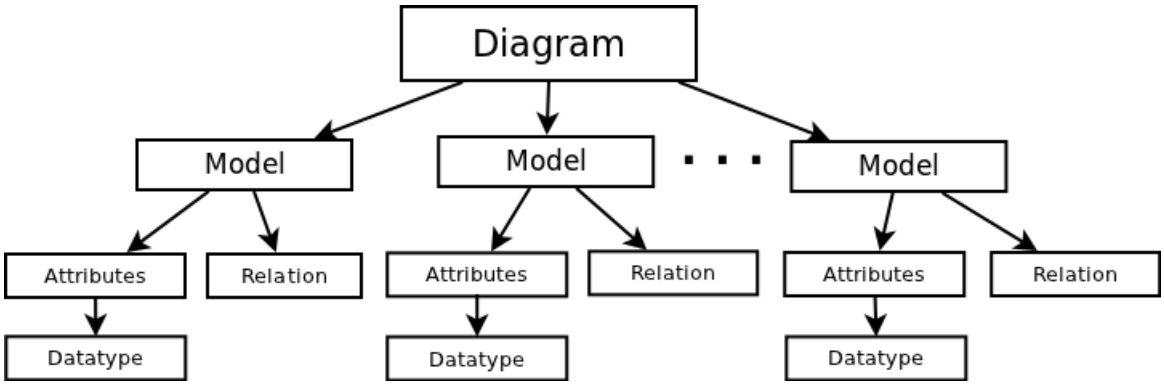


Figura 4.1: Diagrama hierárquico da estrutura XML criada

Supondo o seguinte contexto de software: "*Modelar um aplicativo que seja capaz de registrar projetos e tarefas referentes ao mesmo. Cada projeto deverá possuir um nome, uma data de começo e uma de fim. Cada tarefa deverá possuir um nome, descrição e datas de início e fim. Para os projetos deverá ser registrado, para fins de auditoria, as datas de criação e a última modificação de seus dados.*". Um possível diagrama de classes para esse contexto pode ser analisado na figura 4.2, onde foram omitidos os métodos das classes, pois os mesmos não influenciariam no entendimento da estrutura. A estrutura XML equivalente ao diagrama de classes pode ser vista na figura 4.3 e a sua montagem é realizada através das seguintes transformações:

- as classes do diagrama são representadas por entidades *model*.
- os atributos de cada classe são representados por entidades attribute e possuem um tipo de dado representado pela entidade datatype.
- o relacionamento entre as classes é representado em duas entidades relation, uma em cada *model*, representando a cardinalidade do relacionamento.

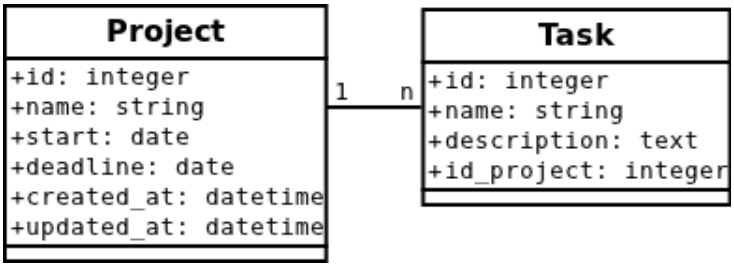


Figura 4.2: Diagrama de classe de exemplo

A estrutura XML criada é simples e não possui todos os recursos que a modelagem de aplicações carece. No entanto, é uma estrutura que permite explorar todos os recursos do RailsOnDia, e, por isso, será utilizada para a obtenção dos resultados.

```xml
<diagram>
  <model name="Projects">
    <attribute name="id">
      <datatype>INTEGER</datatype>
    </attribute>
    <attribute name="name">
      <datatype>STRING</datatype>
    </attribute>
    <attribute name="start">
      <datatype>DATE</datatype>
    </attribute>
    <attribute name="deadline">
      <datatype>DATE</datatype>
    </attribute>
    <attribute name="created_at">
      <datatype>DATETIME</datatype>
    </attribute>
    <attribute name="updated_at">
      <datatype>DATETIME</datatype>
    </attribute>
    <relation type="has_many">Tasks</relation>
  </model>

  <model name="Tasks">
    <attribute name="id">
      <datatype>INTEGER</datatype>
    </attribute>
    <attribute name="name">
      <datatype>STRING</datatype>
    </attribute>
    <attribute name="description">
      <datatype>TEXT</datatype>
    </attribute>
    <attribute name="id_project">
      <datatype>INTEGER</datatype>
    </attribute>
    <relation type="belongs_to">Projects</relation>
  </model>
</diagram>
```

Figura 4.3: Exemplo de estrutura XML

4.3 Modelagens abordadas

4.3.1 Object-relational mapping

O Rails é um framework criado para lidar principalmente com aplicações orientadas à base de dados, metodologia conhecida por *Database-driven Applications*, sendo seu foco principal em bases de dados relacionais. No entanto, pelo fato do framework ser desenvolvido com a tecnologia de orientação a objeto e os bancos de dados com tecnologia relacional, origina-se um problema conhecido por incompatibilidade de impedância objeto-relacional (AMBLER, 2003). Esse problema é resultado da diferente abordagem dada para os dados do sistema por cada uma das tecnologias citadas, ou seja, existe um *gap* semântico entre o que é modelado em relação a código e em relação a banco de dados. De modo a minimizar esse problema, o *Rails* possui uma classe que aplica o conceito de ORM (Object-relational mapping) aos seus dados (RUBY; THOMAS; HANSSON, 2011).

As bibliotecas que implementam o ORM mapeiam as tabelas dos bancos de dados para classes, linhas de uma tabela para objetos de classe e colunas para os atributos. No framework Rails a responsável por esse mapeamento é a classe *ActiveRecord*, que cria uma interface de programação que permite ao programador não utilizar comandos SQL diretamente na aplicação, tornado-se assim o responsável pela comunicação entre a camada de dados e de aplicação. Dessa forma, o programador não necessita tomar conhecimento de como seus objetos serão guardados em banco e, ao mesmo tempo, o responsável pela base de dados não necessita saber como suas tabelas são carregadas para os objetos do sistema (MARSHALL; PYTEL; YUREK, 2007).

Este trabalho aborda o desenvolvimento de uma biblioteca capaz de tirar proveito das características do ORM implementadas pela classe *ActiveRecord*, criando um modo de interação mais avançado que a atual manipulação de arquivos proposta pelo framework.

A interação é realizada através de diagramas onde é possível a criação e edição dos *models* do sistema e seus relacionamentos. Os diagramas são semelhantes a diagramas de classes propostos pela UML, onde classes, atributos e relacionamentos são representados de forma gráfica e clara. No entanto, não será um diagrama de classe completo pois não irá abordar aspectos como operações, múltipla herança, implementação, entre outros conceitos utilizados em uma modelagem orientada à objeto.

O diagrama proposto nessa seção se destina a simplesmente alterar o modo de interação do desenvolvedor Rails, dando-lhe alternativa ao modo de manipulação através de arquivos. O ORM não é de fato um diagrama, mas sim uma técnica de mapeamento que tenta minimizar e resolver o problema de persistir dados de uma aplicação orientada a objetos em um banco relacional. No entanto, é possível criar um diagrama que mostra o mesmo mapeamento realizado pelo ORM, como mostrado nessa seção.

4.3.2 Diagrama de classes

Os diagramas de classes fazem parte da especificação UML e representam a estrutura e as relações das classes que compõem um sistema (FOWLER, 2004). Por ser um diagrama que modela sistemas utilizando o paradigma da orientação a objeto, torna-se possível a utilização desse diagrama para modelagem de aplicativos em Rails. Os relacionamentos de UML que possuem tradução para uma aplicação em Rails são:

- Associação.
- Especialização.
- Agregação.
- Composição.

As associações são implementadas através de métodos do framework Rails. Os métodos pertencem à classe *ActiveRecord* e em seu nome utilizam uma nomenclatura que indica o tipo de relacionamento, e recebem como parâmetro o *model* a ser referenciado. Os métodos existentes para o relacionamento de associação estão expostos na tabela 4.1.

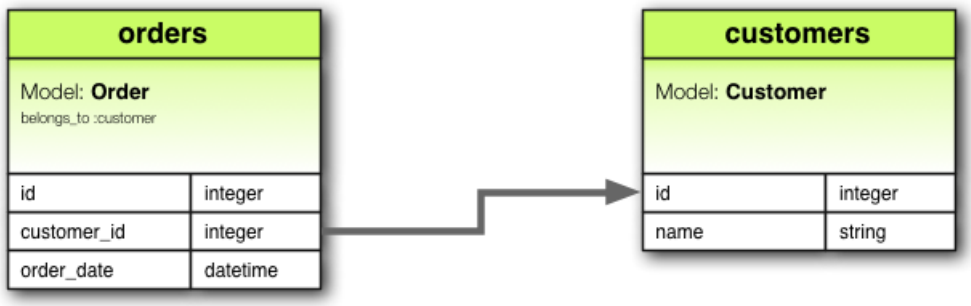
Para que as associações sejam mantidas em banco de dados, cabe ao programador criar colunas que tornem persistentes as informações sobre o relacionamento. Na figura 4.4 pode ser visto o caso em que um *model Order* pertence a um *model Customers* através da utilização do método *belongs_to*. De modo a persistir em banco de dados essa informação, deve ser criada um atributo para o *model Order* que armazene o identificador do *Customer*.

Diagramas de classes não preveem atributos para armazenar o relacionamento entre suas classes. Esse é mais um exemplo de problema de impedância objeto-relacional, no qual para armazenar um relacionamento a nível de orientação a objeto deve ser utilizado um atributo.

Em Rails a especialização é implementada com uma herança de classes em Ruby. Desta forma, os métodos e atributos da classe herdada ficam visíveis àquela que herda, como é o comportamento padrão desse relacionamento. Ao criar uma situação de herança em uma aplicação Rails, as duas classes costumam ser armazenadas na mesma tabela de banco de dados, sendo diferenciado o modelo por um atributo que armazena o tipo de registro. Por exemplo, dadas três classes: *Person*, *Manager* e *Employee*. As duas últimas são uma especialização de *Person* e, por isso, irão herdá-la. Todos os registros que armazenem dados de uma dessas três classes irão ser salvos na mesma tabela. De

Tabela 4.1: Métodos que implementam associações em *Rails*.

Método	**Comportamento**
belongs_to :person	indica que o *model* que define o relacionamento pertence a um *model person*
has_one :person	indica que o *model* que define o relacionamento possui um *model person*
has_many :person	indica que o *model* que define o relacionamento possui vários *models person*
has_and_belongs_to_many :person	indica que o *model* que define o relacionamento pertence e possui vários *models person*




```
class Order < ActiveRecord::Base
  belongs_to :customer
end
```

Figura 4.4: Exemplo de associação *belongs_to*

modo a identificar qual o tipo de registro, deverá ser criada uma coluna no banco de dados que represente esse tipo, diferenciando entre as classes possíveis. Esse é mais um exemplo de impedância encontrada na modelagem por diagrama de classes.

Para implementar o conceito de agregação o *ActiveRecord* possui um método chamado de *composed_of* que realiza a agregação de classes a outras. Apesar do nome sugerir um relacionamento de composição, esse método implementa o relacionamento de agregação (AKITA, 2006).

O relacionamento de composição possui várias maneiras de ser realizado no contexto desse trabalho. A linguagem Ruby permite a criação do conceito de composição através da delegação de métodos entre classes. No entanto, esse meio é muito verboso, pois é necessário especificar todos os métodos que serão compostos da outra classe, o que acaba implicando em uma série de códigos duplicados, o que não é considerada uma boa prática de programação (RUBY; THOMAS; HANSSON, 2011). Prevendo esse problema, existem alguns módulos em Ruby que possuem a intenção de tornar a tarefa de delegação mais enxuta, provendo meios que permitem a delegação de todos métodos necessários em apenas uma linha de código (FLANAGAN; MATSUM, 2009). O framework Rails utiliza um módulo chamado *delegate* para realizar esse controle de quais métodos devem ser delegados a outras classes, criando assim o conceito do relacionamento de composição (AKITA, 2006).

Tendo esses relacionamentos disponíveis para implementação em um ambiente Rails, torna-se possível a criação e manutenção de aplicações através de um diagrama de classes. Os problemas de incompatibilidade entre o software orientado a objeto e o modo de persistência relacional devem ser tratados pelo algoritmo de tradução entre o diagrama e a aplicação. O algoritmo de tradução deverá seguir um padrão de comportamento para tratar o modo de registro do relacionamento entre os modelos. No entanto, por se tratar de um software open-source, é possível a implementação de diferentes comportamentos pelo desenvolvedor. Na figura 4.5 está exemplificado o algoritmo utilizado para a obtenção de resultados deste trabalho.

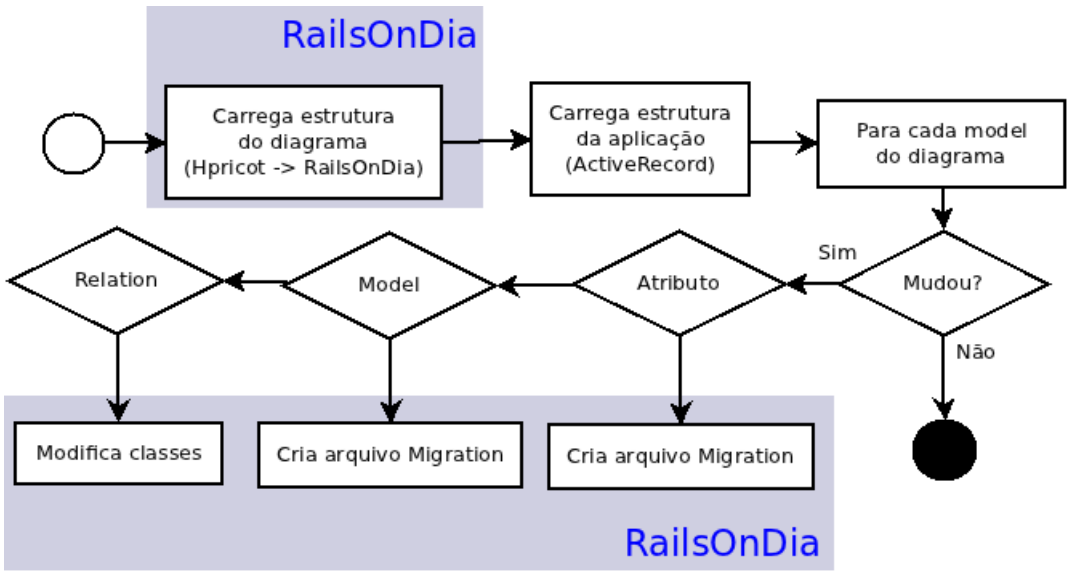


Figura 4.5: Algoritmo de manutenção da estrutura da aplicação.

4.4 RailsOnDia

4.4.1 Funcionamento

A biblioteca proposta, RailsOnDia, é desenvolvida em Ruby e utiliza a classe *ActiveRecord* do framework Rails para extrair informações estruturais da aplicação. Basicamente, a biblioteca possui funções que possibilitam a manipulação dos arquivos Migrations e das classes com os *models* relativos ao sistema modelado.

Para a utilização do RailsOnDia é necessário desenvolver um arquivo em Ruby capaz de interpretar a estrutura do diagrama criado pelo usuário. Essa interpretação é realizada criando um algoritmo de parseamento que identifica as entidades que devem ser modificadas na estrutura da aplicação, através de um arquivo XML que representa a estrutura do modelo. Com os dados obtidos pelo algoritmo de parseamento, torna-se possível a utilização das funções da biblioteca de modo a realizar as modificações necessárias na aplicação. Esse processo está mostrado diagramaticamente na figura 4.6.

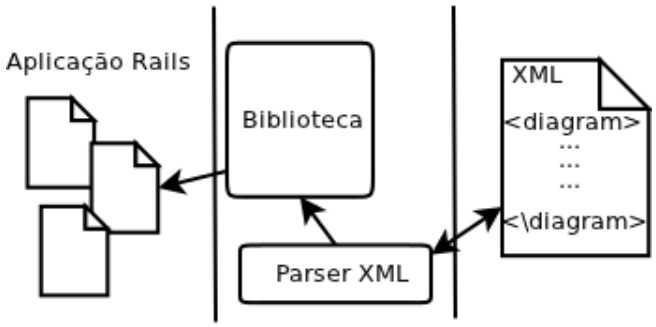


Figura 4.6: Diagrama de comunicação do processo de modelagem proposto.

A biblioteca é composta por um conjunto de classes que implementam as funcionalidades necessárias, ou seja, foram implementados métodos para a criação, modificação e exclusão de *models*, atributos e relacionamentos. A biblioteca foi desenvolvida com o paradigma da orientação a objeto e consiste na representação de um diagrama que possui diversos *models*, que, por sua vez, são compostos por atributos e relacionamentos. Seu diagrama de classes pode ser visto na figura 4.7, na qual estão omitidos os métodos, visto que o diagrama aumentaria de tamanho diminuindo a facilidade de compreensão.

Os *models* são manipulados através da criação de arquivos migrations, que são capazes de criar novas tabelas no banco de dados do sistema. A biblioteca garante a existência de um arquivo com o mesmo nome do *model* com o objetivo de torná-lo acessível ao ser carregado o sistema. De modo semelhante, os atributos de um *model* são tratados através de arquivos de migrations. No entanto, não é necessário o passo de criação de uma classe em Ruby, pois a mesma já deverá existir.

O relacionamento entre os *models* é tratado através da edição da classe dos arquivos envolvidos. Ou seja, ao ser criado um relacionamento entre o *model A* e o *model B* do tipo *model A has_many model B*, será adicionado à classe do *model A* o método de relacionamento *has_many* tendo como parâmetro o *model B*.

A classe *Diagram* não possui métodos ou atributos que influenciem na operação da biblioteca. Essa classe apenas existe para o agrupamento e melhor modelagem do contexto de diagramas. Futuramente essa classe poderá armazenar informações sobre o tipo de modelos utilizados com o objetivo de escolher o tipo de parser a ser utilizado. No

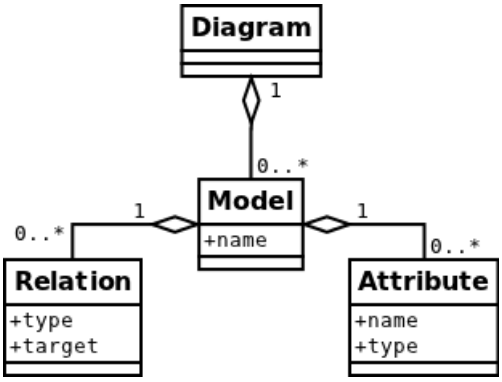


Figura 4.7: Diagrama de classe da biblioteca criada

entanto, este trabalho apenas aborda a leitura de um tipo de diagrama, descrito na seção 4.2, e por isso esse tipo de informação torna-se desnecessária.

A biblioteca cria um arquivo migration para cada alteração que será realizada na estrutura. A composição do nome de um arquivo de migração inclui como chave única um timestamp com precisão de segundos para unicidade dos arquivos, e, por esse motivo, cada uma das migrações necessita ser criada com pelo menos um segundo de diferença da anterior. Visto que, para muitos casos, o computador conseguiria criar várias migrações no espaço de tempo de um segundo, a biblioteca garante que cada uma das migrações possua um timestamp diferente, diferindo de pelo menos um segundo entre cada.

4.4.2 Comparativo com modo textual

Um conceito bastante repetido no ecossistema do framework Ruby on Rails é o DRY. Ele trata de garantir que o programador não repita uma mesma ação mais de uma vez, procurando assim agilizar o trabalho a ser desenvolvido. Atualmente, ao aplicar uma modelagem diagramática, como por exemplo o diagrama de classes, é necessário a transcrição de todos os dados do diagrama para o sistema de forma manual. Ou seja, os dados devem ser digitados e discutidos durante a modelagem do diagrama junto aos envolvidos, e, posteriormente, serem inseridos através de arquivos migrations ou modificações de classes dos *models* no framework Rails. Esse retrabalho pode ser retirado do fluxo de trabalho de um programador que utiliza modelagem, desde que haja uma comunicação e uma troca de dados entre o diagrama e a estrutura aplicação.

O ganho de agilidade ao utilizar tal integração é garantido, visto que uma etapa é retirada do fluxo. Por exemplo, supondo que para a elaboração do diagrama sejam gastos X minutos, sendo desses A minutos para conversa com os envolvidos e B minutos para a inserção dos dados no meio digital. Após aprovado o diagrama, os B minutos serão novamente gastos de modo a atualizar a estrutura da aplicação Rails com os novos dados discutidos com a equipe. Portanto, sempre que houver uma integração entre o diagrama e a aplicação, haverá um ganho através da diminuição de retrabalho da etapa de transcrição entre o modelo e a aplicação.

4.5 Resultados

Os resultados obtidos nesse trabalho foram avaliados frente a um processo de teste dividido em duas partes. Primeiramente, cada funcionalidade da biblioteca foi testada

isoladamente das demais, e, em um segundo momento, foram criados alguns casos de testes que eram composto pela execução de conjuntos de funcionalidades do RailsOnDia.

O processo de teste não pôde ser automatizado, visto que o Rails não possui um modo de testar a ferramenta nativa Rake. O Rake é essencial para o teste do correto funcionamento de um método, pois é ele que executa as modificações criadas pelo RailsOnDia.

Para cada processo de teste isolado foram definidos dois contextos: o ponto de partida e o resultado esperado. O ponto de partida, o mesmo para todas as funcionalidades, foi uma aplicação composta por dois *models* que possuiam um relacionamento entre eles, que pode ser visto na figura 4.8. Tendo essa aplicação, foram testadas as funcionalidades de criação, edição e remoção de *models*, atributos e relações. O resultado esperado foi o correto funcionamento da aplicação com a modificação proporcionada pelo teste.

Também foram definidos o número de passos do processo para a execução da teste. O primeiro passo foi a modificação da estrutura XML que representava o diagrama utilizado para teste, adicionando, editando ou retirando alguma de suas entidades. O segundo passo foi a execução do RailsOnDia. E o terceiro e último passo foi a execução da ferramenta Rake de modo a retificar o correto funcionamento do método.

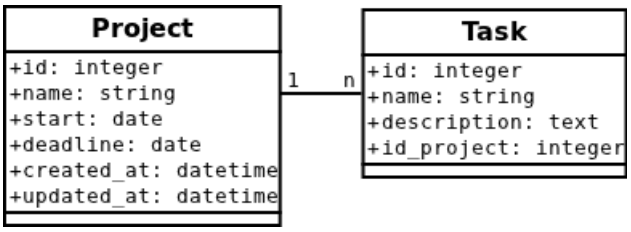


Figura 4.8: Diagrama da aplicação utilizada para testes.

Todos métodos inicialmente apresentaram um correto funcionamento e, após essa validação, passaram por um processo de refatoração de modo a melhorar o código produzido. Após o processo de refatoração, a biblioteca repetiu o processo de testes acima citado de modo a garantir sua integridade. Todas funcionalidades executaram em tempo inferior a um segundo, o que de fato era esperado devido ao baixo processamento à que foram submetidas isoladamente.

Após os testes isolados de funcionalidade, foram executados testes por conjuntos de métodos. Inicialmente esse processo apresentou erros e precisou ser refatorado de modo a não permitir dois arquivos *Migrations* com o mesmo timestamp. Esse problema ocorreu em casos com pouca modificação estrutural, onde mais de um arquivo foi criado dentro do intervalo de um segundo, ficando assim com o mesmo timestamp. De modo a não criar falsos timestamps para os arquivos, a rotina de criação dos arquivos incluiu uma função de *sleep* que força a execução a esperar um segundo até poder continuar seu processamento. Essa característica limita inferiormente o tempo de execução de acordo com o número de arquivos *Migrations*.

O principal resultado positivo desse trabalho foi a elaboração de uma ferramenta que permita a importação de um trabalho já realizado (a modelagem de um diagrama), diminuindo o retrabalho no fluxo de desenvolvimento. A simples importação de dados permite que o passo de codificação não se torne redundante ao processo de modelagem, respeitando assim um dos dogmas do Ruby on Rails, *Don't repeat yourself*.

5 CONCLUSÃO

Com o término do desenvolvimento do RailsOnDia, e a prova de que é possível uma modificação estrutural de uma aplicação em Rails através de scripts em Ruby, o foco volta-se para a criação de uma interface que seja totalmente Web e capaz de tirar proveito das funcionalidades do RailsOnDia. Essa interface tem o intuito de levar o processo de modelagem conceitual para um ambiente completamente Web, de modo a não necessitar de outras ferramentas de modelagem para a correta utilização das funcionalidades do RailsOnDia.

5.1 Limitações

O RailsOnDia foi utilizado como um script em Ruby capaz de carregar a estrutura do framework Ruby on Rails de modo a comparar a estrutura da aplicação com a proposta pelo diagrama modelado. No entanto, o ecossistema do Rails está acostumado a utilizar um conceito chamado de Gem's, que são para o Rails como são os plugins para outras linguagens, ou seja, incluem uma nova funcionalidade ao framework.

Inicialmente, o intuito desse trabalho era a criação de uma Gem que realizasse todas as funcionalidades do RailsOnDia. No entanto, não foi possível a conclusão dessa meta, e o RailsOnDia funcionou como um script externo ao framework, que possui a capacidade de se integrar ao mesmo. Isso se torna um limitante para a utilização da biblioteca em um ambiente de produção, visto que as Gem's possuem um controle de dependências e tendem a acompanhar o desenvolvimento e modificações de versão do framework, enquanto o RailsOnDia não possui essas características.

5.2 Perspectivas

Para trabalhos futuros, o principal objetivo é a criação e publicação da Gem RailsOnDia. Ele será um software open-source compartilhado no ecossistema Rails que possui a intenção de tornar cotidiana a utilização de modelagem, mesmo que superficial, de aplicações baseadas na Web.

Este trabalho poderá servir de base para a criação de um ambiente de modelagem nativo ao Rails através da interface Web dos navegadores. Essa funcionalidade permitirá que a utilização de modelos seja ainda mais intuitiva e corriqueira, caindo assim no fluxo diário de programação dos desenvolvedores. De modo a utilizar as bibliotecas em Ruby e o framework Rails, seria necessária a criação de um servidor responsável por carregar essas ferramentas. No entanto, estuda-se uma possibilidade de transformar a ferramenta para a linguagens client-side, como JavaScript, que seriam capazes de manipular a estrutura sem

a necessidade de um servidor somente para esta finalidade.

REFERÊNCIAS

AKITA, F. **Repensando a WEB com Rails**. [S.l.]: BRASPORT, 2006.

AMBLER, S. **Agile database techniques**: effective strategies for the agile software developer. [S.l.]: Wiley, 2003.

BRAY, T.; PAOLI, J.; SPERBERG, C. M. **Extensible Markup Language(XML) 1.0 W3C Recommendation 10 February 1998**. Acesso em Junho de 2011.

BURBECK, S. **Applications Programming in Smalltalk-80(TM)**: how to use model-view-controller (mvc). Acesso em Junho de 2011.

FLANAGAN, D.; MATSUM, Y. **LINGUAGEM DE PROGRAMAÇAO RUBY, A**. [S.l.]: ALTA BOOKS, 2009.

FOWLER, M. **UML distilled**: a brief guide to the standard object modeling language. [S.l.]: Addison-Wesley, 2004. (The Addison-Wesley object technology series).

KAPPEL, G. **Web engineering**: the discipline of systematic development of web applications. [S.l.]: John Wiley & Sons, 2006.

KOUTSOFIOS, E.; NORTH, S. Drawing graphs with dot. , [S.l.], 2002.

MARSHALL, K.; PYTEL, C.; YUREK, J. **Pro Active record**: databases with ruby and rails. [S.l.]: Apress, 2007. (Apress Series).

OMG. **XML Metadata Interchange Specification**. Acesso em Junho de 2011.

PRESSMAN, R. **Software engineering**: a practitioner's approach. [S.l.]: McGraw-Hill Higher Education, 2010.

PRESSMAN, R.; LOWE, D. **Web engineering**: a practitioner's approach. [S.l.]: McGraw-Hill Higher Education, 2008.

RUBY, S.; THOMAS, D.; HANSSON, D. **Agile Web Development with Rails**. [S.l.]: Pragmatic Bookshelf, 2011. (Pragmatic Bookshelf Series).

SOMMERVILLE, I. **Software engineering**. [S.l.]: Addison-Wesley, 2007. (International computer science series).