Ruby on Rails aplicado às práticas de Banco de Dados Ágeis

Reinaldo Saraiva do Carmo[1], **Ricardo Ramos**[2]

[1]Credishop S/A - Administradora de Cartões de Crédito
Teresina – PI – Brasil

[2]Instituto Federal de Educação Ciência e Tecnologia Piauí
Teresina – PI - Brasil

reinaldo@credishop.com.br, ricardo@cefet.br

Abstract. *This assignment propose that Ruby on Rails framework aplicated a pratical database agile, provide efficiently mechanisms to growing up the chances to sucess until the development process of projects of databases.*

Resumo. *Este trabalho propõe que a adoção do framework Ruby on Rails aplicado às práticas de banco de dados ágeis, forneçam mecanismos eficazes para elevar as chances de sucesso durante o processo de desenvolvimento de projetos de banco de dados.*

1. Introdução

De acordo com Ambler, o processo de desenvolvimento de um software moderno inclui Rational Unified Process (RUP), Extreme Programming (XP), Agile Unified Process (AUP) e Scrum. Todos são modelos evolutivos,ou seja, ágeis por natureza. Desse modo, se faz necessário, dentro das equipes de trabalho, a participação efetiva de um profissional de dados, no sentido de adotar instrumentos e práticas que lhes permitam concluir com efeciência e simplicidade [Ambler 2003].

Não há nenhum aspecto especial sobre os dados de um sistema TI, eles podem ser desenvolvidos de uma forma evolutiva, tal como aspectos não-dados. Mesmo armazenamento de dados pode ser desenvolvidos de uma forma evolutiva. Então, este artigo propõe recolher as melhores práticas de banco de dados ágeis aliada ao desenvolvimento de base de dados evolutiva, aplicado a framework Ruby on Rails.

2. Banco de Dados Ágeis

Há muitas décadas a indústria de software vem buscando técnicas de desenvolvimento para redução dos riscos dos projetos de software, tornando assim essa atividade mais produtiva. Em meados dos anos noventa surgiu uma corrente filosófica proposta por alguns profissionais da indústria de software, mais focado com os aspectos humanos dos projetos de software. Porém, somente em fevereiro de 2001, 17 profissionais experientes se reuniram em Utah (EUA) para discutir suas práticas de desenvolvimento e propuseram alternativas que evitassem processos de desenvolvimento, excessivamente baseados em documentação e formalismos. Naquele momento, decidiram organizar estas propostas sob um nome comum: desenvolvimento ágil de software, que só foi concretizado no lançamento do Manifesto pelo Desenvolvimento Ágil de Software [Ambler 2003].

O desenvolvimento das metodologias ágeis ocorreram a partir da necessidade de avaliação constante de novos requisitos e das complexas exigências para evolução de um

projeto de banco de dados. Um dos aspectos mais centrais destas exigências é a idéia da concepção evolutiva. Em um projeto ágil assume-se que não se pode fixar as exigências de um sistema "UP-FRONT"(de cima à abaixo). Como resultado, o detalhamento exacerbado na fase de concepção, torna-se impraticável no início de um projeto, porém este tem que evoluir através das várias interações do software. Desta forma, os métodos ágeis, em especial Extreme Programming (XP), possuem uma série de ações que tornam esta concepção evolutiva mais prática [Ambler 2003].

Diante deste cenário foi criado por David Heinemeier Hansson, em 2004, a framework Ruby on Rails, desenvolvido em Ruby, a partir das experiências adquiridas na aplicação Basecamp. A framework Ruby on Rails além de ser uma excelente ferramenta para desenvolvimento de aplicações web, ainda permite aplicar práticas de banco de dados agéis [Thomas 2005].

2.1. Práticas

O coração de todo projeto de banco de dados evolutivos são suas práticas que na framework Ruby on Rails está presente desde sua concepção.

Para tornar a programação de aplicações web mais fácil, o Rails faz várias suposições sobre o que cada desenvolvedor precisa para começar. Ele permite que você escreva menos código enquanto faz mais que muitas outras linguagens e frameworks. [Thomas 2005].

A filosofia Rails inclui diversos princípios guia:

- DRY ("Don't Repeat Yourself") - sugere que escrever o mesmo código várias vezes é uma coisa ruim.
- Convenção ao invés de Configuração - significa que o Rails faz suposições sobre o que você quer fazer e como você estará fazendo isto, em vez de deixá-lo mudar cada minúscula coisa através de intermináveis arquivos de configuração.
- REST é um modelo para aplicações web serem organizadas através de recursos e verbos HTTP padrão é o modo mais rápido para proceder.

O Rails é organizado usando a arquietura Modelo, Visão e Controle, bastante conhecido como MVC. Os benefícios desta arquitetura são [Thomas 2005]:

- Isolação entre a lógica de negócios e a interface de usuário
- Facilidade de manter o código DRY
- Manter claro onde tipos de código diferentes pertencem para facilitar a manutenção

2.1.1. Colaboração entre DBAs e Desenvolvedores

Um dos princípios dos métodos ágeis é que as pessoas com diferentes competências e backgrounds necessitam estreitar a comunicação. Entre aqueles que precisam desta estreita interação estão os desenvolvedores e os DBA's. Para que isto aconteça, o DBA tem que fazer-se acessível e disponível. Énecessário colocar em prática a comunicação direta e que estejam próximos uns dos outros visando facilitar algum acordo juntos [Fowler 2003].

Esta colaboração pode ser facilmente obtida com Rails, principalmente por sua arquitetura e princípios, facilitando o aprendizado do indivíduo e da organização, que são

essenciais para aumentar a eficiência em relação ao tempo. Sem o aprendizado os erros são constantes, o que representa desperdício de recursos.

2.1.2. Todo mundo recebe a sua própria base de dados exemplo

É importante para cada desenvolvedor ter a sua própria base de dados de teste, onde poderá experimentar, sem ter que se preocupar às modificações que possam afetar alguém. Muitos DBA's consideram esta prática complexa e difícil de funcionar no mundo real. Com isso, faz-se necessário uma ferramenta que permita a manutenção de forma ágil [Fowler 2003].

O Rails já tem como suporte padrão o SQLite, que é uma aplicação de banco de dados bastante leve e sem servidor. O Rails se utiliza por padrão do SQLite quando um novo projeto é criado, mas você pode sempre modificar isso posteriormente. O banco de dados a ser usado é especificado em um arquivo de configuração, *config/database.yml*. Este arquivo permite configurar o Rails e rodar em três ambientes de execução diferentes [Akita 2006]:

- O ambiente *development* (desenvolvimento) é usado em seu próprio computador enquanto você interage manualmente com a aplicação;
- O ambiente *test* (teste) é usado para rodar testes automatizados ;
- O ambiente *production* (produção) é usado quando a aplicação está pronta para executar em ambiente de produção;

2.1.3. Integração Contínua com desenvolvedores

A Integração Contínua é aplicada ao gerenciamento de código-fonte. O banco de dados pode ser tratado como uma parte do código-fonte, o qual deve ser mantido sob gerenciamento de configuração da mesma forma que o código-fonte [Fowler 2003].

Da mesma forma que o código-fonte, grande parte da integração é tratada pelo sistema de controle versão. Quaisquer alterações no banco de dados requerem corretude, com refatorações automatizado da base de dados. Além disso, o DBA precisa verificar se encaixa dentro do esquema global do banco de dados. Para que isso funcione eficientemente, grandes mudanças não devem surpreender como a integração temporal - daí a necessidade do estreitamento da relação entre o DBA e os desenvolvedores [Fowler 2003].

O Rails tem suporte a integração contínua do esquema do banco de dados através das migrations. Migrations é a forma conveniente de alterarmos o banco de dados de uma maneira organizada e estruturada. É possível editar fragmentos de SQL na mão, mas teríamos a responsabilidade de comunicar aos outros desenvolvedores que eles precisam executá-los. É importante acompanhar as mudanças na máquina de produção toda vez que for feito um deploy [Akita 2006].

Active Record é a base para os models em uma aplicacão Rails, fornecendo independência de banco de dados, alteração na base de dados de forma organizada e estruturada, funcionalidade CRUD básica, capacidade de buscas avançadas e a habilidade de relacionamento entre models, além de controle das mudanças ocorridas no banco de dados [Akita 2006].

O Active Record marca as migrations que já foram executadas, então tudo que se precisa fazer é atualizar o código através de um utilitário de linha de comando, o rake. O Active Record saberá quais migrations devem ser realizadas. Ele também irá atualizar o arquivo db/schema.rb para refletir a estrutura de sua base de dados. As migrations também permitem descrever cada mudança apenas usando a própria lingugagem Ruby. O principal benefício desta estratégia, "assim como grande parte das funcionalidades do Active Record" é a independência do banco de dados. Logo, não será necessário preocupar-se com as diversas variações existentes entre o SQL puro e as funcionalidades específicas dos bancos de dados [Akita 2006].

Migrations são armazenadas em arquivos dentro do diretório *db/migrate*, um para cada classe de migration. O nome dos arquivos tem a forma de YYYYMMDDHHMMSS_create_produtos.rb. O nome da classe de migration deve bater com a última parte do arquivo.

2.1.4. Testando banco de dados

As bases de dados são testadas principalmente para ajudar a estabilizar o desenvolvimento de uma aplicação. E para que funcione de forma eficiente, é necessário que o banco de dados contenha algumas amostragems como exemplo, eliminar todos os erros que possam acontecer durante as mudanças na base de dados, antes que seja colocado na base de dados de produção. Os testes também permitem testar as migrações tanto das base de dados de testes como das legadas ou produção [Ambler 2003].

No Rails, o suporte a testes está presente desde o início, ou seja, todas as aplicações construídas interagem com base de dados de teste, que é gerado na pasta 'test' durante a crição da aplicação. A pasta 'unit' é encarregada de armazenar testes para os *models* da aplicação, a pasta functional armazena testes para os seus controllers e a pasta *integration* deve guardar testes que envolvam interações entre seus *controllers*. O arquivo *test_helper.rb* guarda as configurações padrão para os seus testes [Thomas 2005].

Fixtures são uma forma de organizar os dados de teste; elas ficam na pasta *fixtures*. As fixtures é forma que Rails encontrou para ter amostra de dados. Elas permitem que configuremos a base de dados de teste com informações pré-definidas antes de executar os seus testes. São independentes de banco de dados e assume um formato: YAML. O YAML é formato amigável de descrever a amostra de dados. Abaixo um exemplo de arquivo YAML [Thomas 2005].

Listing 1. Arquivo de fixture YAML

```
reinaldo:
        nome: Reinaldo Saraiva do Carmo
        data_nascimento: 1978-05-07
        profissao: Gerente de TI
katiana:
        name: Katiana Ferreira Oliveira Saraiva
        data_nascimento: 1976-11-22
        profissao: administradora
```

2.1.5. Refatoração em banco de dados

A técnica de refatoração é um conceito conhecido em linguagens de programação orientadas a objeto, mas ela também é usada para bancos de dados. De acordo com Ambler, é quando uma simples mudança no esquema de uma base de dados melhora a sua concepção(projeto), embora mantendo simultaneamente a sua semântica [Ambler 2003]. E para garantir a consistência dos dados as mudanças devem ocorrer em pequenas interações da mesma que é para código fonte [Sadage P. J. 2006].

Para automatizar muitas das refatorações é essencial para o banco de dados que as alterações de esquema e a migração de dados sejam realizadas automaticamente por intermédio de alguma ferramenta, possibilitando sequenciamento das mudanças, garantindo a integração contínua com banco de dados em produção. Esta capacidade pode reverter as alterações automaticamente com pouco esforço [Sadage P. J. 2006].

Em Rails, é pratícavel usufruir de refatorações automáticas, visto que é possível lidar com entidades e tabelas, implementadas a um mapeamento mais direto, ou seja, a entidade carrega todos seus atributos assim como seus detalhes, tornando auto-suficiente em sua persistência. Isso leva para um dos pilares do Rails que é a Convenção, ao invés de Código e Não repetição de código. A partir deste conceito, o framework consegue conectar-se ao banco de dados, pesquisar os campos da tabela através de comandos SQL padrão, usando metaprogramação para criar as propriedades [Thomas 2005].

2.2. Conclusões

A partir deste trabalho verificamos que todos os membros do projeto será capaz de explorar o projeto do banco de dados facilmente, de modo que elas possam saber quais tabelas estão disponíveis e como elas são utilizadas, elevando principalmente a coloboração entre DBA's e desenvolvedores.

Vimos também com adoção das práticas de banco de dados ágeis, evoluir os dados incrementalmente, realizar testes e implementar refatorações no banco de dados. Sendo assim, espera-se que trabalhos futuros sejam capazes de obter outros dados de modo a verificar os resultados em banco de dados que trabalham 24/7.

References

Akita, F. (2006). *Repensando a web com Rails*. Brasport.

Ambler, S. (2003). *Agile Database Techniques: Efective Strategies for the Agile Software Developer*. Wiley Application Development.

Fowler, M. (2003). Evolutionary database design. *http://www.martinfowler.com/articles/evodb.html*, 1:15.

Sadage P. J., S. A. (2006). *Refactoring Databases: Evolutionary Database Design*. Hardcover.

Thomas, D. (2005). *Agile Web Development with Rails: A Pragmatic Guide*. Pragmatic Bookshelf.