Relatório final de estágio

Jônatas Davi Paganini

novembro de 2009

Um ano se passou e o sistema proposto está desenvolvido. Após muitos meses de estudo, análise de outros frameworks, decisões de design e arquitetura, está pronto o primeiro *release* do sistema. As lições aprendidas que pude tirar foram em vários aspectos:

Simplicidade é um dos princípios do Extremme Programming. Através da expressividade da linguagem Ruby e das ferramentas já existentes, é possível se conseguir um ótimo resultado com poucas linhas de código.

Estudo da melhor forma de executar tem sido um dos desafios neste projeto. Cada detalhe dos frameworks utilizados, consegue trazer um recurso exato de forma simples e direta. Por exemplo, em uma necessidade de automaticamente reconhecer os recursos de banco de dados do sistema, é preciso reconhecer os padrões e implementar o método to_liquid para cada objeto que deseja ter acesso na visão. Esta é uma forma interessante de protejer o banco de dados e qualquer tipo de *sql injection*. Para isso, se deseja ter acesso a um atributo, este atributo deve responder pelo método to_liquid. Por exemplo, em uma classe cliente, com um atributo nome, este atributo deve responder pelo método to_liquid. No próprio framework liquid, existe um método auxiliar que gera os métodos to_liquid via meta-programação, precisando apenas usar a seguinte sintaxe:

Listing 1: Implementação do método to_liquid através de um método helper

```
1 class Pessoa < ActiveRecord::Base
2   liquid_methods :nome, :telefone, :enderecos
3 end
```

Desta forma, se este código for usado em uma classe que contenha estes atributos ou métodos, ele irá reconhecer a fonte de dados e os métodos acessíveis. Com esta forma simples de declaração, agora apenas é necessário descobrir seguramente quais atributos podem ir para a visão, ou seja, podem ser vistos na template do liquid.

O uso de testes automatizados ajudou muito no período em que estava sendo desenvolvido o núcleo do *plugin*. Inspirado nos *frameworks* já existentes, nos princípios do *Rails*: *convention over configuration*(convenção ao invés de configuração) e *DRY: Don't repeat yourself*(não se repita) foi possível testar o comportamento da ferramenta perante um sistema de exemplo, foi possível documentar e escrever o *plugin* ao mesmo tempo. Através de técnicas de *Behaviour Driven Development*(Desenvolvimento Orientado a Comportamento) e *frameworks* como o *Cucumber*, foi possível documentar e testar a funcionalidade que o *plugin* oferecia.

Com poucas linhas de código, os testes desenvolvidos nas estórias trouxeram um exemplo de como o sistema funciona. Cada linha da estória sugere uma situação específica de uso do

plugin. Através de exemplos, é possível qualquer desenvolvedor que use o *plugin* consiga também trazer novas funcionalidades e contribuir com o código livre.

Durante o desenvolvimento do estágio, houve a necessidade de automatização dos processos de relatórios de estágio e relatório mensal da empresa. Neste momento, houve o primeiro desafio do *plugin* de usar as templates do framework liquid. Através de apenas um arquivo de texto plano e algumas convenções, o arquivo acompanhamento.textile divide as tarefas e históricos para apresentação do relatório mensal da empresa e o relatório de estágio.

Por padrão, o sistema Troper adotou a visão que todos atributos de um objeto e seus relacionamentos devem ser fornecidos para a template liquid. Desta forma foi preciso apenas navegar pelos modelos fornecidos pelo sistema base e apartir de cada modelo foi invocado o método **liquid_methods** com todos os atributos e relacionamentos do banco de dados.

Listing 2: Implementação do método to_liquid através da meta-programação

```
1
2 for model in self.models
3   if not model.instance_method_already_implemented? "to_liquid"
4     attrs = model.columns.collect(&:name) +
5             model.reflections.keys
6     model.class_eval { liquid_methods *attrs }
7   end
8 end
```

O *plugin* também tem **segurança**. No código acima, o método que "adivinha"os atributos, está limitado a sobrescrição do próprio sistema(linha 2). Ou seja, a implementação só será feita, se o desenvolvedor não o fizer.

Enquanto todos os preparativos de background estavam sendo programados, o conhecimento sobre a interface estava sendo adquirido. Através de livros, fóruns e exemplos, foi possível criar uma interface muito semelhante a proposta em rascunhos. A ferramenta usada para construir a interface, permitiu que fosse totalmente codificado em javascript e html.

A interface do sistema apenas faz referência a um arquivo html. O controlador responsável por trocar informações sobre os modelos é injetado no sistema base apenas para responder sobre quais elementos pode exibir ou não.

A linguagem de programação utilizada neste projeto, teve sim, um único aspecto negativo: a falta de conhecimento dos professores sobre a linguagem e o contexto, tornando um pouco

abstratas as tarefas e explicações sobre o que estava sendo realizado.

A gestão do projeto trouxe mais desenvolvimento de software para o projeto. Na tentativa de automatizar muitos processos burocráticos para documentação, o tempo do desenvolvimento foi compartilhado com as tarefas da gerência do projeto. A dificuldade de trabalhar com documentos formais trouxe ainda o estudo da ferramenta LaTeX e outros *frameworks* como *RedCloth, Textile, Rake, Rak* e ferramentas como *Git, Shell Script* conseguiram integrar o processo de desenvolvimento e acompanhamento do orientador. Com todas essas ferramentas, a integração do sistema tornou-se uma mão cheia de pequenos *frameworks* com funcionalidades bem definidas.

O processo de integração contínua no ambiente de desenvolvimento, é um fato esquecido, ou até mesmo desconhecido pelos professores. Mesmo não fazendo parte da ementa, este problema complica a gestão do projeto. A atitude tomada no projeto atual, trouxe atrasos ao projeto e também foi além do escopo acordado inicialmente. Em meio a esta dualidade, esta foi uma boa experiência e as duas partes têm suas compensações positivas.