4
O Workflow e a Máquina de Regras

O objetivo do workflow e da máquina de regras é definir um conjunto de passos e regras configuráveis. Ao longo de sua execução, um usuário consegue simplificar o seu trabalho, automatizando passos em seu processo que, caso contrário, seriam executados de maneira mais lenta e manual. Esta automatização se deve à configuração de quais regras se aplicam ao projeto e de seu uso.

Neste capítulo, é apresentada toda a estrutura da máquina de regras: como foi definida sua modelagem, como funciona seu fluxo, são detalhados os passos para a criação e uso de uma nova regra e são apresentadas quais regras já foram criadas.

4.1.
Arquitetura da Ferramenta

Foi decidido que a ferramenta *Redmine* (seção 3.5) seria utilizada como base para o desenvolvimento desse trabalho.

Como dito na seção 3.2, o *Redmine* foi desenvolvido utilizando *Ruby on Rails*, que é um framework que utiliza a arquitetura MVC (*model-view-control*). *Rails* provê uma estrutura para a aplicação – o desenvolvedor cria os modelos, visões e controles como partes separadas de funcionalidades e o *Rails* liga todas enquanto a aplicação executa. Uma das vantagens do *Rails* é que a ligação entre as diferentes partes do software é feita através de convenções definidas, dispensando a necessidade de configurações externas de meta-dados (como arquivos XML) para a aplicação funcionar, facilitando o desenvolvimento e customização da ferramenta.

Em aplicações desenvolvidas em Rails, e o Redmine não é exceção, uma requisição enviada pelo usuário é direcionada para um roteador (Ilustração 22), que decide para como a requisição deve ser analisada e para que controlador da aplicação deve ser enviada. A fase final é identificar um método específico

dentro do código do controlador, conhecido como ação. A ação pode buscar dados da requisição em si, pode interagir com o modelo e pode invocar outras ações. Eventualmente, a ação prepara informações para a visão, que apresenta o que é enviado pela ação.

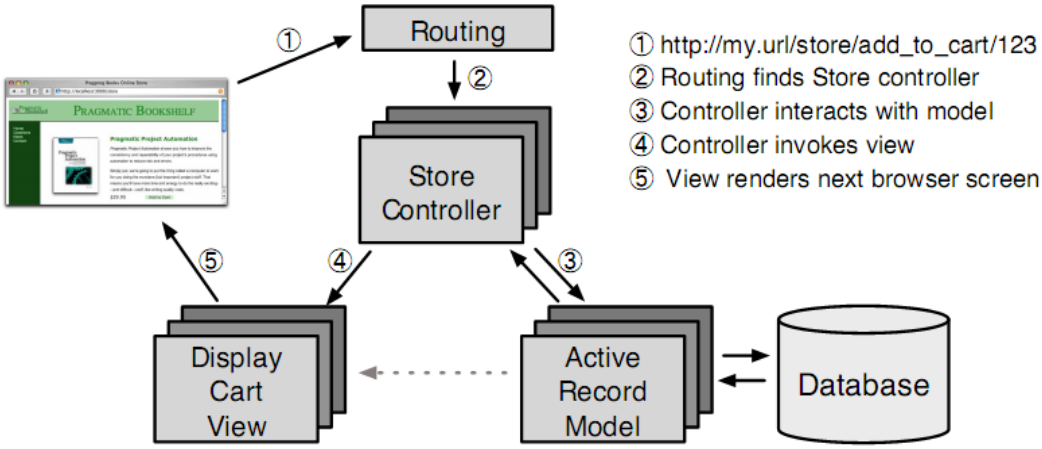

Ilustração 22 - Fluxo de uma requisição no Ruby on Rails

Uma das vantagens das aplicações que utilizam o *framework Ruby on Rails* é que ele já possui sua própria infra-estrutura de testes construída em cada projeto a partir do momento de sua criação. *Rails* cria um conjunto de subpastas para cada categoria de testes, dentro de uma pasta chamada test. As divisões de categoria são as seguintes:

Units: São os testes unitários, responsáveis por cobrir os testes das classes do nível de modelo.

Functionals: testes funcionais responsáveis por testar a lógica dos controladores.

Integration: testes de fluxo entre um ou mais controladores.

Para a criação do conjunto de regras, foi utilizado o ciclo de vida de um objeto em Ruby on Rails. Seu ciclo de vida é controlado pelo próprio *framework*. É possível monitorar e interferir nas transações no banco de acordo com seu tipo, seja uma atualização, criação ou exclusão, conforme mostra a Ilustração 23.

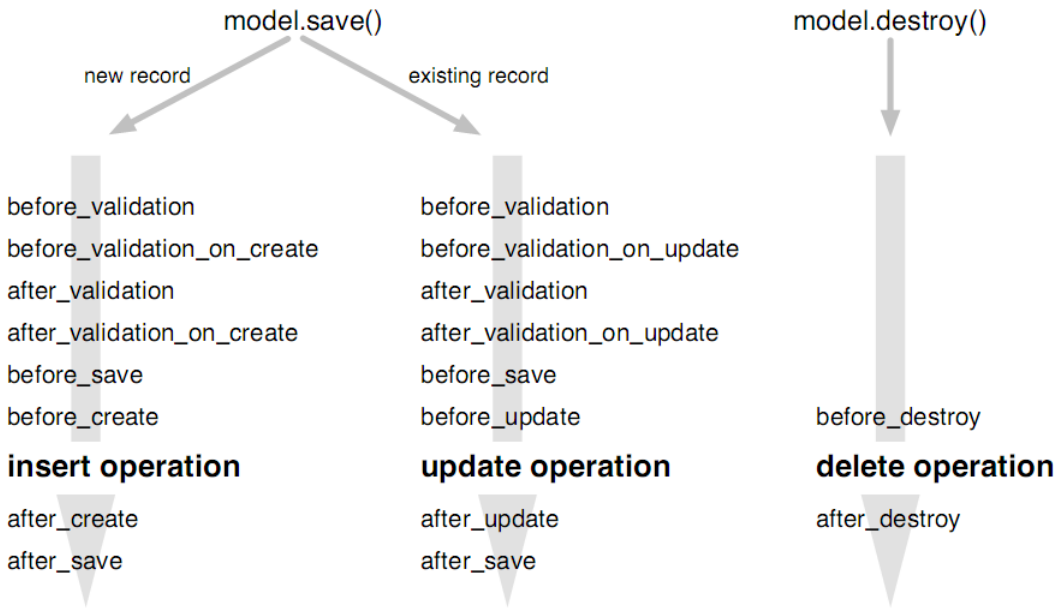

Ilustração 23 - Ciclo de vida de um objeto no Ruby on Rails (Ruby, 2009)

Os nomes acima são métodos que podem ser executados nos diferentes momentos do ciclo de vida de um objeto. Estes métodos são definidos como *callbacks* (Ruby, 2009). Se, por exemplo, desejarmos executar uma ação antes da criação de um objeto no banco, basta sobrescrever o método *before_create*.

4.2. Modelagem

A modelagem das regras teve como base as pesquisas de trabalhos existentes focados em workflows dinâmicos (seção 7). A maioria deles (os que eram baseados em regras) utiliza uma máquina de regras ECA (*Event-Condition-Action*). Neste formato, o início de um evento, partindo que uma condição é atendida, gera uma ação, alterando o estado do workflow.

Pensando nesta modelagem para a ferramenta de processos ágeis, temos a seguinte adaptação:

- Evento: Ação do usuário. Salvar ou atualizar alguma entidade do processo.
- Condição: A ação do usuário quebrar alguma regra de negócio.
- Ação: execução de uma solução para corrigir o caminho do workflow. Esta última está sujeita à execução manual do usuário.

Segundo Rinderle (Rinderle, 2004), um workflow deve poder ser alterado em um nível de instância e em um nível de tipo. O workflow criado permite que

regras sejam alteradas, onde estas alterações afetam todos os projetos que as usam. E permite que regras sejam adicionadas e removidas de projetos, afetando apenas o projeto em questão.

Existem duas entidades importantes na máquina de regras: a regra e suas soluções. Uma regra possui uma condição de aceitação que verifica se sua aplicação é aceita ou não. Quando ela não é aceita, ou seja, as condições da regra não estão sendo atendidas, então suas possíveis soluções são apresentadas para o usuário para corrigir esta falha.

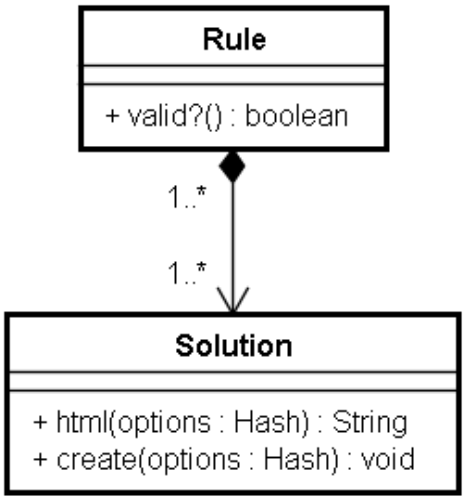

Ilustração 24 - Relacionamento entre Regras e Soluções

A Ilustração acima apresenta o relacionamento entre as regras e as soluções. Uma Regra possui várias soluções e uma solução, por sua vez, pode estar associada a várias regras. Na Ilustração 25, é apresentada a tela de uma regra no sistema, onde uma regra possui duas soluções e estas soluções podem estar associadas a outras regras.

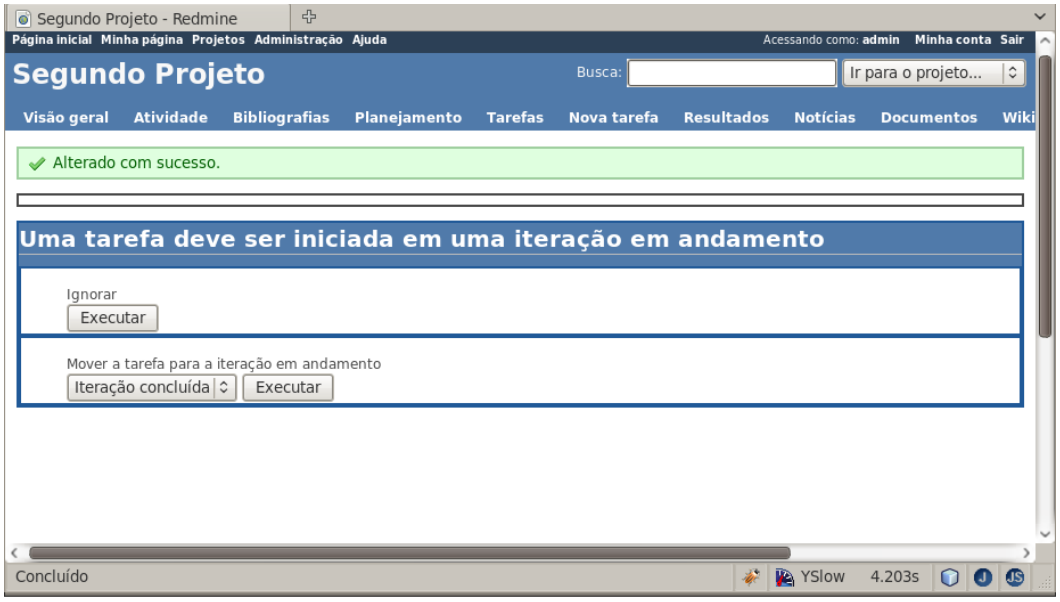

Ilustração 25 - Tela de Regra em Execução

O fluxo do sistema, com relação às requisições e arquitetura do projeto, respeita o padrão de projeto MVC. Nas ilustrações seguintes, são apresentados dois diagramas de seqüência com os dois cenários da execução de uma regra. O primeiro (Ilustração 26) apresenta o fluxo da criação de uma tarefa com sucesso. Onde as regras são verificadas. Quando uma tarefa é salva (*Issue* do diagrama), o sistema verifica a validade das regras (a chamada do método *check_rules_save()*) e prossegue com o curso do sistema, apresentando a tarefa criada para o usuário.

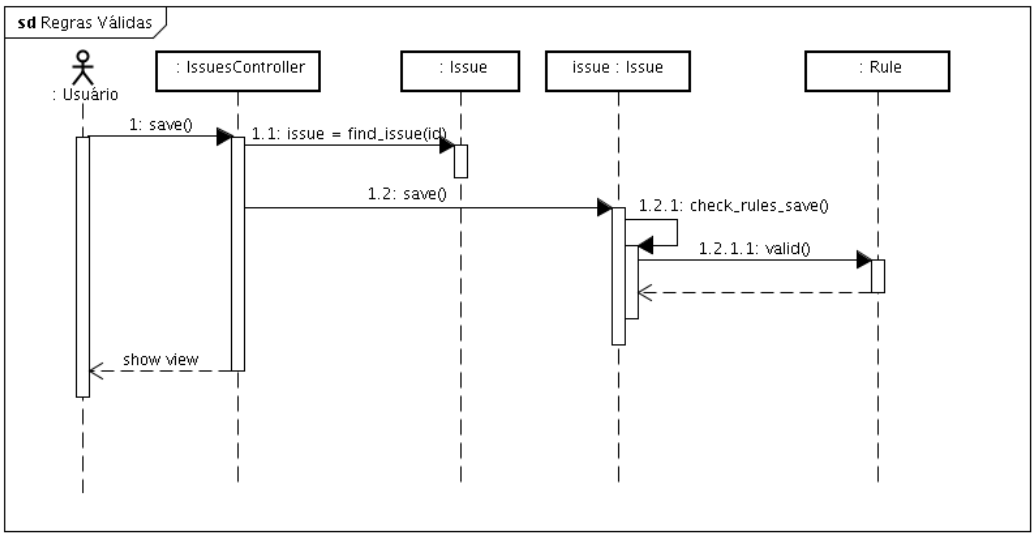

Ilustração 26 - Diagrama de Seqüência sem Regras Inválidas

Na Ilustração 27, no momento em que o sistema verifica que existem regras inválidas, o sistema é redirecionado para a tela de Regras, exibindo a regra e suas soluções para o usuário.

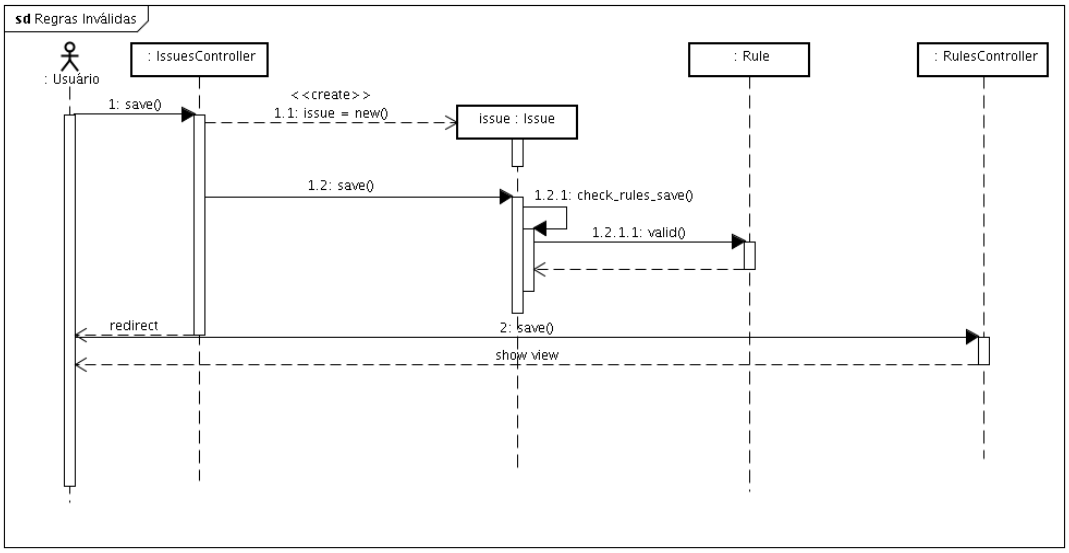

Ilustração 27 - Diagrama de Seqüência com Regras Inválidas

A solução selecionada possui a função de apresentação para o usuário e a de execução. A função de apresentação exibe para o usuário qualquer necessidade para a execução da solução, como textos informativos ou campos a serem preenchidos. A função de execução é responsável por executar a solução escolhida e realizar as alterações no processo.

4.3. Novas Regras

A alteração do workflow e de suas regras segue uma abordagem que não requer informações da instância (Rinderle, 2004), ou seja, novas regras podem ser criadas e incluídas no projeto sem necessidade de revisão das outras regras já criadas. Já que a inclusão de uma regra em nova um projeto representa uma extensão e não uma alteração do workflow já existente.

Abaixo são apresentados os passos necessários para um usuário criar uma nova regra em seu sistema. Regras novas podem atender necessidades específicas para projetos com características particulares.

Como a ferramenta foi desenvolvida usando o framework *Ruby on Rails* e a linguagem *Ruby*, a sintaxe do código-fonte apresentado é da linguagem *Ruby*.

4.3.1.
Como criar uma nova regra

Para criar uma regra, devem ser seguidos os passos:

- Criar nova classe de regra dentro da estrutura do projeto.
- Criar nova classe de solução dentro da estrutura do projeto.
- Cadastrar regra no banco.
- Cadastrar solução no banco.
- Cadastrar regra no projeto.

Estes são os passos necessários para criar uma regra funcionando corretamente dentro de um projeto já existente e são apresentados com detalhes abaixo.

Criar nova classe de regra

Criar uma classe na pasta “app/rules” de sua aplicação. Esta classe deve conter um método chamado *valid?(obj)*. O ponto de interrogação no fim do nome do método significa que o método deve retornar um valor booleano.

```
class TestDrivenRule
  def valid?(issue)
    if issue.is_development?
      return !issue.pre_requirement.nil?
    else
      return true
    end
  end
end
```

Conteúdo do método

Ilustração 28 - Exemplo de Classe de Regra

Se o retorno do método for *true* (verdadeiro), significa que a condição da regra é valida. Caso contrário, suas soluções precisam ser apresentadas para o usuário. O parâmetro “obj” é o objeto referente ao processo sob o qual a regra age. No exemplo acima, o parâmetro foi renomeado para *issue* (tarefa).

Na Ilustração 28, a regra deseja verificar se uma tarefa de desenvolvimento possui uma tarefa de teste como pré-requisito. A regra verifica se a tarefa *(issue)* sendo criada é uma tarefa de desenvolvimento. Depois,

verifica se já possui uma tarefa de pré-requisito. Caso não possua, o método retorna o valor falso, definindo que a regra será disparada como inválida.

Criar classe de solução

As classes de soluções devem ser criadas na pasta “app/rules/solutions” da aplicação.

Esta classe deve conter os seguintes métodos

- html(options={}) - Apresenta os dados do formulário para a submissão da solução.
- create(options) – Executa a solução para resolver a regra. Recebe os dados vindos do formulário do método anterior.

```
class SolutionTddCreateTest
  def self.html(options={})
    output = "Criar uma nova tarefa de teste<br>"
    return output
  end



  def self.create(options)
    issue = options[:issue]
    test = Issue.new issue.attributes
    test.tracker = Tracker.find_by_name "Teste"
    test.subject = "[Teste] #{test.subject}"
    issue.pre_requirements << test
  end
end
```

Ilustração 29 - Classe de Solução de Criação de Tarefa de Teste

A classe acima é uma solução para a regra de *test driven development*. Ela sugere ao usuário criar uma nova tarefa de teste associado como pré-requisito da tarefa de desenvolvimento sendo criada. O método *html()* monta o conteúdo do formulário para exibir para o usuário. E o método *create* é chamado caso o formulário gerado pelo método *html* seja selecionado. Este método recupera a tarefa (variável *issue*) de desenvolvimento, cria e associa como pré-requisito a tarefa de teste (variável *test*).

```ruby
class SolutionTddAssociatePreRequirement
  class Form
    include ActionView::Helpers
  end

  def self.enabled?(options={})
    tests = Issue.find_tests(options[:project])
    return !tests.empty?
  end

  def self.html(options={})
    output = "Associar tarefa a uma tarefa de teste<br>"
    form = Form.new
    tests = Issue.find_tests(options[:project])
    output << form.select_tag(:pre_requirement_id,form.options_for_select(tests.collect{|i| [i.subject,i.id.to_s]}))
    return output
  end

  def self.create(options)
    issue = options[:issue]
    issue.pre_requirements << Issue.find(options[:params][:pre_requirement_id])
  end
end
```

Ilustração 30 - Classe de Solução de Associação com Tarefa de Teste Existente

O outro exemplo apresentado na Ilustração 30 possui um método opcional chamado "*enabled?(options)*", que recebe os mesmos parâmetros do método "*html(options)*". Este método, quando declarado, define se existe alguma condição para que a solução não possa ser executada. Caso não possa, o botão "Executar" em seu formulário estará desabilitado.

A classe interna *Form* é utilizada para chamar bibliotecas auxiliares do Ruby on Rails para montar componentes HTML.

As soluções podem ser reaproveitadas. Caso soluções criadas anteriormente para outras regras se apliquem à nova regra, elas também podem ser associadas.

Cadastrar regra no banco

Para cadastrar a nova regra no banco de dados, é necessário autenticar-se no Redmine com um usuário administrativo e entrar em "Administração/Regras/Nova Regra".

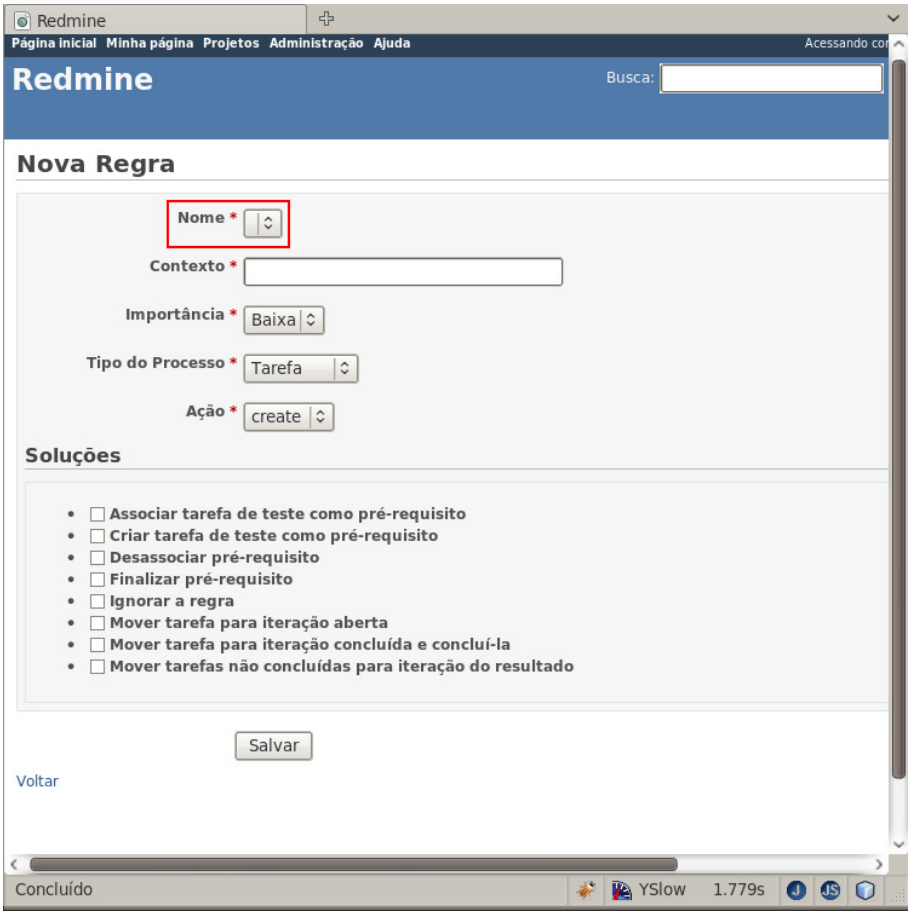

Ilustração 31 - Tela de Cadastro de Nova Regra

O primeiro campo **Nome** busca as classes de regras que se localizam na pasta “app/rules” que ainda não tiveram seus nomes cadastrados no banco. Na figura, o campo está vazio, pois não existem regras que não foram cadastradas no banco em questão.

O campo **Contexto** deve apresentar a mensagem de restrição da regra. Esta mensagem é exibida quando a regra retorna um valor falso do método “*valid?(obj)*”.

No campo **Tipo de Processo**, o usuário deve selecionar em qual entidade do sistema a regra irá agir. Até o momento as opções são: Tarefas e Resultados (seção 5.2). A regra é executada no ciclo de vida da entidade selecionada. E o objeto desta entidade sendo alterado é passado por parâmetro para os métodos “*valid?(obj)*”, “*html(options={})*” e “*create(options)*”.

No campo **Ação** o usuário seleciona sob qual momento do ciclo de vida do objeto a regra será aplicada. Seja na criação, atualização ou todos. Quando a ciclo de vida escolhido é a criação, a regra só é executada uma vez. Na atualização, ela é executada a cada *update* no objeto.

A lista de soluções apresenta quais soluções já foram cadastradas no sistema, permitindo que o usuário selecione quais desejar para que sejam apresentadas caso a regra seja inválida. Se o usuário criar novas soluções, elas devem ser cadastradas antes para que apareçam na lista.

Cadastrar solução no banco

A Ilustração 32 apresenta o cadastro de novas soluções.

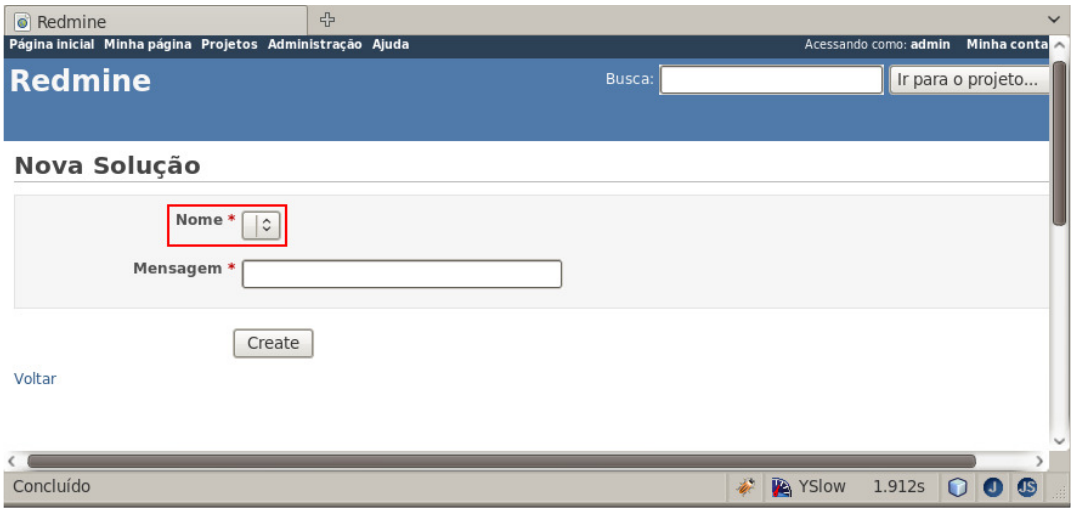

Ilustração 32 - Tela de Cadastro de Nova Solução

Assim como na tela de cadastro de regras, o campo **Nome** apresenta o nome das classes na pasta “app/rules/solutions/” que ainda não foram cadastradas no banco.

O campo **Mensagem** é apenas uma descrição da ação da solução, já que o nome da classe pode não ser muito intuitivo.

Cadastrar regra no projeto

Com a regra nova criada e cadastrada no banco, o usuário deve associá-la ao projeto de interesse. Para visualizar a tela de associação de regras, o usuário deve entrar em um projeto e selecionar em seu menu “Configurações / Regras”.

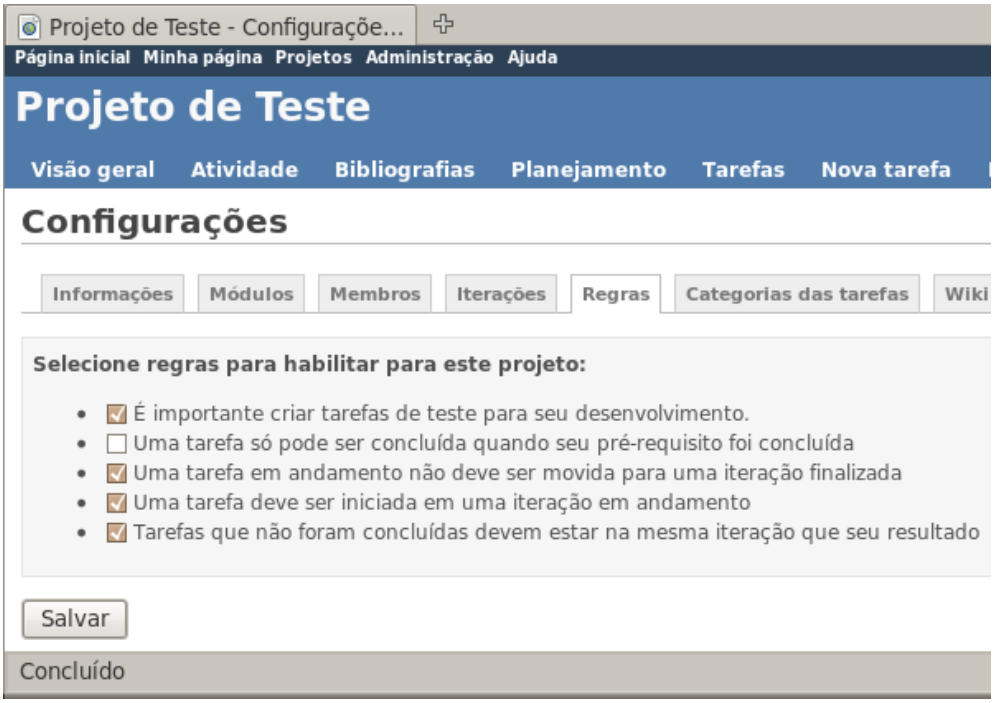

Ilustração 33 - Tela de Associação de Regras a um Projeto

Neste momento as regras “checadas” já estão associadas ao projeto e, dependendo das ações dos usuários, podem ser acessadas.

A Ilustração 34 apresenta uma regra em execução. O contexto da regra é exibido com dois formulários das duas soluções associadas à regra. O usuário seleciona uma das duas soluções, executando o método *create(options)* da classe da solução selecionada.

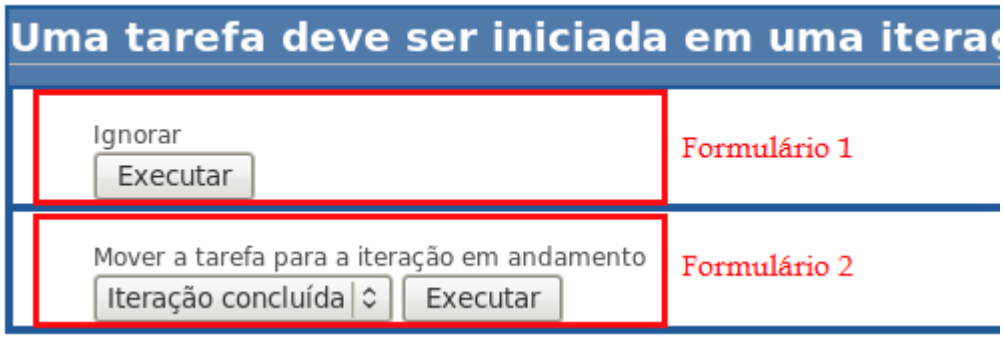

Ilustração 34 - Regra em Execução com duas Soluções

4.4. Regras Incompatíveis

Ao longo da criação das regras e dos projetos, é possível que surjam regras que possuem ações que entram em conflito dentro de uma transação. Esta combinação de regras não pode ser inserida no mesmo projeto.

Para tratar este requisito e, inspirado pela tabela de incompatibilidade de atividades no *AgentWork* (seção 7.1), foi criada uma tabela que mapeia a incompatibilidade entre as regras.

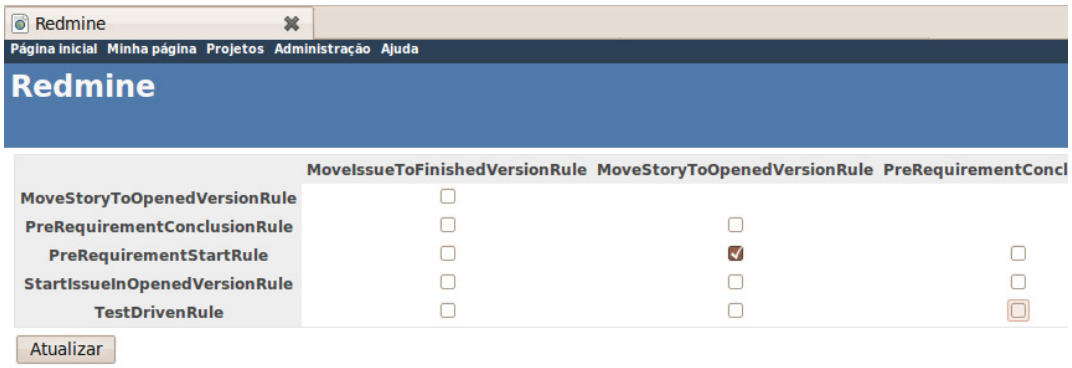

Ilustração 35 - Tabela de Incompatibilidade de Regras

As regras que estão mapeadas como incompatíveis, não podem ser incluídas no mesmo projeto.