engenharia de software magazine

Edição 90

DEVMEDIA

ISSN 1983127-7

Arquitetura de Software
Padrões arquiteturais em Ruby

MEDIÇÃO DE SOFTWARE

Uso do PSM: Practical Software Measurement na prática

Agilidade
Introdução ao uso de testes exploratórios

Planejamento
Saiba como implantar um PMO

Nesta seção você encontra artigos voltados para testes, processo, modelos, documentação, entre outros

Padrões arquiteturais em Ruby

Preservando a arquitetura de sistemas de software no decorrer do seu desenvolvimento

Fique por dentro:

A utilização de padrões e a adoção de boas práticas são recomendações estabelecidas por um projeto arquitetural bem definido. Contudo, no decorrer do projeto, esses padrões tendem a se degradar, fazendo com que seus benefícios sejam anulados. Esse problema é ainda mais severo em sistemas desenvolvidos em linguagens dinâmicas. Assim, este artigo tem como objetivo demonstrar o uso de ArchRuby – uma ferramenta que provê conformidade e visualização arquitetural – para expressar padrões e boas práticas arquiteturais que, uma vez preservados, contribuem para a manutenibilidade do sistema de software.

Sérgio Henrique Mirada

sergio.miranda@dcc.ufmg.br

sergiohenriquemiranda.com.br

Mestre em Ciência da Computação pela Universidade Federal de Minas Gerais, possui mais de oito anos de experiência em desenvolvimento de sistemas, atuando como líder técnico na Dito Internet por seis anos. Sérgio ainda possui larga experiência em linguagens dinâmicas e em projetos arquiteturais voltados à escalabilidade de sistemas de grande porte. Nos dias atuais Sérgio é responsável pela área de Tecnologia da Informação da empresa Coffee & Joy (coffeeandjoy.com.br).

Ricardo Terra

terra@dcc.ufla.br - www.dcc.ufla.br/~terra

Ricardo Terra é doutor em Ciência da Computação pela Universidade Federal de Minas Gerais (UFMG) com período sanduíche na University of Waterloo (UWaterloo) e pós-doutorado no INRIA/Université Lille 1. Desde fevereiro de 2014, é professor adjunto do Departamento de Ciência da Computação (DCC) da Universidade Federal de Lavras (UFLA). É professor permanente do Programa de Pós-Graduação em Ciência da Computação (PPGCC) atuando em Arquitetura de Software e Qualidade de Software. Trabalhou cinco anos como arquiteto Java e hoje coordena o Grupo Java do DCC/UFLA.

A arquitetura de software é geralmente definida como um conjunto de decisões de projeto que tem impacto em cada aspecto da construção e evolução do sistema de software. Isso implica em definições de diversos padrões e boas práticas que devem ser seguidas. No entanto, no decorrer do projeto, esses padrões tendem a se degradar fazendo com que benefícios proporcionados por um projeto arquitetural (e.g., manutenibilidade, escalabilidade, portabilidade) sejam anulados. Esse problema é ainda mais severo em sistemas desenvolvidos em linguagens dinâmicas, onde recursos como invocação dinâmica, construções dinâmicas, eval, etc., tornam os desenvolvedores mais propícios a quebrar a arquitetura planejada.

Diante disso, este artigo demonstra que a preservação de padrões e boas práticas arquiteturais – o que contribui para o processo de manutenção do sistema – pode ser facilmente realizado em sistemas desenvolvidos em linguagens dinâmicas. Para isso, é utilizado um sistema motivador, desenvolvido com a linguagem Ruby, cuja arquitetura contempla diversos padrões e boas práticas arquiteturais comumente encontrados no desenvolvimento de sistemas. Assim, é demonstrado como preservar tais padrões utilizando ArchRuby, onde arquitetos de software definem regras arquiteturais para controlar as dependências que podem ou não ser estabelecidas em

um dado sistema. Mais especificamente, ArchRuby é uma ferramenta que provê conformidade e visualização arquitetural baseada em técnicas de análise estática de código e uma heurística de inferência de tipos para sistemas Ruby.

O restante desse artigo está organizado conforme descrito a seguir. A seção *Arquitetura de Software* faz uma breve introdução a projetos arquiteturais de sistemas de software e a seção *Conformidade e Visualização Arquitetural* descreve as principais técnicas existentes. A seção *Ferramenta ArchRuby* apresenta a solução de conformidade e visualização arquitetural proposta. A seção *Sistema Motivador* descreve o sistema utilizado para ilustrar os padrões e boas práticas arquiteturais. A seção *Regras Arquiteturas e Benefícios para Manutenibilidade* demonstra como traduzir tais padrões em regras arquiteturais e os benefícios associados a tais regras. A seção *Processo de Conformidade Arquitetural* aplica o processo de conformidade arquitetural no sistema motivador e a seção *Visualização da Arquitetura* apresenta visualizações que facilitam o raciocínio arquitetural.

Arquitetura de software

A arquitetura de software tem como responsabilidade entender e projetar a melhor maneira que as diversas partes que compõem um software devem ser estruturadas entre si. Ou seja, é uma ligação crítica existente entre a descrição (e.g., requisitos do software) e o projeto de software. Dessa maneira, o projeto arquitetural é um artefato importante, pois as definições presentes nesse artefato afeta o desempenho, a robustez, a portabilidade, a manutenibilidade, etc. do sistema de software.

Contudo, no decorrer do projeto – devido à falta de conhecimento, prazos curtos, etc. – os padrões definidos pela arquitetura tendem a se degradar. Isso faz com que os benefícios proporcionados por um projeto arquitetural sejam anulados. Esse fenômeno é conhecido como erosão arquitetural e deve ser controlado para não afetar o projeto arquitetural.

Conformidade e visualização arquitetural

Conforme demonstrada anteriormente, a arquitetura é um artefato crucial que precisa ser monitorado e seguido pelos desenvolvedores de software. Conformidade arquitetural é o processo que verifica o grau de consistência entre a arquitetura concreta (i.e., implementação concreta do código fonte) e a planejada. O processo de conformidade arquitetural pode ser estático (i.e., sem executar o sistema alvo) ou dinâmico (i.e., executando o sistema alvo). Esta seção apresenta algumas técnicas de conformidade arquitetural e visualização:

• **Modelos de Reflexão:** essa técnica compara dois modelos: um representando o modelo de alto nível do sistema (e.g., especificado pelo desenvolvedor) e outro representando a implementação concreta do sistema (e.g., produzido por ferramentas de análise estática do código fonte ou coletado durante a execução do sistema). Após a comparação entre os dois modelos é gerado o modelo de reflexão, revelando possíveis divergências e ausências. Divergências indicam interações no código fonte que não são permitidas pelo modelo de alto nível da arquitetura, já ausências indicam interações que são esperadas que acontecessem, embora não tenham sido encontradas;

• **Matriz de Dependência Estrutural (DSM):** o conceito de matriz de dependência foi primeiramente proposto por Baldwin e Clark para mostrar a importância do design modular na indústria de hardware. Depois disso, outros autores reiteraram que DSM poderia também ser utilizada na indústria de software. Uma DSM é uma matriz quadrada onde linhas e colunas representam módulos de um sistema. A ferramenta LDM, por exemplo, representa nas células o número de dependências existentes entre dois módulos. Nessa ferramenta, é possível utilizar regras de *design* para diferenciar dependências de duas formas: A *can-use* B e A *cannot-use* B, indicando que o módulo A pode (ou não) depender do módulo B;

• **Linguagens de restrições:** o objetivo principal de linguagens de restrições é prover uma maneira de especificar dependências estruturais. DCL (*Dependency Constraint Language*) é uma linguagem de domínio específico que suporta a definição de restrições estruturais entre módulos. Em outras palavras, DCL permite a arquitetos de software restringir o espectro de dependências que podem ser estabelecidas em um dado sistema. Para isso, DCL provê restrições para capturar divergências e ausências. Para capturar divergências os arquitetos devem utilizar regras *only can*, *can only* ou *cannot*. Já ausências são detectadas através de regras *must*.

Ferramenta ArchRuby

ArchRuby é uma ferramenta que realiza conformidade arquitetural em sistemas desenvolvidos em linguagens dinâmicas. Mais especificamente, ArchRuby foi desenvolvida para sistemas Ruby. Para atender as particularidades existentes em linguagens dinâmicas, a ferramenta utiliza uma heurística de propagação de tipos. O objetivo principal é detectar violações que representam anomalias arquiteturais e que, portanto, contribuem para a erosão da arquitetura planejada de um sistema. A **Figura 1** apresenta uma visão geral da ferramenta.

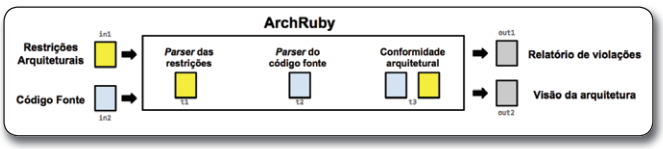


Figura 1. Ferramenta ArchRuby

A ferramenta recebe como entrada o arquivo de definição das restrições arquiteturais (**in1**) e o código fonte do sistema alvo (**in2**). A ferramenta realiza o *parser* das restrições arquiteturais (**t1**) e do código fonte (**t2**). Assim, o processo de conformidade arquitetural (**t3**) detecta pontos do código fonte que não respeitam a arquitetura planejada do sistema alvo. Como resultado, a ferramenta gera um relatório textual (**out1**) que detalha as violações detectadas (localização no código fonte, regra violada, etc.), e dois modelos de alto nível da arquitetura, um baseado em modelos de reflexão e outro em DSM, para melhor visualizar as violações identificadas (**out2**). As dependências – arestas no modelo de reflexão e células no matriz de

dependências – que representam as violações são destacadas. Em seguida é explicado o que é considerado divergência e ausência para ArchRuby. Além disso, é demonstrado também como os módulos, juntamente com suas restrições, devem ser definidos. Por fim, é exemplificado o funcionamento da ferramenta em um sistema motivador.

• **Divergências:** para detectar divergências – dependências que existem no código fonte mas não são prescritas pela arquitetura planejada – para cada modulo nós definimos aqueles que são permitidos depender (*allowed*) ou não (*forbidden*);

• **Ausências:** de forma similar, para detectar ausências – dependências que não existem no código fonte, mas são obrigatórias pela arquitetura planejada – para cada módulo nós definimos aqueles que são obrigatórios (*required*);

• **Restrições arquiteturais:** são especificadas em uma linguagem de domínio específico em formato YAML, amplamente utilizada no ecossistema Ruby. Mais especificamente, cada módulo do sistema deve ser especificado como a seguir:

```
<id_módulo>:
 (files | gems):'<padrão> {, <padrão>}'
 [(allowed | forbidden):'<id_módulo> {, <id_módulo>}']
 [(required):'<id_módulo> {, <id_módulo>}']
```

onde **<id_módulo>** é o nome do módulo que está sendo especificado. Módulos podem ser compostos por arquivos (*files*) ou Gems (*gems*) e devem ser definidos por pelo menos um **<padrão>**, delimitado por vírgulas. Basicamente, em **padrão** pode-se utilizar *shell glob* (disponível na biblioteca padrão de arquivos da linguagem Ruby) para assinalar múltiplos arquivos. Em resumo, para definir um módulo, é necessário definir seu nome, quais arquivos ou *gems* fazem parte do módulo, e quais são suas restrições arquiteturais;

• **Interface de utilização:** a ferramenta é um Gem para Ruby. Sua execução é feita por meio de linha de comando. Dessa maneira, qualquer fábrica de software pode incorporá-la ao seu processo de desenvolvimento independentemente do ambiente utilizado. O comando a seguir ilustra sua utilização:

```
archruby --arch_def_file=<caminho_arquivo_arquitetural> --app_root_
path=<caminho_sistema_alvo>
```

Sistema motivador

Para ilustrar padrões e boas práticas arquiteturais comumente aplicadas no desenvolvimento de sistemas Ruby, utiliza-se um sistema denominado TechVideos que automatiza a busca de vídeos na Internet relacionados a tecnologias atuais. Basicamente, um usuário pode realizar seu *login* utilizando sua conta do GitHub, marcar um vídeo como favorito, inserir observações nos vídeos assistidos e gerar um relatório em formato PDF com todos os vídeos já assistidos juntamente com suas observações. Por se tratar de um sistema web, o *framework* Rails é utilizado para dar suporte ao desenvolvimento.

A arquitetura do sistema TechVideos segue o padrão arquitetural Model-View-Controller (MVC), conforme ilustrado na

Figura 2. A camada de visão pode acessar serviços oferecidos pela camada de controle. Consequentemente, a camada de controle utiliza serviços fornecidos pela biblioteca *ActionController* (o qual faz parte do *framework* Rails) e é composta por classes responsáveis por intermediar a comunicação entre a camada de visão e de modelo. Já a camada de modelo contém objetos de negócio, objetos de transferência de dados e objetos de acesso a dados, e também faz a utilização do *framework ActiveRecord* para a interação com o banco de dados.

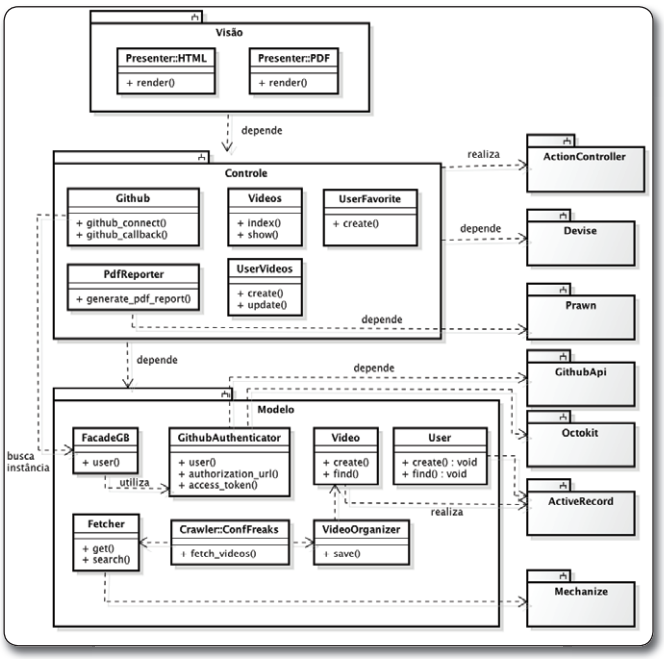


Figura 2. Arquitetura do sistema TechVideos

Regras arquiteturas e seus benefícios para manutenibilidade

Nesta seção, utiliza-se a arquitetura do sistema TechVideos para ilustrar diversos padrões e boas práticas arquiteturais, os quais serão (i) brevemente descritos, (ii) traduzidos em regras arquiteturais especificadas em ArchRuby e (iii) relacionados os seus principais benefícios em termos de manutenibilidade.

Arquitetura em camadas

A arquitetura em camada é organizada com o intuito de camadas superiores utilizarem serviços fornecidos por camadas imediatamente inferiores. Em outras palavras, suponha uma arquitetura dividida em três camadas onde a camada M0 é a camada mais inferior e a camada M2 é a camada mais superior. Se a camada M2 estabelecer dependência com a camada M0, estará violando o padrão arquitetural subjacente. Para especificar que o sistema TechVideos adota uma arquitetura em camadas, especifica-se a seguinte definição para o módulo **view**:

```
1 view:
2 files:'app/views/**/*.rb, app/presenters/**/*.rb'
3 forbidden:'model'
```

Nas linhas 1 e 2, informa-se o nome do módulo e especifica-se os arquivos que o compõem. Na linha 3, são indicadas as restrições arquiteturais desse módulo. Mais especificamente, o módulo está proibido de estabelecer dependência com o módulo **model**, o qual refere-se à camada de modelo.

Benefícios para a manutenibilidade: a restrição colocada garante que a arquitetura em camada não seja violada, uma vez que o modulo **view** não está autorizado a acessar o módulo **model**. Isso torna o sistema mais fácil de manter, pois ArchRuby garante que modificações na camada de modelo não afetem a camada de visão. Em outras palavras, se for preciso realizar manutenção para trocar toda a camada de modelo, a camada de visão não sofrerá impacto.

Padrões estruturais

Sistemas orientados a objetos se baseiam na interação entre diversos objetos. Contudo, a interação entre eles pode se tornar complicada, por exemplo, algumas interfaces podem exigir parâmetros complexos de serem obtidos. Nesse sentido, o padrão de projeto *Facade* simplifica a interação entre objetos provendo uma interface que oculta detalhes de implementação, facilitando a interação entre classes clientes e classes provedora de serviços. Para demonstrar esse padrão, os módulos **github_controller** e **crawler** são definidos conforme a **Listagem 1**.

Listagem 1. Definição dos módulos github_controller e crawler.

```
1 github_controller:
2  files: 'app/controllers/github_controller.rb'
3  allowed: '..., gb_facade, ...'
4
5 crawler:
6  files: 'app/crawler/**/*.rb'
7  allowed: '..., video_organizer, ...'
```

Nas linhas 1-2 e 5-6 são definidos o nome e quais arquivos definem os módulos. Na linha 3 são indicadas as restrições arquiteturais. Permite-se que o módulo **github_controller** acesse o módulo **gb_facade**, que implementa o padrão *Facade* para acessar a classe que provê acesso a dados obtidos do GitHub. De forma similar, na linha 7, permite-se que módulo **crawler** acesse o módulo **video_organizer**, que implementa o padrão *Facade* para interagir com objeto responsável por manter os vídeos salvos.

Benefícios para a manutenibilidade: a preservação dessas restrições garante uma comunicação simplificada entre classes que utilizam funcionalidades que são providas por outras classes, facilitando o entendimento e manutenção do sistema. Além disso, classes clientes não serão afetadas se for preciso alterar classes utilizadas pelos módulos que implementam o padrão *Facade*. Certamente, ArchRuby garante que as classes clientes não dependam de tais classes, pois isso caracterizaria uma violação arquitetural.

Segregação de Interface

Em sistemas orientados a objetos, é comum observar objetos que dependem de outros que fornecem interfaces com muitas funcionalidades. É preferível estabelecer dependência com interfaces simples e especializadas. A segregação de interface relata exatamente o que foi mencionado anteriormente. Para atender a segregação de interface, o módulo **fetcher** é definido como a seguir:

```
1 fetcher:
2  files: 'lib/fetcher.rb'
3  allowed: 'mechanize'
```

Essa definição faz com que o módulo **fetcher** esteja permitido a acessar apenas o módulo **mechanize**. Dessa forma, ArchRuby garante que módulos do sistema que necessitarem de realizar acesso ou *parser* de *sites* externos devem depender do módulo **fetcher**.

Benefícios para a manutenibilidade: a interface oferecida pelo módulo **mechanize** é extensa e bastante genérica. O módulo **fetcher** oferece uma interface mais especializada, i.e., com um menor número de métodos. Isso faz com que os clientes do módulo **fetcher** dependam de uma interface enxuta. Além disso, o sistema fica mais fácil de ser mantido e atualizado, caso seja necessário trocar a biblioteca que oferece acesso a *sites*, basta dar manutenção no módulo **fetcher**. Em outras palavras, ArchRuby garante que, nesse cenário, o módulo **fetcher** será o único afetado e seus clientes não terão impacto.

Responsabilidade única

Uma boa prática em sistemas orientados a diz respeito a responsabilidades que objetos possuem. É desejável que um objeto possua apenas uma única responsabilidade, i.e., uma classe deve possuir apenas uma única razão para ser alterada. Para ilustrar o princípio de única responsabilidade, o módulo **user** é definido como a seguir:

```
1 user:
2  files: 'app/models/user.rb'
3  required: 'activerecord'
```

Essa definição obriga o módulo **user** a estabelecer alguma forma de dependência com o módulo **activerecord**. Mais importante, como não foram especificados os módulos com o qual **user** pode estabelecer dependência, qualquer dependência estabelecida que não seja com **activerecord** será considerada uma violação arquitetural.

Benefícios para a manutenibilidade: a preservação dessa restrição garante que o módulo user possua somente uma única responsabilidade: gravar dados dos usuários no banco de dados. Assim, esse módulo possui apenas uma única razão para ser alterado: quando for necessário substituir ou modificar o módulo activerecord. ArchRuby então preserva um módulo mais robusto, uma vez que é focado em apenas uma tarefa específica.

Controle de dependências externas

Atualmente, sistemas web tendem a se acoplar a diversos recursos externos, seja através da utilização de *frameworks*

ou bibliotecas que proveem acesso a APIs. No entanto, esse acoplamento deve ser controlado de forma que o acesso a *frameworks* e APIs externas seja feito somente por classes específicas. Para ilustrar o controle de dependências externas, os módulos **github_authenticator** e **pdf_generator** são definidos como na **Listagem 2**.

Listagem 2. Exemplo de controle das dependências externas – definição de módulos.

```
1 github_authenticator:
2  files: 'lib/github_authenticator.rb'
3  allowed: 'github_api, github_octokit'
4
5 pdf_generator:
6  files: 'lib/pdf_generator.rb'
7  allowed: 'pdf'
```

Essas definições estipulam, respectivamente, que o módulo **github_authenticator** pode acessar as bibliotecas que fornecem acesso à API do GitHub e que o módulo **pdf_generator** pode acessar o módulo que fornece uma biblioteca para manipular arquivos em formato PDF.

Benefícios para a manutenibilidade: a definição dessas restrições garante que o acoplamento com os módulos externos ao sistema esteja restrito a módulos específicos. Isso aumenta a robustez e facilita a manutenibilidade do sistema. Portanto, ArchRuby garante que o acesso a funcionalidades externas esteja centralizado em classes específicas, e não espalhado por todo o sistema.

Processo de conformidade arquitetural

Nesta seção, utiliza-se as restrições arquiteturais previamente especificadas para o sistema TechVideos para ilustrar o processo de conformidade arquitetural realizado pela ferramenta ArchRuby. É importante mencionar que foram propositalmente introduzidas violações arquiteturais para melhor ilustrar o reporte de violações.

Detectando divergências

É importante ressaltar que as classes pertencentes a camada de visão não devem acessar classes da camada de modelo. Dessa forma, uma divergência arquitetural foi propositalmente introduzida na classe **Presenters::PDF** (pertencente a camada de visão), cujo código fonte pode ser observado na **Listagem 3**.

A classe **PDF** define o método **render** que tem a função de preencher a variável de instância **data**. Esse código aparenta não violar a regra arquitetural em questão, i.e., não parece acessar classes de modelo. No entanto, a ferramenta ArchRuby possui, além de análise estática convencional, uma heurística de inferência de tipos que permitiu ArchRuby identificar que a variável **collection** (que está sendo passada para o método **render** na linha 4) pode ser do tipo **UserVideo**, classe pertencente à camada de modelo. Essa heurística analisa todas as classes do sistema que invocam o método **render**, como a classe **PdfReportController** (vide **Listagem 4**).

Listagem 3. Exemplo de divergência arquitetural.

```
01 Module Presenters
02  class PDF
03   ...
04   def render(collection)
05    @data << "Videos visualizados"
06     collection.each do | collection |
07     @data << "#{collection.video.title}: #{collection.observation}"
08    end
09    @data
10   end
11  ...
12  end
13 end
```

Listagem 4. Código da classe PdfReportController.

```
01 class PdfReportController < ApplicationController
02  def generate_pdf_report
03    pdf = PdfGenerator.new
04    collection = UserVideo.where(user_id: current_user.id)
05    texts = Presenters::PDF.new.render(collection)
06    send_data pdf.render(texts), type: "application/pdf", disposition: "inline"
07  end
08 end
```

Essa classe define o método **generate_pdf_report** que atribui um objeto da classe **UserVideo** à variável **collection** (linha 4). Mais importante é que essa variável é utilizada como argumento na chamada ao método **render** da classe **Presenters::PDF**. Isso configura uma divergência arquitetural, já que as classes da camada de visão não podem depender de classes da camada de modelo. Como consequência, ArchRuby reporta essa anomalia arquitetural no relatório textual gerado pela ferramenta, como demonstra a **Listagem 5**.

Listagem 5. Descrição do ArchRuby informando anomalia arquitetural.

```
01 divergence:
02   class_origin: Presenters::PDF
03   line_origin: 04
04   class_target: UserVideo
05   module_origin: view
06   module_target: user_video
07   constraint: module 'view' cannot depend on module 'user_video'
```

Basicamente, o relatório indica o tipo de violação detectada (linha 1), a classe de origem (linha 2), a linha de código onde a violação foi encontrada (linha 3), a classe que está sendo acessada de forma não autorizada (linha 4), os módulos de origem e destino (linhas 5-6) e uma mensagem explicando a restrição violada (linha 7).

Detectando ausências

É importante ressaltar que as classes pertencentes à camada de modelo devem obrigatoriamente estabelecer dependência com a classe ActiveRecord. Dessa forma, ausências foram propositalmente causadas na classe **User** (pertencente à camada de modelo), cujo código fonte pode ser observado a seguir:

```
01 class User
02  devise :database_authenticatable, :registerable,
      :recoverable, :rememberable, :trackable, :validatable
03 end
```

A classe **User** é responsável por armazenar os dados dos usuários no banco de dados. Embora essa classe deva depender do *framework* de persistência de dados ActiveRecord (módulo **activerecord**), tal dependência não está sendo estabelecida. Isso denota uma ausência arquitetural, uma vez que as definições arquiteturais obrigam o módulo da classe **User** estabelecer dependência com o módulo **activerecord**. Como consequência, ArchRuby reporta essa anomalia arquitetural no relatório textual gerado pela ferramenta (vide **Listagem 6**).

O reporte de uma ausência é similar ao reporte de uma divergência, conforme pode ser observado na seção anterior. No entanto, no caso de ausências a linha de origem e a classe de origem (linhas 3-4) não são preenchidas, uma vez que não existe a dependência.

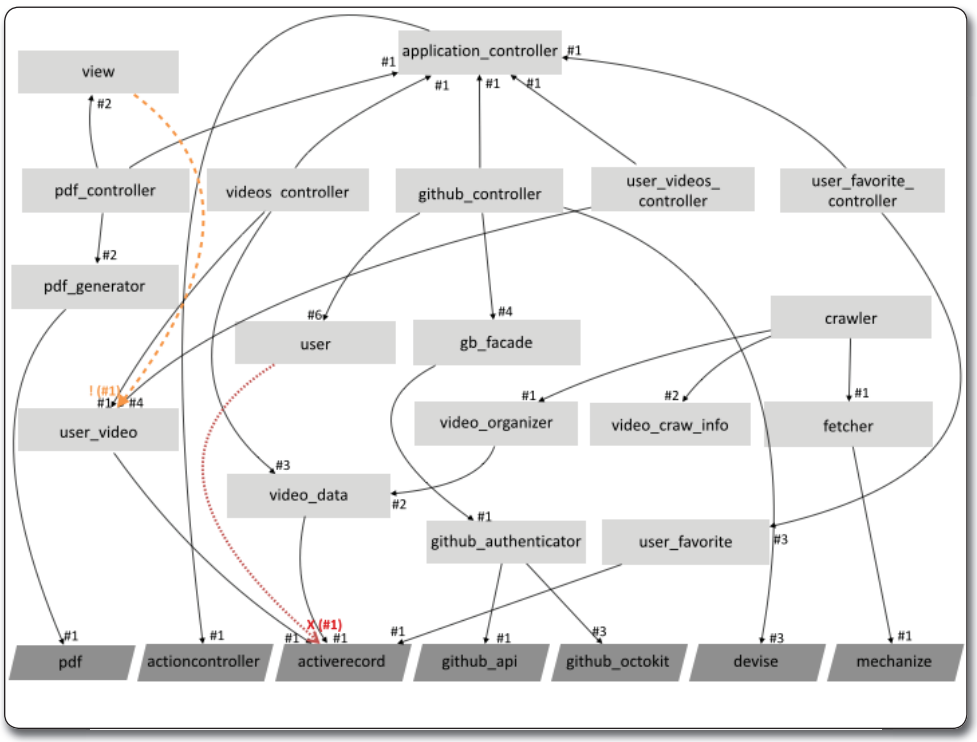


Figura 3. Modelo de Reflexão gerado automaticamente por ArchRuby

Visualização da arquitetura

No intuito de prover uma melhor forma de reporte das violações detectadas, ArchRuby complementa o relatório textual com duas visualizações de alto nível da arquitetura: uma baseada em Modelos de Reflexão e outra em Matrizes de Dependência Estrutural (DSMs). A **Figura 3** ilustra o modelo do sistema TechVideos automaticamente gerado pela ferramenta ArchRuby, o qual é baseado em uma adaptação sutil do modelo de reflexão proposto por Murphy. Nessa visualização, os vértices retangulares na cor cinza claro representam módulos internos do sistema (e.g., **github_controller**) e vértices em trapézio na cor cinza escuro representam módulos externos ao sistema (e.g., **activerecord**). As arestas representam conexões existentes entre os módulos, as quais são diferenciadas quando se trata de violações arquiteturais. Por exemplo, divergências são representadas por uma aresta tracejada na cor laranja, que pode ser visualizada entre os módulos **view** e **user_video**. Já ausências são representadas por arestas pontilhadas na cor vermelha, que pode ser visualizada entre os módulos **video_data** e **activerecord**. Esse modelo de alto nível auxilia os arquitetos a refletirem sobre a arquitetura concreta do sistema. Porém, por ser baseado em grafos, modelos de reflexão apresentam problemas de escalabilidade à medida que o sistema cresce.

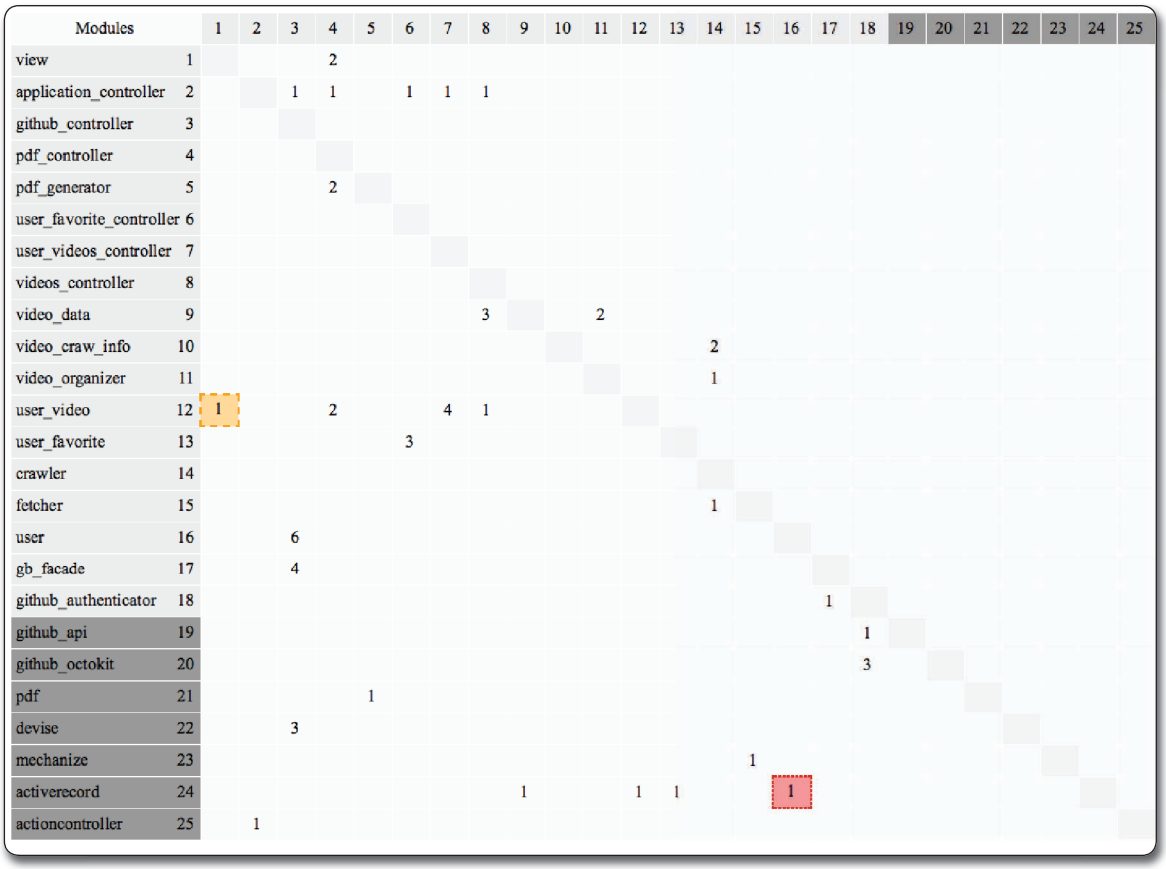
Para resolver o problema de escalabilidade, a ferramenta ArchRuby também gera uma saída baseada em Matrizes de Dependência Estrutural (DSM). A **Figura 4** ilustra a DSM do sistema TechVideos automaticamente gerada pela ferramenta ArchRuby.

Listagem 6. Descrição do ArchRuby informando mais uma anomalia arquitetural.

```
01 absence:
02   class_origin: User
03   line_origin:
04   class_target:
05   module_origin: user
06   module_target: activerecord
07   constraint: not implementing a required module
```

Nessa visualização, linhas e colunas na cor cinza claro representam módulos internos do sistema enquanto que, em cinza escuro, representam módulos externos. Já as células, representam a quantidade de referências existentes entre os módulos, e são diferenciadas quando se trata de violações arquiteturais. Células com bordas laranjas tracejadas representam divergências arquiteturais. Por exemplo, é possível notar uma divergência entre o módulo **view**, pertencente à camada de visão, e o módulo **user_video** (pertencente a camada de modelo). Já células com bordas vermelhas pontilhadas representam ausências arquiteturais. Por exemplo, existe uma ausência arquitetural entre o módulo **user**, pertencente à camada de modelo, e o módulo **activerecord** (*framework* de persistência de dados).

O uso de padrões e a adoção de boas práticas sempre foi recomentado no desenvolvimento de software. Contudo, no decorrer do projeto, esses padrões tendem a se degradar, fazendo com que seus benefícios sejam anulados. Nesse contexto, a tarefa de manutenção de software torna-se ainda mais árdua.



Modules	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
view 1				2																					
application_controller 2			1	1		1	1	1																	
github_controller 3																									
pdf_controller 4																									
pdf_generator 5				2																					
user_favorite_controller 6																									
user_videos_controller 7																									
videos_controller 8																									
video_data 9								3			2														
video_craw_info 10														2											
video_organizer 11														1											
user_video 12	1			2			4	1																	
user_favorite 13						3																			
crawler 14																									
fetcher 15														1											
user 16			6																						
gb_facade 17			4																						
github_authenticator 18																	1								
github_api 19																		1							
github_octokit 20																		3							
pdf 21					1																				
devise 22			3																						
mechanize 23															1										
activerecord 24									1			1	1			1									
actioncontroller 25		1																							

Figura 4. DSM gerada automaticamente pelo ArchRuby

Diante disso, a partir de um sistema motivador cuja arquitetura engloba diversos padrões e boas práticas arquiteturais, foi demonstrado como utilizar a ferramenta ArchRuby para preservar tais padrões durante todo o ciclo de desenvolvimento do sistema de software. Além disso, foram apresentados os benefícios que a conformidade arquitetural, i.e., a não violação da arquitetura planejada, traz para a manutenibilidade do sistema.

Por fim, foram ilustrados dois modelos de alto nível da arquitetura implementada do sistema – os quais são gerados automaticamente pela ferramenta ArchRuby – que têm o intuito de prover uma visualização gráfica das violações arquiteturais detectadas. Essas visualizações podem inclusive apoiar novos membros da equipe de desenvolvimento a entender como o sistema está estruturado.

Links e Referências:

Página do projeto ArchRuby
http://aserg.labsoft.dcc.ufmg.br/archruby/manual_pt.html

[1] David Lorge Parnas. Software aging. Em 16th International Conference on Software Engineering (ICSE), páginas 279–287, 1994.

[2] Sergio Miranda, Marco Tulio Valente e Ricardo Terra. ArchRuby: Conformidade e visualização arquitetural em linguagens dinâmicas. In VI Congresso Brasileiro de Software: Teoria e Prática (CBSoft), Sessão de Ferramentas, páginas 17-24, 2015.

[3] Sergio Miranda, Marco Tulio Valente e Ricardo Terra. Architecture conformance checking in dynamically typed languages. Journal of Object Technology, páginas 15(3):1-34, 2016.

[4] Leonardo Passos, Ricardo Terra, Renato Diniz, Marco Tulio Valente e Nabor Mendonça. Static architecture-conformance checking: An illustrative overview. IEEE Software, 27(5):82–89, 2010.

[5] Gail Murphy, David Notkin e Kevin Sullivan. Software reflexion models: Bridging the gap between source and high-level models. Em 3rd Symposium on Foundations of Software Engineering (FSE), páginas 18–28, 1995.

[6] Ricardo Terra e Marco Tulio Valente. A dependency constraint language to manage object-oriented software architectures. Software: Practice and Experience, 32(12):1073–1094, 2009.

[7] Robert Martin. Agile Software Development: Principles, Patterns, and Practices. Pearson, 2002.