UNIVERSIDADE FEDERAL DE ITAJUBÁ

PROGRAMA DE PÓS-GRADUAÇÃO EM
CIÊNCIA E TECNOLOGIA DA COMPUTAÇÃO

APLICAÇÃO DA TÉCNICA DE SCAFFOLDING PARA A CRIAÇÃO DE SISTEMAS CRUD

Danillo Goulart Magno

UNIFEI
Itajubá
Setembro, 2015

UNIVERSIDADE FEDERAL DE ITAJUBÁ
PROGRAMA DE PÓS-GRADUAÇÃO EM
CIÊNCIA E TECNOLOGIA DA COMPUTAÇÃO

Danillo Goulart Magno

APLICAÇÃO DA TÉCNICA DE SCAFFOLDING PARA A CRIAÇÃO DE SISTEMAS CRUD

Dissertação de Mestrado submetida ao Programa de Pós-Graduação em Ciência e Tecnologia da Computação como parte dos requisitos para obtenção do Título de Mestre em Ciência e Tecnologia da Computação.

Área de Concentração: Sistemas de Computação

Orientador: Prof. Dr. Enzo Seraphim

UNIFEI
Itajubá
Setembro, 2015

Dados Internacionais de Catalogação na Publicação (CIP)

Magno, Danillo Goulart.
M198a Aplicação da Técnica de Scaffolding para a Criação de Sistemas CRUD / Danillo Goulart Magno. – Itajubá: UNIFEI, 2015.
xxii + 102 f. : il. ; 29.7

Orientador: Enzo Seraphim.
Dissertação (Mestrado em Ciência e Tecnologia da Computação) – Universidade Federal de Itajubá, Itajubá, 2015.

1. Scaffolding. 2. CRUD. 3. Metaprogramação. 4. Templates. 5. Gerador de Código-fonte. I. Seraphim, Enzo, orient. II. Universidade Federal de Itajubá. III. Título.

CDU 004.4'2

Exemplar correspondente à versão final do texto, após a realização da defesa, incluindo, portanto, as devidas considerações dos examinadores.

Typeset in pdfLaTeX

Folha de Aprovação

Dissertação **aprovada** pela Banca Examinadora em 29 de Setembro de 2015, conferindo ao autor o título de **Mestre em Ciência e Tecnologia da Computação**.

Dr. Enzo Seraphim _____

Orientador

Universidade Federal de Itajubá

Dr^a. Célia Leiko Ogawa Kawabata _____

Membro da Banca

Instituto Federal de São Paulo, Campus São Carlos

Dr. Edmilson Marmo Moreira _____

Membro da Banca

Universidade Federal de Itajubá

Felicidade?
Disse o mais tolo: "Felicidade não existe".
O intelectual: "Não no sentido lato".
O empresário: "Desde que haja lucro".
O operário: "Sem emprego, nem pensar".
O cientista: "Ainda será descoberta".
O místico: "Está escrito nas estrelas".
O político: "Poder".
A igreja: "Sem tristeza, impossível. Amém".
O poeta riu de todos, e, por alguns minutos, foi feliz.

Sérgio Vaz

A todos que me fazem mais feliz.

Agradecimentos

Esta seção pode ser traiçoeira se os devidos cuidados não forem tomados. Uma palavra errada e pessoas podem se achar menos queridas do que as outras, ou pior, se acharem esquecidas por não serem mencionadas aqui. Tantos a agradecer (família, amigos, educadores, internet, instituição, agências de fomento, etc) em um espaço tão limitado quanto as folhas de um papel. Eu gostaria de deixar claro que sou imensamente grato a todos que de alguma forma contribuíram para o meu sucesso. Apesar disso, alguns nomes merecem ser citados. Minhas sinceras desculpas caso eu tenha esquecido de alguém, não foi por querer.

Obviamente começarei com a base de toda a minha essência nesta vida fugaz: Deus. Ele que me manteve firme até hoje. Meus agradecimentos *Àquele* que compreende meus temores e anseios. Inspira paz e solenidade na *Sua* infinita bondade. Obrigado.

Bondosos e humildes, atenciosos e compreensivos; gostaria de agradecer a meus pais *Romilo* e *Ivone* pela enorme paciência durante todos esses anos de espera. Garanto que farei valer cada sacrifício que vocês fizeram por mim. Amo vocês. Dedico este mestrado a vocês. Obrigado por tudo.

Preciso agradecer também a meus queridos irmãos pela imensa ajuda durante este percurso. *Aninhakizy92magno*, com seus cookies gostosos mas nada agradáveis visualmente. Usava todo seu conhecimento culinário para fornecer guloseimas para a família... pelo menos você tentava, obrigado. Lutador, educador físico, atleta, youtuber, andarilho de slackline. Amado por todos da família. Gostaria de agradecer também ao queridinho. Respeito e gratidão a você meu irmão *Egmagnofitness*. Justo e solidário por ceder seu notebook para uma boa causa, obrigado por essa e várias outras ajudas.

Obrigado aos meus amigos de todas as horas: os integrantes da *família rep'n'roll*. Serão sempre considerados minha segunda família. Estiveram sempre ao meu lado, nos momentos bons e ruins. Muito animados e dispostos a passar o ano todo estudando, principalmente se for ano de copa do mundo e eleições. Obtive valores que servirão por toda minha vida. Lembrarei de todos vocês e voltarei sempre que possível.

Habilidoso cientista da computação e grande amigo, gostaria de deixar aqui meu agradecimento especial ao professor *Enzo*. Inspira o melhor das pessoas e as tranquiliza em momentos de aflição ou descrédito. Responsável pela orientação deste trabalho, ele me forneceu toda a ajuda que eu precisei durante o percurso.

Agradeço a todos os professores que auxiliaram a minha formação acadêmica durante este mestrado. Sempre prontos a ajudar e esclarecer dúvidas. Seus esforços e dedicação facilitaram meu processo de aprendizado durante o curso.

Eu gostaria de agradecer a instituição pelo programa de mestrado e pela oportunidade, obrigado *Theodomiro*. Também gostaria de agradecer a existência da *Internet*, facilitadora de novos aprendizados, conectora de pessoas e totalmente relevante ao programa de Mestrado em Ciência e Tecnologia da Computação. Também agradeço aos colegas do programa pela companhia.

Agradecimentos especiais aos amigos *João* e *Helaine* pelo apoio durante o mestrado, à *CAPES* pelo auxílio financeiro, e aos membros da *Banca Examinadora* pela disponibilidade para avaliação deste trabalho.

Enfim, agradeço a tudo e a todos que contribuíram para que eu pudesse terminar com êxito mais essa etapa tão importante em minha vida. Obrigado!

Resumo

A maioria dos sistemas computacionais comerciais utilizam algum tipo de banco de dados para o armazenamento persistente de informações. Esses sistemas geralmente possuem operações de inserção, leitura, edição e remoção dessas informações, e são conhecidos como sistemas CRUD. A codificação dessas operações demanda tempo, e, consequentemente, recursos. *Scaffolding* é uma técnica que utiliza a camada de modelo do padrão MVC para gerar automaticamente as camadas de visão e controle de um sistema CRUD. Isso aumenta a produtividade do desenvolvedor de *softwares*. Este trabalho propõe a implementação de um sistema que utiliza-se da técnica de *scaffolding* para automatizar o processo de criação de sistemas CRUD. O sistema *Metaffolder* foi desenvolvido para cumprir esta proposta e adicionou a técnica de *scaffolding* ao *framework Play*. O *Metaffolder* possui um mecanismo que utiliza reflexão e anotações para extrair os metadados de todas as classes de modelo. Os *templates* das classes de visão e controle foram criados visando maior usabilidade do sistema CRUD gerado. A combinação dos metadados extraídos e dos *templates* predefinidos resultaram na geração dos códigos-fonte das camadas de visão e controle. Foram realizados experimentos que compararam o aumento de produtividade ao se utilizar a técnica de *scaffolding* dos *softwares Metaffolder* e *Rails*. Também foram comparados os níveis de usabilidade desses *softwares* em relação à técnica de *scaffolding*, e dos sistemas CRUD gerados pelo uso dela. Os resultados comprovaram o aumento de produtividade ao se utilizar *scaffolding*, assim como bons níveis de usabilidade do *software Metaffolder* e do sistema CRUD gerado pela técnica.

Palavras-chave: *Scaffolding*, CRUD, Metaprogramação, *Templates*, Gerador de Código-fonte.

Abstract

Use of The Scaffolding Technique to Build CRUD Systems

Most commercial computer systems use some kind of database for persistent storage of information. These systems, also known as CRUD systems, usually have operations for insertion, reading, editing and removal of this information. Coding these operations takes time, and therefore resources. Scaffolding is a technique that uses the model layer of the MVC pattern to generate the view and controller layers of a CRUD system. This technique increases the software developer productivity. This paper proposes the implementation of a software that uses the scaffolding technique to automate the building process of CRUD systems. The Metaffolder system was developed to achieve this proposal and to add the scaffolding technique to the Play framework. The Metaffolder has a mechanism that uses reflection and annotations to extract the metadata of all model classes. The templates of view and controller classes were created aiming better usability of the CRUD system made by the technique. The combination of the extracted metadata and default templates resulted in the generation of source code of the view and controller layers. The conducted experiments compare the increase in productivity when using the scaffolding technique of the Metaffolder system and Rails framework. The experiments also compare the usability level of the scaffolding technique and the CRUD system generated by it. The results showed an increase in productivity when using scaffolding, as well as good usability level of the scaffolding technique and the CRUD system generated by it.

Keywords: *Scaffolding, CRUD, Metaprogramming, Templates, Source Code Generator.*

Sumário

Lista de Figuras

Lista de Tabelas

Abreviaturas e Siglas

API	–	Application Programming Interface
CoC	–	Convention over Configuration
CPF	–	Cadastro de Pessoa Física
CRC	–	Classe Responsabilidades Colaboradores
CRUD	–	Create Read Update Delete
CSS	–	Cascading Style Sheets
DAL	–	Data Access Layer
DDR	–	Double Data Rate
DML	–	Data Manipulation Language
ECC	–	Error-Correcting Code
EMF	–	Eclipse Modeling Framework
FQN	–	Full Qualified Name
HTML	–	HyperText Markup Language
HTTP	–	HyperText Transfer Protocol
IBM	–	International Business Machines
IDE	–	Integrated Development Environment
JDK	–	Java Standard Edition Development Kit
ISBN	–	International Standard Book Number
JPA	–	Java Persistence API
JSF	–	JavaServer Faces
JSON	–	JavaScript Object Notation
JSP	–	JavaServer Pages
JSR	–	Java Specification Request
JVM	–	Java Virtual Machine
LASER	–	Laboratório de Segurança e Engenharia de Redes
MIT	–	Massachusetts Institute of Technology

MVC – Model View Controller
NoSQL – Not Only SQL
RAM – Random Access Memory
RG – Registro Geral
SBT – Simple Build Tool
SQL – Structured Query Language
SUS – System Usability Scale
UML – Unified Modeling Language
UNIFEI – Universidade Federal de Itajubá
URL – Uniform Resource Locator
XML – eXtensible Markup Language
W3C – World Wide Web Consortium
WWW – World Wide Web

Capítulo 1

Introdução

A história da comunicação foi marcada por diversas invenções que revolucionaram o modo como a informação é transmitida entre as pessoas. A prensa de Gutenberg possibilitou a confecção de livros em larga escala. Com o telefone de Graham Bell duas pessoas poderiam ter uma conversa a distância. O rádio tornou-se um meio de comunicação de massas: a transmissão da informação percorria grandes distâncias e alcançava milhares de pessoas ao mesmo tempo. A televisão ainda é uma das principais opções de entreterimento das famílias contemporâneas. Apesar disso, uma invenção em especial pode ser considerada como a que mais influenciou a história da humanidade recente, tanto na comunicação quanto em diversas outras áreas: a *internet*.

A internet é um mecanismo capaz de disseminar mundialmente informações e possibilita a interação e colaboração entre pessoas, independente da localização geográfica (LEINER et al., 2009). A internet causou uma revolução sem precedentes no mundo da comunicação e da tecnologia. Ela integra todas as funcionalidades das invenções citadas anteriormente: consegue transmitir texto, áudio e vídeo para o mundo todo.

Licklider e Clark (1962) foram os autores do primeiro documento relatando interações sociais por meio de redes de computadores. Licklider imaginava um conjunto de computadores globalmente interconectados com dados acessíveis rapidamente de qualquer site, conceito muito parecido com a internet dos dias de hoje (LEINER et al., 2009).

Vários estudos foram realizados em diversas áreas da informática para que os conceitos básicos da internet se tornassem realidade. Organizações governamentais, universidades e fabricantes se uniram para solucionar diversos problemas que foram surgindo, tais como: interconexão entre redes de diferentes arquiteturas, transmissão eficiente de dados, infraestrutura operacional, suprir interesses da comunidade científica, despertar interesse da comunidade emergente de usuários domésticos, etc. Diversas tecnologias surgiram dessa união e a internet amadureceu muito desde a sua criação, mas o surgimento da tecnologia *Web* pode ser considerado como marco inicial de profundas mudanças tecnológicas, sociais, políticas e econômicas na humanidade.

Berners-Lee foi o fundador da *World Wide Web*, também conhecida como WWW ou *web* (BERNERS-LEE, 1989). Muitas de suas ideias iniciais ainda existem na web atual: descentra-

lização da informação, acesso independente de *hardware* e *software*, facilidade de criação e compartilhamento de conteúdo. Berners-Lee também foi o fundador da W3C (*World Wide Web Consortium*), instituição responsável por padronizar os protocolos e tecnologias utilizadas na construção da *web*, de modo que o seu conteúdo seja o mais acessível possível. Atualmente a maioria das grandes empresas são membros desse consórcio: *Facebook*, *Google*, *Microsoft*, *IBM*, *Twitter*, *Oracle*, etc (W3C, 2014a). Esta padronização e união contribuiram fortemente para a rápida evolução e popularização da *web* e da internet.

Uma pesquisa realizada pela empresa *Internet World Stats* revela que o numero de usuários da internet aumentou quase 7 vezes de 2000 a 2014 e que 40% da população mundial utiliza a internet atualmente (IWS, 2014). Esse crescimento sem precedentes influenciou positivamente a área de desenvolvimento *web*. Novas metodologias, tecnologias e ferramentas foram criadas para que desenvolvedores se tornassem mais produtivos e pudessem acompanhar as exigências da *web* moderna.

Os *frameworks web* são importantes ferramentas de desenvolvimento. Desenvolvedores conseguem economizar uma quantidade significativa de tempo na construção de *websites* quando se utiliza os *frameworks* apropriados (SCHWARTZ, 2014).

Segundo Fayad e Schmidt (1997), *frameworks* orientados a objetos representam uma aplicação abstrata formada por blocos pré-fabricados de *software* que os programadores podem usar, estender ou adaptar para uma solução específica. Estes blocos normalmente representam exigências básicas de uma aplicação *web*, otimizações para melhoria de desempenho e automações de código para aumento de produtividade. Exemplos de exigências básicas são HTTP (*HyperText Transfer Protocol*), HTML (*HyperText Markup Language*), CSS (*Cascading Style Sheets*), banco de dados, transações, servidor *web*, etc. Exemplos de otimizações são memória *cache*, concorrência, escalabilidade, etc. Exemplos de automações de código são *templates* HTML, geradores de códigos-clichê (*boilerplate code*) transparentes ao desenvolvedor e de uso específico do *framework*, geradores de códigos-fonte visíveis e disponíveis ao desenvolvedor.

Ainda segundo Fayad e Schmidt (1997), os principais benefícios da utilização de *frameworks* orientados a objetos são:

- *Modularidade*: *frameworks* modulares ajudam a localizar mais rapidamente comportamentos inesperados, provenientes de alterações em blocos de *software*.
- *Reusabilidade*: Interfaces estáveis garantem a reusabilidade, pois suas implementações genéricas podem ser usadas em novas aplicações.
- *Extensibilidade*: As implementações genéricas podem ser estendidas para casos mais particulares, essencial para adição de novas funcionalidades em aplicações existentes.

1.1 Motivação

Atualmente várias empresas estão em busca de soluções de *software* que utilizem tecnologias *web* ao invés dos tradicionais *softwares* nativos. O principal motivo dessa escolha está intimamente relacionado ao crescimento e popularização da internet: um número cada vez maior de pessoas acessam a internet diariamente, ou seja, maior contato dessas empresas com possíveis clientes. Além disso, soluções *web* possuem os benefícios intrínsecos dessa tecnologia: zero atualizações, zero instalação, alta disponibilidade, alta acessibilidade, etc.

Geralmente estas soluções de *software* empresariais, seja *web* ou nativo, possuem algum mecanismo de manipulação e persistência de dados. Esse mecanismo deve possuir as 4 funções básicas de armazenamento persistente: criação, leitura, edição e remoção (HELLER, 2007). Essas funções são popularmente conhecidas como CRUD (*Create-Read-Update-Delete*).

O ritmo dinâmico e acelerado da internet cria usuários de aplicações *web* cada vez mais exigentes. Ferramentas, princípios, convenções e padrões de desenvolvimento precisam ser utilizados pelos desenvolvedores para aumentar a produtividade, qualidade final e cumprir prazos cada vez menores. *Frameworks*, IDEs (*Integrated Development Environment*), bibliotecas e geradores de código são algumas das ferramentas que programadores utilizam para se tornarem mais produtivos.

O mercado atual possui diversos *frameworks* de desenvolvimento *web*, a maioria *open source*. Esses *frameworks* fornecem ao programador todas as funcionalidades necessárias para a criação de aplicações *web* modernas. Além disso, a maioria dos *frameworks web* utilizam o padrão de arquitetura MVC (*Model-View-Controller*). Esse padrão separa uma aplicação em 3 camadas: modelo, visão e controle (REENSKAUG, 1979).

A divisão em camadas possibilita o uso da técnica de *scaffolding*: geração de código automático baseado em informações da camada de modelo. Essa técnica aumenta a produtividade, diminui erros e é utilizada em vários *frameworks web* atuais, tais como: *Ruby on Rails*, *Grails*, *Spring*, *CakePHP*, etc. Apesar de também ser atual e popular (TYPESAFE, 2014b), o *framework Play* (TYPESAFE, 2014c) ainda não possui a funcionalidade de *scaffolding*.

Portanto, as principais motivações para o desenvolvimento deste trabalho são:

- Sistemas CRUD são amplamente utilizados por empresas em aplicações *web*.
- A maioria dos sistemas implementados atualmente são orientados a modelo e utilizam MVC.
- Código gerado automaticamente aumenta a produtividade e diminui erros.
- Os *frameworks web* mais utilizados atualmente implementam a técnica de *scaffolding*.

- Apesar de muito utilizado por grandes empresas, o *framework Play* ainda não utiliza a técnica de *scaffolding*.

1.2 Objetivos

Este trabalho propõe a implementação de uma ferramenta que, a partir das entidades da camada de modelo, irá gerar todo o código-fonte necessário para a criação de um sistema CRUD funcional. A criação de código será automática e transparente ao desenvolvedor.

Foram utilizados os seguintes métodos e tecnologias na implementação da ferramenta proposta:

- Convenção e/ou configuração para definir os nomes e locais dos arquivos gerados.
- Reflexão e anotações para encontrar todas as classes anotadas como entidades e seus respectivos atributos.
- Metaprogramação para armazenar em metadados todas as informações necessárias sobre as entidades encontradas.
- Reaproveitar ferramentas fornecidas pelo *framework Play* para reduzir dependências externas.
- Geradores de *templates* para criação de código compilável a partir dos metadados existentes.

O principal objetivo desse trabalho é fornecer ao usuário da ferramenta implementada um aumento de produtividade no desenvolvimento de sistemas. A utilização da técnica de *scaffolding* garantirá este aumento: um sistema CRUD será gerado automaticamente a partir das classes da camada de modelo. Como consequência, o tempo gasto na codificação das camadas de controle e visão será consideravelmente reduzido. Além disso, geradores de código reduzem o erro humano causado pela codificação manual, e isso também influencia o aumento da produtividade. Tanto a ferramenta quanto o sistema gerado por ela deverão ser agradáveis aos seus usuários, ou seja, deverão possuir boa usabilidade.

1.3 Organização do trabalho

Este trabalho está dividido em capítulos, cada um simbolizando uma fase no desenvolvimento de uma pesquisa científica. A síntese de cada capítulo é listada a seguir:

- *Capítulo 2 - Fundamentação Teórica*: discute sobre os conceitos necessários para o desenvolvimento de um gerador de código. Os mais importantes são: convenções, padrões arquiteturais, operações CRUD, metaprogramação, e técnicas de geração de código.
- *Capítulo 3 - Metaffolder, Aplicando a Técnica de Scaffolding*: descreve o sistema desenvolvido neste trabalho, mostrando seu funcionamento e os detalhes mais relevantes de sua implementação. Ao final desse capítulo é apresentado um exemplo de utilização do sistema, com suas respectivas classes de entrada e arquivos de controle e visão gerados na saída.
- *Capítulo 4 - Experimentos e Resultados*: explica a metodologia dos experimentos realizados e apresenta os dados e resultados obtidos.
- *Capítulo 5 - Conclusões*: apresenta as conclusões obtidas da análise dos resultados, as contribuições primárias e secundárias, e sugere melhorias para trabalhos futuros.

Capítulo 2

Fundamentação Teórica

Este Capítulo apresenta a fundamentação teórica necessária para o desenvolvimento deste trabalho. A Seção 2.1 enfatiza a importância de convenções em projetos de *softwares*. A Seção 2.2 explica o padrão arquitetural MVC e os seus benefícios. A Seção 2.3 elucida os conceitos básicos de metaprogramação e suas principais ferramentas da linguagem Java: *Annotations* e *Reflections*. A Seção 2.4 apresenta uma breve descrição sobre o significado do acrônimo CRUD e sua utilização no mercado atual. A Seção 2.5 descreve a técnica de *scaffolding* e sua utilização nos *frameworks Rails* e *Grails*. Finalmente, a Seção 2.6 faz uma breve comparação entre este trabalho e outros trabalhos similares, que também utilizam a técnica de *scaffolding*.

2.1 Convenções

Convenções são regras, normas, critérios ou acordos preestabelecidos e aceitos pela maioria de uma sociedade. Esse conceito também pode ser aplicado à engenharia de *software*. Existem vários tipos de convenções em computação e cada uma delas pode variar entre si. Por exemplo, um mesmo tipo de convenção pode variar conforme linguagem de programação ou local de trabalho. No exemplo do Código 2.1, a convenção de nome para métodos e atributos varia entre as linguanges de programação Java (GOSLING et al., 2013) e *Ruby* (MATSUMOTO, 1993).

Código 2.1: Convenção de nome para métodos e atributos

```
1 // Codigo Java
2 public void exemploDeMetodo() {
3     int exemploDeAtributo = 0;
4 }
5
6 // Codigo Ruby
7 def exemplo_de_metodo
8     exemplo_de_atributo = 0
9 end
```

Particularmente, convenções com o objetivo de padronizar código são muito importantes em computação. Em ambientes corporativos, a padronização de código é muito utilizada, pois agrega ao processo de desenvolvimento de *software* os seguintes benefícios:

- *Integração de código:* sistemas complexos geralmente possuem múltiplas equipes. Código padronizado possibilita a união desses códigos e minimiza problemas de compatibilidade.
- *Diminui comunicações desnecessárias:* o código padronizado facilita a leitura e entendimento do mesmo, portanto membros de uma equipe passam menos tempo explicando o código e mais tempo trabalhando.
- *Manutenção:* o programador responsável pela manutenção de código pode não ser o mesmo que o criou, mas como o código é padronizado sua compreensão será facilitada.
- *Ferramentas de automação:* ferramentas de automação podem utilizar os mesmos padrões de código dos desenvolvedores para gerar código automaticamente. Como exemplo, pode-se citar a geração automática de métodos de acesso (leitura e escrita) a partir dos nomes de variáveis (*getters* e *setters*).

2.1.1 Convenção Sobre Configuração

Convenção sobre Configuração (*CoC - Convention over Configuration*) é um paradigma de desenvolvimento de *software* muito usado em *frameworks* configuráveis. Seu objetivo é simples: evitar configurações desnecessárias através do uso de convenções. Apesar disso, sua adoção não foi imediata, pois a área de metaprogramação ainda não era muito difundida e não existiam convenções bem definidas na época (CHEN, 2006).

Frameworks de armazenamento persistente são um bom exemplo da evolução desse paradigma ao longo dos anos. A primeira geração desses *frameworks* dependia de arquivos de configuração para a criação de suas tabelas relacionais. O Código 2.2 mostra uma classe de usuário contendo dois atributos: identificador (`id`) e senha (`password`). Nesses *frameworks*, o mapeamento de classes e atributos em tabelas e campos era feito com a criação de arquivos XML (*eXtensible Markup Language*) contendo informações da persistência.

Código 2.2: Classe `User`

```
1 @Entity
2 public class User {
3     @Id
4     private String id;
5     private String password;
6     //-- getters e setters --//
7 }
```

O Código 2.3 exibe um arquivo XML do *framework Hibernate* que era usado no mapeamento da classe do Código 2.2. O *framework* utilizava esse arquivo XML para a criação dos comandos SQL (*Structured Query Language*) utilizados na criação das tabelas relacionais.

Código 2.3: Configuração XML para Hibernate

```
1 <hibernate-mapping>
2 -/<class name="User" table="USERS">
3   <id name="id" column="ID" type="string"/>
4   <property name="password" column="PASSWORD" type="string"/>
5 </class>
6 </hibernate-mapping>
```

O Código 2.4 exibe o código SQL gerado para criar a tabela da classe `User`, mostrada no Código 2.2. Atualmente, esse código SQL é gerado automaticamente utilizando metaprogramação e convenção de nomes, ou seja, não depende de configurações manuais.

Código 2.4: Comando SQL para criação da tabela

```
1 CREATE TABLE USERS (
2   ID VARCHAR(20) NOT NULL,
3   PASSWORD VARCHAR(20),
4   PRIMARY KEY(ID)
5 );
```

O paradigma *CoC* herda todas as vantagens de se utilizar convenções: menor curva de aprendizagem de novas tecnologias, menos linhas de código, menos decisões, sistemas prontos mais rapidamente, equipe mais sincronizada, etc.

Sistemas ainda podem ser personalizados e configurados normalmente caso as convenções não satisfaçam os requisitos do programador. Apesar disso, é importante tentar evitar configurações que fujam às convenções predeterminadas, pois sistemas inteiros que dependam dessas convenções podem falhar (MARTIN; MARTIN, 2006).

Resumidamente, Convenção sobre Configuração é um paradigma no qual uma configuração padrão bem definida é adotada na falta de configurações manuais. Uma configuração

bem definida é aquela que foi exaustivamente utilizada em vários casos de sucesso, ou seja, é uma convenção entre desenvolvedores.

2.2 Padrões Arquiteturais

Sistemas de *software* podem ser divididos em subsistemas. Padrões arquiteturais determinam as responsabilidades e relacionamentos de cada subsistema. Esses padrões apresentam esquemas estruturais fundamentais para sistemas de *software* (BUSCHMANN et al., 1996).

Padrões arquiteturais possuem um nível maior de abstração quando comparados com padrões de projeto (KAISLER, 2005; BUSCHMANN et al., 1996). São estratégias mais sofisticadas que abragem um grande número de componentes e propriedades de um sistema de *software*. Sua utilização influencia diretamente a estrutura e organização geral desse sistema (KAISLER, 2005).

2.2.1 Model-View-Controller

Atualmente, um padrão arquitetural muito difundido é o padrão MVC, que foi criado por Reenskaug (1979) e divide um sistema em camadas. Essa divisão faz com que o desenvolvimento do produto final seja mais modular e eficiente. A abstração em camadas é muito utilizada em *frameworks web* modernos, pois facilita a criação de sistemas cada vez mais coesos e menos acoplados. Alguns dos *frameworks* mais famosos que utilizam MVC: *Grails* (PIVOTAL, 2014), *Rails* (HANSSON, 2003), *Play* (TYPESAFE, 2014c), etc.

A Figura 2.1 ilustra o padrão MVC. A camada de modelo é uma aproximação por *software* de um determinado processo ou sistema do mundo real. A apresentação, união das camadas de visão e controle, é a camada acessível ao usuário. Em um sistema de compras *online*, a camada de modelo seria os objetos `carrinho`, `produto` e `usuario` por exemplo. A camada de visão seria as páginas HTML e os cliques do *mouse* seriam interpretados pela camada de controle. A descrição de cada camada é feita a seguir.

1. *Camada de modelo*: representa os dados e funcionalidades de uma determinada aplicação. Regras de negócio dão significado a esses dados e definem como eles são manipulados. Essa camada também atualiza a camada de visão sempre que houver alterações no estado dos dados (SOMMERVILLE, 2011; FOWLER, 2002). Geralmente encapsula uma camada inferior específica para a manipulação de dados, conhecida como DAL (*Data Access Layer*). Ela apresenta as funcionalidades CRUD de armazenamento persistente. A camada de modelo é totalmente independente das camadas de visão e controle.

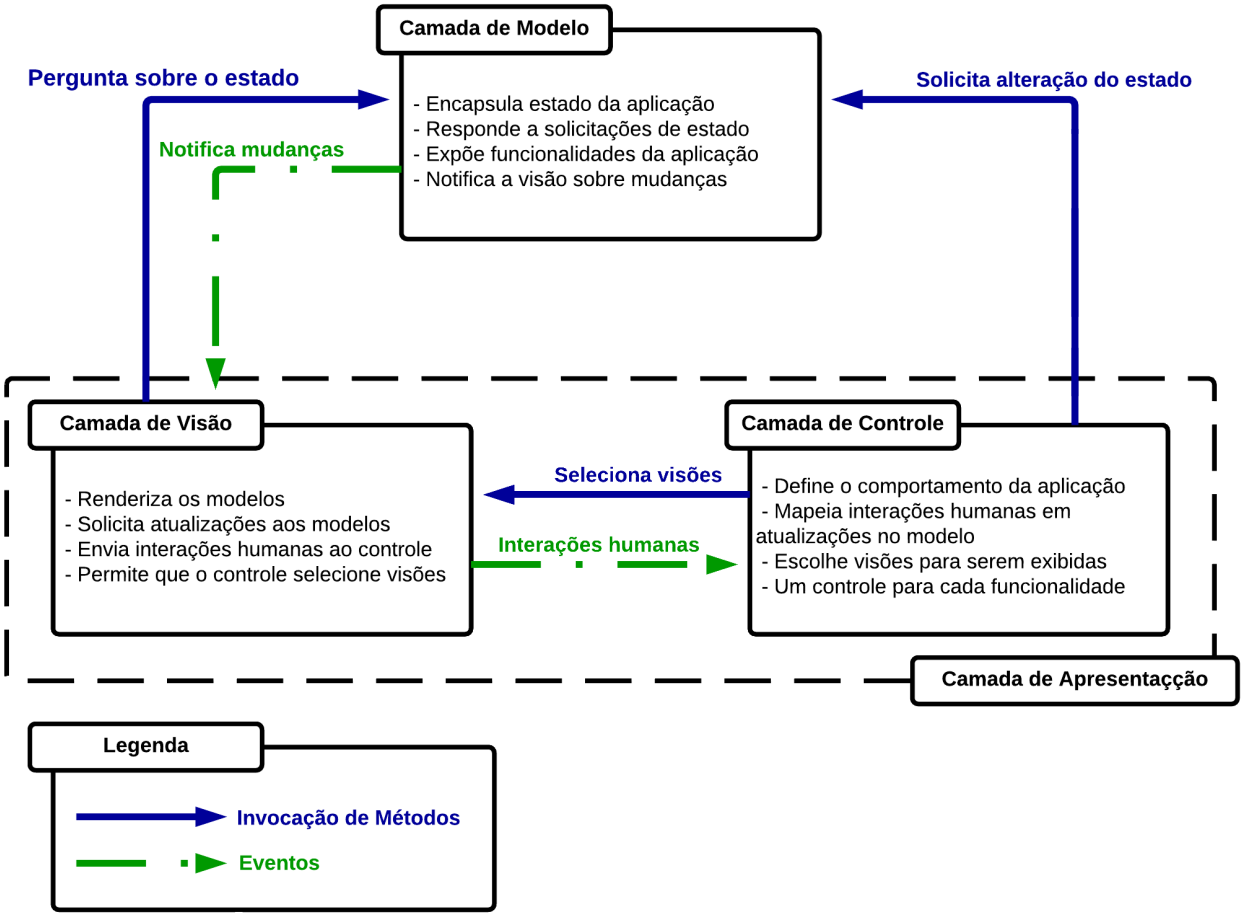


Figura 2.1: Padrão Arquitetural MVC
Fonte: Adaptada de Singh et al. (2002)

2. *Camada de visão*: é formada por várias interfaces de visão (*views*). Cada interface é responsável por apresentar informações ao usuário. Um modelo pode ser apresentado por múltiplas interfaces, ou seja, uma mesma informação pode ser apresentada ao usuário de diferentes maneiras (BUSCHMANN et al., 1996). Em aplicações *web* modernas, os tipos mais populares de apresentação são XML, JSON (*JavaScript Object Notation*) e HTML e o reponsável pela interação entre interface e usuário é o navegador (*browser*). Os navegadores mais conhecidos são: *Firefox*, *Internet Explorer* e *Google Chrome*.

3. *Camada de controle*: realiza a mediação entre as camadas de visão e modelo. É a camada responsável por toda a lógica e comportamento do sistema. Recebe eventos da camada de visão e os processa. Em aplicações *web*, esses eventos são normalmente solicitações `GET` ou `POST`. Esses eventos são convertidos em ações a serem realizadas pela camada de modelo (alteração do modelo) e/ou camada de visão (renderização de nova *view*).

A separação da camada de modelo é considerada como um dos princípios fundamentais de bom desenvolvimento de *software*. No padrão MVC essa separação gera duas situações importantes: modelo é separado da apresentação e visão é separada de controle. Essas duas situações resultam nas seguintes vantagens (+) e desvantagens (–):

+ *Automação de testes*: ferramentas de teste que simulam interações do usuário na camada de visão são lentas e complicadas. Testes feitos diretamente nos objetos da camada de modelo são muito mais eficientes e fáceis de serem automatizados (FOWLER, 2002).

+ *Desenvolvedores especializados*: cada camada utiliza conceitos e tecnologias diferentes em sua implementação. Como as camadas são independentes entre si, pode-se utilizar desenvolvedores com conhecimentos específicos de alguma determinada camada. Por exemplo, *web designers* para criação de páginas HTML, desenvolvedores para programação dos controles e analistas para modelagem do sistema (FOWLER, 2002).

+ *Várias representações de um modelo*: a variedade de representações de modelos é uma prática comum em aplicações *web*. Um mesmo *controller* pode ser responsável por diferentes *views* de um mesmo modelo. Essas *views* podem ser abertas e fechadas em tempo de execução para representar esse modelo. Qualquer alteração em uma das representações é refletida em todas as outras (BUSCHMANN et al., 1996; FOWLER, 2002).

+ *Separação de controle e visão*: não existia separação entre controle e visão nas primeiras versões do padrão MVC (REENSKAUG, 1979; FOWLER, 2002). Atualmente a maioria dos *frameworks web* separa essas duas camadas, pois promove mão de obra especializada, clareza de código e reusabilidade de um mesmo *controller* para diferentes *views*.

– *Interdependência entre controle e visão*: impossibilita reuso de objetos de uma camada sem utilizar seu objeto correspondente da outra camada. A única exceção é se alguma *view* for estática, ou seja, somente leitura e sem interações com o usuário (BUSCHMANN et al., 1996).

– *Forte dependência das camadas de visão e controle*: apesar do modelo ser independente das outras camadas, o contrário não é verdadeiro. As camadas de visão e controle são altamente dependentes da camada de modelo. Se o modelo sofrer alteração, todas as *views* e *controllers* relacionados a esse modelo precisam ser atualizadas (BUSCHMANN et al., 1996).

2.3 Metaprogramação Java

A linguagem de programação Java é uma linguagem de alto-nível de propósito geral, compilada, concorrente, baseada em classes, orientada a objetos, fortemente tipada, influenciada por aspectos de outras linguagens de programação bem sucedidas (GOSLING et al., 2013).

Metaprogramação é um conceito poderoso e é utilizado na implementação das principais bibliotecas e *frameworks* Java. As Seções a seguir descrevem as principais caracteristicas envolvidas no paradigma de metaprogramação.

2.3.1 Metadados

Metadados são informações bem organizadas que descrevem, explicam, localizam ou facilitam a obtenção, uso ou manipulação de uma fonte de informação. De um modo geral, metadados são dados que descrevem dados (NISO, 2004).

NISO (2004) classifica os metadados em 3 tipos:

- *Metadados descritivos*: são dados que auxiliam a pesquisa e localização de um determinado objeto, ou seja, dados que descrevem o próprio objeto. Por exemplo, os elementos *nome* e *cpf* descrevem o objeto *pessoa* e podem ser utilizados para a sua localização. Alguns elementos normalmente utilizados na descrição de objetos: RG, CPF, nome, título, ISBN, latitude, longitude, *hashs*, etc.
- *Metadados estruturais*: são dados que indicam como objetos são distribuídos ou organizados. Esses dados especificam a estruturação do objeto. Por exemplo, o catálogo de uma biblioteca determina como os livros são organizados, as páginas de um livro determinam a ordem dos capítulos. Objetos de um catálogo de biblioteca normalmente são estruturados em: páginas, números, letras, capítulos, seções, alíneas, diagramas, plantas, blocos, etc.
- *Metadados administrativos*: são dados que influenciam decisões importantes relacionadas ao tratamento de um objeto. Essas decisões normalmente envolvem permissões de acesso, direitos autorais, segurança, controle, etc.

A utilização de metadados em computação apresenta uma infinidade de aplicações: identificação de arquivos de texto, áudio e vídeo; marcadores em páginas HTML para otimização de *engines* de busca; arquivos de configuração em *frameworks*; mapeamento de classes em tabelas; marcadores de conteúdo para facilitar a filtragem de determinado assunto;

marcadores de código que influenciam nas decisões do compilador; atributos de leitura e escrita de arquivos; etc.

Os metadados que foram utilizados nesse trabalho são todos os elementos descritivos, estruturais e administrativos de uma classe Java, tais como: nome, tipo, atributos e seus controladores de acesso, atributos herdados, pacote, local em disco, anotações, etc.

2.3.2 Metaprogramação

Programas de computador analisam e transformam dados de entrada (*input*) em dados de saída (*output*). Metaprogramação é quando os dados de entrada e de saída também são programas de computador. De uma maneira análoga a metadados, pode-se definir metaprogramação como sendo programas que manipulam programas. Análise e transformação em tempo de compilação constitui metaprogramação estática e em tempo de execução constitui metaprogramação dinâmica (LöWE; NOGA, 2002).

Damasevicius e Stuikys (2008) afirmam que vários conceitos da metaprogramação estão sendo utilizados sem o devido conhecimento dos seus autores. A justificativa é que esses conceitos não receberam a devida análise e categorização. Os conceitos mais comuns são: *metalinguagem*, *linguagem-objeto*, *programa-objeto*, *metaprograma* e *metassistema*.

A metalinguagem é qualquer linguagem utilizada para discutir, descrever ou analisar outra linguagem (CZARNECKI; EISENECKER, 2000). Também é considerada como sendo a linguagem na qual o metaprograma é escrito. A linguagem-objeto é a linguagem do programa-objeto. O programa-objeto é o programa produzido pelo metaprograma. Programas-objeto são programas comuns, programados em alguma linguagem de programação, são o código-fonte do programa. Um metassistema ou sistema de metaprogramação é a união de metaprograma, programa-objeto e suas respectivas linguagens de programação (Figura 2.2).

Metaprogramas são programas especialmente projetados para analisar e transformar outros programas. Um metaprograma é um programa que pode construir programas-objeto, combinar programas-objeto menores em maiores, observar características de programas-objeto. Exemplos de metaprogramas: geradores de programas, analizadores, compiladores, interpretadores (SHEARD, 2000).

Os metaprogramas podem ser classificados como analisadores ou geradores de programas (Figura 2.2). Um analisador de programa extrai as características descritivas, estruturais e administrativas mais relevantes de um programa-objeto e as interpreta em um determinado contexto. Essa análise produz resultados que podem auxiliar programadores nas fases da criação de *software*: métricas de *software*, fluxogramas, diagramas, otimizadores, refatoradores.

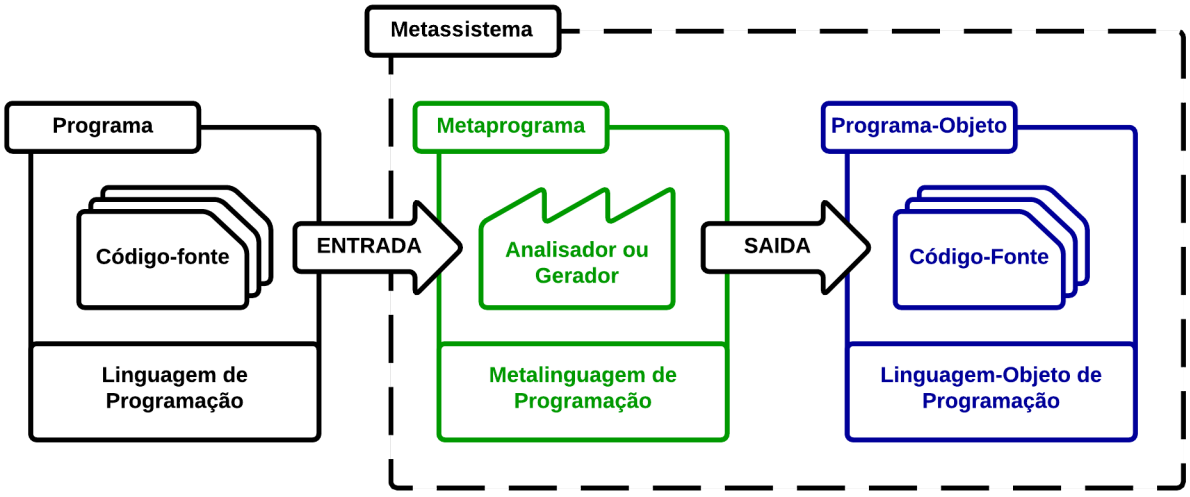


Figura 2.2: Metaprogramação: caso geral

Geradores de programas são implementados para solucionar problemas específicos. Vários programas-objeto podem ser gerados, um para cada tipo de problema. Os programas-objeto criados são soluções específicas para determinado problema, portanto são mais eficientes do que suas versões não-geradas (SHEARD, 2000).

Sistemas de metaprogramação que possuem metalinguagem igual a linguagem-objeto são considerados homogêneos (SHEARD, 2000). Somente em metassistemas homogêneos existe reflexão, definida como um conjunto de funções da linguagem, capazes de acessar e alterar informações de metaprogramas em tempo de execução. A introspecção define as funções que acessam o estado de execução do metaprograma e a intercessão define as funções que alteram esse estado (OLIVEIRA, 2012).

Forman (2005) define reflexão como sendo a capacidade de um programa analisar o próprio estado de execução, e de alterar esse estado de acordo com o que for encontrado na análise. A Figura 2.3 demonstra essa definição: a saída do metaprograma é o seu próprio estado, inspecionado e alterado por mecanismos reflexivos, sendo reutilizado como uma nova entrada.

2.3.3 Java Annotations API

Anotação (*annotation*) é um mecanismo genérico que associa informação a elementos da linguagem Java. A API (*Application Programming Interface*) foi criada a partir da especificação JSR175 (*Java Specification Request*) (ORACLE, 2004). É padrão na linguagem desde a versão 5 e se encontra no pacote `java.lang.annotation`. Anotações não influenciam diretamente no comportamento do programa, que somente pode ser alterado pela ferramenta ou biblioteca responsável por essas anotações. Podem ser acessadas em tempo de execução ou em tempo de compilação por métodos de introspecção e intercessão.

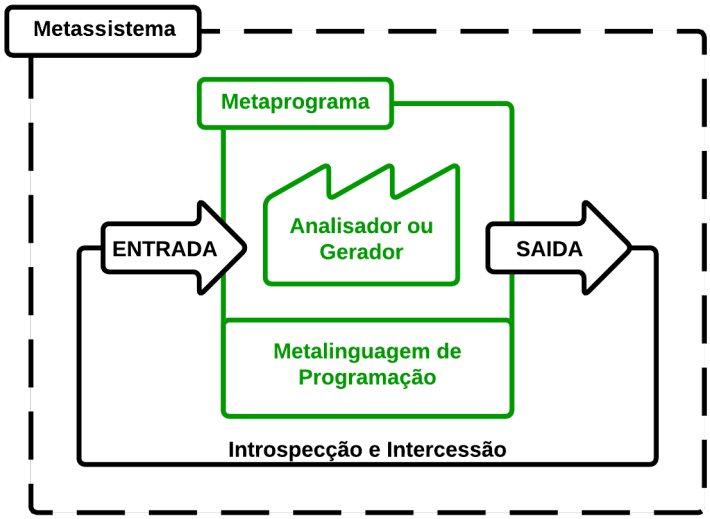


Figura 2.3: Metaprogramação: reflexão

As principais vantagens de *annotations* são: erros detectados em tempo de compilação; amplamente utilizadas por bibliotecas e *frameworks*; substitui arquivos de configuração; pode economizar recursos computacionais; sua leitura é nativa, ou seja, não exige bibliotecas externas. A principal desvantagem é uma possível poluição de código caso as convenções adotadas não sejam suficientes para o desenvolvedor. As anotações passariam a ser consideradas como fragmentos do arquivo de configuração anexados ao código-fonte.

Anotações são essencialmente metadados, ou seja, elas acrescentam alguma informação útil ao elemento anotado, mas não fazem parte do código do mesmo. Os elementos Java que podem ser anotados são: pacote (*package*), classe (*class*), enumerador (*enum*), interface (*interface*), atributo (*field*), método (*method*), parâmetro (*formal parameter*), construtor (*constructor*) e variável local (*local variable*) (GOSLING et al., 2013).

Anotações são indicadas pela inicial `@` e sua sintaxe e utilização podem ser observadas no Código 2.5. A anotação `@Entidade` é um modificador da classe `Usuario` (assim como a palavra reservada `public`), e possui o tipo-anotação (*annotation type*) `Entidade`.

Código 2.5: Uso da anotação `@Entidade`

```
1 @Entidade(value = "USUARIO", persistente = true)
2 public class Usuario {...}
```

Os métodos de um tipo-anotação também são conhecidos como elementos, e guardam informações relacionadas à anotação, na forma de pares elemento-valor (GOSLING et al., 2013). No exemplo do Código 2.5, a anotação `@Entidade` possui dois pares elemento-valor. Essa é a sintaxe padrão de uma anotação (*normal annotation*). No primeiro par, o elemento

`value` recebe o valor `"USUARIO"`. No segundo par, o elemento `persistente` recebe o valor `true`.

Toda anotação tem um tipo-anotação associado a ela que precisa ser declarado para ser criado (ORACLE, 2004). Sua declaração se assemelha à declaração de uma interface tradicional. O símbolo `@` precede a palavra reservada `interface` e indica a declaração do tipo-anotação, como pode ser observado no Código 2.6, que define a anotação `@Entidade`.

Código 2.6: Declaração do tipo-anotação `Entidade`

```
1  /*
2  *  ---informacoes para documentacao---
3  */
4  @Inherited
5  @Documented
6  @Retention(RetentionPolicy.RUNTIME)
7  @Target(ElementType.TYPE)
8  @interface Entidade {
9      String value() default "";
10     boolean persistente() default true;
11 }
```

Elementos com valores `default` podem ser omitidos da anotação. A sintaxe de anotação marcadora (*marker annotation*) é utilizada quando todos os elementos são declarados como `default` ou quando o tipo-anotação possui zero elementos (GOSLING et al., 2013). Na linha 5 do Código 2.7 a anotação `@Entidade` utiliza a sintaxe de anotação marcadora, já que seus dois elementos (`value` e `persistente`) possuem valores padrão (Código 2.6) e foram omitidos.

Código 2.7: Sintaxes da anotação `@Entidade`

```
1  @Entidade(value = "UM_NOME", persistente = false)
2  public class SintaxeGeral {...}
3
4  //Omissao de elementos default
5  @Entidade
6  public class SintaxeMarkerAnnotation {...}
7
8  //Omissao do elemento value e elementos default
9  @Entidade("UM_NOME")
10 public class SintaxeSingleElementAnnotation {...}
```

O elemento `value` é uma convenção de nome para uso da sintaxe de anotação de elemento único (*single-element annotation*) e pode ser omitido da anotação quando for o único elemento declarado no seu tipo-anotação correspondente, ou quando todos os outros

elementos declarados possuírem valores padrão (GOSLING et al., 2013). Na linha 9 do Código 2.7 a anotação `@Entidade("UM_NOME")` utiliza a sintaxe de anotação de elemento único.

Anotações que anotam tipo-anotação são conhecidas como meta-anotações (*meta-annotations*). O pacote `java.lang.annotation` possui as meta-anotações `@Documented`, `@Inherited`, `@Retention` e `@Target`:

@Documented documenta anotações. Ferramentas de documentação irão criar *links* automaticamente para tipos-anotação anotados com `@Documented`. Por exemplo, se a classe `Usuario` (Código 2.5) for documentada, um *link* para a documentação do tipo-anotação `Entidade` é criado automaticamente, pois a mesma está anotada com `@Documented` (Código 2.6).

@Inherited indica que um tipo-anotação deve ser herdado automaticamente por subclasses da classe anotada. Só apresenta funcionalidade quando anotada em tipos-anotação específicos para classes. Por exemplo, subclasses da classe `Usuario` (Código 2.5) herdariam a anotação `@Entidade`, pois a mesma está anotada com `@Inherited` (Código 2.6).

@Retention determina se a anotação estará disponível somente no código-fonte, em tempo de compilação ou em tempo de execução. Anotações disponíveis em tempo de execução são manipuláveis por bibliotecas de reflexão.

@Target um tipo-anotação somente será aplicável aos tipos de elementos Java restringidos pela meta-anotação `@Target`. Por exemplo, o tipo-anotação `Entidade` (Código 2.6) somente será aplicável aos elementos dos tipos `class`, `enum`, `interface`, ou `@interface`.

2.3.4 Java Reflections API

A API *Reflections* possui um conjunto de métodos de introspecção e intercessão, capaz de examinar ou modificar o estado de uma aplicação em tempo de execução (*runtime*). *Reflections* também oferece meios para se instanciar objetos, invocar métodos ou alterar valores de variáveis (JENKOV, 2008). Essa flexibilidade da API traz diversos benefícios e possibilidades ao programador, mas como qualquer ferramenta mal utilizada, pode causar sérios problemas.

Os principais problemas que desenvolvedores inexperientes podem encontrar são: falhas e restrições de segurança, complexidade de código, problemas de desempenho, quebra de portabilidade. A API consegue acessar e alterar atributos privados (`private`), ferindo

o princípio do encapsulamento. Além disso, aplicações podem estar rodando em ambientes restritivos, quebrando funcionalidades reflexivas que foram implementadas para ambientes não-controlados. Saber quando e como usar reflexão evita uma complexidade desnecessária de código. Códigos reflexivos envolvem resolução dinâmica de tipos, prejudicando o desempenho de aplicações com vários laços de repetição. Refatoração de código pode causar quebra de portabilidade em tempo de execução, com erros imprevisíveis e de difícil localização (ORACLE, 2011b; FORMAN, 2005).

Na linguagem Java, a classe `Class` é a mais importante para reflexão, pois possui todos os métodos que expõem o estado da aplicação ao desenvolvedor. A partir deles, pode-se obter informações importantes sobre os vários tipos de elementos da linguagem Java, encapsulados em diferentes classes e localizados no pacote `java.lang.reflect`. As principais classes são `Field`, `Method` e `Constructor`, para os elementos de atributos, métodos e construtores respectivamente. Esses elementos também são considerados *membros* de uma classe, pois implementam a interface `Member`. Outros tipos de elementos também são encapsulados, tais como: modificadores de controle de acesso, parâmetros de tipo para elementos genéricos, hierarquia de classes (OLIVEIRA, 2012; ORACLE, 2011a).

A obtenção do objeto `Class` é feita de 3 maneiras:

- *Herança da classe `Object`*: todos os objetos instanciados herdam o método `getClass` da classe `Object`, raiz da hierarquia de classes da linguagem Java. Esse método retorna a classe associada a esse objeto em tempo de execução (OLIVEIRA, 2012). A linha 7 do Código 2.8 exibe um exemplo de utilização do método herdado `getClass`.
- *Nome totalmente qualificado da classe*: o FQN (*Fully Qualified Name*) é o nome da classe seguido do pacote em que ela se encontra (JENKOV, 2008). A classe `Class` fornece o método estático `forName(...)` para invocar, em tempo de execução, classes a partir do seu FQN. A linha 10 do Código 2.8 exibe um exemplo dessa invocação.
- O literal `class` pode ser usado em nomes de classes para representar, em tempo de compilação, sua classe relacionada. A linha 13 do Código 2.8 mostra a obtenção da classe `Exemplo` através do seu literal `class`.

Código 2.8: Obtenção de classes para reflexão

```
1 //FQN = mestrado.Exemplo
2 import mestrado;
3 public class Exemplo {...}
4
```

```
5  // Metodo getClass() herdado de Object
6  Exemplo e = new Exempplo();
7  Class clazz = e.getClass();
8
9  // Metodo estatico utilitario forName(...)
10 Class clazz = Class.forName("mestrado.Exemplo");
11
12 // Uso do literal class
13 Class clazz = mestrado.Exemplo.class;
```

Membros de uma classe podem ser considerados públicos ou declarados. Membros públicos são todos os elementos da classe analisada que são acessíveis publicamente, inclusive os herdados. Membros declarados são todos os elementos codificados diretamente na classe, com qualquer grau de visibilidade, mas não incluindo elementos herdados.

A instrospecção do estado de uma aplicação é realizada a partir da obtenção dos membros de classe. A Tabela 2.1 exibe os métodos que expoem ao desenvolvedor o estado dos elementos `Field`, `Method` e `Constructor`. Esse estado poderá ser manipulado de acordo com a necessidade da aplicação.

Tabela 2.1: Métodos de introspecção da classe `Class`

Tipo de Membro	Descrição	Elemento	Método de Introspecção
Público	Elementos públicos da própria classe e elementos públicos herdados de suas superclasses.	`Field`	`getField(String nome)` `getFields()`
		`Method`	`getMethod(String nome, Class<?> parametros)` `getMethods()`
		`Constructor`	`getConstructor(Class<?> parametros)` `getConstructors()`
Declarado	Elementos da própria classe com qualquer grau de acesso, mas sem herança de outros elementos.	`Field`	`getDeclaredField(String nome)` `getDeclaredFields()`
		`Method`	`getDeclaredMethod(String nome, Class<?> parametros)` `getDeclaredMethods()`
		`Constructor`	`getDeclaredConstructor(Class<?> parametros)` `getDeclaredConstructors()`

Objetos `Field` podem ter seus valores alterados; objetos `Method` podem ser invocados passando zero ou mais parâmetros; objetos `Constructor` são os responsáveis por instanciar novos objetos. Com isso, as tarefas de atribuição de valores, invocação de métodos e criação de objetos tornam-se responsabilidade do programa, e não mais do desenvolvedor. Os principais métodos responsáveis por essa alteração de estado (intercessão) são mostrados na Tabela 2.2.

Tabela 2.2: Métodos de intercessão das classes `Field`, `Method` e `Constructor`

Elemento	Método de Intercessão	Descrição

`Field`	`set(Object obj, Object v)`
	`setBoolean(Object obj, boolean v)`
	`setByte(Object obj, byte v)`
	`setChar(Object obj, char v)`
	`setDouble(Object obj, double v)`
	`setFloat(Object obj, float v)`
	`setInt(Object obj, int v)`
	`setLong(Object obj, long v)`
	`setShort(Object obj, short v)`
		Cada tipo primitivo tem um método específico para se atribuir valores. O parâmetro obj é o objeto que possui o elemento Field no qual o método set é invocado. O parâmetro v é o novo valor atribuido a esse elemento.

`Method`	`invoke(Object obj, Object... argumentos)`	O parâmetro obj é o objeto que possui o elemento Method no qual o método invoke é invocado. O segundo parâmetro corresponde aos mesmos argumentos utilizados na assinatura do método referenciado.
`Constructor`	`newInstance(Object... argumentos)`	Cria uma nova instância do objeto ao qual o construtor pertence, com os argumentos de inicialização especificados.

2.4 Sistemas CRUD

Atualmente, a maioria dos sistemas comerciais utilizam banco de dados para manipulação e armazenamento persistente de dados. Para que isso seja possível, é necessário a utilização de algumas operações básicas, comuns a todos esses sistemas e independente de tecnologia ou meio de armazenamento. São elas: criação, leitura, edição e remoção de dados. Essas 4 operações são o núcleo de qualquer estrutura de armazenamento de dados e ficaram popularmente conhecidas pelo acrônimo CRUD, que se origina das palavras *Create* (criar), *Read* (ler), *Update* (atualizar) e *Delete* (deletar).

Sistemas CRUD são sistemas que normalmente seguem um padrão estrutural bem definido: possuem uma base de dados para armazenamento persistente; uma camada de abstração com as 4 operações básicas para a manipulação dos dados, conhecida como camada de acesso a dados ou DAL; e o sistema propriamente dito, que utiliza esta camada para implementar suas funcionalidades e usar os dados armazenados. Geralmente a camada de modelo encapsula a camada de acesso a dados, para maior controle do banco de dados (Figura 2.4).

O termo CRUD normalmente é associado a sistemas que utilizam alguma base de dados relacional, ou seja, manipulação por SQL e persistência em tabelas. Apesar disso, sistemas com qualquer tipo de camada de persistência são considerados CRUD, pois precisam implementar as 4 funcionalidades básicas para funcionar corretamente. Exemplo disso são as bases de dados não-relacionais que utilizam NoSQL (*Not Only SQL*) para manipular seus dados e a persistência geralmente é orientada a documentos ou grafos, utilizando arquivos JSON ou XML.

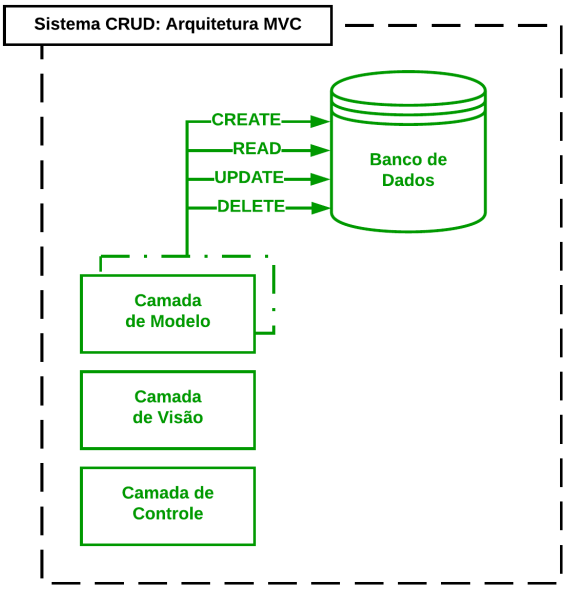


Figura 2.4: Estrutura básica de um Sistema CRUD

Uma linguagem de manipulação de dados (do inglês *Data Manipulation Language* ou DML) possui estruturas (verbos) que implementam as 4 funcionalidades CRUD. A DML mais conhecida é a SQL. Apesar de não ser especificamente uma DML, o protocolo HTTP também possui verbos para as operações CRUD. A Tabela 2.3 faz uma associação entre o acrônimo CRUD e os seus correspondentes em SQL e HTTP.

Tabela 2.3: Operações CRUD em SQL e HTTP

CRUD	SQL	HTTP
Create	insert	put / post
Read (Retrieve)	select	get
Update	update	put / patch
Delete (Destroy)	delete	delete

2.5 Scaffolding

Em engenharia civil, a técnica de cimbramento (*scaffolding* em inglês) representa uma estrutura de suporte provisório durante o período de construções, normalmente composta por andaimes e escoras. Em programação, a técnica de *scaffolding* gera todas as estruturas fundamentais ao funcionamento de uma aplicação *web*. Ela recebeu esse nome pois faz uma alusão à técnica de cimbramento da engenharia civil: ambas representam estruturas temporárias que beneficiam os seus utilizadores. Em engenharia civil, a estrutura de andaimes é conhecida como *scaffold*. O análogo em programação é o sistema CRUD gerado.

A técnica de *scaffolding* cria automaticamente as estruturas mais significativas de um sistema de dados persistentes: o mapeamento entre objetos, banco de dados e interfaces CRUD. Essas estruturas estão intimamente relacionadas entre si e dependem do padrão MVC, portanto a técnica somente poderá ser utilizada para a criação de sistemas CRUD com padrão MVC. Outras ferramentas não relacionadas à *scaffolding* permitem gerar outros componentes, tais como: camada de segurança, integração de sistemas, testes unitários, etc (ADAMUS et al., 2010).

Neste trabalho, o termo *scaffolder* é atribuído ao programa ou módulo que utiliza a técnica de *scaffolding* para a geração automática de código, ou seja, geradores de código. Um dos mecanismos mais comuns de geração de código de um *scaffolder* é o baseado em *templates* (FRANKY; PAVLICH-MARISCAL, 2012).

Geradores de código são ferramentas que aceitam como entrada os requisitos funcionais de um sistema, e produzem na saída os códigos-fonte que implementam esses requisitos (GERACI, 1990).

O *scaffolder* é basicamente uma ferramenta para automatizar a programação de tarefas repetitivas, que do contrário seriam codificadas a mão. A programação se torna repetitiva quando a aplicação precisa seguir estruturas e padrões de desenvolvimento. Um sistema CRUD é um caso típico de aplicação que possui tarefas repetitivas, pois é estruturado pela arquitetura MVC. Todo sistema CRUD possui várias entidades que modelam a aplicação. Para cada entidade existe métodos de acesso ao banco de dados, interfaces com o usuário, métodos que controlam toda a lógica da aplicação, etc. Codificar tudo isso manualmente consome mais tempo e possui um risco maior de erros (COHEN-ZARDI, 2013).

Existe uma certa controvérsia entre vários desenvolvedores experientes sobre os reais benefícios da utilização da técnica de *scaffolding*. Como qualquer ferramenta, é preciso saber como e quando utilizá-la, para ter o máximo de aproveitamento. É necessário entender que desenvolvedores podem usar a técnica quando necessário, mas não podem depender exclusivamente dela para a solução de todos os problemas.

Vários desenvolvedores argumentam que a utilização da técnica de *scaffolding* não é recomendada para programadores iniciantes por questões didáticas. Apesar disso, se o código gerado for claro e legível, tanto programadores iniciantes quanto experientes podem tirar proveito dos benefícios do uso da técnica.

É recomendado utilizar a técnica de *scaffolding* apenas uma vez, para gerar a estrutura inicial do projeto rapidamente. Esse método de desenvolvimento de software é conhecido como prototipação (SOMMERVILLE, 2011). Uma segunda utilização da técnica, resultante de modificações na camada de modelo, sobrescreveria as camadas de visão e controle, fazendo com que todo o código adicionado manualmente seja descartado. Sistemas de controle de versão podem recuperar os arquivos sobrescritos acidentalmente, evitando que código seja perdido.

Classes utilitárias (*helpers*) também podem ser geradas pela técnica de *scaffolding*. Elas servem para facilitar a implementação da lógica da aplicação gerada e não devem ser alteradas manualmente pelos programadores. Além disso, outras abstrações também podem ser geradas pelo *scaffolder*, tais como arquivos XML, CSS, LESS, etc. O desenvolvedor tem a flexibilidade de excluir trechos de código ou arquivos, caso julgue necessário.

Personalizações no *scaffolder* devem ser feitas até certo ponto: o desenvolvedor não deve gastar mais tempo nisso do que resolvendo tarefas da lógica da aplicação. O ideal seria ter uma equipe responsável somente pela manutenção do *scaffolder*, enquanto outras equipes tratariam dos problemas específicos da lógica da aplicação.

Dependendo de como os geradores de código são utilizados, a técnica de *scaffolding* pode ser classificada como dinâmica ou estática. O *scaffolding* dinâmico é a geração e/ou alteração de código-fonte em tempo de execução. O uso dessa técnica não é recomendável, pois pode adicionar dependências do *scaffolder* ao programa gerado; o programador não consegue personalizar o código, pois é sempre sobrescrito; ocorre um aumento no consumo de memória da aplicação final. Essa técnica é útil somente em casos bem específicos, como sistemas CRUD pouco complexos por exemplo. O *scaffolding* estático é o mais conhecido e indicado: gera código em tempo de compilação, através de comandos no *console* do *framework*.

A utilização da técnica pode trazer diversos benefícios, listados a seguir:

- *Personalizações*: bons geradores de código fornecem opções para refinar o resultado final. Se o gerador for *open source*, programadores poderão fazer personalizações diretamente na ferramenta, caso seja fácil e não demande muito tempo.
- *Produtividade*: o código gerado economiza horas de codificação manual, fazendo com que protótipos funcionais sejam gerados em minutos. Os desenvolvedores ficam livres para focar seu tempo na solução de problemas relacionados à lógica da aplicação.
- *Padrão de qualidade*: código feito a mão é instável pois não segue padrões, varia de pessoa para pessoa, um mesmo problema utiliza diferentes soluções, etc. O código criado por um gerador é mais estável: erros encontrados no código gerado podem ser reparados no gerador, assim novos códigos serão gerados sem os erros anteriores. Quanto mais utilizado, mais refinado e otimizado será o *scaffolder* em futuras versões. Bons geradores prezam por geração de códigos claros e de fácil entendimento.
- *Consistência:* o código gerado mantém padrões estruturais e convenções entre diversos projetos. Programadores têm mais facilidade de entender códigos gerados para novos projetos. Dificilmente surgirá erros no código criado, pois ele foi usado com sucesso em projetos anteriores.

- *Dependência Zero: scaffolding* estático não agrega nenhuma dependência externa ao sistema gerado. Todas as classes geradas ficam disponíveis ao programador.

2.5.1 Scaffolding com Templates

Geradores de código baseados em *templates* transformam dados em código-fonte. *Templates* são basicamente arquivos de texto que utilizam como modelo códigos-fonte de projetos bem sucedidos. Projetos bem sucedidos são aqueles que foram testados e utilizados exaustivamente pela comunidade, ou seja, podem ser considerados como modelos de desenvolvimento de sistemas. Todos os trechos do código considerados genéricos são substituídos por parâmetros, que serão interpretados por um gerador de *templates*. Os trechos genéricos mais comuns para substituição são: nome de pacote, nome de classe, nome de projeto, nome de variáveis, etc. *Templates* também podem ter mecanismos de controle de fluxo, que serão analisados e resolvidos pelo gerador de código. Isso garante mais flexibilidade ao desenvolvedor de *templates* (FRANKY; PAVLICH-MARISCAL, 2012).

Contexto é a estrutura responsável por armazenar informações relativas aos arquivos de *template*. Todo parâmetro presente em um *template* possui uma referência no contexto, para que valores sejam atribuídos e o código-fonte seja gerado. Um contexto pode ter mais atributos do que o número de parâmetros de um *template*. Isso possibilita que um mesmo contexto seja reaproveitado em diferentes *templates*. O contexto pode ser qualquer estrutura que armazene informações, por exemplo, instâncias de classes ou arquivos. A informação de um contexto é geralmente representada pela união entre metadados, convenções e configurações.

O Código 2.9 apresenta dois tipos de estruturas de armazenamento de informações. A estrutura representada pelas linhas 2 a 5 são do tipo objeto Java. As linhas 8 a 12 representam uma estrutura do tipo arquivo JSON. Os nomes `classe`, `tipoId` e `objeto` são os parâmetros presentes no *template*, que serão substituidos pelos seus valores correspondentes (`Onibus`, `String` e `onibus`) no processo de geração de código.

Código 2.9: Exemplo de contextos

```
1 //Instancia da classe Contexto na linguagem Java
2 Contexto c = new Contexto();
3 c.setClasse("Onibus");
4 c.setTipoId("String");
5 c.setObjeto("onibus");
6
```

```
//Arquivo JSON
{
  "classe": "Onibus",
  "tipoId": "String",
  "objeto": "onibus"
}
```

O código-fonte criado pelo gerador de *templates* é o resultado da união entre contexto e *template*: parâmetros são substituidos pelos seus valores correspondentes no contexto. De um modo geral, a técnica de *scaffolding* utilizando *templates* pode ser representada pela Figura 2.5.

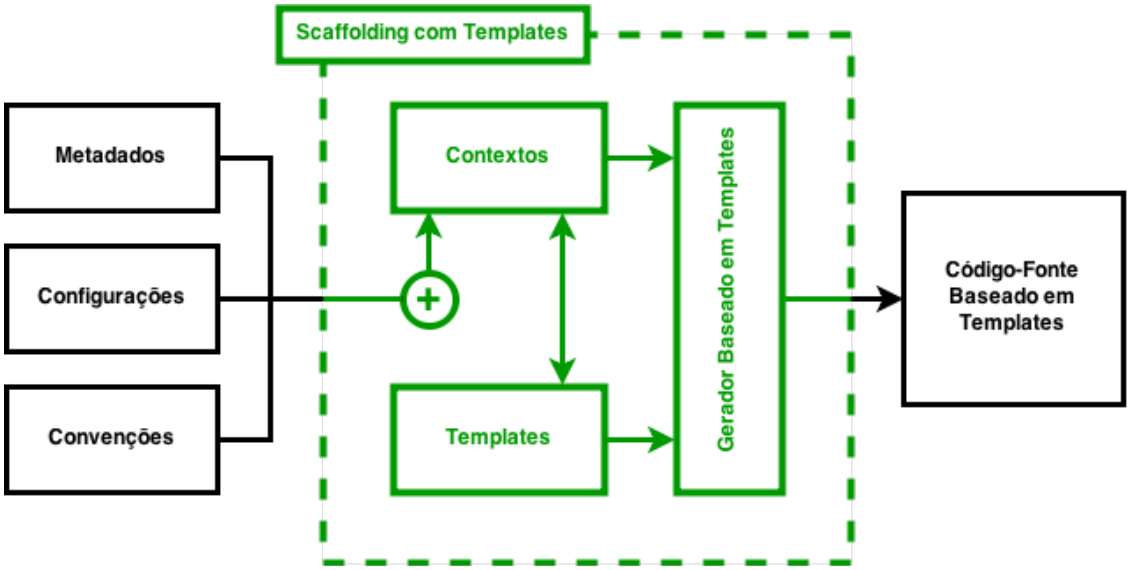


Figura 2.5: Scaffolding baseado em Templates

Como exemplo, o Código 2.10 mostra um fragmento de código-fonte de algum projeto bem-sucedido. Esse fragmento é usado como base para a criação de um *template*. Todas as ocorrências das palavras `Carro` e `carro` (trechos genéricos) são parametrizadas. Além disso, a palavra `Long` também é transformada em parâmetro, para maior flexibilidade. O restante do texto é copiado exatamente como o original.

Código 2.10: Exemplo de fragmento de código-fonte

```
public static Result edit(Long id) {
    Form<Carro> carroForm =
        form(Carro.class)
            .fill(Carro.find.byId(id));
    return ok(editForm.render(id, carroForm));
}
```

Após as devidas substituições, o *template* criado é apresentado no Código 2.11. A sintaxe para a representação dos parâmetros varia de acordo com a ferramenta de *templates* utilizada. Nesse exemplo, foram criados os seguintes parâmetros, utilizando a sintaxe de chaves duplas: `{{classe}}`, `{{objeto}}` e `{{tipoId}}`.

Código 2.11: Exemplo de *template* baseado no Código 2.10

```
1 public static Result edit({{tipoId}} id) {
2     Form<{{classe}}> {{objeto}}Form =
3         form({{classe}}.class)
4             .fill({{classe}}.find.byId(id));
5     return ok(editForm.render(id, {{objeto}}Form));
6 }
```

O Código 2.12 mostra o código-fonte criado pelo gerador de *templates*, utilizando o *template* do Código 2.11 e o contexto do Código 2.9.

Código 2.12: Exemplo de código gerado

```
1 public static Result edit(String id) {
2     Form<Onibus> onibusForm =
3         form(Onibus.class)
4             .fill(Onibus.find.byId(id));
5     return ok(editForm.render(id, onibusForm));
6 }
```

O mecanismo de substituição entre parâmetro do *template* e valor do contexto varia de acordo com a ferramenta de *templates* utilizada. Normalmente são utilizados mecanismos de reflexão para essa substituição. Nos exemplos anteriores, se o parâmetro `{{classe}}` é encontrado, espera-se que o contexto tenha um atributo `classe` ou um método `getClasse()` (caso seja um objeto Java); ou uma chave `classe` (caso seja um arquivo JSON). O valor desse atributo, método ou chave é substituido no *template*, gerando um código-fonte novo.

Franky e Pavlich-Mariscal (2012) resumem o ciclo de vida de um gerador de *templates* em 4 fases:

1. É escolhido algum código-fonte de projetos anteriores bem-sucedidos que implemente a funcionalidade desejada.
2. O código-fonte servirá como base para a criação de um *template*, ou seja, todo texto genérico (nome de classes, pacotes, variáveis, etc) será parametrizado. Novas funcionalidades serão adicionadas ao *template* nessa fase, caso necessário.

3. Geradores de código são implementados com base no *template* criado. O primeiro gerador cria a estrutura básica de um sistema e novos geradores evoluem a partir dela.

4. Desenvolvedores podem alterar o código gerado. O gerador não poderá ser utilizado novamente pois as alterações realizadas serão perdidas.

Apesar de flexíveis e de rápido aprendizado, geradores de código baseado em *templates* possuem alguns pontos negativos. Novas versões do gerador podem fazer com que versões anteriores dos sistemas criados se tornem incompatíveis com as atuais. Um novo sistema precisa ser gerado e testado a cada alteração nos *templates*, para correção de erros de compilação ou execução, caso sejam encontrados (FRANKY; PAVLICH-MARISCAL, 2012).

2.5.2 Scaffolding no Framework Rails

Ruby on Rails (também conhecido como RoR ou simplesmente *Rails*) é um *framework web* completo, de código aberto, sob licença MIT (*Massachusetts Institute of Technology*), e focado em desenvolvimento produtivo. Foi projetado especificamente para que programadores tenham facilidade e satisfação em produzir códigos elegantes e harmoniosos, pois favorece o uso de convenção sobre configuração. É o *framework* para desenvolvimento *web* sem complicações ou sofrimentos (HANSSON, 2003).

O *framework Rails* utiliza somente *scaffolding* estático para geração de código. O comando `generate` é o responsável por toda a criação de código do *framework* e aceita diversos argumentos, cada um representando uma camada de abstração de uma aplicação *web* típica. A ferramenta consegue gerar os modelos e toda a integração com o banco de dados, a camada de visão com as páginas HTML, arquivos CSS e javascript, a camada de controle que interliga modelo e visão, testes de integração e desempenho, arquivo de rotas, etc. Além disso, o desenvolvedor pode adicionar novos geradores ou criar um para alguma aplicação mais específica. Os argumentos mais comuns são `controller`, `model` e `scaffold` (Figura 2.6).

A conexão entre a camada de visão e a camada de controle é feita pelo arquivo de rotas do *framework*. As ações padrão para uma aplicação CRUD são representadas pelos métodos `index`, `new`, `create`, `show`, `edit`, `update` e `destroy`, localizados na camada de controle. Dessas 7 ações, 4 são ligadas à páginas HTML da camada de visão. As outras 3 são ações a serem executadas pelo *framework* (Tabela 2.4).

```
$ rails generate
Usage: rails generate GENERATOR [args] [options]

...

Please choose a generator below.

Rails:
  model
  controller
  scaffold
  ...
```

Figura 2.6: Comando `generate` do frameworks Rails
Fonte: Adaptada de Rails (2014)

Tabela 2.4: Rotas geradas para o modelo `HighScore`

Verbo HTTP	Caminho	Página na Camada de Visão	Ação (método) no controlador	Descrição
GET	/high_scores	index.html.erb	index	Lista todos os objetos `high_score`
GET	/high_scores/new	new.html.erb	new	formulário HTML para criar novos `high_score`
POST	/high_scores		create	cria novo `high_score`
GET	/high_scores/:id	show.html.erb	show	mostra high_score com o id `:id`
GET	/high_scores/:id/edit	edit.html.erb	edit	formulário HTML para alterar `high_score` com id :id
PUT/PATCH	/high_scores/:id		update	altera `high_score` com id `:id`
DELETE	/high_scores/:id		destroy	deleta `high_score` com id `:id`

O gerador de *controllers* cria a camada de apresentação da aplicação, com as classes de controle e suas páginas correspondentes na camada de visão. Cada `action` do controlador representa um método associado a uma página na camada de visão. O exemplo da Figura 2.7 cria o controlador `credit_card_controller.rb` com os métodos `open`, `debit`, `credit` e `close`. Cada método é associado automaticamente às páginas de mesmo nome da camada de visão, por meio de um arquivo de rotas, que também é criado ou atualizado.

```
$ rails generate controller
Usage: rails generate controller NAME [action action] [options]

...
...

Example:
    rails generate controller CreditCard open debit credit close

        Controller: app/controllers/credit_card_controller.rb
        Views:      app/views/credit_card/open.html.erb
                    app/views/credit_card/debit.html.erb
                    app/views/credit_card/credit.html.erb
                    app/views/credit_card/close.html.erb

        ...
```

Figura 2.7: Comando `generate controller` e exemplo de uso
Fonte: Adaptada de Rails (2014)

Os argumentos do gerador de modelos (`generator` com argumento `model`) são os atributos da classe a ser criada. O gerador `scaffold` também espera por argumentos que representem atributos de uma classe, mas cria toda a aplicação ao invés de gerar somente a camada de modelo. O programador pode usar o gerador `scaffold_controller` caso já possua as classes de modelo.

A Figura 2.8 mostra o uso do gerador `scaffold` para criação da classe `HighScore` com os atributos `game` e `score`. Na verdade, todo a computação é delegada a outros geradores mais específicos (`invoke`). A linha `route resources :high_scores` corresponde à conexão entre camada de visão e controle, representada anteriormente pela Tabela 2.4. A partir disso, todos os arquivos necessários são criados (`create`) para a geração de um sistema CRUD totalmente funcional.

```
$ bin/rails generate scaffold HighScore game:string score:integer
    invoke  active_record
    create    db/migrate/20130717151933_create_high_scores.rb
    create    app/models/high_score.rb
    invoke    test_unit
    ...
    invoke  resource_route
     route    resources :high_scores
    invoke  scaffold_controller
    create    app/controllers/high_scores_controller.rb
    invoke    erb
    create      app/views/high_scores
    create      app/views/high_scores/index.html.erb
    create      app/views/high_scores/edit.html.erb
    create      app/views/high_scores/show.html.erb
    create      app/views/high_scores/new.html.erb
    ...
```

Figura 2.8: Comando `generate scaffold` e arquivos criados
Fonte: Adaptada de Rails (2014)

2.5.3 Scaffolding no Framework Grails

Grails é um *framework* completo de desenvolvimento de aplicações *web*, com código aberto e lincença *Apache*. Foi construído para a Máquina Virtual Java (do inglês *Java Virtual Machine* ou JVM). Utiliza a linguagem de programação *Groovy* e o paradigma de convenção sobre configuração. Programadores conseguem ser produtivos em um ritmo constante durante o processo de desenvolvimento de aplicações *web* (PIVOTAL, 2014).

A técnica de *scaffolding* foi movida para um *plugin* nas versões mais recentes do *Grails*. Desacoplamento de funcionalidades é um recurso poderoso dos *frameworks* e reduz dependências desnecessárias. O *plugin* é adicionado por padrão, mas o programador pode removê-lo caso decida não utilizar a técnica.

O *framework Grails* possui dois tipos de *scaffolding*: dinâmico, utilizando comandos `static` nas próprias classes de controle; e estático, utilizando a linha de comando.

Os Códigos 2.13 e 2.14 mostram como realizar a geração automática, em tempo de execução, do controlador `HighScoreController`, ou seja, aplicação de *scaffolding* dinâmico. O *framework* busca por um atributo estático com nome `scaffold`. Se encontrado e seu valor for um booleano `true` (Código 2.13), o *framework* irá utilizar convenção de nomes para criar métodos referentes à classe de modelo `HighScore`. Caso necessário, o desenvolvedor pode alterar esse comportamento passando explicitamente o nome da classe a ser referenciada pelo *framework*. No exemplo do Código 2.14, o *framework* irá referenciar a classe de modelo com nome `Placar`.

Código 2.13: Scaffolding Dinâmico na classe `HighScoreController` para o modelo `HighScore`

```
1 //Scaffolding para o modelo HighScore
2 class HighScoreController {
3     static scaffold = true
4 }
```

Código 2.14: Scaffolding Dinâmico na classe `HighScoreController` para o modelo `Placar`

```
1 //Scaffolding para o modelo Placar
2 class HighScoreController {
3     static scaffold = Placar
4 }
```

As ações implementadas em tempo de execução são as seguintes: `index`, `show`, `edit`, `delete`, `create`, `save` e `update`. Essas ações representam métodos na classe de controle e interfaces CRUD na camada de visão. Novos métodos podem ser adiciondos ou sobrescritos (Código 2.15).

Código 2.15: Adicionando e sobrescrevendo métodos na classe `BookController`

```
1 class BookController {
2     static scaffold = Livro
3
4     def changeAuthor() {
5         def b = Book.get(params.id)
6         b.author = Author.get(params["author.id"])
7         b.save()
8     }
9
```

```
10      def show() {
11          def book = Book.get(params.id)
12          log.error(book)
13          [bookInstance : book]
14      }
15  }
```

O programador provavelmente precisará personalizar a lógica ou aparência de uma aplicação. Isso não é possível com *scaffolding* dinâmico. Sobrescrever muitos métodos acaba tendo o mesmo efeito que codificá-los manualmente. Com geração estática esses problemas são solucionados (ROCHER et al., 2014). Após a execução de um comando, toda a aplicação fica disponível ao programador em tempo de compilação.

Diferente do *framework Rails*, o *Grails* não possui um gerador de modelos. Portanto, todos os geradores disponíveis dependem da entrada de classes de modelo. O Código 2.16 mostra os 3 comandos possíveis para geração estática das camadas de controle (`generate-controller`), visão (`generate-views`), ou ambas (`generate-all`). As mesmas ações implementadas em tempo de execução são geradas em tempo de compilação. Os comandos são executados no terminal do sistema operacional do desenvolvedor.

Código 2.16: Comandos para geração estática no framework Grails

```
1  //Gerador da camada de controle
2  > grails generate-controller HighScore
3
4  //Gerador da camada de visao
5  > grails generate-views HighScore
6
7  //Gerador da camada de apresentacao
8  > grails generate-all HighScore
```

2.6 Trabalhos Similares

Existem vários estudos relacionados à geração automática de código. Apesar disso, apenas alguns estão relacionados às tecnologias *web*. É importante ressaltar que a filosofia básica do trabalho desenvolvido nesta dissertação independe de tecnologias, e é encontrada em vários trabalhos correlatos: fornecer um sistema pronto a partir de alguma representação da camada de modelo. Esta representação pode ser por diagramas UML (*Unified Modeling Language*), classes Java, arquivos XML, tabelas relacionais, etc (SILVA et al., 2011; DAISSAOUI, 2010; MRACK et al., 2006). Além disso, quase todas as IDEs mais modernas possuem mecanismos de geração de código a partir dessas representações (NETBEANS, 2010; SANCHEZ, 2008; MALYSHEV, 2006).

O trabalho de Silva et al. (2011) desenvolve uma ferramenta para criação de sistemas *web* CRUD utilizando as tecnologias *Servlets*, JSP (*JavaServer Pages*), *Hibernate* e *jQuery*. A entrada do programa são *scripts* SQL para criação de tabelas relacionais. A partir dessas tabelas, o programador deve realizar algumas configurações antes que os códigos-fonte sejam gerados. O gerador cria as classes da camada de modelo baseado no *framework Hibernate*, a camada de controle utiliza *Servlets* e a camada de visão são as páginas JSP com *jQuery*. O programador deve criar uma nova aplicação, com pastas apropriadas para as camadas MVC, e copiar manualmente todos os arquivos gerados pela ferramenta.

O trabalho de Daissaoui (2010) cria um sistema *web* CRUD a partir de um diagrama UML das classes do projeto, independente de linguagens de programação ou tecnologias. O algoritmo criado no estudo utiliza as tecnologias JSP e *Struts* como base. A estrutura do *framework Struts* é mapeada em metamodelos. O programa utiliza essa estrutura para desenvolver o algoritmo de geração de dados. Esse algoritmo usa uma lista genérica de métodos, ou seja, pode ser adaptado conforme casos de uso. A partir disso, é gerado código para os 4 métodos CRUD. As regras desse algoritmo são mapeadas pelo *framework* EMF (*Eclipse Modeling Framework*) da IDE *Eclipse*, gerando um arquivo XML que poderá ser utilizado para implementar o mesmo algoritmo em diferentes linguagens de programação.

O trabalho de Mrack et al. (2006) utiliza uma abordagem um pouco diferente dos citados anteriormente. A entrada do programa são classes Java. A configuração é feita utilizando anotações nas próprias classes. Apesar disso, o programa não gera as camadas de controle e visão para futuras edições, caso o programador julgue necessário. O sistema CRUD gerado precisa ser conectado manualmente a alguma tecnologia de armazenamento persistente. Por ser baseado em eventos, se o programador quiser novas funcionalidades (novas anotações), terá que codificar tudo manualmente.

A maioria das IDEs modernas possuem mecanismos de geração automática de sistemas CRUD, mas com algumas restrições significativas: suportam somente a tecnologia JSF (*JavaServer Faces*); suportam somente a tecnologia JPA (*Java Persistence API*); alterações nos *templates*, quando possíveis, são custosas e não-documentadas. A IDE *Netbeans* aceita como entrada tabelas relacionais ou entidades JPA (NETBEANS, 2010). A IDE *IntelliJ IDEA* possui essa funcionalidade mas só utiliza entidades JPA como entrada (MALYSHEV, 2006). A IDE *Eclipse* não possui geração de aplicações CRUD por padrão, mas possui um *plugin* para esse propósito (SANCHEZ, 2008). Projetos criados no *Play* podem ser exportados para o *Eclipse* ou *Intellij IDEA*. Das IDEs citadas, somente a *Intellij IDEA* suporta o *framework Play* nativamente.

O trabalho proposto é específico para o *framework Play*, portanto somente assimila as idéias básicas dos seus correlatos. A representação da camada de modelo é feita somente por classes Java. Toda a conexão com o banco de dados, inclusive a criação das tabelas, é feita

automaticamente. O sistema final gera todos os códigos das camadas de visão e controle, possibilitando maior personalização. A aplicação gerada utiliza tecnologias do mercado atual.

2.7 Considerações Finais

Este Capítulo apresentou os conceitos, ferramentas, convenções, padrões arquiteturais de projeto e técnicas utilizados como fundamentação teórica para o desenvolvimento de um metaprograma gerador de código. Todos os tópicos apresentados neste Capítulo são utilizados na implementação desse metaprograma ou no entendimento do sistema CRUD (programa-objeto) gerado pelo mesmo.

Os conceitos de metadados e reflexão são a base das APIs *Annotation* e *Reflection*, ferramentas responsáveis pela metaprogramação na linguagem Java. Metaprogramas geradores de código precisam necessariamente utilizar essas ferramentas, direta (Java APIs) ou indiretamente (bibliotecas de terceiros que implementam as Java APIs). Convenções precisam ser adotadas para minimizar configurações desnecessárias. O padrão MVC é a estrutura básica do *framework web* utilizado e do programa-objeto gerado. A técnica de *scaffolding* utiliza convenções, padrões arquiteturais e metaprogramas bem definidos para gerar programas-objeto, que neste trabalho será um sistema CRUD funcional.

Capítulo 3

Metaffolder, Aplicando a Técnica de Scaffolding

Este Capítulo apresenta o sistema *Metaffolder*, desenvolvido a partir dos conceitos apresentados no Capítulo anterior. O principal objetivo desse sistema é a implementação e aplicação da técnica de *scaffolding* descrita na fundamentação teórica. No *Metaffolder*, as camadas de visão e controle de um sistema CRUD são geradas automaticamente, a partir da análise das informações referentes às classes da camada de modelo.

Foi proposto um esquema para representar a técnica de *scaffolding* utilizada no sistema *Metaffolder*. Este esquema foi dividido em 3 partes: entrada, *Metaffolder*, e saída. Todos os componentes desse esquema são descritos brevemente na Seção 3.1 e aprofundados nas Seções 3.2, 3.3 e 3.4, respectivamente.

A Seção 3.2 detalha a entrada do sistema: a estruturação das pastas e arquivos, as configurações e anotações disponíveis, e como as classes de modelo devem ser anotadas. A Seção 3.3 apresenta o sistema *Metaffolder*, seu diagrama de classes e todos os conceitos e tecnologias utilizados no seu desenvolvimento. A Seção 3.4 descreve a saída do sistema: todos os arquivos criados são exibidos, incluindo códigos-fonte da camada de controle e páginas HTML da camada de visão. A saída do terminal também é exibida.

3.1 Esquema Proposto

O esquema proposto foi criado com o objetivo de modularizar a aplicação da técnica de *scaffolding* em sistemas CRUD com arquitetura MVC. Esta estrutura modular foi implementada no sistema *Metaffolder* (Figura 3.1).

O esquema está dividido em 3 partes principais: *Entrada*, *Metaffolder*, e *Saída*. Cada parte está subdividida em estruturas, representadas pelas linhas tracejadas. Cada estrutura está dividida em um ou mais componentes, que são abstrações da teoria envolvida na técnica de *scaffolding*. Cada componente é explicado a seguir.

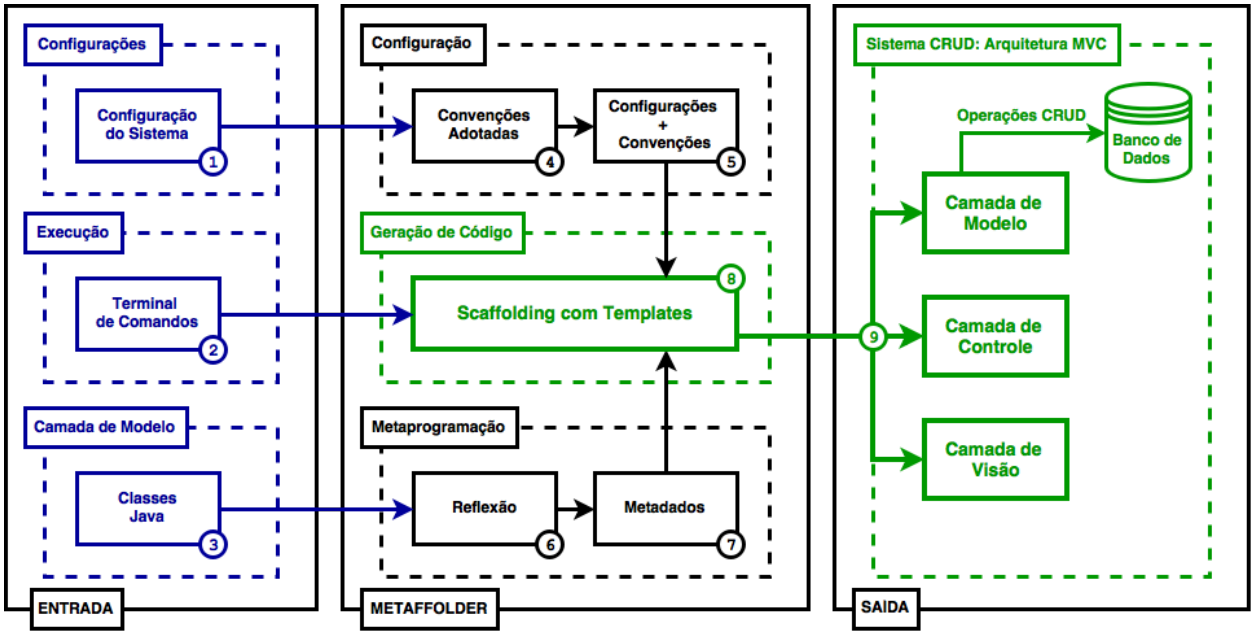


Figura 3.1: Esquema proposto para implementação da técnica de scaffolding

Entrada

1. *Configuração do Sistema*: caso necessário, é disponibilizado ao usuário várias configurações para alterar as convenções adotadas pelo sistema. Essas configurações garantem ao desenvolvedor maior flexibilidade de personalização. Com isso o sistema segue o paradigma de convenção sobre configuração.

2. *Terminal de Comandos*: o usuário do sistema utiliza o terminal, estendido do *framework Play*, para iniciar o processo de análise de classes e geração automática de código. Isso é feito através da execução do comando `metaffold`.

3. *Classes Java*: programas que implementam a técnica de *scaffolding* esperam que o desenvolvedor entre com algum tipo de representação da camada de modelo, tais como: diagramas UML, tabelas relacionais, classes orientadas a objetos, etc. Neste trabalho, a camada de modelo é representada por classes Java, anotadas com a anotação `@Entity` da biblioteca JPA, requisito do sistema *Metaffolder*. As anotações `@Html`, `@Html.Input`, `@Html.TextArea` e `@Exclude` são fornecidas ao usuário para alteração do comportamento padrão do sistema.

Metaffolder

4. *Convenções Adotadas*: Todas as convenções do sistema são representadas por este componente. Geralmente são utilizadas convenções de nomes de pastas e arquivos, tipos de chaves primárias, formatação de código, etc. Uma convenção é basicamente

uma configuração padrão, ou seja, uma configuração adotada pelo sistema caso seu usuário não altere configurações na entrada.

5. *Configurações + Convenções*: responsável pela união de todas as configurações do desenvolvedor e convenções do sistema. Essa união é representada pela classe `MetaffoldConf` e é utilizada como entrada para o gerador de código baseado em *templates*.

6. *Reflexão*: é a responsável pela introspecção das classes de entrada e pelo mapeamento de todas as características consideradas importantes, tais como nome do modelo, atributos, herança, chaves primárias, etc. Resumidamente, recebe classes Java como entrada e produz metadados referentes a essas classes como saída.

7. *Metadados*: é basicamente a representação computacional das estruturas mapeadas pelo componente da reflexão. É toda a informação obtida no processo de reflexão, armazenada em duas classes criadas para este propósito: `MetaModel` e `MetaField`. Estas classes são usadas como entrada para o gerador de código baseado em *templates*.

8. *Scaffolding com Templates*: O núcleo do *Metaffolder*. Possui toda a lógica para a geração de código, inclusive geradores internos para manipulação de *templates*. Mecanismos internos transformam convenções, configurações e metadados em estruturas de contexto, que são usadas juntamente com *templates* predefinidos para a geração dos códigos-fonte da arquitetura MVC.

Saída

9. *Sistema CRUD*: Este componente representa a saída do sistema *Metaffolder*. As camadas de controle e visão são criadas. A camada de modelo é reutilizada para compor a arquitetura MVC do sistema CRUD criado. Esta saída será utilizada pelos desenvolvedores como auxílio para o desenvolvimento de sistemas mais complexos.

3.2 Entrada

Normalmente um programa espera que seu usuário adicione uma entrada para que cálculos computacionais sejam realizados e uma saída seja produzida. A entrada do sistema *Metaffolder* utiliza 3 estruturas do *framework Play*: seu mecanismo de configuração, seu terminal de comandos e sua arquitetura MVC. Adicionalmente, anotações são disponibilizadas ao usuário para alteração do comportamento padrão do sistema *Metaffolder*.

Tanto o mecanismo de configuração quanto a arquitetura MVC são organizados em diretórios. A Figura 3.2 exibe os diretórios e arquivos relevantes ao sistema *Metaffolder*, para uma aplicação de exemplo chamada `webapp`:

- O diretório `webapp` é a raiz da aplicação. Este diretório é nomeado de acordo com o nome da aplicação.
- O diretório `app` contém os arquivos compiláveis da aplicação, divididos em 3 pastas, cada uma representando uma camada da arquitetura MVC: `models`, `views`, `controllers`.
- Os arquivos da pasta `models` são a entrada do sistema *Metaffolder*. Ao aplicar a técnica de *scaffolding*, são criados arquivos Java na pasta `controllers` e arquivos HTML na pasta `views`, representando a saída do sistema *Metaffolder*.
- O diretório `conf` possui o arquivo `application.conf`, responsável pela configuração da aplicação. Com esse arquivo, o usuário pode alterar o comportamento do sistema.

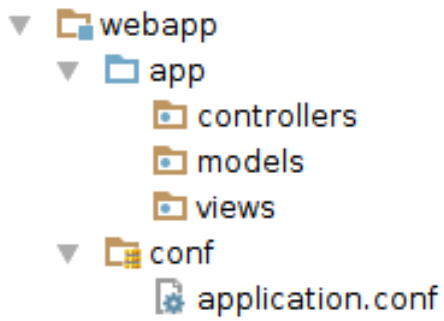


Figura 3.2: Anatomia *Play*: diretórios e arquivos relevantes ao sistema *Metaffolder*

O terminal do *framework Play* fornece ao seu usuário vários comandos avançados para maior controle da aplicação, tais como: `compile`, `test`, `run`, `inspect`, etc. Este terminal foi estendido através do sistema *Metaffolder* para fornecer ao seu usuário o comando `metaffold`, responsável pela aplicação da técnica de *scaffolding*. Um sistema CRUD será criado a partir das classes de modelo, das convenções adotadas e das configurações do desenvolvedor.

3.2.1 Configurações

As configurações são normalmente inseridas em linhas, cada uma representando uma configuração do tipo par chave-valor. Por exemplo, um arquivo `application.conf` com a linha `metaffold.domain = "minha.empresa"` adiciona a configuração com chave

`metaffold.domain` e valor `"minha.empresa"`, responsável por adicionar o subpacote `minha.empresa` a todos os códigos-fonte gerados.

O Código 3.1 exibe as configurações disponíveis ao desenvolvedor. Um valor padrão é adotado caso as configurações não sejam inseridas no arquivo `application.conf`. Os valores adicionados são apenas um exemplo. É aconselhável que o desenvolvedor utilize as configurações padrão sempre que possível, pois agrega os benefícios do paradigma de convenção sobre configuração, mencionados no Capítulo anterior.

Código 3.1: Exemplo de configurações adicionadas ao arquivo `application.conf`

```
1 //Configuracoes                          // Valores padrao
2 metaffold.domain = "minha.empresa"       // ""
3 metaffold.modeslFolder = "modelos"       // "models"
4 metaffold.controllersFolder = "controles" // "controllers"
5 metaffold.viewsFolder = "paginas"        // "views"
6 metaffold.controllerSuffix = "s"         // "Controller"
7 metaffold.formSuffix = "F"               // "Form"
8 metaffold.listSuffix = "L"               // "List"
9 metaffold.showSuffix = "S"               // "Show"
```

O Código 3.2 exibe a estrutura gerada a partir de uma classe de exemplo `minha.empresa.modelos.Usuario` e das configurações do Código 3.1. O comportamento do sistema foi personalizado de acordo com as necessidades do desenvolvedor.

Código 3.2: Comportamento personalizado pelo desenvolvedor

```
1 minha/empresa/modelos/Usuario
2 minha/empresa/controles/Usuarios
3 minha/empresa/paginas/usuario/usuarioF
4 minha/empresa/paginas/usuario/usuarioL
5 minha/empresa/paginas/usuario/usuarioS
```

O sistema adotará automaticamente as configurações padrão se nenhuma configuração do *Metaffolder* for encontrada no arquivo `application.conf`. O Código 3.3 exibe a estrutura gerada a partir de uma classe de exemplo `models.Usuario`, sem configurações adicionais.

Código 3.3: Comportamento padrão do sistema

```
1 models/Usuario
2 controllers/UsuarioController
3 views/usuario/usuarioForm
4 views/usuario/usuarioList
5 views/usuario/usuarioShow
```

3.2.2 Anotações

Em um sistema CRUD, normalmente cada atributo de uma entidade é representado na camada de visão por elementos HTML de formulário (*form*) ou por colunas em uma tabela (*table*). Os elementos HTML utilizados pelo sistema são `input`, `textarea` e `select`. O elemento `input` pode ser personalizado utilizando diversos tipos predefinidos na linguagem HTML, tais como: `radio`, `checkbox`, `number`, `date`, `file`, etc. Seu tipo padrão é o `text` (W3C, 2014b).

O tipo do atributo de uma entidade é mapeado pelo sistema e transformado no seu correspondente da camada de visão. A Tabela 3.1 exibe os tipos de atributos da camada de modelo e seus correspondentes na camada de visão. O tipo `select` é mapeado somente quando o atributo `List` representa uma lista de entidades (classes da camada de modelo). O tipo `textarea` só pode ser mapeado manualmente através de uma anotação interna do sistema (`@Html.TextArea`).

Tabela 3.1: Mapeamento entre tipos da camada de modelo e visão

Tipos	
Modelo	Visão
String	input type="text"
Number	input type="number"
Date	input type="date"
Boolean	input type="checkbox"
File	input type="file"
Enum	imput type="radio"
List	select
—	textarea

O sistema fornece ao desenvolvedor duas anotações para alterar o processo padrão de mapeamento de tipos entre camada de modelo e visão: `@Html.Input` e `@Html.TextArea`. Elas servem para atribuir significados de visão aos atributos da camada de modelo. Os tipos de *inputs* predefinidos na linguagem HTML podem ser passados na anotação `@Html.Input`. Por exemplo, um atributo `senha` pode ser considerado um elemento HTML de senha, se anotado com a anotação `@Html.Input("password")`.

A anotação `@Html` serve para alteração de características mais gerais de um elemento de formulário HTML, tais como: rótulo, visiblidade, permissões de edição, ou se o atributo será ignorado da camada de visão. Atributos representando chaves-primárias são ignorados por padrão. O comportamento padrão para rótulos é o próprio nome do atributo.

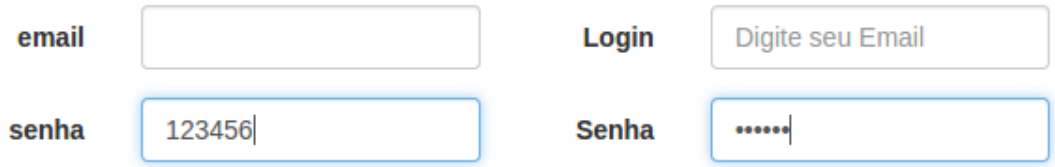
Todas as classes de modelo precisam ser anotadas com `@javax.persistence.Entity` para que elas sejam encontradas pelo gerador de metadados do sistema. Ela é a anotação padrão da biblioteca JPA para representar entidades persistentes. Apesar de cobrir a maioria dos casos de uso atualmente, o desenvolvedor pode implementar um novo gerador de metadados caso não utilize essa anotação no seu projeto. O sistema fornece a anotação `@Exclude` caso o desenvolvedor não queira que alguma classe participe do processo de *scaffolding*. Classes anotadas com `@Exclude` não serão encontradas pelo gerador de metadados e o código das camadas de visão e controle referente a essa classe não será gerado.

O Código 3.4 exibe a classe `ExemploUsuario`, devidamente anotada. Possui a anotação `@Entity` para ser considerada uma entidade válida. A anotação `@Id` também faz parte da biblioteca JPA e transforma o atributo `id` em chave primária. As anotações `@Html.Input("password")` e `@Html.Input("email")` adicionam significado aos atributos `senha` e `email` respectivamente. A anotação `@Html` personaliza os rótulos na camada de visão.

Código 3.4: Classe `ExemploUsuario` com as anotações do sistema *Metaffolder*

```
1  package sub.pkg.models.exe.mplo;
2  // imports das anotacoes
3
4  @Entity
5  public class ExemploUsuario {
6
7      @Id
8      private Long id;
9
10     @Html("Login")
11     @Html.Input(value = "email", placeholder = "Digite seu Email")
12     private String email;
13
14     @Html("Senha")
15     @Html.Input("password")
16     private String senha;
17
18     //-- getters e setters --//
19 }
```

A Figura 3.3 mostra os atributos da classe `ExemploUsuario` gerados na camada de visão sem (Figura 3.3(a)) e com (Figura 3.3(b)) as anotações do sistema *Metaffolder*. Na Figura 3.3(a) o campo `senha` é apenas uma entrada de texto qualquer, podendo significar qualquer coisa. Na Figura 3.3(b), o mesmo campo representa um entrada de senha e seu comportamento muda. Além disso, a anotação `@Html` adiciona rótulos mais significativos e apresentáveis ao usuário.



(a) Sem anotações (b) Com anotações

Figura 3.3: Adicionando significado aos atributos de uma classe

3.3 *Metaffolder*

Esta seção descreve o sistema *Metaffolder* e todos os conceitos e tecnologias utilizados em sua modelagem e implementação. O diagrama de classes do sistema é exibido para melhor entendimento de sua arquitetura. É explicado o funcionamento das principais tecnologias presentes na implementação do sistema.

3.3.1 Modelagem

Diagramas auxiliam visualmente o entendimento de um sistema e facilitam sua implementação. Apesar disso, métodos descritivos também são necessários, pois elucidam responsabilidades e comportamentos de cada classe. Cartões CRC (Classe-Responsabilidades-Colaboradores) (BECK; CUNNINGHAM, 1989) conseguem descrever com clareza cada classe do sistema.

A Figura 3.4 apresenta o diagrama utilizado para modelar o sistema. O estereótipo «`uses`» indica que um lado da relação usa o outro lado de alguma maneira. O estereótipo «`creates`» indica que um lado da relação instancia um objeto referente ao outro lado. O estereótipo «`classes`» indica que um lado da relação fornece todas as classes do *framework Play* ao outro lado. O estereótipo «`confs`» indica que um lado da relação fornece todas as configurações do *Play* ao outro lado.

O diagrama de classes da Figura 3.4 foi dividido em 3 pacotes principais:

metaffold.meta metaprogramação do sistema. Possui as classes responsáveis pela criação e armazenamento dos metadados que representam as classes de modelo, também chamados de metamodelos e meta-atributos. Também possui anotações que modificam o comportamento do gerador de metadados e das páginas HTML criadas.

metaffold.codegen contém as classes responsáveis pela geração de código compilável. Utiliza os metadados criados no pacote `metaffold.meta`. O núcleo do sistema se encontra nesse pacote: o mecanismo que realiza o *scaffolding* utilizando convenções, configurações, metadados e arquivos de *templates*.

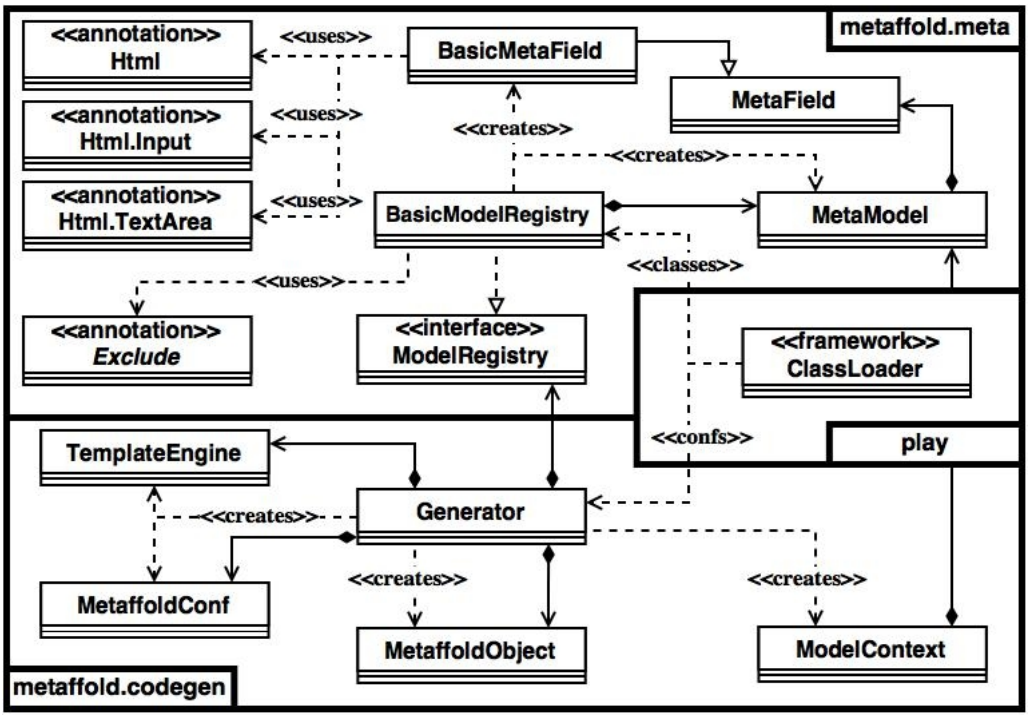


Figura 3.4: Diagrama de classes do sistema *Metaffolder*

play apesar de não ser um pacote do sistema *Metaffolder*, é fundamental para o seu funcionamento, portanto foi incluso no diagrama de classes. Esse pacote contém o *classloader* do *framework Play*, responsável pela conexão entre o *framework* e o sistema *Metaffolder*. Ele fornece ao sistema todas as classes de modelo e configurações do *Play*.

Os cartões CRC são uma ferramenta de análise e modelagem de sistemas, utilizada para definir suas classes e responsabilidades. Cada cartão é dividido em 3 partes (BECK; CUNNINGHAM, 1989):

Nome nome da classe descrita. Pode conter campos opcionais para melhor descrição da classe (*superclasse, subclasse, implementa, implementada por*).

Responsabilidades representa o comportamento da classe, suas obrigações a serem cumpridas.

Colaboradores classes que auxiliam na execução de comportamentos mais complexos.

A união entre modelos visuais e descritivos resulta em uma implementação mais fiel ao sistema proposto. O comportamento e implementação do sistema *Metaffolder* pode ser entendido mais facilmente com a análise tanto dos cartões CRC quanto do diagrama da Figura 3.4. As classes do sistema são descritas a seguir.

A classe `ClassLoader` desempenha um papel fundamental: é a responsável pela conexão entre *Play* e *Metaffolder*. Fornece acesso a todas as classes carregadas pelo *framework*, utilizado pela classe `BasicModelRegistry` para encontrar as classes da camada de modelo. Também fornece os valores de todas as chaves de configuração do *framework Play*, utilizados pela classe `MetaffoldConf` para criação da configuração do sistema *Metaffolder*.

As anotações `@Html.Input` e `@Html.TextArea` são responsáveis por adicionar significado aos atributos de uma classe de modelo. Também alteram o comportamento padrão da geração dos atributos de uma classe na camada de visão, juntamente com a anotação `@Html`. As 3 anotações são utilizadas pela classe `BasicMetaField`. A anotação `@Exclude` é utilizada pela classe `BasicModelRegistry` como um filtro de classes, ou seja, não será gerado código automático para classes anotadas com essa anotação.

A classe `MetaModel` encapsula todas os atributos presentes nas classes de modelo, utilizando a classe `MetaField`. Um metamodelo se relaciona com vários meta-atributos. Um metamodelo também armazena sua chave-primária, muito utilizada pelo sistema para consultas e geração adequada das páginas de visão.

A classe `MetaField` armazena todas as características de um atributo, incluindo suas anotações, tipo, nome, tipo HTML correspondente, etc. Apesar disso, todas as atribuições de valores (métodos `set`) e configurações baseadas em anotações (métodos `isAnnotationPresent`) são feitas pela sua classe filha `BasicMetaField`.

A classe `ModelRegistry` é considerada a geradora de metamodelos do sistema. É a responsável por criar e armazenar todos os metamodelos e seus respectivos meta-atributos. Fornece uma interface para personalização no modo de busca das classes de modelo. A `BasicModelRegistry` é a implementação padrão utilizada no sistema *`Metaffolder`*. Ela utiliza o *classloader* do *Play* para buscar todas as classes do *framework* que estejam anotadas com a anotação `@javax.persistence.Entity`.

A classe `MetaffoldConf` representa o objeto de configuração do sistema. Todas as configurações do `framework Play` são passadas a ela pelo `Generator`. A classe é a responsável pela filtragem somente das configurações do sistema *Metaffolder*. É criada pela classe `Generator`.

A classe `ModelContext` representa o contexto que será utilizado pela ferramenta de *templates* para geração de código. Ela constrói esse contexto a partir dos metadados criados pela classe `ModelRegistry` e das configurações da classe `MetaffoldConf`. É criada pela classe `Generator`.

A classe `MetaffoldObject` representa o arquivo que será gerado com a aplicação da técnica de *scaffolding*. Possui 3 atributos, todos utilizados pela ferramenta de *templates* do sistema: arquivo de *template*, contexto (classe `ModelContext`) e arquivo de saída.

A classe `TemplateEngine` é considerada a ferramenta de *templates* do sistema. É a responsável por transformar contextos e *templates* em código-fonte compilável. Essa transformação é o último passo da técnica de *scaffolding*.

A classe `Generator` corresponde ao núcleo do sistema *Metaffolder*. É a responsável pela criação e coordenação das várias classes necessárias para a aplicação da técnica de *scaffolding*. Recebe todas as configurações do *framework Play* e as repassa para a classe `MetaffoldConf`. Obtém os *templates* correspondentes a cada contexto. Cada instância de `MetaffoldObject` representa um arquivo a ser gerado pela classe `TemplateEngine`.

3.3.2 Implementação

No esquema proposto anteriormente (Figura 3.1), o sistema *Metaffolder* foi dividido em 3 estruturas: *configuração*, *metaprogramação*, e *geração de código*. Cada estrutura possui um papel importante no funcionamento do sistema e serão explicadas a seguir.

Configuração

Quanto menor o tempo gasto configurando um sistema, maior a produtividade. Apesar disso, é impossível que as configurações padrão sirvam para diversos casos de uso, fazendo com que meios de configuração sejam necessários para casos mais particulares de uso do sistema.

O sistema *Metaffolder* utiliza várias convenções que podem ser configuradas caso necessário. A localização e nomeação dos arquivos a serem criados depende do nome da classe de modelo, do seu diretório e das configurações do sistema.

O *framework Play* utiliza internamente a bilbioteca `typesafe.config` (TYPESAFE, 2015) como mecanismo de configuração. Um arquivo chamado `reference.conf` é utilizado para armazenar todas as configurações padrão do sistema *Metaffolder*. Este arquivo é interno, transparente ao desenvolvedor e representa o componente *Convenções Adotadas* do esquema proposto (Figura 3.1).

O arquivo `reference.conf` utilizado pelo sistema *Metaffolder* é exibido no Código 3.5. Todos os valores adotados foram baseados nas convenções de nomenclatura utilizadas pelo *framework Play* e pela comunidade Java.

Código 3.5: Arquivo de configuração `reference.conf`, utilizado pelo *Metaffolder*

```
metaffold.domain = ""
metaffold.modelsFolder = "models"
metaffold.controllersFolder = "controllers"
metaffold.viewsFolder = "views"
metaffold.controllerSuffix = "Controller"
metaffold.formSuffix = "Form"
metaffold.listSuffix = "List"
metaffold.showSuffix = "Show"
```

A resolução do valor para determinada chave é feita seguindo o algoritmo do Código 3.6. Resumidamente, quando um usuário insere uma chave no arquivo `application.conf`, ele automaticamente sobrescreve o valor padrão dessa chave, caso esteja contida no arquivo `reference.conf`. Isso proporciona maior grau de personalização ao usuário do sistema.

Código 3.6: Algoritmo para resolução de chaves de configuração

```
algoritmo
    se chave contida em application.conf então
        retorna valor da chave
        fim_algoritmo
    senão
        para_cada reference.conf encontrado faça
            se chave contida em reference.conf então
                retorna valor da chave
                fim_algoritmo
            fim_se
        fim_para
    fim_se
    lança_exceção "chave nao encontrada"
fim_algoritmo
```

Com o *classloader* do *framework Play*, é possível obter um objeto proveniente da biblioteca `typesafe.config`. Uma filtragem é necessária, pois o objeto possui todos os pares de configuração chave-valor do *framework Play*, incluindo o `application.conf` e configurações de outros módulos que também utilizem arquivos `reference.conf`. O `Generator` delega à classe `MetaffoldConf` a filtragem dos pares chave-valor referentes ao sistema *Metaffolder*. Isso é feito no construtor da classe `MetaffoldConf`. Portanto, após instanciada, essa classe representa o componente *Configurações + Convenções* do esquema proposto (Figura 3.1). A Figura 3.5 resume o funcionamento do mecanismo de configuração do *Metaffolder*.

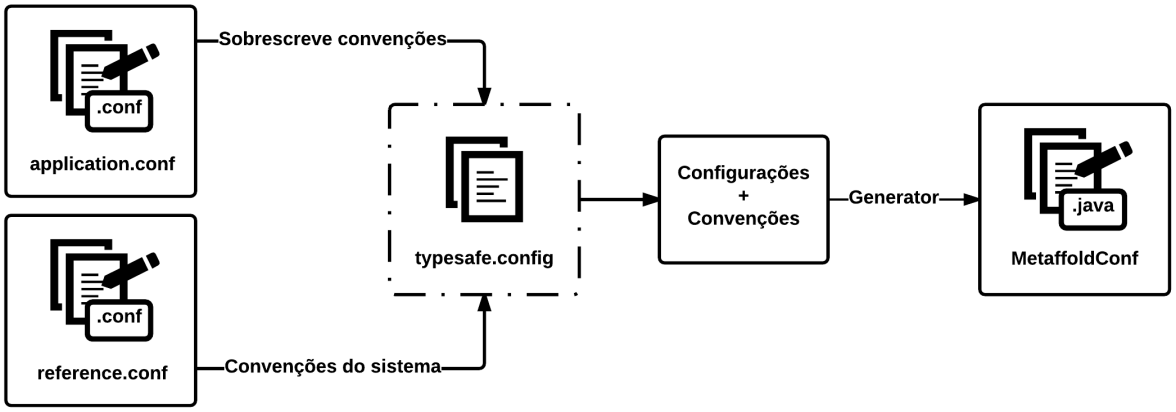


Figura 3.5: Mecanismo de configuração do sistema *Metaffolder*

Metaprogramação

A interface `ModelRegistry` permite que desenvolvedores criem seus próprios métodos para busca e armazenamento de modelos. Apesar disso, o sistema já possui uma implementação básica que cobre diversos casos de uso. Ela busca todas as classes anotadas com `@javax.persistence.Entity`, anotação padrão da biblioteca JPA utilizada em diversos *frameworks* de persistência de dados. Outros prováveis casos de uso: localização dos modelos utilizando outra anotação, localização por pacote, etc.

A classe `BasicModelRegistry` utiliza a biblioteca `org.reflections` (MAMO, 2013), uma ferramenta que facilita o uso de reflexão na linguagem Java e inclusa nativamente no *framework Play*. A implementação da biblioteca utiliza os mecanismos de reflexão da API *Java Reflections*.

A classe `ClassLoader` fornece ao `BasicModelRegistry` o *classloader* do *framework Play*, com todas as suas classes e arquivos carregados, como mostrado anteriormente no diagrama de classes da Figura 3.4. Esse *classloader* é utilizado pela biblioteca `org.reflections` para a localização das classes de modelo do sistema CRUD.

A biblioteca de reflexão precisa saber onde e o que pesquisar. O *classloader* indica onde pesquisar. A classe `TypeAnnotationsScanner`, da própria biblioteca, indica o tipo da pesquisa. A classe `@Entity` indica o que pesquisar. Metamodelos (classe `MetaModel`) e seus respectivos meta-atributos (classe `MetaField`) são criados a partir das classes de modelo obtidas neste processo. A Figura 3.6 resume o mecanismo responsável pela criação dos metamodelos do sistema *Metaffolder*.

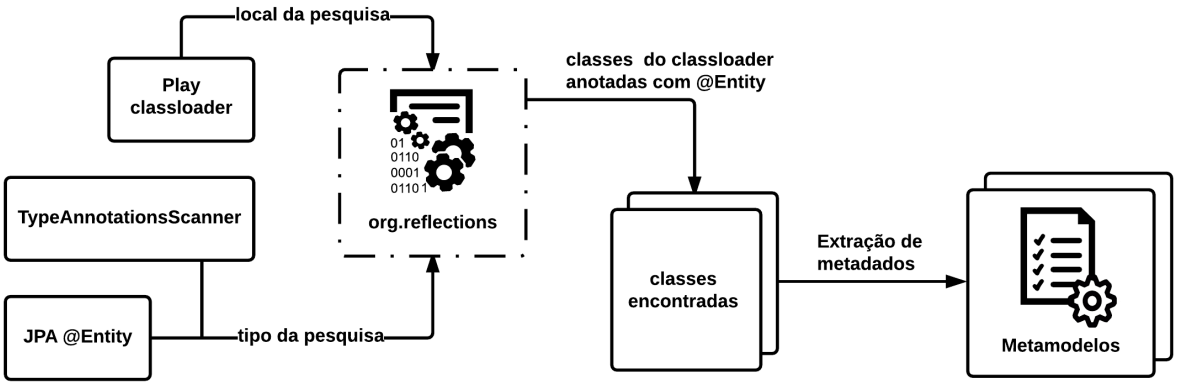


Figura 3.6: Mecanismo de criação de metamodelos do sistema *Metaffolder*

Geração de Código

A classe `Generator` representa o núcleo do sistema *Metaffolder*. Ela coordena todas as interações entre as diferentes classes do sistema, como mostrado anteriormente no diagrama da Figura 3.4. Seu papel é a geração de código das camadas de controle e visão.

De acordo com a fundamentação teórica deste trabalho, um contexto depende de metadados, configurações e convenções para ser criado. Analogamente, a classe `ModelContext` depende de uma instância da classe `MetaModel` e de uma instância da classe `MetaffoldConf`.

Um contexto também pode ser considerado como um conjunto de pares chave-valor, representando informações relativas ao metamodelo. As chaves utilizadas para a classe `ModelContext` são descritas a seguir:

- *nameClass*: é o nome da classe de modelo, sem alterações.
- *nameVar*: é o nome da classe de modelo, mas com a primeira letra minúscula. O resultado final segue a convenção de nomenclatura de métodos da linguagem Java.
- *domainDot*: representa o pacote localizado antes do pacote base de armazenamento de classes da camada de modelo, sufixado com o caracter `"."`.
- *dotRelativePackage*: representa o pacote localizado depois do pacote base de armazenamento de classes da camada de modelo, prefixado com o caracter `"."`.
- *keyName*: nome do atributo que representa a chave primária da classe de modelo.
- *keyType*: tipo declarado do atributo que representa a chave primária da classe de modelo.

- *fields*: é uma lista de todos os atributos da classe de modelo, com suas principais características, tais como nome, tipo, tipo HTML, anotações, etc.
- *conf*: é o objeto `MetaffoldConf`. Todas as configurações do sistema podem ser utilizadas em *templates*.

A Figura 3.7 mostra um exemplo de criação de contexto. A classe `ModelContext` é criada a partir da união de informações referentes ao modelo (classes `MetaModel` e `MetaField`) e referentes às configurações do sistema *Metaffolder* (classe `MetaffoldConf`). Todo o conteúdo do contexto criado fica armazenado na classe `ModelContext`. Esta classe é utilizada posteriormente pela *engine* de *templates* para substituição de parâmetros e criação de arquivos.

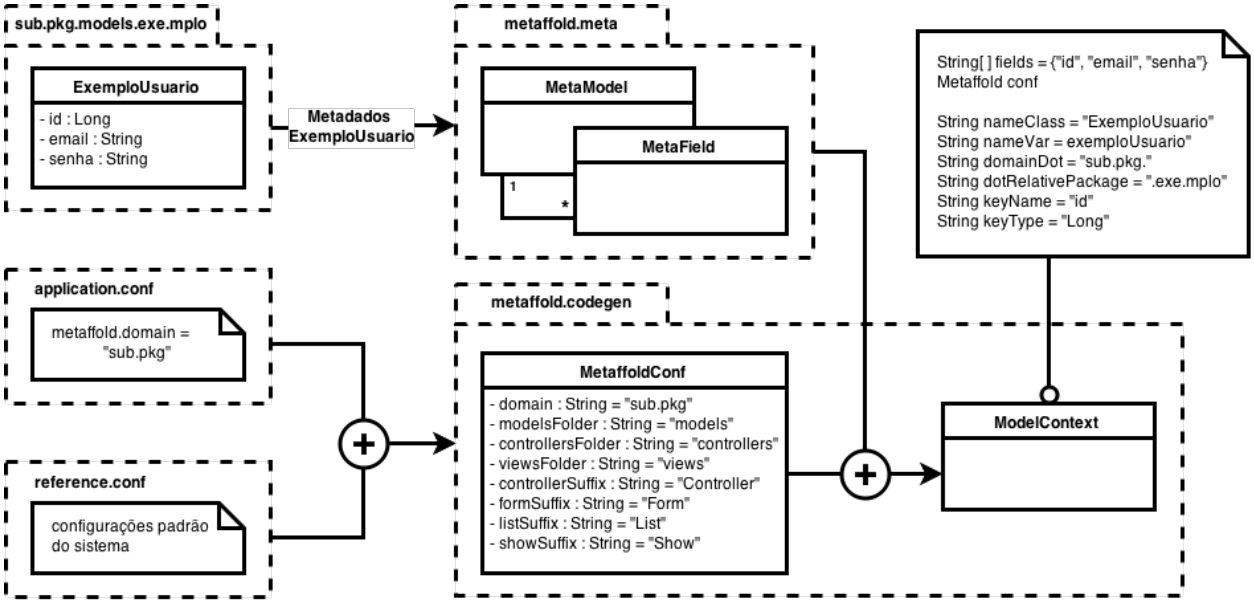


Figura 3.7: Exemplo da criação de um contexto para a classe `ExemploUsuario`

O objeto `MetaffoldObject` é apenas uma facilidade de implementação. Ele armazena o contexto, o arquivo que será criado e o template que será utilizado. Por exemplo, utilizando o contexto da Figura 3.7, seria criado um objeto `MetaffoldObject` com o template `controller.template` e um arquivo `app/sub/pkg/controllers/exe/mplo/MeuExemploController.java`. O mesmo procedimento é realizado para a criação dos arquivos da camada de visão.

Cada modelo gera um contexto. Com as informações do contexto é possível criar os 4 arquivos dependentes do modelo (*controller*, *list*, *show*, *form*). A lista de objetos `MetaffoldObject` representa todos os arquivos que serão criados pelo sistema. Basta utilizar alguma ferramenta para transformação de contextos e *templates* em código, também conhecida como *engine de templates*.

A biblioteca *JMustache* foi escolhida para ser a *engine* de *templates* do sistema. Ela é leve, eficiente, não possui dependências, possui sintaxe simples e aceita qualquer tipo de objeto Java como contexto (BAYNE, 2010). A classe `TemplateEngine` utiliza as funcionalidades da biblioteca *JMustache* para gerar arquivos a partir dos objetos `MetaffoldObject`.

Parâmetros de um template são resolvidos de acordo com o seguinte algoritmo (BAYNE, 2010):

- Se o contexto for um objeto do tipo `Map`, o método `Map.get` será usado.
- Se um método não `void` com o mesmo nome que a variável existir, ele será chamado.
- Se um método não `void getBar` (para a variável `bar`) existir, ele será chamado.
- Se um atributo com o mesmo nome que a variável existir, seu conteúdo será usado.

Na biblioteca *JMustache*, todo texto indicado por chaves duplas é um marcador. Marcadores `{{#x}}` e `{{/x}}` delimitam uma seção com o objeto `x` como contexto. O marcador especial `{{.}}` exibe o valor do contexto atual com o método `toString()`, particularmente útil quando iterando dentro de *arrays* ou listas. Atributos de objetos de um contexto podem ser acessados utilizando a notação de ponto, igual à utilizada na linguagem Java.

O Código 3.7 exibe um exemplo de uso da ferramenta *JMustache*, utilizando o contexto mostrado na Figura 3.7. Tanto a entrada (linha 5) quanto a saída (linha 13) são representadas por variáveis do tipo `String`. O comportamento do sistema *Metaffolder* é semelhante a este exemplo, a única diferença é que a entrada e saída são representadas por arquivos.

Código 3.7: Exemplo de uso da ferramenta *JMustache*

```
1  //contexto da classe ExemploUsuario
2  //ModelContext ctx
3
4  //template de exemplo
5  String template =
6      "Atributos da classe {{nameClass}}" + "\n" +
7      "{{#fields}}"                       + "\n" +
8      "   {{nameVar}}.{{.}}"              + "\n" +
9      "{{/fields}}"                       + "\n" +
10     "controller: {{domainDot}}{{conf.controllersFolder}}
11         {{dotRelativePackage}}.{{nameClass}}{{conf.controllerSuffix}}";
12
13 String saida = Mustache.compiler()
14                       .compile(template)
15                       .execute(ctx);
16
```

```
17 /* saida
18 Atributos da classe ExemploUsuario
19     exemploUsuario.id
20     exemploUsuario.email
21     exemploUsuario.senha
22 controller: sub.pkg.controllers.exe.mplo.ExemploUsuarioController
23 */
```

Os *templates* do sistema foram adaptados de uma aplicação de demonstração do *Play*, inclusa no *framework* e destinada a usuários iniciantes (TYPESAFE, 2014a). Essa decisão se deve ao fato de que o propósito do *scaffolding* é gerar sistemas básicos, funcionais e de fácil entendimento, ou seja, gerar um sistema que servirá como alicerce para o desenvolvimento de um sistema mais complexo.

Todos os *templates* utilizados pelo sistema podem ser sobrescritos pelo desenvolvedor, basta criar um arquivo na pasta `conf/templates` com o mesmo nome do *template* a ser sobrescrito. O Apêndice A exibe o conteúdo do arquivo `controller.template`. Os outros *templates* seguem o mesmo princípio: possuem texto estático e parâmetros, representados por marcadores `{{ }}`. A lista a seguir descreve todos os *templates* utilizados pelo sistema *Metaffolder*:

controller.template modelo para criação das classes de controle do sistema. Possui todos os métodos necessários em um sistema CRUD, levando em consideração verbos do arquivo de rotas. Por exemplo: possui o método `edit` para chamar um formulário de edição (verbo `GET`); e possui um método `update` (verbo `POST`) para salvar essa edição no banco de dados.

form.template responsável pela criação do formulário HTML de um objeto, que pode ser utilizado tanto para criação quanto para edição de objetos. Rotas e métodos de controle verificam se a chave-primária de um objeto é válida: em caso positivo (chave não nula), o formulário é usado para edição; em caso negativo (chave nula) o mesmo formulário é utilizado para criação.

list.template lista todos os objetos de uma determinada classe de modelo. Cada classe possui seu próprio arquivo `list.scala.html`. Cada objeto da lista possui um *link* para suas páginas de alteração e visualização. Também possui uma opção para exclusão do objeto.

show.template exibe um objeto. Como exemplo, essa página poderia ser utilizada em ambientes restritivos, para usuários que possuem privilégios somente de leitura.

application.template representa a classe de controle da aplicação. Normalmente possui métodos de segurança e outros que dizem respeito a aplicação como um todo. O sistema *Metaffolder* utiliza este template para criar o arquivo `Application.java`,

responsável por exibir a página inicial `index.scala.html`, ou seja, possui apenas um método `index()` que invoca essa página.

index.template página inicial do sistema CRUD gerado. Possui *links* para as páginas `list` de cada classe de modelo.

layout.template possui todas as importações de arquivos CSS e *JavaScript* da aplicação criada. Todas as outras páginas HTML herdam características visuais dessa página.

menu.template menu superior do sistema. Possui *links* para todas as páginas `list`, para acesso rápido aos objetos do sistema. Esse menu é responsável por melhorar a navegabilidade do sistema CRUD.

messages.template é o arquivo de internacionalização do sistema CRUD gerado. Todos os textos dependentes de um idioma estão armazenados neste arquivo. A extensão de cada arquivo `messages` representa um idioma e precisam ser criados manualmente, caso necessário. Por exemplo, `messages.en-US` para inglês dos Estados Unidos. A classe `play.i18n.Messages` é a responsável pela internacionalização e também utiliza um mecanismo de par chave-valor. O funcionamento da classe `Messages` é intuitivo e é apresentado no Código 3.8.

Código 3.8: Exemplo de internacionalização

```
1  //Arquivo messages
2  button.label.save = Salvar Endereço
3
4  //Arquivo messages.en-US
5  button.label.save = Save Address
6
7  //Classe de internacionalização
8  String saida = Messages.get("button.label.save")
9  // local padrão   -> saida = Salvar Endereço
10 // Estados Unidos -> saida = Save Address
```

routes.template arquivo de rotas do sistema CRUD. Utiliza um conjunto de linhas para representar a conexão entre camada de controle e camada de visão. Cada linha é uma rota do sistema. Cada linha é dividida em 3 partes: verbo HTML, endereço URL (*Uniform Resource Locator*) e método de controle. O Código 3.9 mostra um exemplo de arquivo `routes` com 4 rotas. O caractere # representa uma linha de comentário. O endereço URL é relativo ao endereço do servidor *web* utilizado. Em uma aplicação local, normalmente utiliza-se o endereço `http://localhost:9000` para acesso ao servidor. Em uma aplicação remota o endereço de acesso representa um domínio *web*, por exemplo `http://www.meudominio.com`. O caractere `:` representa passagem de parâmetro. Por exemplo, o acesso ao endereço `http://localhost:9000/Usuario/5` invoca o método `UsuarioController.show(5);`

Código 3.9: Exemplo de arquivo de rotas

```
1 #Verbo   Endereço URL          Controller
2 GET      /                     Application.index
3 GET      /Usuario/list         UsuarioController.index
4 GET      /Usuario/:id          UsuarioController.show(Long id)
5 POST     /Usuario/:id/delete   UsuarioController.delete(Long id)
```

3.4 Saída

A saída do sistema *Metaffolder* é o conjunto de arquivos gerados a partir de metadados representando as classes da camada de modelo. São gerados classes de controle, páginas HTML, um arquivo de rotas representando as ações padrão de uma aplicação CRUD e um arquivo para internacionalização do sistema. O sistema CRUD final é a somatória de todos os arquivos gerados mais a camada de modelo.

A Figura 3.8 exibe o exemplo que será utilizado nesta Seção para explicar e apresentar a saída do sistema *Metaffolder*. A classe `Administrador` é uma classe de modelo e utiliza algumas anotações do sistema. Os arquivos gerados serão direcionados para o subpacote `br.edu.unifei`, portanto a configuração `metaffold.domain = "br.edu.unifei"` deve ser adicionada ao arquivo `application.conf` para que o sistema *Metaffolder* se comporte adequadamente. O restante das configurações adicionadas são os valores padrão e poderiam ser omitidas. Elas foram adicionadas somente para facilitar a visualização e entendimento do exemplo.

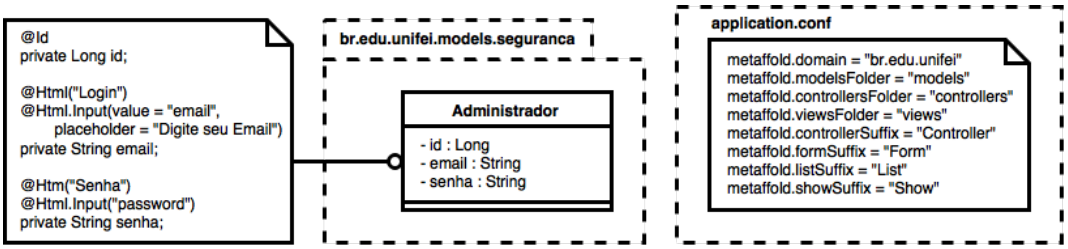


Figura 3.8: Exemplo de entrada para demonstração da saída do sistema *Metaffolder*

Todos os arquivos do sistema CRUD são gerados automaticamente pelo *Metaffolder* após a execução do comando `metaffold` no terminal do *framework Play*. A Figura 3.9 exibe a saída gerada pelo sistema *Metaffolder*.

```
[exemplo] $ metaffold
[info] Updating {file:/home/danillo/workspace/experimento1/treinamento/}root...
[info] Resolving jline#jline;2.12.1 ...
[info] Done updating.
[info] Compiling 2 Scala sources and 3 Java sources to /home/danillo/workspace/experimento1/treinamento/target/scala-2.11/classes...
[pool-10-thread-2] INFO org.reflections.Reflections - Reflections took 4 ms to scan 2 urls, producing 9 keys and 20 values
[pool-10-thread-2] INFO Metaffolder - [Criado] app/br/edu/unifei/controllers/seguranca/AdministradorController.java
[pool-10-thread-2] INFO Metaffolder - [Criado] app/br/edu/unifei/views/seguranca/administrador/administradorForm.scala.html
[pool-10-thread-2] INFO Metaffolder - [Criado] app/br/edu/unifei/views/seguranca/administrador/administradorList.scala.html
[pool-10-thread-2] INFO Metaffolder - [Criado] app/br/edu/unifei/views/seguranca/administrador/administradorShow.scala.html
[pool-10-thread-2] INFO Metaffolder - [Criado] app/br/edu/unifei/controllers/Application.java
[pool-10-thread-2] INFO Metaffolder - [Criado] app/br/edu/unifei/dao/Dao.java
[pool-10-thread-2] INFO Metaffolder - [Criado] app/br/edu/unifei/views/index.scala.html
[pool-10-thread-2] INFO Metaffolder - [Criado] app/br/edu/unifei/views/menu.scala.html
[pool-10-thread-2] INFO Metaffolder - [Criado] app/br/edu/unifei/views/layout.scala.html
[pool-10-thread-2] INFO Metaffolder - [Criado] conf/routes
[pool-10-thread-2] INFO Metaffolder - [Criado] conf/messages
[pool-10-thread-2] INFO Metaffolder - [Criado] app/br/edu/unifei/views/tags/twitterBootstrap3.scala.html
[pool-10-thread-2] INFO Metaffolder - [Criado] public/javascripts/main.js
[pool-10-thread-2] INFO Metaffolder - [Criado] public/stylesheets/main.css
[success] Total time: 4 s, completed Aug 7, 2015 5:38:19 PM
[exemplo] $ _
```

Figura 3.9: Terminal do *framework Play*, após execução do comando `metaffold`

A estrutura final do sistema CRUD gerado é representada por todos os diretórios e arquivos do mesmo. Ela depende das configurações do desenvolvedor e das classes de modelo. A Figura 3.10 exibe a estrutura final gerada a partir do exemplo de entrada da Figura 3.8. É importante ressaltar que apenas os diretórios relevantes ao sistema *Metaffolder* são exibidos na figura.

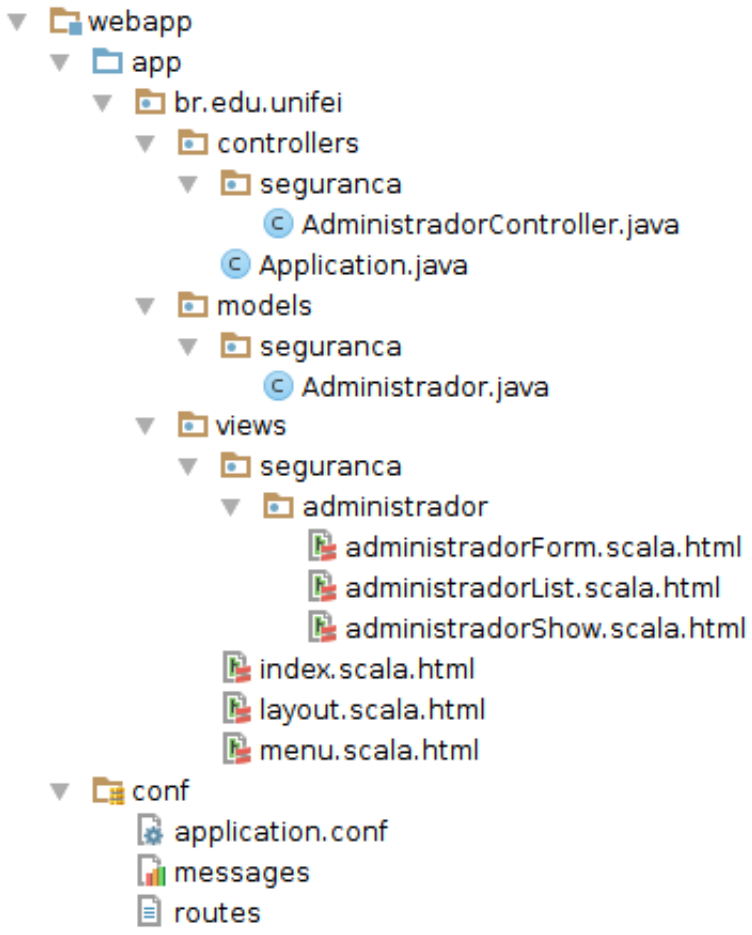


Figura 3.10: Estrutura do sistema CRUD gerado

A Figura 3.11 exibe o relacionamento entre páginas da camada de visão, rotas e métodos de controle. Como explicado anteriormente, uma rota possui um verbo HTTP (`GET` ou `POST` em navegadores *web*), um endereço URL e um método de controle. O usuário realiza alguma interação no navegador *web* que resulta em uma requisição HTTP com um endereço URL. Esta requisição é tabelada no arquivo de rotas do sistema CRUD e o método de controle correspondente é invocado. O método realiza as computações necessárias, geralmente envolvendo a camada de modelo e o banco de dados, e retorna uma nova página de visão ao usuário.

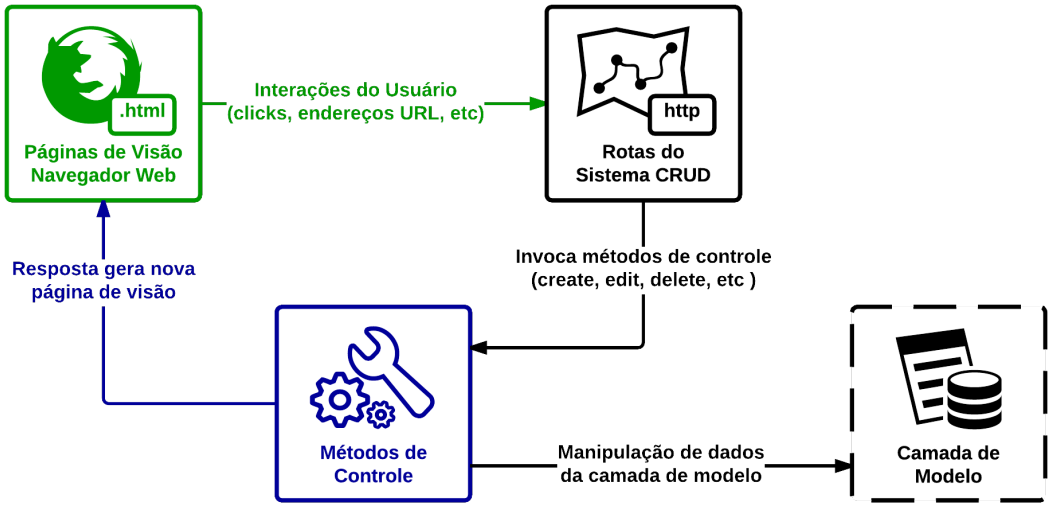


Figura 3.11: Relacionamento entre páginas de visão / rotas / métodos de controle

O diagrama da Figura 3.12 representa os relacionamentos mais importantes do sistema CRUD gerado (linhas do arquivo `routes`). A página da camada de visão `administradorList.scala.html` fornece acesso a todas as funcionalidades do sistema CRUD gerado, ou seja, possui *links* para as páginas `administradorForm.scala.html` (criação ou alteração de objetos) e `administradorShow.scala.html` (visualização de objetos). Também possui uma opção para exclusão de objetos. Cada *link* possui verbos `GET` ou `POST`: o verbo `GET` representa a leitura de um objeto de modelo no banco de dados; o verbo `POST` representa a alteração de um objeto de modelo no banco de dados. Seus significados podem ser alterados de acordo com a aplicação em que são utilizados.

A classe `AdministradorController` possui todos os métodos necessários em um sistema CRUD. Da Figura 3.12 tem-se os seguintes métodos: `index`, `create`, `save`, `show`, `edit`, `update`, `delete`. Além disso, a classe ainda possui o método `page` para paginação de objetos. O mecanismo de paginação aumenta a eficiência do sistema pois carrega somente um número limitado de objetos, reduzindo o consumo de memória, espaço e processamento gastos pelo servidor e banco de dados.

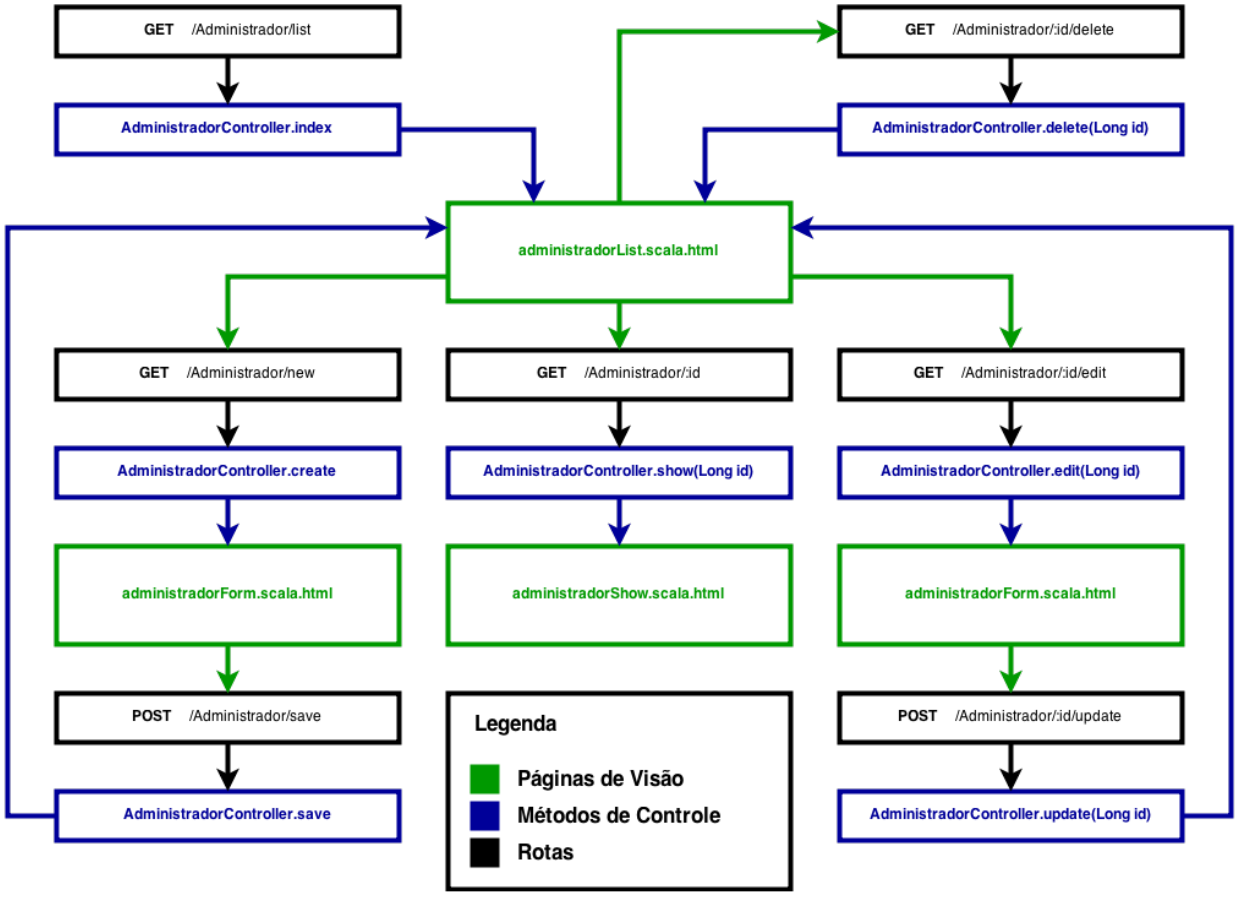


Figura 3.12: Relacionamentos do sistema CRUD gerado

As páginas da camada de visão representam a interface disponível ao usuário para a interação com o sistema gerado. A Figura 3.13 exibe todas as páginas HTML criadas a partir do exemplo de entrada da Figura 3.8. É a única parte do sistema que fica exposta ao usuário final: camada de controle, modelos, rotas, configurações, etc. são de interesse apenas do desenvolvedor. Os nomes destacados em vermelho foram usados somente para identificação das páginas e não fazem parte da saída apresentada ao usuário final. A página `index.scala.html`, não presente na figura, é a página inicial do sistema gerado, contendo *links* para as páginas `list.scala.html` de cada classe da camada de modelo. Foram criados alguns objetos apenas para facilitar a visualização e entendimento das funcionalidades do sistema CRUD gerado.

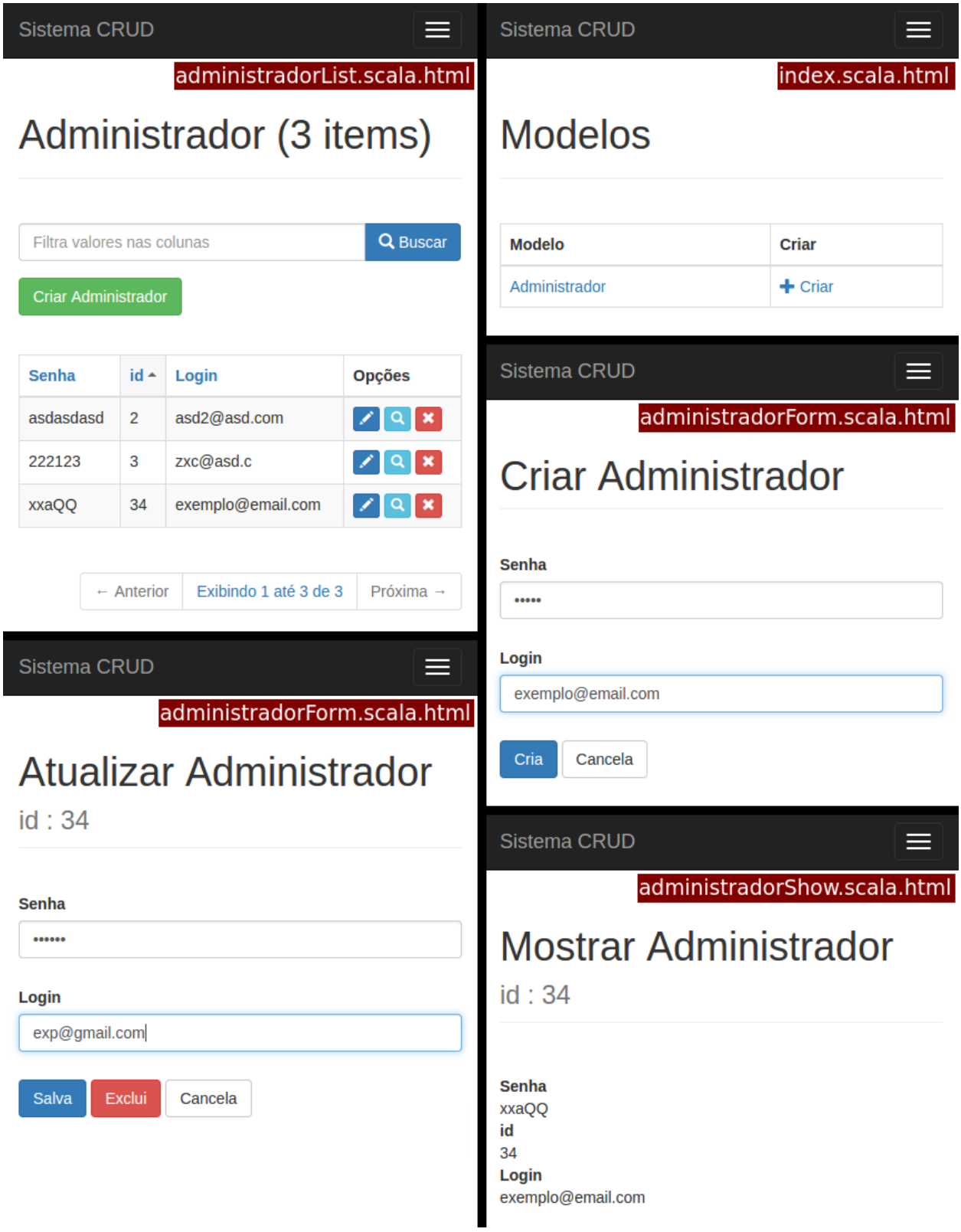


Figura 3.13: Camada de visão do sistema CRUD gerado

3.5 Considerações Finais

Este Capítulo apresentou todos os processos envolvidos no desenvolvimento e implementação da técnica de *scaffolding*, com a finalidade de criar automaticamente um sistema CRUD a partir de classes de modelo. A técnica foi dividida em 3 partes distintas: entrada, *Metaffolder* e saída. Os principais mecanismos e tecnologias utilizados no sistema *Metaffolder* foram apresentados e explicados em detalhes. Um exemplo de entrada foi utilizado para demonstrar o funcionamento da técnica de *scaffolding*, implementada no sistema *Metaffolder*. Foram apresentados a saída no terminal do *Play*; os diretórios, rotas e camada de visão do sistema CRUD gerado.

A Tabela 3.2 exibe uma comparação entre as características do sistema *Metaffolder* e dos sistemas analisados na fundamentação teórica deste trabalho. A primeira coluna representa características importantes relacionadas à aplicação da técnica de *scaffolding*. Essa coluna foi dividida em características mais gerais e características específicas do sistema CRUD gerado. As demais colunas representam os sistemas que aplicam a técnica de *scaffolding*.

Tabela 3.2: Comparação das principais características relacionadas à técnica de scaffolding

Sistemas Analisados que Aplicam a Técnica de Scaffolding	*Metaffolder*	*Rails*	*Grails*	*SILVA et al.*	*DAISSAOUI*	*MRACK et al.*	*Netbeans*	*Eclipse*	*Intellij IDEA*
Características Gerais									
Aplica o scaffolding em classes de modelo	x		x			x	x	x	
Camadas de visão e controle editáveis	x	x	x	x	x		x	x	x
Camada de visão estilizada por CSS	x	x	x				x	x	x
Convenção sob Configuração	x	x	x						
Configuração por anotações	x					x			
Características do Sistema CRUD Gerado									
Suporte a internacionalização	x	x	x						
Tecnologias atuais	x	x	x				x	x	x
CSS padrão agradável ao usuário	x		x						
Ferramenta de busca por objetos (*search*)	x								
Itens de navegação (menus, links, etc)	x		x						

Da tabela, observa-se que o sistema *Metaffolder* e o sistema CRUD gerado pela técnica de *scaffolding* apresentam características compatíveis com as dos principais *frameworks web* e IDEs do mercado. Também percebe-se que os trabalhos acadêmicos analisados (SILVA et al., DAISSAOUI, MRACK et al.) possuem uma deficiência de características importantes, se comparados com os demais sistemas.

Capítulo 4

Experimentos e Resultados

Este Capítulo apresenta os experimentos realizados para verificar se a utilização do sistema *Metaffolder* garante o aumento da produtividade no desenvolvimento de sistemas. O tempo gasto na criação de um sistema CRUD foi medido utilizando dois métodos: codificação manual e aplicação da técnica de *scaffolding*. Os tempos obtidos foram comparados para demonstrar o aumento da produtividade. Adicionalmente, foram realizados testes e medições de tempo para a validação das funcionalidades CRUD do sistema gerado. Os participantes dos experimentos responderam um questionário para verificar o nível de usabilidade da ferramenta *Metaffolder* e do sistema gerado por ela. Os mesmos experimentos foram realizados com o *framework Rails*, que também possui a técnica de *scaffolding*. A comparação entre *Metaffolder* e *Rails* forneceu uma melhor análise dos dados obtidos.

Um estudo realizado por Nielsen (2012) diz que apenas 5 participantes são suficientes na maioria dos testes de usabilidade. Apesar disso, o grupo de participantes dos experimentos foi composto por 10 alunos da Universidade Federal de Itajubá (UNIFEI), para um melhor resultado na parte quantitativa dos experimentos. Os alunos são da área de *Tecnologia e Informação*, portanto possíveis usuários do *software Metaffolder*. Eles possuem conhecimento intermediário na linguagem Java e nenhum conhecimento nas demais tecnologias e *softwares* utilizados.

A Seção 4.1 descreve o ambiente de testes utilizado nos experimentos, com suas características de *hardware* e *software*. A Seção 4.2 explica todos os processos envolvidos na realização dos experimentos. A Seção 4.3 exibe a análise dos dados obtidos e seus respectivos resultados.

4.1 Preparação do ambiente de testes

Para que um experimento baseado em comparações seja realizado, é necessário garantir que todas as condições entre os objetos alvos de comparação sejam as mais parecidas possíveis, para que o erro na análise final seja mínimo. No caso de um experimento computacional, todos os computadores utilizados devem possuir o mesmo *hardware*, os mesmos sistemas operacionais, as mesmas configurações, os mesmos *softwares* instalados.

Softwares de virtualização de sistemas operacionais são ideais para esse tipo de situação, pois possibilitam a criação de uma máquina virtual específica para determinado experimento, sem alterações significativas no sistema operacional hospedeiro (a única alteração é a instalação do *software* de virtualização).

Os experimentos foram realizados no Laboratório de Segurança e Engenharia de Redes (LASER) da UNIFEI. Todos os computadores desse laboratório possuem a seguinte especificação técnica: processador Intel Xeon X3450 - 2,66GHz - 8MB de *cache* - 4 núcleos e 8 *threads*; 8GB de memória RAM DDR3 1333MHz *Dual Channel* e controle de correção de erros (ECC); disco rígido de 466GB SATA2; placa de vídeo NVIDIA QUADRO 600. A máquina virtual foi criada utilizando 6GB de memória RAM, 4 núcleos e 128MB de memória de vídeo.

O sistema operacional hospedeiro é o *Microsoft Windows 8.1 Professional* com o *software* de virtualização *VirtualBox*. A máquina virtual foi criada somente com as configurações mínimas e os seguintes *softwares*:

- Sistema Operacional Ubuntu 14.04 (instalação padrão)
- Editor de texto Sublime Text 3 Build 3083
- Java Standard Edition Development Kit 1.8.0_45 (JDK 8)
- Play Framework 2.4.2
- Ruby 2.2.1p85 (2015-02-26 revision 49769)
- Ruby on Rails 4.2.3

4.2 Metodologia

Foram realizados dois experimentos comparativos entre si, o primeiro utilizou o sistema *Metaffolder* e o segundo o *framework Rails*. Cada experimento possui as fases de treinamento, execução e avaliação. As fases de treinamento e execução são subdivididas em duas e três tarefas, respectivamente. Ambas possuem as tarefas de aplicação da técnica de *scaffolding* e verificação das funcionalidades CRUD do sistema gerado. A fase de execução possui uma tarefa adicional: a codificação manual do sistema criado pela técnica de *scaffolding*. Na fase de avaliação, os participantes responderam um questionário para determinar o nível de usabilidade da aplicação da técnica de *scaffolding* e do sistema CRUD gerado por ela. Um roteiro foi criado para os participantes e se encontra no Apêndice B. Ele contém todos os passos para a realização das tarefas.

Um resumo estrutural da metodologia utilizada neste trabalho é listado a seguir. Os símbolos *T1* a *T5* representam o tempo que cada participante gastou para realizar cada tarefa das fases de treinamento e execução nos experimentos 1 e 2, respectivamente.

```
Experimento 1 - Sistema Metaffolder
      i - Fase de Treinamento
            a) Aplicação da técnica de scaffolding - T1
            b) Verificação das funcionalidades CRUD - T2
     ii - Fase de Execução
            a) Aplicação da técnica de Scaffolding - T3
            b) Verificação das funcionalidades CRUD - T4
            c) Codificação manual do sistema gerado - T5
    iii - Fase de Avaliação
            a) Aplicação do questionário de usabilidade.

Experimento 2 - Framework Rails
      i - Fase de Treinamento
            a) Aplicação da técnica de Scaffolding - T1
            b) Verificação das Funcionalidades CRUD - T2
     ii - Fase de Execução
            a) Aplicação da técnica de Scaffolding - T3
            b) Verificação das funcionalidades CRUD - T4
            c) Codificação manual do sistema gerado - T5
    iii - Fase de Avaliação
            a) Aplicação do questionário de usabilidade.
```

4.2.1 Fase de Treinamento

A fase de treinamento tem o objetivo de apresentar aos participantes todas as tecnologias usadas nos experimentos. Essa fase é necessária pois nenhum dos participantes tem conhecimento prévio dos *softwares* utilizados e os tempos obtidos não retratariam bem a realidade.

Nesta fase, um avaliador (o próprio autor deste trabalho) ficou responsável pela execução de todos os passos do roteiro e os participantes apenas observam. Quando o avaliador termina a execução desses passos, os participantes tentam reproduzir o que foi ensinado em um novo cenário, para que o tempo gasto seja medido. Durante a execução das tarefas, o avaliador fica disponível para esclarecer qualquer dúvida que os participantes tiverem.

Na primeira tarefa, o participante cria as classes de modelo fornecidas no roteiro e aplica a técnica de *scaffolding* para verificar se todos os arquivos são gerados corretamente. Isso

é feito com a inicialização do sistema CRUD gerado. Se nenhum erro for exibido, pode-se garantir que não existem erros de compilação (sistema *Metaffolder*) ou erros de sintaxe (*framework Rails*). Nesta fase são utilizadas as classes de modelo da Figura 4.1.

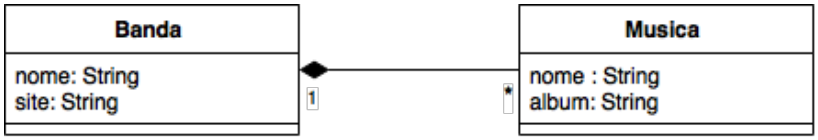


Figura 4.1: Diagrama de classes usado na 1ª tarefa da fase de treinamento

Na segunda tarefa, o participante utiliza o sistema CRUD gerado para verificar se ele está funcionando corretamente, ou seja, sem erros em tempo de execução. O participante interage com o sistema para verificar o funcionamento de algumas ações básicas, presentes na maioria dos sistemas CRUD atuais. Essa verificação é feita através da manipulação de objetos referentes às classes utilizadas como modelo. Os objetos usados nessa tarefa são exibidos na Figura 4.2.

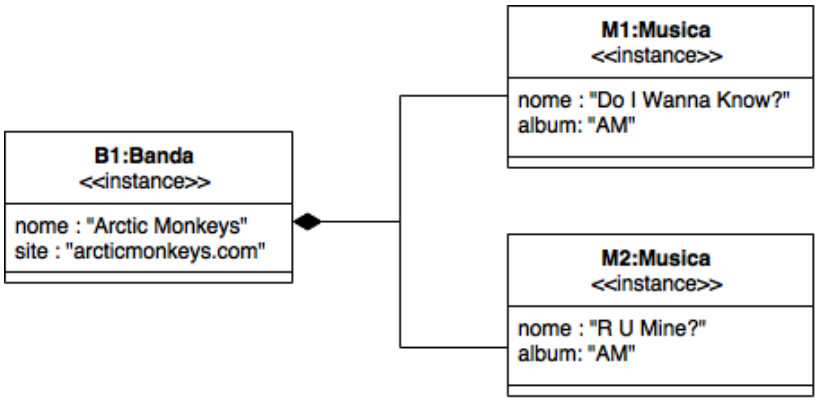


Figura 4.2: Diagrama de objetos usado na 2ª tarefa da fase de treinamento

4.2.2 Fase de Execução

A fase de execução tem como objetivo demonstrar o aumento na produtividade ao se aplicar a técnica de *scaffolding*. Nessa fase, o tempo gasto na realização das tarefas é mais condizente com a realidade, pois os participantes já estão treinados.

As tarefas realizadas nesta fase devem ser parecidas com as da fase de treinamento, já que os participantes precisam reproduzir o aprendizado em algo que já tenham experiência. A diferença é que desta vez o avaliador apenas observa e os participantes realizam o experimento sem qualquer tipo de ajuda, apenas com as orientações do roteiro. O tempo gasto pelos participantes é novamente medido.

Os passos necessários para se realizar a primeira tarefa são parecidos com os da fase de treinamento, a única diferença é que novas classes de modelo são utilizadas para a

aplicação da técnica de *scaffolding*. A Figura 4.3 mostra as classes usadas na primeira tarefa.

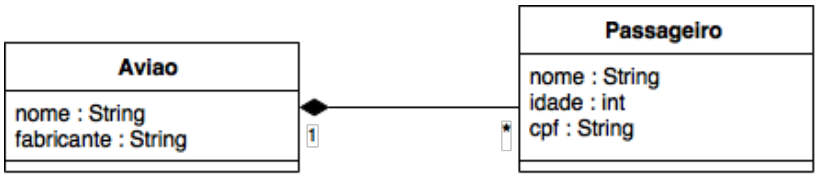


Figura 4.3: Diagrama de classes usado na 1ª tarefa da fase de execução

Na segunda tarefa da fase de execução, os participantes também verificam as funcionalidades CRUD do sistema gerado. A diferença é que um novo diagrama de objetos é usado nesta fase. Ele é exibido na Figura 4.4.

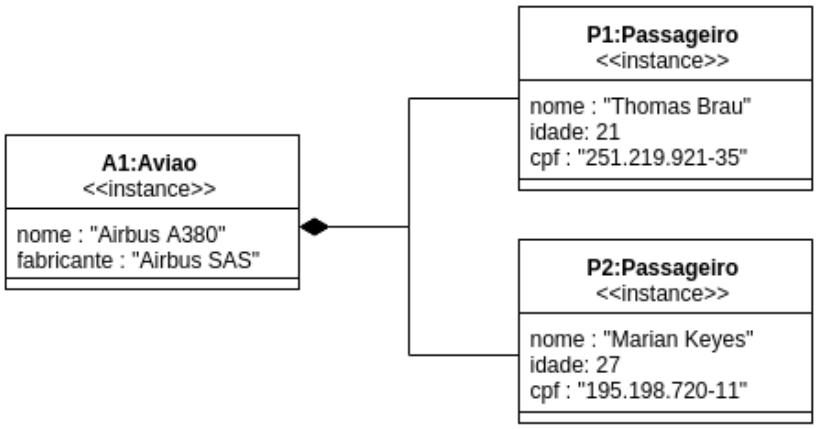


Figura 4.4: Diagrama de objetos usado na 2ª tarefa da fase de execução

A terceira tarefa representa a codificação manual do sistema CRUD gerado com o *scaffolding*. A comparação entre os tempos da primeira e terceira tarefa pode servir como indicativo do aumento da produtividade ao se utilizar a técnica de *scaffolding*. A codificação manual está presente somente na fase de execução, pois não faz sentido analisar a produtividade do usuário de um sistema que ele não sabe utilizar.

Os participantes codificaram as classes de modelo e todos os arquivos gerados automaticamente pela técnica de *scaffolding*. Os participantes puderam usar quaisquer técnicas de codificação durante o processo, inclusive copiar e colar, desde que se copiasse trechos de código já digitados anteriormente.

4.2.3 Fase de Avaliação

Na fase de avaliação é medido o nível de usabilidade dos *softwares* (*Metaffolder* e *Rails*) e sistemas CRUD gerados. Para isso foi utilizado o questionário SUS (*System Usability Scale*) de Brooke (1996), muito conhecido na literatura. A avaliação de usabilidade é necessária

porque um bom *software* somente estará completo quando possuir uma boa aceitação dos seus usuários.

O questionário SUS é constituído de 10 questões em escalas de 1 a 5, onde 1 significa "Discordo Totalmente" e 5 significa "Concordo Totalmente". As questões do questionário SUS se encontram no Anexo A. O resultado final é calculado utilizando os seguintes critérios (BROOKE, 1996):

- Para as questões ímpares, o valor final da questão será a resposta do participante menos 1.
- Para as questões pares, o valor final da questão será 5 menos a resposta do participante.
- Após correções, todas as questões estarão em uma escala de 0 a 4 (4 sendo a resposta mais positiva).
- Soma-se todas as questões. O total é um valor de 0 a 40.
- Multiplica-se o total por 2,5. O resultado final indica a pontuação SUS do questionário, e varia de 0 a 100.

Apesar da pontuação SUS variar de 0 a 100, ela não pode ser considerada como uma porcentagem. Um estudo feito por Sauro (2011) contou com 5000 usuários para avaliar 500 sistemas diferentes com o questionário SUS. A pontuação média foi de 68 pontos na escala de 0 a 100. Portanto, pode-se utilizar esse resultado como parâmetro de avaliação da usabilidade de um sistema: uma pontuação superior a 68 pontos indica que o sistema está acima da média, e, consequentemente, com um bom nível de usabilidade.

4.3 Resultados

Todos os dados obtidos nos experimentos podem ser consultados no Apêndice C. Eles foram analisados e são apresentados nesta Seção para elaboração das conclusões deste trabalho.

A Figura 4.5 apresenta o tempo médio de cada tarefa dos experimentos realizados. Foi utilizada a escala logarítmica para uma melhor representação visual de todos os tempos obtidos. Observa-se que a maioria dos tempos são menores no *framework Rails* do que no sistema *Metaffolder*. Isso já era esperado porque o *Rails* foi o precursor da técnica de *scaffolding* em *frameworks web* e por isso foi escolhido para comparação nos experimentos.

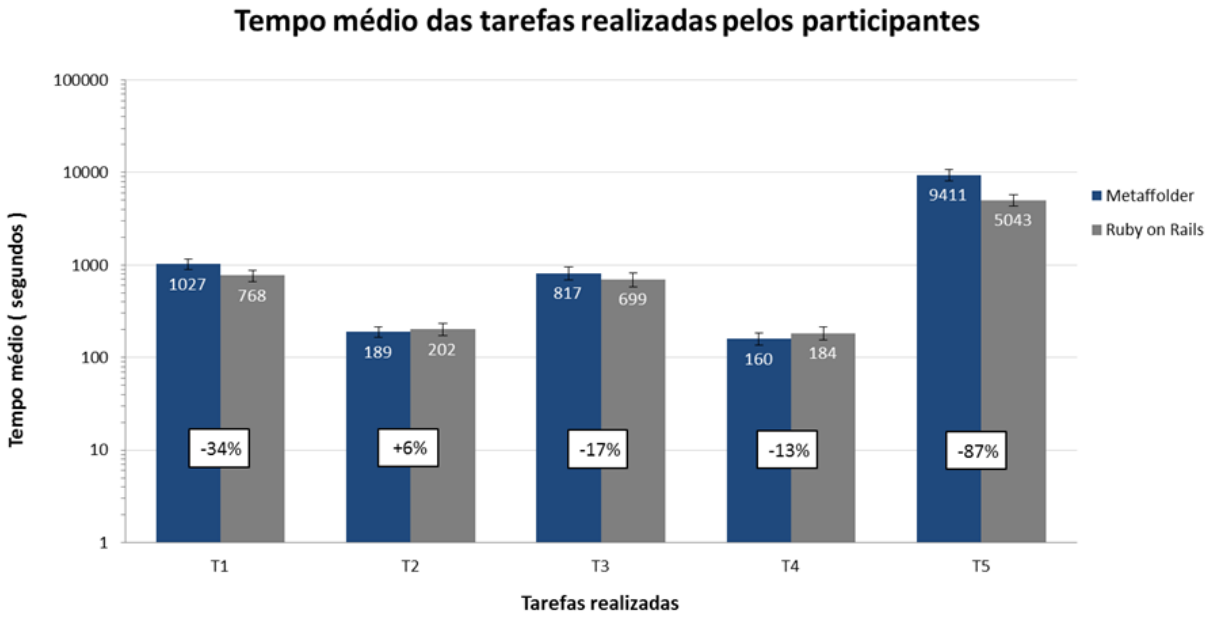


Figura 4.5: Tempo médio das tarefas realizadas pelos participantes

A porcentagem de redução (positiva) ou de aumento do tempo (negativa) em relação ao *Rails* também está na Figura 4.5. Na aplicação do *scaffolding* na fase de treinamento (T1), os participantes do *Metaffolder* gastaram aproximadamente 35% mais tempo do que o *framework Rails*. Após treinamento, essa diferença caiu para 17% (T3). Dessa comparação pode-se concluir que mesmo gastando um pouco a mais de tempo para aplicar a técnica de *scaffolding*, o *Metaffolder* possui um tempo próximo ao do *Rails*, se seus usuários já estiverem familiarizados (treinados) com o sistema.

Os tempos médios T2 e T4 são semelhantes, tanto no sistema *Metaffolder* quanto no *framework Rails*. Com isso, pode-se afirmar que os sistemas CRUD gerados são funcionalmente similares e fáceis de usar, inclusive para usuários sem conhecimento prévio (não-treinados).

Apesar do tempo de codificação manual do sistema CRUD ser aproximadamente 90% maior do que o do *Rails*, isso não é necessariamente um ponto negativo: a técnica de *scaffolding* criará todo o código necessário automaticamente, caso seu usuário decida utilizá-la.

A Figura 4.6 demonstra a melhora de desempenho dos participantes após a fase de treinamento. Da figura, observa-se que os usuários do sistema *Metaffolder* aproveitaram melhor o treinamento, tanto na aplicação da técnica de *scaffolding* (T1 e T3) quanto na verificação das funcionalidades CRUD do sistema gerado (T2 e T4). Um melhor aproveitamento pode indicar facilidade de uso da técnica e do sistema CRUD gerado ou facilidade de aprendizado dos participantes, por exemplo.

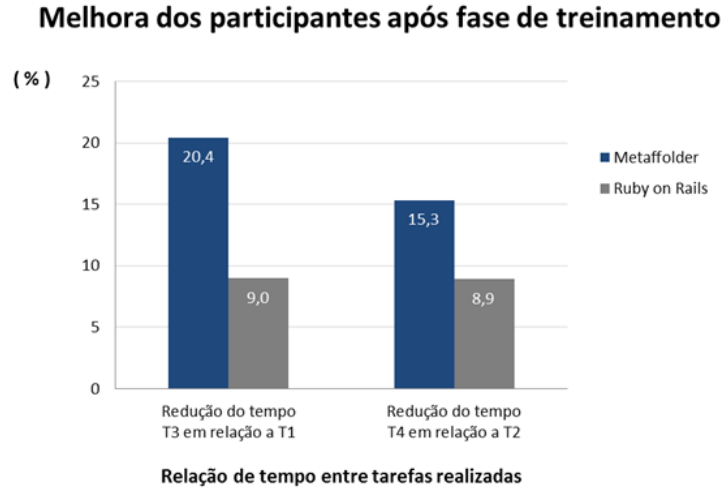


Figura 4.6: Melhora dos participantes após fase de treinamento

A Figura 4.7 apresenta o comparativo feito entre os tempos T3 e T5. Ela mostra a eficiência da técnica de *scaffolding* (T3) se comparada com a codificação manual (T5). Pode-se observar uma redução de tempo próxima a 90%, tanto para o *Metaffolder* quanto para o *Rails*. Com esses dados fica evidente o aumento da produtividade no desenvolvimento de sistemas CRUD ao se utilizar a técnica de *scaffolding*. Como observação, vale mencionar que este ganho é referente à codificação das camadas de controle e visão de apenas duas classes de modelo. A redução de tempo seria ainda maior com a aplicação da técnica em mais classes de modelo.

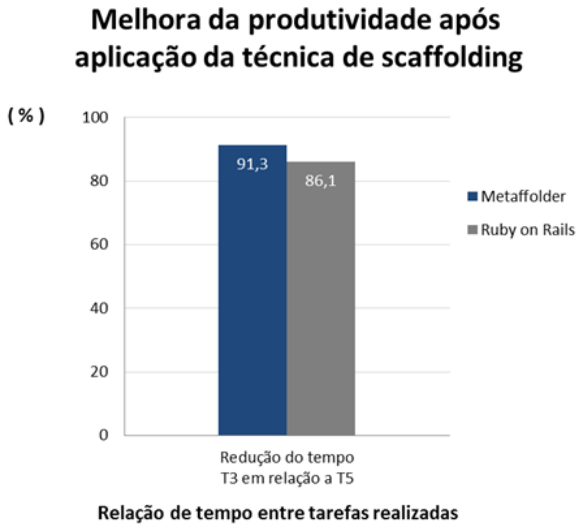


Figura 4.7: Melhora da produtividade após aplicação da técnica de scaffolding

Finalmente, a Tabela 4.1 apresenta as pontuações do questionário SUS obtidas pelos *softwares Metaffolder* e *Rails*. O sistema *Metaffolder* obteve uma média de 83,7 pontos com desvio padrão de 7,3 pontos. O *framework Rails* obteve uma média de 86,0 pontos com

desvio padrão de 3,8 pontos. Apesar de menor, a pontuação do *Metaffolder* está bem próxima da pontuação do *Rails*.

Tabela 4.1: Pontução SUS dos *softwares Metaffolder* e *Ruby on Rails*

	Pontuação SUS	
Participante	Metaffolder	Ruby on Rails
1	92,5	92,5
2	95,0	85,0
3	85,0	85,0
4	77,5	82,5
5	70,0	85,0
6	80,0	82,5
7	85,0	85,0
8	82,5	92,5
9	80,0	87,5
10	87,5	82,5
Média	83,7	86,0
Desvio Padrão	7,3	3,8

Dos estudos de Sauro (2011), pode-se concluir que ambos os *softwares* obtiveram notas satisfatórias (maiores que 68) e podem ser considerados sistemas com bons níveis de usabilidade.

4.4 Considerações Finais

Os experimentos apresentados neste Capítulo demonstram o aumento de produtividade ao se utilizar a técnica de *scaffolding* para criação de sistemas CRUD. A técnica de *scaffolding* implementada no sistema *Metaffolder* foi utilizada por um grupo de participantes em um experimento de medição de tempos. O aumento da produtividade foi comprovado pela análise dos tempos de codificação manual e automática (*scaffolding*). Cada participante também respondeu um questionário para avaliar o nível de usabilidade do *Metaffolder* e seu sistema CRUD gerado. A análise dos resultados do questionário SUS comprovou bons níveis de usabilidade do *Metaffolder* e seu sistema CRUD gerado. Adicionalmente, foram comprovadas a produtividade e usabilidade referentes ao *framework Rails*.

Capítulo 5

Conclusão

A crescente demanda por sistemas *web* de qualidade, em prazos cada vez menores, é um dos principais motivos para a constante evolução das ferramentas disponíveis ao desenvolvedor. Atualmente, *frameworks web* desempenham um papel fundamental no desenvolvimento de aplicações: possuem um conjunto de soluções amplamente testadas e consolidadas que aumentam a produtividade de seus usuários. Das soluções disponíveis, a opção de geração automática de código foi o objeto de estudo deste trabalho.

Vários *frameworks web* modernos possuem a funcionalidade de geração de sistemas CRUD a partir de classes de modelo, também conhecida como *scaffolding*. Apesar do ganho na produtividade e redução de erros humanos, o *framework Play* ainda não tinha essa funcionalidade. O principal objetivo deste trabalho foi propor e implementar uma solução de *software* que aumentasse, com o uso da técnica de *scaffolding*, a produtividade dos usuários do *framework Play*. O *software Metaffolder* foi desenvolvido para cumprir este objetivo.

O requisito mínimo teórico para a implementação da técnica é a utilização do padrão arquitetural MVC. A criação das camadas de visão e controle do sistema CRUD final depende de informações da camada de modelo. O mecanismo de *templates* foi adotado para a criação dessas camadas. A literatura fornecida neste trabalho elucidou esse e outros aspectos envolvidos nessa técnica.

A arquitetura da ferramenta foi baseada no conceito de convenção sobre configuração, ou seja, utilizou convenções das comunidades *Play* e Java para evitar configurações desnecessárias. Apesar disso, o sistema desenvolvido fornece mecanismos de personalização, caso seja necessário.

Com o objetivo de diminuir dependências externas, grande parte das tecnologias internas do *framework Play* foram reaproveitadas na implementação do sistema. As principais foram: biblioteca de reflexão `org.reflections` e biblioteca de configuração `typesafe.config`. A ferramenta *JMustache* foi a escolhida como *engine* de *templates* por ser compacta, eficiente, com zero dependências e fácil de usar.

Os resultados obtidos nos experimentos comprovaram que o objetivo de aumentar a produtividade utilizando *scaffolding* foi alcançado. O *software Metaffolder* foi comparado

com o *Rails*, precursor da técnica de *scaffolding* em *frameworks web* modernos, para a obtenção de conclusões com maior significado. Percebeu-se que mesmo sendo um pouco inferior, o desempenho do *software Metaffolder* foi muito próximo ao do *framework Rails*, com relação tanto à técnica de *scaffolding* quanto ao nível de usabilidade. Este resultado foi considerado como positivo e enfatizou o êxito no cumprimento dos objetivos propostos neste trabalho.

5.1 Contribuições

As principais contribuições deste trabalho são:

- Implementação do sistema *Metaffolder*, uma solução de *software* que utiliza a técnica de *scaffolding* para criar sistemas CRUD a partir de classes de modelo e *templates* predefinidos.
- Os experimentos demonstraram que a técnica de *scaffolding* do sistema *Metaffolder* promove o aumento de aproximadamente 90% na produtividade comparado à codificação manual de um sistema CRUD. Consequentemente, houve uma diminuição de erros de codificação do sistema CRUD gerado.

As contribuições secundárias são:

- Expansão da literatura sobre o tema relacionado à técnica de *scaffolding*. Não existe muito conteúdo teórico sobre o mesmo, por se tratar de um tema mais técnico e recente.
- Definição de um esquema modular para aplicação da técnica de *scaffolding* em sistemas CRUD com arquitetura MVC. Este esquema foi implementado no sistema *Metaffolder*, dividindo-o em 3 estruturas distintas: configuração, metaprogramação e geração de código.
- Dados experimentais e suas respectivas análises, tanto do *Metaffolder* quanto do *Rails*, incluindo todos os tempos das tarefas e respostas do questionário SUS. O sistema *Metaffolder* obteve uma pontuação SUS média de 83,7 e o *framework Rails* de 86,0 pontos.

5.2 Dificuldades Encontradas

Várias dificuldades foram encontradas durante a realização deste trabalho. As mais relevantes são comentadas nesta Seção.

Inicialmente, foi escolhido o *ObInject* (CARVALHO et al., 2013) para a persistência de dados do sistema CRUD gerado, um *framework* proprietário do Orientador deste trabalho. Apesar de eficiente na inserção e consulta a dados, o *ObInject* ainda não possuía todas as 4 operações CRUD implementadas, portanto precisou ser substituído por outro *framework* durante o desenvolvimento do *Metaffolder*.

A literatura relacionada à técnica de *scaffolding* é praticamente inexistente. Apesar de existir muitos artigos referentes à geradores de código (SILVA et al., 2011; DAISSAOUI, 2010; COHEN-ZARDI, 2013; GERACI, 1990; FRANKY; PAVLICH-MARISCAL, 2012), o conteúdo referente à técnica de *scaffolding* é bem abstrato e pouco detalhado (RAILS, 2014; ROCHER et al., 2014; ADAMUS et al., 2010). Foi criado um esquema teórico da implementação da técnica em um sistema para diminuir essa abstração e facilitar o entendimento de suas várias partes e inter-relacionamentos.

Durante sua constante evolução (versões 1.2.x, 2.x, 2.2.x, 2.3.x e 2.4.x), o *framework Play* realizou algumas mudanças nas suas estruturas internas que dificultaram um pouco a implementação do *Metaffolder*. A versão final do sistema deveria usar o mecanismo de *Injeção de Dependência* (JSR-299 Expert Group, 2009) usado na versão 2.4.x do *Play* e muito usado em *softwares* modernos. O mecanismo de leitura de *classloader* acabou sendo adotado por causa de problemas técnicos encontrados com o uso de injeção de dependência.

A partir da versão 2.x, o *Play* começou a usar a ferramenta SBT (*Simple Build Tool*) (TYPESAFE, 2014d), que usa a linguagem *Scala* (ODERSKY, 2003) para a adição de novos comandos ao seu terminal. Portanto, foi necessário aprender o mínimo sobre o funcionamento da ferramenta e da linguagem para que o comando `metaffold` fosse adicionado com sucesso ao *framework Play*.

5.3 Trabalhos Futuros

Apesar da ferramenta ser adaptável aos gostos do desenvolvedor, não é interessante que ele gaste muito tempo em configurações. As seguintes sugestões são apresentadas para dar continuidade ao desenvolvimento deste trabalho e melhorar a usabilidade da ferramenta:

- *Banco de dados de templates*: a sintaxe da *engine* de *templates* é simples e intuitiva, permitindo que novos *templates* sejam criados para personalização do sistema. Apesar disso, muitos desenvolvedores não acham essa ideia interessante. Um banco de dados com os *templates* dos casos de uso mais frequentes poderia ser criado. Para não aumentar o tamanho da biblioteca, os arquivos ficariam salvos na nuvem. Um menu interativo para escolha do *template* seria disponibilizado ao usuário pelo terminal do *framework*.

- *Mais opções para localização de modelos*: apesar de muito utilizada, não são todos os projetos desenvolvidos que utilizam a anotação `@Entity`. Novas implementações da interface `ModelRegistry` seriam fornecidas ao usuário do sistema *Metaffolder*. Em último caso, a ferramenta poderia disponibilizar uma anotação própria, por exemplo `@Crud.Include`, para identificar classes de modelo.

- *Módulo de segurança*: disponibilizar ao desenvolvedor a opção de adicionar um módulo de segurança ao sistema CRUD gerado, com opções de cadastro e autenticação de usuários. Este módulo seria parecido com os utilizados nos exemplos fornecidos pela documentação do *framework Play*.

- *Mais opções para engines de templates*: desacoplar do sistema a biblioteca *JMustache*, para que o usuário possa escolher a *engine* de *templates* que mais lhe agradar.

- *Gerador de templates*: bons *templates* são reutilizados inúmeras vezes. Apesar disso, é necessário um tempo maior no seu processo de criação. Uma solução para este problema seria a implementação de um gerador de *templates*. Um arquivo texto é utilizado como entrada. O usuário entraria com palavras-chave, por exemplo `nome = "Admin"`. O programa subistituiria todas as ocorrênicas da palavra `Admin` pelo marcador `{{nome}}`. O resultado final seria um *template* reutilizável a partir de um código pronto.

Referências Bibliográficas

ADAMUS, R.; KOWALSKI, T. M.; KULIBERDA, K.; WIŚLICKI, J.; BLEJA, M. Tools supporting generation of"data-intensive"applications for a web environment. **Automatyka/Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie**, v. 14, p. 951–960, 2010.

BAYNE, M. **JMustache - A Java implementation of the Mustache templating language**. [S.l.], 2010. Disponível em: <https://github.com/samskivert/jmustache>. Acesso em: 7 de Setembro de 2014.

BECK, K.; CUNNINGHAM, W. A laboratory for teaching object oriented thinking. **SIGPLAN Not.**, ACM, New York, NY, USA, v. 24, n. 10, p. 1–6, set. 1989. ISSN 0362-1340.

BERNERS-LEE, T. **Information Management: A Proposal**. Genf, März 1989. Disponível em: <http://www.w3c.org/History/1989/proposal.html>.

BROOKE, J. Sus-a quick and dirty usability scale. **Usability evaluation in industry**, London, v. 189, n. 194, p. 4–7, 1996.

BUSCHMANN, F.; MEUNIER, R.; ROHNERT, H.; SOMMERLAD, P.; STAL, M. **Pattern-oriented Software Architecture: A System of Patterns**. New York, NY, US: John Wiley & Sons, Inc., 1996. ISBN 0-471-95869-7.

CARVALHO, L. O.; SERAPHIM, T. F. P.; JR., C. T.; SERAPHIM, E. Obinject: a noodmg persistence and indexing framework for object injection. **JIDM**, v. 4, n. 3, p. 220–235, 2013. Disponível em: <http://seer.lcc.ufmg.br/index.php/jidm/article/view/240>.

CHEN, N. **Convention over Configuration**. [S.l.], 2006. Disponível em: <http://softwareengineering.vazexqi.com/files/pattern.html>. Acesso em: 11 de Outubro de 2014.

COHEN-ZARDI, D. **Code Generation: good or evil?** [S.l.], 2013. Disponível em: <http://blog.softfluent.com/2013/03/07/code-generation-good-or-evil/>. Acesso em: 11 de Novembro de 2014.

CZARNECKI, K.; EISENECKER, U. W. **Generative Programming: Methods, Tools, and Applications**. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 2000. ISBN 0-201-30977-7.

DAISSAOUI, A. Applying the mda approach for the automatic generation of an mvc2 web application. In: LOUCOPOULOS, P.; CAVARERO, J.-L. (Ed.). **RCIS**. [S.l.]: IEEE, 2010. p. 681–688.

DAMASEVICIUS, R.; STUIKYS, V. Taxonomy of the fundamental concepts of metaprogramming. **124X Information Tecnhology and Control**, v. 27, p. 124–132, 2008.

FAYAD, M.; SCHMIDT, D. C. Object-oriented application frameworks. **Commun. ACM**, ACM, New York, NY, US, v. 40, n. 10, p. 32–38, out. 1997. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/262793.262798>.

FORMAN, I. **Java reflection in action**. Greenwich, CT: Manning Pearson Education, 2005. ISBN 1-932394-18-4.

FOWLER, M. **Patterns of enterprise application architecture**. 1. ed. Boston: Addison-Wesley Professional, 2002. 560 p. ISBN 0-321-12742-0.

FRANKY, M.; PAVLICH-MARISCAL, J. Improving implementation of code generators: A regular-expression approach. In: **Informatica (CLEI), 2012 XXXVIII Conferencia Latinoamericana En**. [S.l.: s.n.], 2012. p. 1–10.

GERACI, A. Ieee standard glossary of software engineering terminology. **IEEE Std 610.12-1990**, p. 1–84, Dec 1990.

GOSLING, J.; JOY, B.; STEELE JR., G. L.; BRACHA, G.; BUCKLEY, A. **The Java Language Specification, Java SE 7 Edition**. 1st. ed. [S.l.]: Addison-Wesley Professional, 2013. ISBN 0133260224, 9780133260229.

HANSSON, D. H. **Ruby on Rails Framework**. [S.l.], 2003. Disponível em: <http://rubyonrails.org/>. Acesso em: 15 de Setembro de 2014.

HELLER, M. **REST and CRUD: the Impedance MismatchWeb application framework**. Framingham, MA, US, 2007.

IWS. **World Internet Users Statistics and 2014 World Population Stats**. Bogota, Colombia, 2014. Disponível em: <http://www.internetworldstats.com/stats.htm>. Acesso em: 8 de Setembro de 2014.

JENKOV, J. **Java Reflection Tutorial**. [S.l.], 2008. Disponível em: <http://tutorials.jenkov.com/java-reflection/index.html>. Acesso em: 11 de Novembro de 2014.

JSR-299 Expert Group. **JSR-299: Contexts and Dependency Injection for the Java EE platform**. [S.l.], dez. 2009. Disponível em: <http://download.oracle.com/otndocs/jcp/web_beans-1.0-fr-eval-oth-JSpec/>.

KAISLER, S. **Software paradigms**. Hoboken, N.J: Wiley-Interscience, 2005. ISBN 0-471-48347-8.

LEINER, B. M.; CERF, V. G.; CLARK, D. D.; KAHN, R. E.; KLEINROCK, L.; LYNCH, D. C.; POSTEL, J.; ROBERTS, L. G.; WOLFF, S. A brief history of the internet. **SIGCOMM Comput. Commun. Rev.**, ACM, New York, NY, US, v. 39, n. 5, p. 22–31, out. 2009. ISSN 0146-4833. Disponível em: <http://doi.acm.org/10.1145/1629607.1629613>.

LICKLIDER, J. C. R.; CLARK, W. E. On-line man-computer communication. In: **Proceedings of the May 1-3, 1962, Spring Joint Computer Conference**. New York, NY, USA: ACM, 1962. (AIEE-IRE '62 (Spring)), p. 113–128. Disponível em: <http://doi.acm.org/10.1145/1460833.1460847>.

LöWE, W.; NOGA, M. L. Metaprogramming applied to web component deployment. **Electr. Notes Theor. Comput. Sci.**, v. 65, n. 4, p. 106–116, 2002.

MALYSHEV, E. **JSF Application in Just Two Clicks**. [S.l.], 2006. Disponível em: <http://blog.jetbrains.com/idea/2006/11/jsf-application-in-just-two-clicks/>. Acesso em: 21 de Novembro de 2014.

MAMO, R. **Reflections - Java runtime metadata analysis**. [S.l.], 2013. Disponível em: <https://github.com/ronmamo/reflections>. Acesso em: 27 de Agosto de 2014.

MARTIN, M.; MARTIN, R. **Agile Principles, Patterns, and Practices in C#**. Pearson Education, 2006. ISBN 9780132797146. Disponível em: <http://books.google.com.br/books?id=hckt7v6g09oC>.

MATSUMOTO, Y. **Ruby Programming Language**. [S.l.], 1993. Disponível em: <https://www.ruby-lang.org/en/>. Acesso em: 2 de Outubro de 2014.

MRACK, M.; MOREIRA Álvaro de F.; PIMENTA, M. Merlin: Interfaces crud em tempo de execução. **XIII Sessão de Ferramentas do SBES**, Florianópolis, SC, p. 79–84, 2006.

NETBEANS. **Generating a JavaServer Faces 2.x CRUD Application from a Database**. [S.l.], 2010. Disponível em: <https://netbeans.org/kb/docs/web/jsf20-crud_pt_BR.html>. Acesso em: 21 de Outubro de 2014.

NIELSEN, J. **How Many Test Users in a Usability Study?** [S.l.], 2012. Disponível em: <http://www.nngroup.com/articles/how-many-test-users/>. Acesso em: 7 de Setembro de 2014.

NISO. **Understanding metadata**. Bethesda, MD, 2004. ISBN 1-880124-62-9.

ODERSKY, M. **The Scala Programming Language**. [S.l.], 2003. Disponível em: <http://www.scala-lang.org/>. Acesso em: 10 de Outubro de 2014.

OLIVEIRA, M. F. de. **Metaprogramação e Metadados de Persistência e Indexação para o Framework Object-Injection**. Dissertação (Mestrado) — Universidade Federal de Itajubá, Itajubá, MG, 2012.

ORACLE. **A Metadata Facility for the Java Programming Language**. [S.l.], 2004. Disponível em: <https://jcp.org/en/jsr/detail?id=175>. Acesso em: 15 de Setembro de 2014.

____. **Package java.lang.reflect**. [S.l.], 2011. Disponível em: <https://docs.oracle.com/javase/7/docs/api/java/lang/reflect/package-summary.html>. Acesso em: 17 de Agosto de 2014.

____. **Trail: The Reflection API**. [S.l.], 2011. Disponível em: <https://docs.oracle.com/javase/tutorial/reflect/>. Acesso em: 3 de Dezembro de 2014.

PIVOTAL. **Grails Framework**. [S.l.], 2014. Disponível em: <https://grails.org/>. Acesso em: 17 de Outubro de 2014.

RAILS, R. on. **The Rails Command Line**. [S.l.], 2014. Disponível em: <http://guides.rubyonrails.org/command_line.html>. Acesso em: 5 de Dezembro de 2014.

REENSKAUG, T. M. H. **Models-Views-Controllers**. [S.l.], dez. 1979. Disponível em: <http://heim.ifi.uio.no/~trygver/1979/mvc-2/1979-12-MVC.pdf>. Acesso em: 14 de Outubro de 2014.

ROCHER, G.; LEDBROOK, P.; PALMER, M.; BROWN, J.; DALEY, L.; BECKWITH, B.; HOTARI, L. **Grails Scaffolding**. [S.l.], 2014. Disponível em: <http://grails.org/doc/latest/guide/scaffolding.html>. Acesso em: 13 de Novembro de 2014.

SANCHEZ, J. **Crudo Plugin**. [S.l.], 2008. Disponível em: <http://crudo.sourceforge.net/>. Acesso em: 5 de Novembro de 2014.

SAURO, J. **Measuring Usability With The System Usability Scale (SUS)**. [S.l.], 2011. Disponível em: <http://www.measuringu.com/sus.php>. Acesso em: 22 de Novembro de 2014.

SCHWARTZ, M. **Web application framework**. [S.l.], 2014. Disponível em: <http://docforge.com/wiki/Web_application_framework>. Acesso em: 12 de Setembro de 2014.

SHEARD, T. Accomplishments and research challenges in meta-programming. In: **In 2nd Int. Workshop on Semantics, Applications, and Implementation of Program Generation, LNCS 2196**. [S.l.]: Springer-Verlag, 2000. p. 2–44.

SILVA, F.; PERRI, C.; ALMEIDA, L. Desenvolvimento de uma ferramenta assistente para criação de aplicações crud em java na web. **Colloquium Exactarum**, v. 2, n. 2, 2011. ISSN 2178-8332. Disponível em: <http://revistas.unoeste.br/revistas/ojs/index.php/ce/article/view/460>.

SINGH, I.; STEARNS, B.; JOHNSON, M. **Designing Enterprise Applications with the J2EE Platform**. 2nd. ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN 0-201-78790-3.

SOMMERVILLE, I. **Software Engineering**. Boston: Pearson, 2011. ISBN 978-0-13-703515-1.

TYPESAFE. **Activator Computer Database Java**. [S.l.], 2014. Disponível em: <https://github.com/typesafehub/activator-computer-database-java>. Acesso em: 10 de Setembro de 2014.

____. **Case Studies & Stories**. San Francisco, US, 2014.

____. **Play Framework Documentation**. [S.l.], 2014. Disponível em: <https://www.playframework.com/>. Acesso em: 27 de Outubro de 2014.

____. **SBT - The interactive build tool**. [S.l.], 2014. Disponível em: <http://www.scala-sbt.org/>. Acesso em: 14 de Outubro de 2014.

____. **Typesafe Config**. [S.l.], 2015. Disponível em: <https://github.com/typesafehub/config>. Acesso em: 11 de Outubro de 2014.

W3C. **Current Members - W3C**. [S.l.], 2014. Disponível em: <http://www.w3.org/Consortium/Member/List>. Acesso em: 16 de Novembro de 2014.

____. **A vocabulary and associated APIs for HTML and XHTML**. [S.l.], 2014. Disponível em: <http://www.w3.org/TR/html5/forms.html>. Acesso em: 11 de Dezembro de 2014.

Apêndice A

Template da Camada de Controle

Este Apêndice exibe o arquivo de *template* utilizado pelo *Metaffolder* para criação das classes de controle do sistema CRUD gerado.

```
1  package {{domainDot}}{{conf.controllersFolder}}{{dotRelativePackage}};
2
3  import play.mvc.*;
4  import play.data.*;
5  import play.i18n.*;
6  import static play.data.Form.*;
7
8  import java.util.*;
9
10 {{#imports}}
11 import {{domainDot}}{{conf.modelsFolder}}{{dotRelativePackage}}.*;
12 {{/imports}}
13 import {{domainDot}}{{conf.viewsFolder}}.html{{dotRelativePackage}}.{{nameVar}}.*;
14
15 import static {{domainDot}}{{conf.daoFolder}}.Dao.*;
16
17 public class {{nameClass}}{{conf.controllerSuffix}} extends Controller {
18
19     public static final String[] searchFields = {
20         {{searchableFields}}
21     };
22
23     public  Result index() {
24         return redirect(
25             {{domainDot}}{{conf.controllersFolder}}{{dotRelativePackage}}.routes.{{
                   nameClass}}{{conf.controllerSuffix}}.page(0, "{{keyName}}", "")
26         );
27     }
28
29     public  Result page(int page, String orderBy, String search) {
30         return ok(
31             {{nameVar}}{{conf.listSuffix}}.render(
32                 {{nameVar}}Dao.page(page, orderBy, searchFields, search),
33                 orderBy, search
34             )
```

```
35          );
36      }
37
38      public  Result show({{keyType}} {{keyName}}) {
39          {{nameClass}} obj = {{nameVar}}Dao.get({{keyName}});
40
41          return ok({{nameVar}}{{conf.showSuffix}}.render(obj));
42      }
43
44      public  Result edit({{keyType}} {{keyName}}) {
45          Form<{{nameClass}}> f = form({{nameClass}}.class).fill(
46              {{nameVar}}Dao.get({{keyName}})
47          );
48          return ok({{nameVar}}{{conf.formSuffix}}.render({{keyName}}, f));
49      }
50
51      public  Result update({{keyType}} {{keyName}}) {
52          Form<{{nameClass}}> f = form({{nameClass}}.class).bindFromRequest();
53          if(f.hasErrors()) {
54              return badRequest({{nameVar}}{{conf.formSuffix}}.render({{keyName}}, f)
                    );
55          }
56
57          {{nameClass}} obj = f.get();
58          obj.{{keyName}} = {{keyName}};
59
60          {{#fields}}
61          {{#hasMappedBy}}
62          obj.{{fieldName}}.forEach(o -> o.{{metafield.mappedBy}} = obj);
63          {{/hasMappedBy}}
64          {{/fields}}
65          {{nameVar}}Dao.update(obj);
66
67          flash(SUCCESS_KEY, Messages.get(
68              "crud.flash.update.success", "{{nameClass}}", "{{keyName}}", obj.{{
                    keyName}}));
69
70          return index();
71      }
72
73      public  Result create() {
74          Form<{{nameClass}}> f = form({{nameClass}}.class);
75
76          return ok(
77              {{nameVar}}{{conf.formSuffix}}.render(null, f)
78          );
79      }
80
```

```
81      public  Result save() {
82          Form<{{nameClass}}> f = form({{nameClass}}.class).bindFromRequest();
83          if(f.hasErrors()) {
84              return badRequest({{nameVar}}{{conf.formSuffix}}.render(null, f));
85          }
86
87          {{nameClass}} obj = f.get();
88
89          {{#fields}}
90          {{#hasMappedBy}}
91          obj.{{fieldName}}.forEach(o -> o.{{metafield.mappedBy}} = obj);
92          {{/hasMappedBy}}
93          {{/fields}}
94          {{nameVar}}Dao.save(obj);
95
96          flash(SUCCESS_KEY, Messages.get(
97              "crud.flash.create.success", "{{nameClass}}", "{{keyName}}", obj.{{
                    keyName}}));
98
99          return index();
100     }
101
102     public  Result delete({{keyType}} {{keyName}}) {
103         {{nameVar}}Dao.delete({{keyName}});
104
105         flash(SUCCESS_KEY, Messages.get(
106             "crud.flash.delete.success", "{{nameClass}}", "{{keyName}}", {{keyName
                    }}));
107
108         return index();
109     }
110
111     // Helper para selects
112     public static Map<String,String> options() {
113         Map<String, String> options = new LinkedHashMap<>();
114         for({{nameClass}} obj: {{nameVar}}Dao.all())
115             options.put(obj.{{keyName}} + "", obj.toString());
116
117         return options;
118     }
119 }
```

Apêndice B

Roteiro Experimental

Introdução

Este roteiro deve ser seguido para a obtenção dos dados que serão utilizados para análise e conclusões referentes ao software *Metaffolder*.

Ele está dividido em 2 experimentos, cada um com três fases: treinamento, execução e avaliação. Na fase de treinamento você deverá observar o avaliador enquanto ele realiza as tarefas deste roteiro. Quando ele acabar você deverá repetir tudo o que aprendeu nesta fase, basicamente seguir passo a passo as orientações desse roteiro. O avaliador poderá ser consultado para qualquer dúvida ou problema técnico. Na fase de execução você tentará executar tarefas parecidas mas sem qualquer auxílio do avaliador. Na fase de avaliação você responderá um questionário de usabilidade.

Cada experimento possui as fases de treinamento, execução e avaliação. As fases de treinamento e execução são subdivididas em duas e três tarefas, respectivamente. Ambas possuem as tarefas de aplicação da técnica de scaffolding e verificação das funcionalidades CRUD do sistema gerado. A fase de execução possui uma tarefa adicional: a codificação manual do sistema criado pela técnica de scaffolding. O tempo gasto na execução de cada tarefa é cronometrado e anotado para análise. Portanto, ao final dos experimentos, cada participante terá gerado 10 tempos: 5 utilizando o sistema *Metaffolder* e 5 utilizando o framework Rails.

O resumo deste roteiro é exibido a seguir:

```
Experimento 1 - Sistema Metaffolder
      i - Fase de Treinamento
            a) Aplicação da técnica de scaffolding - T1
            b) Verificação das funcionalidades CRUD - T2
     ii - Fase de Execução
            a) Aplicação da técnica de Scaffolding - T3
            b) Verificação das funcionalidades CRUD - T4
            c) Codificação manual do sistema gerado - T5
    iii - Fase de Avaliação
            a) Aplicação do questionário de usabilidade.

Experimento 2 - Framework Rails
      i - Fase de Treinamento
            a) Aplicação da técnica de Scaffolding - T1
            b) Verificação das Funcionalidades CRUD - T2
     ii - Fase de Execução
            a) Aplicação da técnica de Scaffolding - T3
            b) Verificação das funcionalidades CRUD - T4
            c) Codificação manual do sistema gerado - T5
    iii - Fase de Avaliação
            a) Aplicação do questionário de usabilidade.
```

Experimento 1 - Sistema Metaffolder

Nesta parte será utilizado o sistema *Metaffolder* para a realização das tarefas. O avaliador demonstrará passo a passo como aplicar a técnica de scaffolding para gerar um sistema CRUD a partir de classes de modelo. O funcionamento do sistema gerado também será demonstrado pelo avaliador. Observe seus passos atentamente. Ao final da demonstração você deverá realizar as mesmas tarefas e o tempo será cronometrado. Você poderá consultar o avaliador caso tenha quaisquer dúvidas. Todos os tempos obtidos serão salvos em uma folha de dados *online,* no *link* `http://goo.gl/forms/Zscpo5FsB8`.

Fase de Treinamento [Avaliador]

- 1ª Tarefa - Aplicação da Técnica de Scaffolding

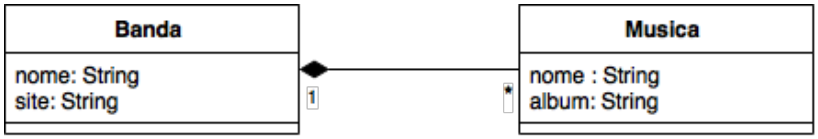


Figura B.1: Experimento 1 - Treinamento - Diagrama de Classes 1

1. Abra o terminal com o atalho CTRL+ALT+T.
2. Navegue até a pasta de trabalho com o comando cd ~/workspace.
3. Crie uma nova pasta para o experimento 1 (sistema *Metaffolder*): mkdir experimento1.
4. Entre na pasta criada: cd experimento1.
5. Utilize o framework Play para criar uma nova aplicação: activator new treinamento play-java.
6. Entre na pasta criada pelo framework: cd treinamento.
7. Abra o editor de texto Sublime Text 3: subl ..
8. Exclua o diretório **app** usando o menu lateral do Sublime Text. Todos os arquivos necessários serão criados pelo *Metaffolder*.
9. Exclua o arquivo **conf/routes**.
10. No arquivo **project/plugins.sbt** descomente o plugin do Ebean (linha 21).
11. No arquivo **project/plugins.sbt** adicione a linha

```
1    addSbtPlugin("masters" % "sbt-metaffolder"  % "1.0.0")
```

12. No arquivo **build.sbt** ative os plugins do `Ebean` e `Metaffolder` alterando a linha 5.

```
1    lazy val root = (project in file(".")).enablePlugins(PlayJava,
         PlayEbean, Metaffolder)
```

13. No arquivo **conf/application.conf** descomente as linhas 33 a 36 para ativar o banco de dados.
14. No arquivo **conf/application.conf** adicione os modelos ao banco de dados com a linha abaixo.

```
1    ebean.default = ["models.*"]
```

15. Crie o arquivo **app/models/Banda.java**.

```
1  package models;
2
3  import javax.persistence.*;
4  import java.util.List;
5
6  @Entity
7  public class Banda {
8      @Id
9      public Long id;
10     public String nome;
11     public String site;
12
13     @OneToMany(mappedBy = "banda", cascade = CascadeType.ALL)
14     public List<Musica> musicas;
15
16     public String toString() {
17         return getClass().getSimpleName() + "@" + id;
18     }
19 }
```

16. Crie o arquivo **app/models/Musica.java**.

```
1  package models;
2
3  import javax.persistence.*;
4
5  @Entity
6  public class Musica {
7      @Id
8      public Long id;
9      public String nome;
10     public String album;
11
12     @ManyToOne(optional = false)
13     public Banda banda;
14
15     public String toString() {
16         return getClass().getSimpleName() + "@" + id;
17     }
18 }
```

17. No terminal execute o comando `activator` para iniciar o framework Play.
18. Execute o comando `metaffold` para a aplicação da técnica de scaffolding. Vários arquivos serão gerados automaticamente pelo *Metaffolder*.
19. Inicie o sistema CRUD gerado com o comando `run`.

20. Abra o Firefox e entre no endereço localhost:9000.
21. Clique no botão "Apply this script now!" para criar o banco de dados do sistema gerado.

- 2ª Tarefa - Verificação das Funcionalidades CRUD do Sistema Gerado

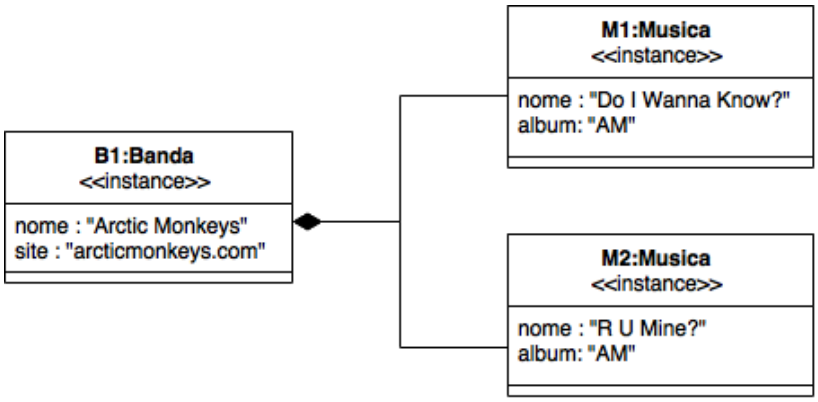


Figura B.2: Experimento 1 - Treinamento - Diagrama de Objetos 1

22. Crie a instância `B1`. Esta operação testa a criação de objetos (Create).
23. Crie as instâncias `M1` e `M2`.
24. Visualize o objeto `B1`. Esta operação testa a leitura de objetos (Read). Repare que os objetos `M1` e `M2` estão contidos em `B1`. Isso garante o funcionamento correto do relacionamento entre as classes Banda e Musica.
25. Remova a associação entre `B1` e `M2` e salve. Esta operação testa a alteração de objetos (Update). Perceba que o objeto `M2` foi excluído. Isso garante a composição entre as classes `Banda` e `Musica`, isto é, todo objeto `Musica` exige um relacionamento com um objeto `Banda`.
26. Exclua o objeto `B1`. Esta operação testa a exclusão de objetos (Delete). O objeto `M1` também foi excluído. Esta operação também garante a composição entre as classes `Banda` e `Musica`.

Fase de Treinamento [Estudante]

- 1ª Tarefa - Aplicação da Técnica de Scaffolding
  - Digite o comando ~/timer start no terminal para iniciar um cronômetro em segundo plano.
  - Repita os passos 1 a 21 executados pelo avaliador. Ele está a disposição para dúvidas ou problemas técnicos.

  - Digite o comando ~/timer stop no terminal para encerrar o cronômetro. Anote no campo T1 do formulário online a duração exibida no terminal.

- 2ª Tarefa - Verificação das Funcionalidades CRUD do Sistema Gerado

  - Digite o comando ~/timer start no terminal para iniciar um cronômetro em segundo plano.
  - Repita os passos 22 a 26 executados pelo avaliador. Ele está a disposição para dúvidas ou problemas técnicos.
  - Digite o comando ~/timer stop no terminal para encerrar o cronômetro. Anote no campo T2 do formulário online a duração exibida no terminal.

Fase de Execução [Estudante]

- 1ª Tarefa - Aplicação da Técnica de Scaffolding

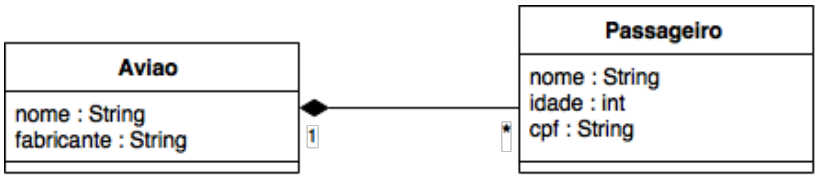


Figura B.3: Experimento 1 - Execução - Diagrama de Classes 2

  - Digite o comando ~/timer start no terminal para iniciar um cronômetro em segundo plano.
  - Repita os passos 1 a 21 executados pelo avaliador na fase de treinamento, mas desta vez utilize o diagrama de classes da Figura B.3. Siga os passos com calma e atenção. O avaliador não poderá ser consultado nesta fase do experimento. Obs: Ao repitir o passo 5, altere o comando para activator new execucao play-java.
  - Digite o comando ~/timer stop no terminal para encerrar o cronômetro. Anote no campo T3 do formulário online a duração exibida no terminal.

- 2ª Tarefa - Verificação das Funcionalidades CRUD do Sistema Gerado

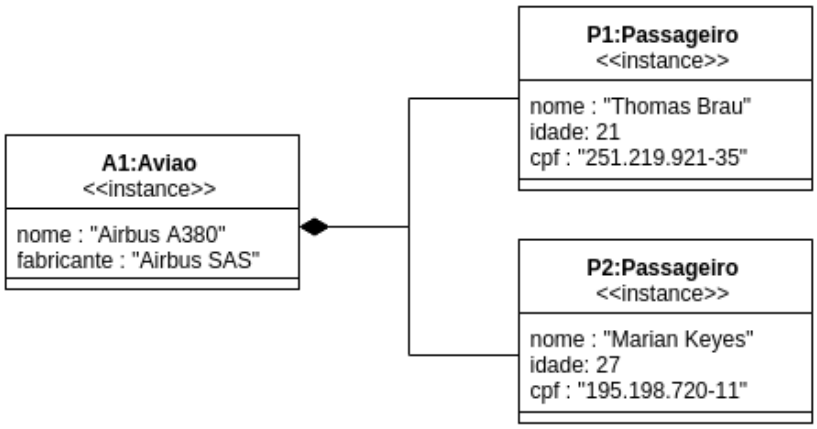


Figura B.4: Experimento 1 - Execução - Diagrama de Objetos 2

  - Digite o comando ~/timer start no terminal para iniciar um cronômetro em segundo plano.
  - Repita os passos 22 a 26 executados pelo avaliador na fase de treinamento, mas desta vez utilize o diagrama de objetos da Figura B.4. Siga os passos com calma e atenção. O avaliador não poderá ser consultado nesta fase do experimento.
  - Digite o comando ~/timer stop no terminal para encerrar o cronômetro. Anote no campo T4 do formulário online a duração exibida no terminal.

- 3ª Tarefa - Codificação Manual

  - Digite o comando ~/timer start no terminal para iniciar um cronômetro em segundo plano.
  - Abra o terminal com o atalho CTRL+ALT+T.
  - Navegue até a pasta do primeiro experimento: cd /workspace/experimento1.
  - Crie a pasta **codificacao**: mkdir codificacao.
  - Abra o editor de texto Sublime Text 3: subl .. Observe que dentro da pasta **experimento1** existe as pastas **treinamento**, **execucao** e **codificacao**.
  - Divida a janela do Sublime ao meio com o atalho SHIFT+ALT+8. Isso facilitará o processo de cópia. Deixe o arquivo original em uma metade e o novo arquivo na outra.
  - Copie manualmente todos os arquivos da pasta **execucao/app** para a pasta **codificacao/app**, isto é, cada arquivo deverá ser criado e digitado, do zero. A cópia de trechos de código somente está permitida para trechos já digitados por você. Isso pode ser útil em arquivos com conteúdo semelhante, que é o caso da camada de visão. Obs.: comentários não precisam ser copiados.

  - Repita o procedimento anterior para os arquivos **routes** e **messages** da pasta **execucao/conf**.
  - Digite o comando ~/timer stop no terminal para encerrar o cronômetro. Anote no campo T5 do formulário online a duração exibida no terminal.

Fase de Avaliação [Estudante]

- Tarefa - Aplicação do Questionário de Usabilidade
  - Você deverá responder um questionário referente à usabilidade da técnica de scaffolding do *Metaffolder*, e ao sistema gerado por ela. O questionário é simples, rápido e se encontra online no link `http://goo.gl/forms/oOoqxRWOql`.

Experimento 2 - Framework Rails

Nesta parte será utilizado o framework Rails para a realização dos procedimentos. O avaliador demonstrará passo a passo como aplicar a técnica de scaffolding para gerar um sistema CRUD a partir de classes de modelo. O funcionamento do sistema gerado também será demonstrado pelo avaliador. Observe seus passos atentamente. Ao final da demonstração você deverá realizar o mesmo procedimento e o tempo será cronometrado. Todos os tempos obtidos serão salvos em uma folha de dados online, no link `http://goo.gl/forms/6HGRAJ8LVH`. Vale ressaltar que o scaffolding do Rails não gera relacionamentos na camada de visão e isto precisou ser feito manualmente. Ele também não gera a navegação entre modelos diferentes (links).

Fase de Treinamento [Avaliador]

- Aplicação da Técnica de Scaffolding

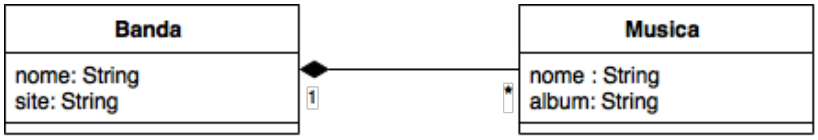


Figura B.5: Experimento 2 - Treinamento - Diagrama de Classes 1

1. Abra o terminal com o atalho CTRL+ALT+T.
2. Navegue até a pasta de trabalho com o comando cd ~/workspace.
3. Crie uma nova pasta para o experimento 2 caso ela não exista (framework Rails): mkdir experimento2.

4. Entre na pasta criada: cd experimento2.
5. Utilize o framework Rails para criar uma nova aplicação: rails new treinamento.
6. Entre na pasta criada pelo framework: cd treinamento.
7. Crie a classe Banda (Figura B.5) e aplique a técnica de scaffolding: rails g scaffold Banda nome site
8. Crie a classe Musica (Figura B.5) e aplique a técnica de scaffolding: rails g scaffold Musica nome album banda:references
9. Abra o editor de texto Sublime Text 3: subl ..
10. Altere o arquivo **app/models/banda.rb**.

```
1 class Banda < ActiveRecord::Base
2   has_many :musicas, dependent: :delete_all
3
4   def to_s
5     Banda.name.demodulize + "@" + self.id.to_s
6   end
7 end
```

11. Altere o arquivo **app/models/musica.rb**.

```
1 class Musica < ActiveRecord::Base
2   belongs_to :banda
3   validates_presence_of :banda
4
5   def to_s
6     Musica.name.demodulize + "@" + self.id.to_s
7   end
8 end
```

12. No terminal digite o comando rake db:migrate
13. No arquivo **app/controllers/bandas_controller.rb** modifique a linha 72 permitindo mais um parâmetro.

```
1     params.require(:banda).permit(:nome, :site, musica_ids: [])
```

14. No arquivo **app/views/bandas/_form.html.erb** adicione o código abaixo entre as linhas 21 e 22.

```
1 <div class="field">
2     <%= f.label :musica_id %><br />
3     <%= f.select :musica_ids, Musica.all.collect {|x| [x.to_s, x.id]},
          {}, multiple: true %>
4 </div>
```

15. No arquivo **app/views/musicas/_form.html.erb** substitua a linha 24 pela linha abaixo.

```
1 <%= f.select :banda_id, Banda.all.collect {|x| [x.to_s, x.id]}, {} %>
```

16. No terminal, inicie o sistema CRUD gerado com o comando rails server.
17. Abra o Firefox e entre no endereço localhost:3000.

- Verificação das Funcionalidades CRUD do Sistema Gerado

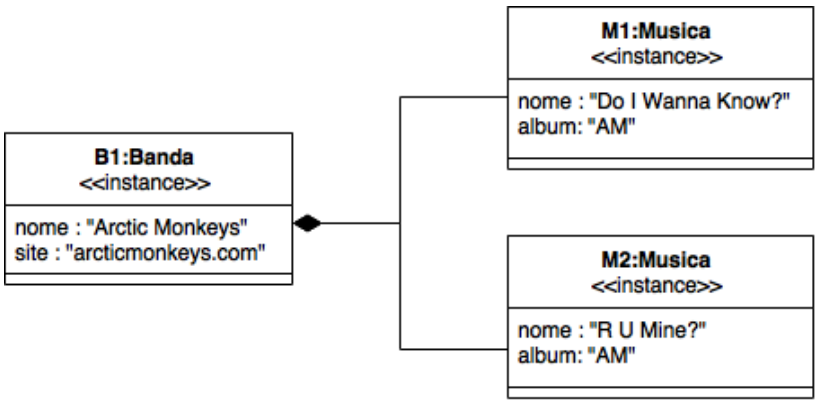


Figura B.6: Experimento 2 - Treinamento - Diagrama de Objetos 1

18. Acesse o endereço localhost:3000/bandas.
19. Crie a instância `B1`. Esta operação testa a criação de objetos (Create).
20. Acesse o endereço localhost:3000/musicas.
21. Crie as instâncias `M1` e `M2`.
22. Visualize o objeto `B1`. Esta operação testa a leitura de objetos (Read). Repare que os objetos `M1` e `M2` estão contidos em `B1`. Isso garante o funcionamento correto do relacionamento entre as classes Banda e Musica.
23. Remova a associação entre `B1` e `M2` e salve. Esta operação testa a alteração de objetos (Update). Perceba que o objeto `M2` foi excluído. Isso garante a composição entre as classes `Banda` e `Musica`, isto é, todo objeto `Musica` exige um relacionamento com um objeto `Banda`.
24. Exclua o objeto `B1`. Esta operação testa a exclusão de objetos (Delete). O objeto `M1` também foi excluído. Esta operação também garante a composição entre as classes `Banda` e `Musica`.

Fase de Treinamento [Estudante]

- Aplicação da Técnica de Scaffolding
  - Digite o comando `./timer start` no terminal para iniciar um cronômetro em segundo plano.
  - Repita os passos 1 a 17 executados pelo avaliador. Ele está a disposição para dúvidas ou problemas técnicos.
  - Digite o comando `./timer stop` no terminal para encerrar o cronômetro. Anote no campo T1 do formulário online a duração exibida no terminal.
- Verificação das Funcionalidades CRUD do Sistema Gerado
  - Digite o comando `./timer start` no terminal para iniciar um cronômetro em segundo plano.
  - Repita os passos 18 a 24 executados pelo avaliador. Ele está a disposição para dúvidas ou problemas técnicos.
  - Digite o comando `./timer stop` no terminal para encerrar o cronômetro. Anote no campo T2 do formulário online a duração exibida no terminal.

Fase de Execução [Estudante]

- Aplicação da Técnica de Scaffolding

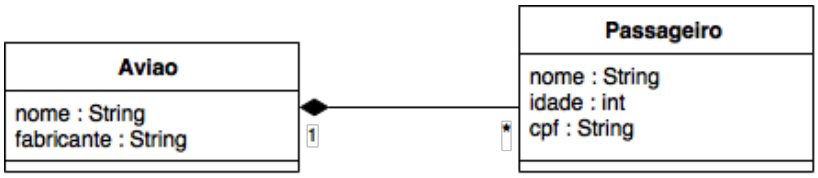


Figura B.7: Experimento 2 - Execução - Diagrama de Classes 2

  - Digite o comando `~/timer start` no terminal para iniciar um cronômetro em segundo plano.
  - Repita os passos 1 a 17 executados pelo avaliador na fase de treinamento, mas desta vez utilize o diagrama de classes da Figura B.7. Siga os passos com calma e atenção. O avaliador não poderá ser consultado nesta fase do experimento. Obs: Ao repitir o passo 5, altere o comando para `rails new execucao`.
  - Digite o comando `~/timer stop` no terminal para encerrar o cronômetro. Anote no campo T3 do formulário online a duração exibida no terminal.

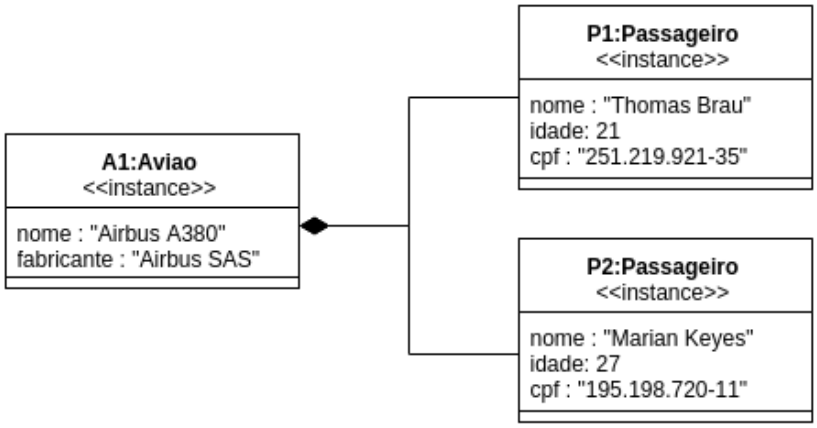


Figura B.8: Experimento 2 - Execução - Diagrama de Objetos 2

- Verificação das Funcionalidades CRUD do Sistema Gerado
  - Digite o comando `~/timer start` no terminal para iniciar um cronômetro em segundo plano.
  - Repita os passos 18 a 24 executados pelo avaliador na fase de treinamento, mas desta vez utilize o diagrama de objetos da Figura B.8. Siga os passos com calma e atenção. O avaliador não poderá ser consultado nesta fase do experimento.
  - Digite o comando `~/timer stop` no terminal para encerrar o cronômetro. Anote no campo T4 do formulário online a duração exibida no terminal.
- 3ª Tarefa - Codificação Manual
  - Digite o comando `~/timer start` no terminal para iniciar um cronômetro em segundo plano.
  - Abra o terminal com o atalho `CTRL+ALT+T`.
  - Navegue até a pasta do segundo experimento: `cd /workspace/experimento2`.
  - Crie a pasta **codificacao**: `mkdir codificacao`.
  - Abra o editor de texto Sublime Text 3: `subl ..` Observe que dentro da pasta **experimento2** existe as pastas **treinamento**, **execucao** e **codificacao**.
  - Divida a janela do Sublime ao meio com o atalho `SHIFT+ALT+8`. Isso facilitará o processo de cópia. Deixe o arquivo original em uma metade e o novo arquivo na outra.
  - Copie manualmente todos os arquivos da pasta **execucao/app** para a pasta **codificacao/app**, isto é, cada arquivo deverá ser criado e digitado, do zero. A cópia de trechos de código somente está permitida para trechos já digitados por você. Isso pode ser útil em arquivos com conteúdo semelhante, que é o caso da camada de visão. Obs.: comentários não precisam ser copiados.

  - Repita o procedimento anterior para todos os arquivos com extensão **.rb** da pasta **execucao/db**.
  - Repita o procedimento anterior para o arquivo **routes.rb** da pasta **execucao/config**.
  - Digite o comando `~/timer stop` no terminal para encerrar o cronômetro. Anote no campo T5 do formulário online a duração exibida no terminal.

Fase de Avaliação [Estudante]

- Tarefa - Aplicação do Questionário de Usabilidade
  - Você deverá responder um questionário referente à usabilidade da técnica de scaffolding do framework Rails, e ao sistema gerado por ela. O questionário é simples, rápido e se encontra online no link `http://goo.gl/forms/oOoqxRWOql`.

Apêndice C

Dados Experimentais

Este Apêndice apresenta todos os dados obtidos no experimentos realizados neste trabalho. A Tabela C.1 exibe o tempo gasto por cada um dos 10 participantes na execução dos experimentos referentes aos *softwares Metaffolder* e *Rails*. Foram calculados a média e o desvio padrão de cada tempo. As tabelas Tabela C.2 e Tabela C.3 apresentam as respostas do questionário e as pontuações SUS relativas aos *softwares Metaffolder* e *Rails* respectivamente. Também foram calculados a média e o desvio padrão das pontuações obtidas.

Tabela C.1: Tempo gasto na execução das tarefas dos Experimentos 1 e 2 (em segundos)

	Experimento 1 - Metaffolder					Experimento 2 - Ruby on Rails				
	Fase de Treinamento		Fase de Execução			Fase de Treinamento		Fase de Execução		
Participante	T1	T2	T3	T4	T5	T1	T2	T3	T4	T5
1	958	204	733	177	7283	668	177	591	160	3825
2	989	175	702	112	10384	682	223	592	209	5051
3	1063	220	832	181	10981	682	247	602	232	6021
4	1268	216	1095	190	8985	851	171	746	163	6029
5	968	215	721	154	9642	905	188	804	168	4903
6	1114	168	939	162	7877	600	207	511	173	4777
7	1161	157	883	144	11361	880	197	780	137	5020
8	785	160	643	147	8771	903	181	851	212	5530
9	1026	180	854	163	9619	698	173	727	172	4477
10	942	191	763	174	9209	807	251	782	214	4794
Média	1027	189	817	160	9411	768	202	699	184	5043
Desvio Padrão	134	24	134	23	1283	113	30	115	30	677

Tabela C.2: Respostas dos participantes (escala de 1 a 5) e a pontuação SUS do sistema *Metaffolder*

	Experimento 1 - Metaffolder										
	Respostas (Questões 1 a 10)										
Participante	1	2	3	4	5	6	7	8	9	10	Pontuação SUS
1	5	1	5	1	4	2	5	2	5	1	92,5
2	5	1	5	1	4	2	5	1	5	1	95,0
3	4	1	5	1	4	2	3	1	5	2	85,0
4	4	2	4	2	5	2	4	1	4	3	77,5
5	4	2	3	1	4	1	3	3	4	3	70,0
6	4	2	4	1	4	2	4	1	4	2	80,0
7	4	1	4	1	5	1	4	2	4	2	85,0
8	4	2	5	1	4	2	5	1	4	3	82,5
9	5	2	4	1	4	1	3	2	4	2	80,0
10	4	1	5	1	5	1	4	1	3	2	87,5
										Média	83,5
										Desvio Padrão	7,3

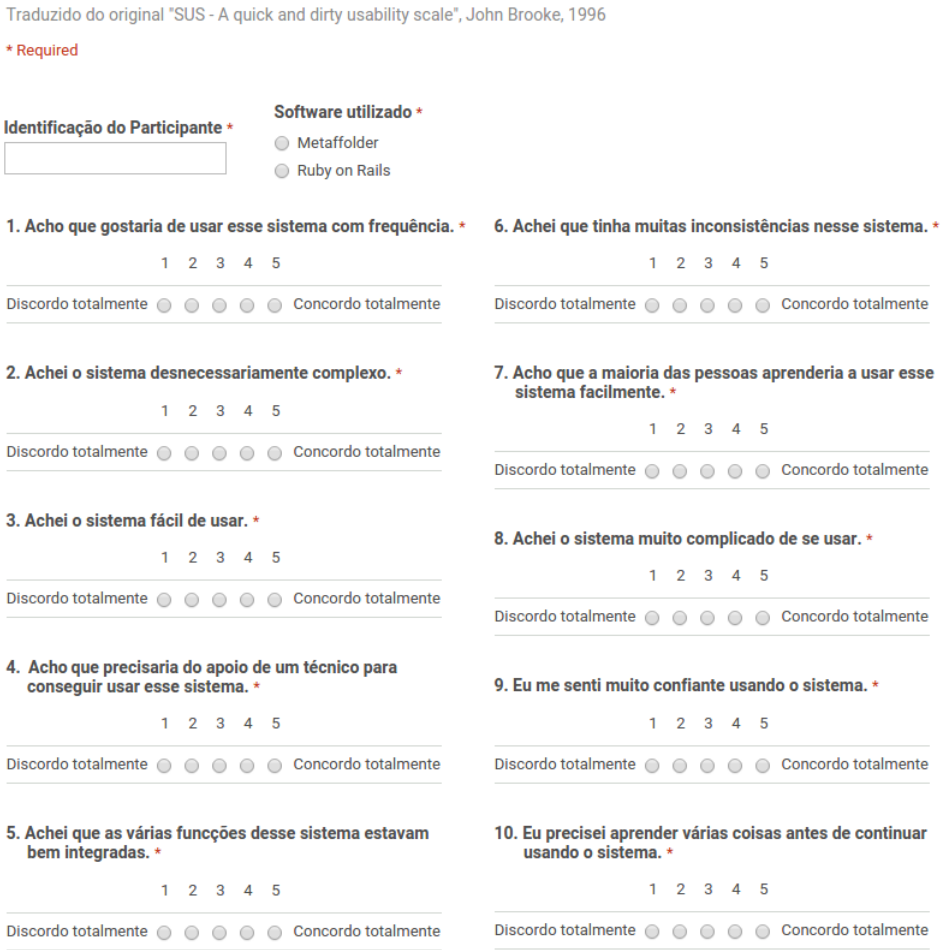
Tabela C.3: Respostas dos participantes (escala de 1 a 5) e a pontuação SUS do *framework Rails*

	Experimento 2 - Ruby on Rails										
	Respostas (Questões 1 a 10)										
Participante	1	2	3	4	5	6	7	8	9	10	Pontuação SUS
1	4	1	4	1	5	1	5	1	5	2	92,5
2	5	2	5	1	5	1	4	2	4	3	85,0
3	4	2	4	1	5	1	4	2	4	1	85,0
4	5	1	4	2	5	1	5	2	3	3	82,5
5	4	1	5	1	4	2	4	2	4	1	85,0
6	4	1	4	2	5	1	4	1	4	3	82,5
7	4	2	5	1	5	1	5	2	4	3	85,0
8	4	1	5	1	5	1	5	1	4	2	92,5
9	5	1	4	2	5	1	4	2	5	2	87,5
10	4	1	5	1	4	2	4	2	4	2	82,5
										Média	86,0
										Desvio Padrão	3,8

Anexo A

Questionário de Avaliação de Usabilidade

Este Anexo apresenta o questionário utilizado para determinar o nível de usabilidade dos *softwares Metaffolder* e *Rails* e seus respectivos sistemas CRUD. O questionário foi criado no Google Forms. Ele se encontra disponível online no link `http://goo.gl/forms/oOoqxRWOql`.



Traduzido do original "SUS - A quick and dirty usability scale", John Brooke, 1996

* Required

Identificação do Participante *

Software utilizado *
- Metaffolder
- Ruby on Rails

1. Acho que gostaria de usar esse sistema com frequência. *

1 2 3 4 5

Discordo totalmente ○ ○ ○ ○ ○ Concordo totalmente

2. Achei o sistema desnecessariamente complexo. *

1 2 3 4 5

Discordo totalmente ○ ○ ○ ○ ○ Concordo totalmente

3. Achei o sistema fácil de usar. *

1 2 3 4 5

Discordo totalmente ○ ○ ○ ○ ○ Concordo totalmente

4. Acho que precisaria do apoio de um técnico para conseguir usar esse sistema. *

1 2 3 4 5

Discordo totalmente ○ ○ ○ ○ ○ Concordo totalmente

5. Achei que as várias funcções desse sistema estavam bem integradas. *

1 2 3 4 5

Discordo totalmente ○ ○ ○ ○ ○ Concordo totalmente

6. Achei que tinha muitas inconsistências nesse sistema. *

1 2 3 4 5

Discordo totalmente ○ ○ ○ ○ ○ Concordo totalmente

7. Acho que a maioria das pessoas aprenderia a usar esse sistema facilmente. *

1 2 3 4 5

Discordo totalmente ○ ○ ○ ○ ○ Concordo totalmente

8. Achei o sistema muito complicado de se usar. *

1 2 3 4 5

Discordo totalmente ○ ○ ○ ○ ○ Concordo totalmente

9. Eu me senti muito confiante usando o sistema. *

1 2 3 4 5

Discordo totalmente ○ ○ ○ ○ ○ Concordo totalmente

10. Eu precisei aprender várias coisas antes de continuar usando o sistema. *

1 2 3 4 5

Discordo totalmente ○ ○ ○ ○ ○ Concordo totalmente

Anotações